A Consistency-based Approach to Knowledge Base Refinement

Tri M. Cao and Paul Compton
School of Computer Science and Engineering
University of New South Wales,
Sydney 2052, Australia
tmc@cse.unsw.edu.au

Abstract

In this work, we outline an approach to incrementally building knowledge-based systems based on tightly controlling the order of evaluation of the knowledge components of the system. The order of evaluation is based on two relations, sequence and correction, that correspond to two kinds of changes that an expert may wish to make to a knowledge base. A consistency notion based on past performance is also introduced. Knowledge base refinement is structured so that newly added knowledge has one of these relations with existing knowledge components in the system and the consistency condition is preserved. We further propose that the knowledge components added might be any knowledge-based systems or programs rather than rules. This proposal is a generalisation of the Ripple-Down Rule incremental approach to building knowledge-based systems.

Introduction

A rule based system can be described a set of rules, an inference engine to enable the rules to be evaluated, mechanisms to control the order in which rules are evaluated and the working memory which contains input data plus any output from rules that fire. The control information for how the rules are evaluated can be encoded as a *dependency graph* (Apt, Blair, & Walker 1988; Colomb 1999) or implicitly stated in the software that implements the inference engine. Even though there has been some move towards second generation expert systems (which contain a conceptual model of the domain), the majority of expert systems developed to date have been rule based.

The importance of controlling the order of rule evaluation should be noted. It is often assumed that declarative programming avoids the programmers and knowledge engineers having to think about the order of evaluation. This is certainly not the case in rule-based systems and is probably not true for any declarative programming. Rule-based systems include control mechanisms such as conflict resolution strategies to decide which rule should be fired (its conclusion added to working memory) first if a number of rules are satisfied by the data. These strategies are essentially heuristic; e.g, fire the rule with greatest number of conditions first, so that there will be circumstance where the ordering is inappropriate. The knowledge engineer then changes the rules to make the conflict resolution strategy give the right conclusion. Conflict resolution applies to the order in which rule fires; the inference engine also evaluates rules in a particular order, which will need to be taken into account. For example, the MYCIN backward chaining inference engine evaluates rule conditions in the order in which the conditions occur in the rule. The MYCIN knowledge engineers therefore organised the order of conditions in rules so that when the system asks the user about a particular rule condition, it does this in an order that is appropriate for the domain. Much of the following is about developing a very explicit ordering for evaluation and firing so that they cannot and do not need to be controlled by the person adding rules. That is, knowledge acquisition becomes more declarative if firing and evaluation and firing ordering becomes more procedural.

This description of a rule-based system can be generalised if we consider that a single rule plus inference engine can be considered as a program: that is, given some input the rule evaluation may result in some output (if the rule fires). A rule-based system can thus be generalised to a set of programs, mechanisms to control the order in which these programs are run and working memory where the original input data plus any output from the various programs is stored. Of course what makes such a system a knowledge- based is that the programs (or rules) are added to capture the preferences and beliefs of the owner (supervisor, teacher) of the system in some sort of knowledge acquisition process.

In such systems there would be two situations where knowledge acquisition was required: firstly the system's output is correct but incomplete and secondly that part of whole of the output is wrong. If we assume that we cannot look inside the system to fix it and that the system has a lot of value so that we do not want to discard it, then these errors must be fixed by adding a second KBS so that the output of the first is passed to the second to have extra information added or to have the output from the first system replaced in the specific circumstances where it has been found to be wrong.

This is what must happen at the atomic rule level, except hidden by the editing that occurs.

- If we have a system with a single rule then we add an extra rule to deal with some new circumstance.

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

- If the rule fires inappropriately we add extra conditions to it to restrict the circumstance in which it fires and add a new rule to give the correct conclusion in the circumstances. Despite the two step editing, what is happening logically is that the new rule specifies the circumstances in which its conclusion should be given rather than the previous rule. That is if both rules fire the conclusion of the later rule replaces that of the initial rule.

If we cannot edit the rule, (or the black box that has produced the incorrect output), all we can do is to add a program that replaces the output of the first program in certain circumstances.

