Predicting the Behavior of a Knowledge Base

Jean-Luc DORMOY

Electricité de France Research Center
IMA/TIEM
1, avenue du Général de Gaulle
92141 Clamart Cedex
FRANCE
E-mail: dormoy@cli21cz.edf.fr

Abstract

This paper presents the underlying ideas of a knowledge-based system, called Shal, which, given a knowledge base, attempts to capture the key features of its behavior. The idea is that a priori understanding the behavior of a knowledge base can help to achieve the multiple tasks involved in knowledge engineering, in particular interpreting and/or compiling, contradiction discovery, and acquisition of new knowledge. Moreover, the concepts Shal uses, e.g. reasoning about sets, symbolic constraint propagation, and simple theorem proving mechanisms, can in turn be incorporated into the basic language of the knowledge-based system, thus extending its scope. As Shal syntactically is a normal knowledge base, this positive feedback can be injected into the Shal design process itself: as soon as a new concept or reasoning mode is defined, it can be used to improve the behavior of Shal and the acquisition of new knowledge to be embodied into Shal's knowledge base. This is the conventional bootstrapping process, we apply here to knowledge acquisition. This key feature makes Shal an indefinitely extendable system. We already run four successive versions of Shal, each one using the previous one, and we describe here the improvements Shal brought to the initial inference engine.

Keywords: Meta-Knowledge, Production Systems, Knowledge Processing.

Area: Automated Reasoning (sub-area: rule-based reasoning).

Submitted to AAAI 1990.

1 Introduction

The official claim of a knowledge base (KB) designer is that his system captures knowledge from a specific area. The unofficial claim is that the system exhibits the *intended behavior*. This is due in part to the narrowness of today's AI systems: in the case of rule-based systems, an expert's rule can hardly ever be implemented in a "computer rule", but in a set of productions, plus procedural tricks for control. Furthermore, what is available to the designer is, roughly speaking, an interpreter or a compiler. This "knowledge processor" imposes in turn some constraints on the knowledge base, because of the choice of its "logic". And it merely *runs* the knowledge base. Moreover, it is based on a general algorithm (e.g. RETE or TREAT), which is supposed to be efficient enough for all the possible KBs. The result is that the intelligence exhibited by a KB is not the system's, but the designer's.

All this is the well-known bad side of things. As we did not pretend to solve all these frustrations, we tackled a shorter-term problem: We designed a production system, called Shal, which aims at discovering the main features of the behavior of a production system. The idea is that, when given a knowledge base, we know or we think much more about it than conventional knowledge processors, such as inference engines. There are two main reasons for that. Firstly, there exists *implicit* knowledge which is not contained within the knowledge base. For example, we know the kind of Working Memory (WM) it should be run on. Secondly, we have knowledge about knowledge-based systems that enables us to deduce things about a knowledge base. This work reports how we attempted to capture this kind of knowledge and to embody it into a knowledge-based system.

In order to achieve its goals, Shal is able to analyze the RHS-LHS[1] links between rules, to group rules in sets presenting an interesting global pattern, and to reason about the growth of classes of objects in the Working Memory (WM). It can also use implicit knowledge, i.e. constraints, about the objects handled by the KB; this knowledge must be given by the KB designer. It can take advantage of its rough predictions to help the standard interpreter to be more efficient, or to synthesize a compiled version of some subsets of productions. It might also discover errors by confronting its own deductions with the constraints provided by the user. Naturally, it can also display the main features of the expected behavior of the KB.

Moreover, Shal is designed as a production system, in order to take advantage of its own knowledge. Indeed, Shal was designed (and is still being extended) through a sequence of bootstrapping steps.

Eventually, the KB designer still has trouble, as previously described, but Shal can help him to figure out how his KB system behaves.

Section 2 gives a detailed example. Section 3 shows how Shal works, and what it can deduce. Section 4 emphasizes various aspects of Shal's design process. Section 5 gives an overview of related work.

2 A motivating example

[1] LHS: Left-Hand Side. RHS: Right-Hand Side.

This section describes, through an example, *what* Shal does. The description of the example is somewhat long and tedious, but this was necessary to give a realistic idea of our system. However, it must be pointed out that the range of application of Shal is not limited to this example. This will be fully described in Section 3.

2.1 Description of the example

We show here a simple but non simplistic example: a knowledge base made up of two rules, which aims at performing *arc consistency* in a Constraint Satisfaction Problem (CSP):

```
Rule RemovePossibleValue
If  x PossiblyIsInDomainOf X
    (x Through (X,Y)) HasNumberOfConnections 0
Then
    (x PossiblyIsInDomainOf X) is Removed

Rule MaintainNumberOfConnectionsWhenRemoved
If  (x PossiblyIsInDomainOf X) is Removed
    y PossiblyIsInDomainOf Y
    x IsInImageOf (y Through (Y,X))
    (y Through (Y,X)) HasNumberOfConnections N
Then
    (y Through (Y,X)) HasNumberOfConnections N-1
```

A CSP consists of a set of variables, each having a given domain, and submitted to a set of constraints. Solving a CSP means assigning values to the variables from their respective domains so that all the constaints are satisfied.

For instance, the following CSP (see also Fig. 1(a)):

Dom(X) = {x_1,x_2,x_3}, Dom(Y) = {y_1,y_2}, Dom(Z) = {z_1,z_2,z_3,z_4},

Constraints on (X,Y), (Y,Z), and (Z,X):

(X,Y): {(x_1,y_1),(x_2,y_1),(x_3,y_2)}

(Y,Z): {(y_1,z_1),(y_1,z_2),(y_1,z_3)}

(Z,X): {(z_1,x_1),(z_2,x_2),(z_3,x_3),(z_4,x_3)}

has two solutions: *(x_1,y_1,z_1)* and *(x_2,y_1,z_2)*.

Performing arc consistency means propagating the possible values of the variables by considering each constraint locally. The basic principle of arc consistency is that, if x is a possible value of X, and if there is no couple (x,y) in a given constraint (X,Y), where y is a possible value of Y, then x should no longer be considered as a possible value of X. The advantage of arc consistency is that it is far simpler a problem than the whole CSP. It has been shown [Mackworth & Freuder, 1985] that arc consistency is polynomial - though the full problem is NP-complete. An optimal algorithm is given in [Mohr & Anderson, 1986].

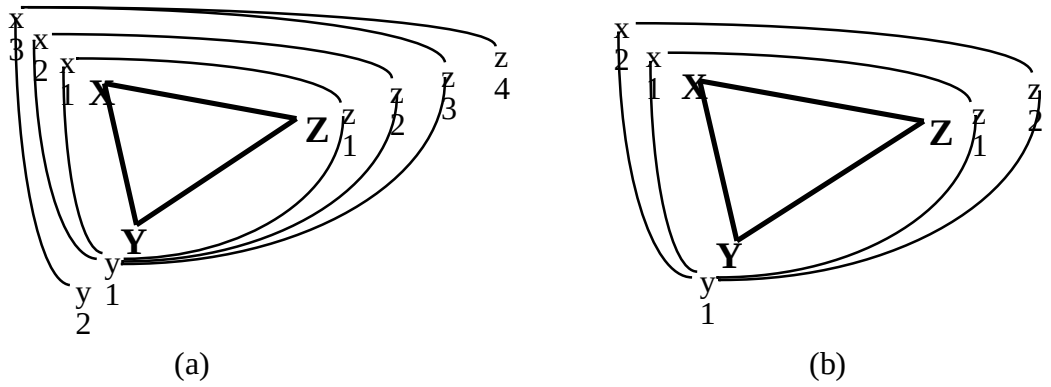

Figure 1: Graphical representation of a CSP (before and after arc consistency)