Cases that are used to test the knowledge base are of central importance. The only possible of way of characterising, testing or evaluating a knowledge base is via a set of cases. As will be discussed, these cases may not test every possible behaviour of the KBS; the testing is only as good as the cases available. However, regardless of the quality of the cases, they are the only way of characterising the system.

This can be seen in the example of a single rule: If the single rule is an overgeneralisation and gives the wrong conclusion for a case, it is equally that the new rule that replaces the conclusion of the first rule, will also be an overgeneralisation. If the expert overgeneralised the first time, how can he or she be relied upon to do better next time. If the new rule is to result in an improvement there must be a case or cases which correctly fire the first rule and are assumed to specify its scope, which need to be tested to see that they are still handled correctly by the first rule rather than the correction rule. The philosophical arguments for why an expert can never be relied on are outlined in (Compton & Jansen 1990).

A central insight of this work is that is that the two tasks of adding knowledge to add conclusions or replace conclusions do not have to be implicit in knowledge engineering. Rather systems can be recursively structured so that all knowledge acquisition is explicitly achieved by adding a KBS/program/rule to augment or replace output. Secondly, we hypothesise that there can be no advantage in carrying out these tasks in an implicit way by editing the knowledge base. Rather, there is risk or introducing other errors in such editing. The assumption here is that the circumstances in which knowledge is appropriate are never fully defined (and cannot be) so that it is inappropriate to hope for a perfect fix for such knowledge, the fix will never be complete. Ultimately the only way to handle errors is to provide knowledge of the circumstances where output has to be added to or replaced. And this knowledge will also need to be augmented or replaced.

Rule-level versions of these ideas are known as Ripple-Down Rules (RDR). Various RDR approaches have been developed and applied to a range of domains including: pathology (Edwards & Compton 1993), configuration (Compton *et al.* 1998), heuristic search (Beydoun & Hoffmann 2000), document management using multiple classification (Kang *et al.* 1997), and resource allocation (Richards & Compton 1998) with considerable success.

The generalisation here is an extension of a previous generalisation applying only to rules (Compton & Richards 2000). Beydoun and Hoffman have also generalised RDR with a multiple RDR knowledge base approach, Nested RDR (NRDR). However, their linking between knowledge bases is via intermediate conclusions of concepts as used in heuristic classification (Clancey 1985). The generalisation here attempts to control the linking between knowledge-bases themself in an RDR-like fashion.

Basic Concepts

Input

The inputs of each system are called cases. A case is the data, relationships in the data and any theory that may be provided as input to the KBS plus the output of the KBS. Note that a KBS cannot change the case it is provided with; it can only add to it. That is a KBS linked to a blackboard could not delete information from the blackboard, it could only add information. This does not mean the KBS cannot decide there is something wrong with the information it is provided. It means that if the case were rerun after being processed, the same output would be provided again. The the output would be the original input plus some sort of statement that there is something wrong with the input.

Informally, we define a case as a finite list of atomic objects which can be originally given or as output from evaluation. The atomic object representation will depend on the underlying language used. For example, if the language used is a first order language, the objects will be the set of atomic formulas. For a configuration task, the objects here are variable assignments. The order of the list is important because it keeps track of the order of evaluation.

Output

Output can be either information of some type that is added to the case, or a request for some other agent to add information to the case. If information is added, it may be a classification, a design or theory or it may be adding relations to the data; e.g. in a resource allocation problem resources are assigned to users of those resources, and perhaps temporal or spatial relations are also added specifying the temporal sequence and relative locations of resources.

If the output is a request for another agent to add information, this agent may be a human who provides other information or a program for example carrying out a calculation, another KBS of the type defined above or another kind of knowledge based system.

In particular, a special symbol, called NO_OUTPUT is used when no output can be derived from the current input. That is the indication the system does not have knowledge relevant to the case in hand.

Primary rules

A primary rule (or a clausal rule) is a formula of the form

$$O \longleftarrow L_1, L_2, \ldots, L_m$$

where the L_i's are conditions that refer to information in the case and the condition will be true or false depending on whether the relevant object is in the case. O is either an atomic object or a request for a additional information or the special symbol, NO_OUTPUT.

RDR Agent

An RDR agent manages how control is passed between the various programs used and how data is posted to the blackboard and passed to programs. Note that the programs called may themselves be other RDR agents who may have their own blackboards and programs which they call. The RDR agent is simply a blackboard controller, but one which organises how programs are called. The RDR agent does not have any explicit human knowledge, as the control structure it learns is determined simply by whether a correction or an additional knowledge base is invoked.

Control mechanism

The control mechanisms here are very simple and are of three types. One type of control mechanism handles requests for specific programs (see output above), the other organises the sequence of KBS independent of requests for specific agents. The sequence is determined by two types of relations between KBS: sibling and correction relations. These two relations are determined by the knowledge acquisition process.

For completeness there is also a general control mechanism that after each addition to the case (output posted to the blackboard), the whole reasoning process restarts with the first KBS. This is not a strict requirement, but the reasons for this will be outlined under knowledge acquisition.

Information requests If the output from a KBS indicates that a request be posted to another agent to provide information to the blackboard, this is acted on immediately. If the agent is unable to respond or does not respond quickly enough, then the answer NO_OUTPUT is posted. That is, the output from the KBS is the result of the action suggested and the KBS is only considered to have completed its task when a final response is posted.

Sibling relation A case is input to the first KBS which then produces output which is added to the case. This enhanced case is now passed to the second knowledge base and further output is added. Any case passed to the first KBS must be passed to the second KBS. That is, the original KBS is replaced by a sequence of KBS. There can be any number of KBS in a sequence of KBS, but the sequence replaces the original KBS (and each consequent sequence). As described below, extra KBS are added in a sibling relationship when knowledge inadequacies are discovered. Since they are added over time, the sibling sequence is always ordered by age or time of addition. Note again that the output from any later KBS cannot change the case it is provided with; i.e. the input plus output from the earlier sequence.

Correction relation A case is input to the first KBS, but is then passed to a second KBS before the output is added to the case (i.e. posted to the blackboard). If the second KBS provides output, this output from the second KBS is added to the case, not the output from the first KBS. If the second KBS does not provide any output then the output from the first KBS is added to the case. Any case passed to the first KBS which produces output, is passed to the second KBS. That is, the original KBS is replaced by a correction sequence of KBS. There can be any number of KBS in a correction sequence of KBS, but the sequence replaces the original KBS (and each consequent sequence).

More than one correction KBS can be added to correct a KBS. In this case, the correction KBSs have a sibling relation. That is, the case is initially passed to the first correction KBS, are then passed to the second correction KBS. If the first correction KBS adds output to the case, the second correction KBS acts as a conventional sibling KBS. However, if the first KBS does not add any output, the output from the original KBS is not immediately added to the case, rather the case is passed to the second correction KBS which may add output to replace the output of the original KBS, or if it too fails to add any output then the original output of the first KBS is added. (The circumstances in which more than one correction may apply will be discussed below)

A KBS can also be any combination of both types of sequences, resulting in a recursive structure, with these two types of relationships possible at every level. Note that a correction rule may be added to a KBS which is a sibling sequence of KBS. In this case all the output which is produced by the sibling sequence is replaced. Alternatively the correction may be added to the particular KBS that caused the error. The knowledge acquisition issues which determine which approach is used will be discussed. However, it should be noted that if a correction KBS replaces a specific KBS rather than a sequence, a case must be passed to a correction KBS before being passed to a sibling KBS to determine the output from the first KBS. That is the evaluation is depth first rather than breadth first.

Repeat inference

After a piece of output is added to a case, control is passed back to the first KBS and inference starts again but with the enhanced case. This process is repeated until the case passes through the system with no more output being added. The reason for repeat inference is that some features of a case may be provided in the initial case on some occasions but on other occasions these same features will be generated as output from a KBS. If KBS which uses these features was developed before the KBS that produces the features, then the first KBS using the features will not be effectively used without repeat inference. The reason inference is repeated as soon as output is generated, is that the supervisor/owner decides extra output is required in the context that the previous repeat inference has been completed. Hence the new KBS, should only be used in the same circumstances.