The CSP example above is represented in the production system working memory in the following way (we give only part of it):

```
   x1 PossiblyIsInDomainOf X   x2 PossiblyIsInDomainOf X
                …
y1 IsInImageOf (x1 Through (X,Y))   x1 IsInImageOf (y1 Through (Y,X))
                …
   (y1 Through (Y,X)) HasNumberOfConnections 2
                …
```

To explain what precedes, `PossiblyIsInDomainOf` describes the range of each variable possible value, `IsInImageOf` describes the images of a possible value through a given constraint, and `HasNumberOfConnections` gives the cardinality of the range of each possible value through all the constraints. This description is redundant and does not correspond to a strict representation of the initial CSP, but it is clear that it could easily be obtained from it.

Now, the two rules act on this working memory in the following way:

- The rule `RemovePossibleValue` removes a possible value x from the domain of variable X if it has no connection through a constraint (X,Y) on X.
- The rule `MaintainNumberOfConnectionsWhenRemoved` propagates a value removed from the domain of a variable X to the variables Y linked to X through a constraint (Y,X). This is done by maintaining the number of connections of the possible values of Y.

It is clear that these two rules implement the "arc consistency principle" mentioned above. If we run them on the example, we get the ranges of possible values represented in Fig. 1 (b). The sequence of rules firings is:

Rule fired	**Main action**
`RemovePossibleValue`	z_4 removed from domain of Z
"	y_2 removed from domain of Y
`MaintainNumberOfConn…`	x_3 through (X,Z) has 1 connection

"	x_3 through (X,Y) has 0 connection
`RemovePossibleValue`	x_3 removed from domain of X
`MaintainNumberOfConn…`	z_3 through (Z,X) has 0 connection
`RemovePossibleValue`	z_3 removed from domain of Z
`MaintainNumberOfConn…`	y_1 through (Y,Z) has 2 connections

2.2 The behavior of a knowledge base

The task of an inference engine is to run a knowledge base. All the existing inference engines are based on a more or less efficient algorithm. However, they do not take into account the particular features of the knowledge base they interprete or compile. In the previous example, it is clear that the two rules will call each other in a recursive manner: as soon as a possible value is removed by rule 1, rule 2 possibly applies to this removed value. In fact, this is the only case when rule 2 can apply, since only rule 1 can remove a value. Moreover, when rule 2 applies, then rule 1 also applies, provided that the number of possible connections of Y through (Y,X) passes from 1 to 0.
All this can be deduced by closely scanning the knowledge base. One might also require extra-knowledge *about* the knowledge base, such that "there is no already-removed value in the initial working memory". This is some kind of *implicit* knowledge. As this kind of fact cannot be deduced from the mere knowledge of the rules, it should be asked to the designer. Analyzing a knowledge base this way and acquiring the required extra-knowledge from the designer is what Shal performs.
In this example, Shal eventually deduces that running rules 1 and 2 is equivalent to calling procedure R_1, which in turn calls R_2:

R_1: Find out (x,X) such that rule 1 apply.
Perform the action of rule 1.
R_2(x,X)

R_2(x,X): Find out (y,Y) such that rule 2 applies.
For each such (y,Y):
Compute N (the number of connections of Y through (Y,X))
Apply the action of rule 2.
If N=1 Then R_2(y,Y)

This is the structural behavior of the knowledge base, as intended by its designer.

2.3 Compiling a knowledge base

What we called in the previous section a "structural behavior" also is an algorithm. In particular, rules 1 and 2 can be compiled by implementing this algorithm. This algorithm can still be refined. In the example, one could for instance state more precisely how the "Find out" actions should be implemented, in particular by deciding that some indexes or pointers on the working memory should be maintained while running the knowledge base. A set of "good" indexes could be:

- Given (x,X), the set of (y,Y) such that

y IsInImageOf (x Through (X,Y))

- Given (x,X,Y), N such that
  (x Through (X,Y)) HasNumberOfConnections N
- The set of (x,X,Y) such that
  (x Through (X,Y)) HasNumberOfConnections 0

(The latter index should not be maintained after R_2 is called).