Semi-formal Specification

In this section, we would like to give a semi-formal description of the proposed system. First, we look at the knowledge base representation. Second, we describe the control mechanism through the evaluation functions.

Representation

A knowledge base K is one of three forms: a primary rule, or is composed from two component knowledge bases by sibling or correction relations. In addition, each knowledge base is associated with a set of input cases, named cornerstone cases. In a more formal way, we can define K recur-

sively:

$$K = \begin{cases} (R, D) \\ (Sib(K_1, K_2), D) \\ (Cor(K_1, K_2), D) \end{cases}$$

where R is a primary rule defined above, K_1, K_2 are knowledge bases and D is the cornerstone case set. Note that the cornerstone case set is attached to both primary rules and composite knowledge bases, they correspond to two different knowledge acquisition techniques described later: global refinement and local refinement.

Knowledge bases can communicate through special $Request$ objects. A $Request$ object contains the address of the agent which will carry out the request and the input data that had been passed to the knowledge base.

Evaluation

The evaluation function $Eval(K, d)$ can be defined recursively as follows

- If d is the case passed to K and K is a primary rule R, which is of the form $A \longleftarrow L_1, L_2, \ldots, L_m$ then $Eval(K, d) = A$.
- If $K = Sib(K_1, K_2)$, let $o_1 = Eval(K_1), o_2 = Eval(K_2)$
  - if o_1 is not NO_OUTPUT and o_1 is not in d then $Eval(K, d) = o_1$,
  - otherwise $Eval(K, d) = o_2$ (note that o_2 can also be NO_OUTPUT).
- If $K = Cor(K_1, K_2)$, let $o_1 = Eval(K_1), o_2 = Eval(K_2)$
  - if o_1 is NO_OUTPUT then $Eval(K, d) = NO\_OUTPUT$, otherwise
  - if o_1 is not NO_OUTPUT and o_2 is NO_OUTPUT then $Eval(K, d) = o_1$,
  - otherwise $Eval(K, d) = o_2$.

From the definition, we can see that the evaluation function returns as soon as there is an output that is to be added to the case. The returned output is the conclusion of a knowledge base where none of its corrections applies to the current input. The repeated evaluation function $RepeatedEval(K, d)$ can be defined as applying $Eval$ to the data until the output does not change. The following algorithm will show $RepeatedEval(K, d)$ is computed.

$o := \{\}$
do
  $d := d \cup o$
  $o := Eval(K, d)$
  if $o = Request$
    then
      $o := getRequestedInformation$
  fi
  if $o = d$
    then
      $RepeatedEval(K, d) = o$;
      exit
  fi
od

Algorithm 0.1 Repeated Evaluation Procedure

The operation of $Request$ can be seen from the algorithm. As the external agent does not have the same control mechanism as RDR agent, we simply send the current data and assign the result to the output.

Knowledge Acquisition

The fundamental strategy for knowledge acquisition is to add knowledge when and if a case is handled incorrectly. This means that knowledge is added for real cases in real circumstances. Secondly since the cost of knowledge acquisition is effectively constant with knowledge base size, knowledge can be added while the system is in actual use and becomes a small but interesting extension to normal work or activity flow.

Of particular importance: it can be noted that since no information is removed from the blackboard there is an implicit assumption that solutions to all problems can be assembled linearly. That is, there is no need for any backtracking; information initially added does not need to be removed. This seems to be a plausible assumption in that although a human may use a propose-and-revise or similar approach to developing a solution, they can provide a linear sequence of justification when they are explaining how they reach a conclusion . The broad knowledge acquisition strategies outlined in the introduction then apply as follows.

In the following knowledge acquisition, we consider the special function $Request$ and the special symbol NO_OUTPUT to be the same as the other conclusion objects when constructing the knowledge base.