These indexes are not necessarily the "natural" ones, as those possibly memorized by an inference engine. They are particularly well adapted to running *this* knowledge base.

2.4 Contradiction discovery

As mentioned above, understanding the behavior of a knowledge base requires to get some "extra-knowledge" about the knowledge base, for instance that a particular class of working memory elements must be empty in the initial working memory. However, this extra-knowledge can also be used to *criticize* or *discover contradictions* in the knowledge base. For instance, imagine that the two rules in the example are embedded in a larger KB, and that for some reason the KB designer has stated that no working memory element mentioning the relation `IsInImageOf` can be removed when running the rules. Assume at least that the following action is added to rule 2:

```
Remove from WM[x IsInImageOf (y Through (Y,X))]
```

Then, there is a potential contradiction.
Indeed, this contradiction can be discovered by confronting two different partial specifications of the same problem: the knowledge base and extra-knowledge. This is no *consistency checking* in the sense of logic. Indeed, Shal does not systematically look for a contradiction, but it can find one out while it acquires knowledge to understand the KB behavior.

3 The Shal knowledge base

We showed in the previous section the kind of things Shal can do. We shall know describe *how* it does it.
From now on, we denote KB the knowledge base to be analyzed by Shal.

3.1 What is Shal?

Basically, knowledge bases considered by Shal are sets of production rules expressed in an OPS5-like language. There are some differencies between the language we use and OPS5, but their descriptions are unimportant to a good understanding of this paper. Shal also is a production system. Thus, analyzing a given knowledge base KB requires it to be translated into Shal's working memory. This is done by a translator, as shown in Figure 2, which transforms rules, premises, actions, variables into corresponding objects belonging to the classes Rule, Premise, etc. These objects are linked by slots or relations which make the description meaningful and non ambiguous. In short, every element of KB is expressed in the frame language of Shal's working memory.

Rules in Shal speak about rules, premises, actions, variables, and so on. There is an example of such a rule in Fig. 2. This is one of the first rules in Shal, it initializes the *positive recursion arcs* between two rules (see Section 3.3).