Global Refinement

The simplest case is to add an extra KBS or to add a replacement KBS which applies to the entire previous KBS. That is, no matter how complex the previous output, it will be replaced by other output in some circumstances. The cornerstone cases are then checked to see if their output is changed and if so whether this is appropriate. If any cornerstone cases have had their output changed inappropriately the application of the added KBS is made more restrictive. (If the extra KBS is a single rule, the user adds further conditions which apply to the case in hand, but not to the cornerstone case.) If the user is to replace some component from the output, we have the following operator

$$K' = (Cor(K, R), D)$$

where K is the original knowledge base, R is the refinement knowledge base and D is the cornerstone case set associated with the new knowledge base K'. D is the union of the the cornerstone cases from K and R. Similarly, if the user chooses to add further components to the output, we have

$$K' = (Sib(K, R), D).$$

Local Refinement

The user looks at the sequence of output and decides that one of the outputs in the sequence have to be replaced. A KBS is added to do this. The case then has to be rerun as

some of the later outputs may be missing or wrong, and perhaps a series of changes need to be made to the case to get all the components right. This generalises to the idea that when the output for a case is being fixed one corrects whatever outputs need correcting in the sequence in which they are provided. If the corrections cause further errors in the sequence, these too are corrected in sequence. The following algorithm shows how this is done. Suppose the input data $d = \{d_1, d_2, \ldots, d_m\}$, we have the output $RepeatedEval(K, d) = \{o_1, o_2, \ldots, o_n\}$:

1. the expert identifies the first wrong output component o_i
2. the expert identifies the component knowledge base K which gives o_i (from the list provided by the system)
3. do a global refinement to K
4. rerun the input data with respect to the new knowledge base
5. if output is correct, stop the process, otherwise, go to step 1.

In each step, the newly added component will only affect the performance of the local knowledge base.

Conclusion

Previous work on RDR has been explicit about attaching a rule to another rule using a correction relation and been explicit about the use of cases. However, it has been less explicit about the sequence relationship except for (Compton & Richards 2000). Because of the lack of focus on the sequence relationship some RDR systems have included other control mechanisms such as conflict resolution strategies. In this paper we have reduced all inference control to the two relations of sequence and correction and elaborated these relations.

We have further proposed that these relations can be used between knowledge-based systems or other programs as well as between rules. Again cases are used to initiate and guide knowledge acquisition. We suggest that this generalisation should enable extremely powerful RDR-like systems to be developed.

We have also suggested that perhaps all knowledge acquisition can be reduced to correcting or adding to knowledge using these two relations and that perhaps the success of RDR comes from explicitly ensuring that knowledge is added to existing knowledge using one or other of these relations, rather than allowing essentially uncontrolled editing. We are not able to prove such a conjecture at this stage, but would suggest that an RDR approach does seem to facilitate easier knowledge-based system development than free editing.

Our hope is that the generalisation outlined here will lead to more sophisticated systems being assembled from more complex components, but that this is incrementally with similar ease to rule-based RDR.

References

Apt, K. R.; Blair, H. A.; and Walker, A. 1988. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, 89–148. Morgan Kaufmann Publishers Inc.

Beydoun, G., and Hoffmann, A. 2000. Incremental acquisition of search knowledge. *Journal of Human-Computer Studies* 52:493–530.

Clancey, W. J. 1985. Heuristic classification. *Artificial Intelligence* 27:289–350.

Colomb, R. 1999. Representation of propositional expert systems as partial functions. *Artificial Intelligence* 109(1-2):187–209.

Compton, P., and Jansen, R. 1990. A philosophical basis for knowledge acquisition. *Knowledge Acquisition* 2:241–257.

Compton, P., and Richards, D. 2000. Generalising ripple-down rules (short paper). In *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management*, 380–386. Springer-Verlag.

Compton, P.; Ramadan, Z.; Preston, P.; Le-Gia, T.; Chellen, V.; and Mullholland, M. 1998. A trade-off between domain knowledge and problem solving method power. In Gaines, B., and Musen, M., eds., *11th Banff KAW Proceeding*, 1–19.

Edwards, G., and Compton, P. 1993. Peirs: A pathologist maintained expert system for the interpretation of chemical pathology reports. *Pathology* 25:27–34.

Kang, B.; Yoshida, K.; Motoda, H.; and Compton, P. 1997. A help desk system with intelligence interface. *Applied Artificial Intelligence* 11:611–631.

Richards, D., and Compton, P. 1998. Taking up the situated cognition challenge with ripple down rules. *Journal of Human-Computer Studies* 49:895–926.