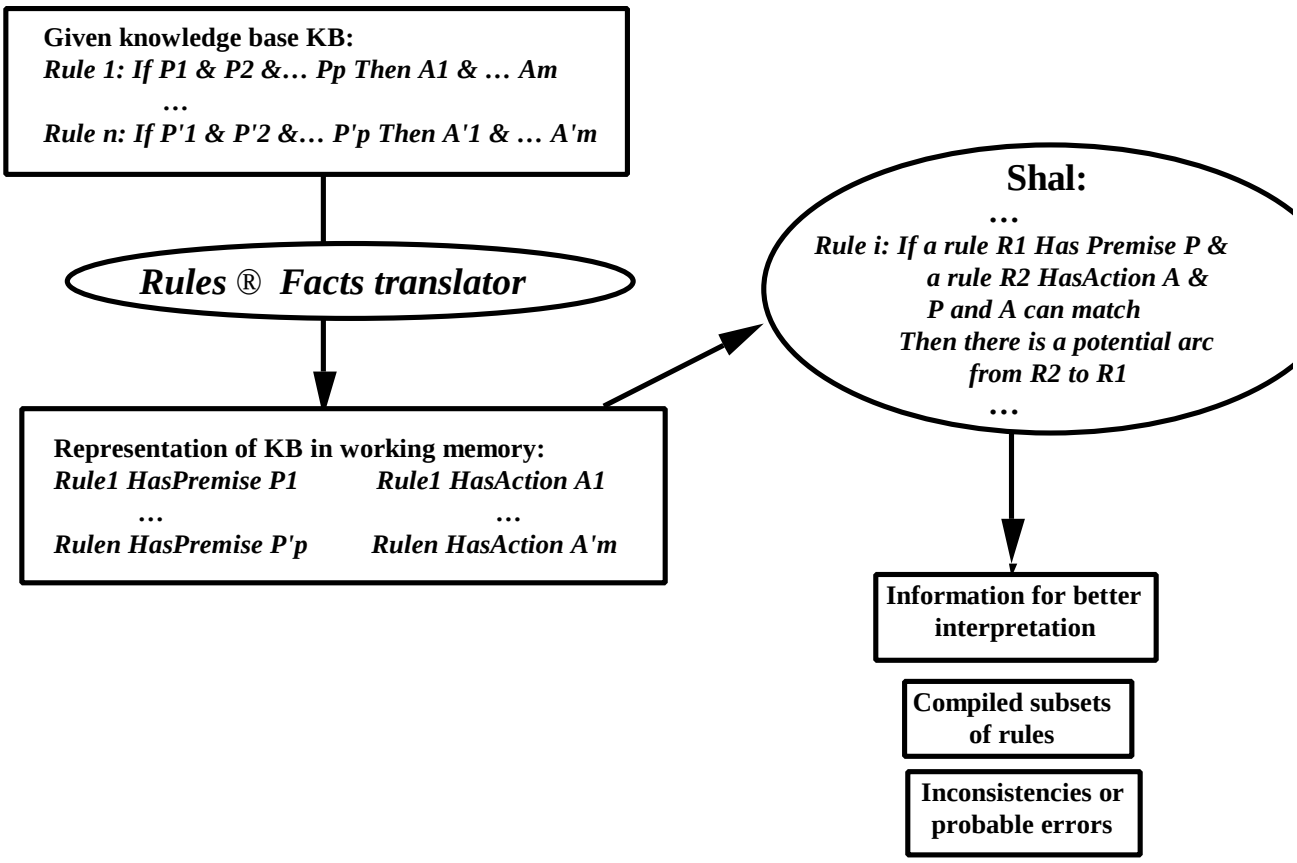

Figure 2: How Shal analyzes a given knowledge base

3.2 Knowledge about a knowledge base

As said in section 2, understanding the KB behavior requires some extra-knowledge, i.e. knowledge about KB. The time being, the language for expressing this knowledge is based on the following concepts:

Classes and sets: Classes and sets of objects can be defined, and the slots or relations used in KB can be typed using them. The main difference with the conventional notion of a class is that a class C is for us a function which maps an initial working memory WM_0 and a nonnegative integer n onto a set $C(WM_0,n)$. This set is the actual set of objects corresponding to the specification of C after the n^{th} rule firing, when running KB with WM_0 as initial working memory. In the example of Section 2, if C is the class of working memory elements mentioning the relation `HasNumberOfConnections`, then $C(WM_0,3)$ contains `((x3 Through (X,Y)) HasNumberOfConnections 1)` and `((x3 Through (X,Z)) HasNumberOfConnections 2)`.

Temporal constraints on classes: Time is for us the discrete time of which instants are the rules firings. Moments are intervals of instants. For example, one can state that a particular class is steady, decreasing, or increasing when running the rules. One can also state that a particular class is empty at a particular instant or during a particular moment. These constraints must hold for any "reasonable" initial working memory.

One can also state some relations between classes and objets, for instance membership or inclusion. These relations can be restricted to hold for some given instants or moments, possibly quantified. One can also manipulate union, intersection, cartesian product and projection of classes, which also are classes.

Cardinality conditions on classes: One can state cardinality conditions on finite classes: minimum, maximum, or exact number of elements. In particular, one can state that a relation is one-to-one.

This knowledge is incorporated into Shal's working memory using an ad hoc representation.

3.3 How rules are linked together

Let's turn now to the kind of things Shal studies and deduces.

The main tool for understanding a KB behavior is studying the potential recursions between rules. There are two kinds of recursion: positive and negative. A positive recursion between rules R_2 and R_1 occurs when performing an action of R_2 adds an instanciation to a premise of R_1. A negative recursion occurs when, instead of adding, this removes an instanciation. Potential recursions can be suspected when a rule action symbolically matches a rule premise. In the example of Section 2, there is a potential positive recursion between the action of rule 2 `(y Through (Y,X)) HasNumberOfConnections N-1` and the premise of rule 1 `(x Through (X,Y)) HasNumberOfConnections 0`. The rule of Shal shown in Fig. 3 installs the positive recursions.

Hence, we have two oriented graphs, the rules being their vertices. Several things can be done on them:

Proving that a recursion is fictitious. This means that the potential recursion can never happen. It is only "syntactic illusion". That can be proved using extra-knowledge. For example, if rule R_2 mentions the actions P(X,Y) and Q(X,a) and rule R_1 the premises P(Z,U) and Q(Z,b), where a and b are two distinct constants, and if Q is one-to-one, then the recursion between P(X,Y) and P(Z,U) is fictitious.

Proving that a recursion almost surely happens. This means that, if the potential recursion between R_2 and R_1 actually happens, then R_1 applies under a simple condition. This is the case in the example of Section 2, since rule 1 applies after rule 2 under the single condition N=1.

Other things can be done, such as recognizing that a particular subset of rules has a specific pattern (see Section 3.5).

The main tool for studying the potential recursions is symbolic propagation. This means that Shal tries to propagate the symbolic variable bindings of rule R_2

throughout the premises of rule R_1. In the example of Section 2, the recursion between rule 2 and rule 1 is possible provided that the variables of rule 1 can be substituted by the variable bindings of rule 2 according to:

x_{R_1}/y_{R_2}, X_{R_1}/Y_{R_2}, Y_{R_1}/X_{R_2}, $0/N_{R_2}-1$

The condition $0/N_{R_2}-1$ is equivalent to $N_{R_2}=1$, and this variable binding of rule 1 turns out to be correct under this single condition.

It must be pointed out that this symbolic propagation is efficient only by making an intensive use of the implicit knowledge provided by the user or discovered by Shal. In the previous example, the fact that `HasNumberOfConnections` is one-to-one is essential to draw the conclusion.

3.4 Simple theorem proving

Proving that a potential recursion is fictitious, or on the contrary that it almost always happens, may not be straightforward. Hence, Shal controls this task by performing some kind of simple theorem proving. The first step is to state some interesting problems to solve, namely "Does a given potential recursion between rules R_2 and R_1 happen?". This is transformed into "Does there exist instants n_2 and n_1, with $n_2 < n_1$, such that the variable binding of R_2 at instant n_2 provides R_1 with a correct variable binding at n_1 through the recursion?". This formulation is used to properly denote the respective variable bindings of R_2 and R_1. Then, Shal imposes some necessary conditions for a proposition to be true, and it states some actions to be performed to check these conditions. This enables it to reason locally. Consider the example given above, where R_2 mentions the actions P(X,Y) and Q(X,a) and R_1 the premises P(Z,U) and Q(Z,b), with a and b two distinct constants, and Q a one-to-one relation. There is no specific rule in Shal for this particular case. Instead, it first states that a necessary condition for the recursion to happen is X/Z and Y/U, then that Q(X,a), and Q(Z,b), then, that a=b, which contradicts a?b. The actions to be performed are different kinds of symbolic propagation.

3.5 Discovering sets of rules having a particular pattern

Another thing Shal can do is trying to recognize subsets of rules having an already-known pattern. We call this a global pattern. Here are some examples of such patterns:

Attached subrules: It is the case of the example in Section 2. Rule 1 is attached to rule 2. As soon as rule 2 applies, and if the condition N=1 is fulfilled, then rule 1 triggers. This pattern is useful when a rule is attached to a single rule or to a subset of rules having a classification pattern.

Classification rules: This pattern is made up of rules which are close to each other in the KB, and which all contain the same pattern in their left-hand-side:

Rule i If P(X,Y) & … Then …
…
Rule i+p If P(X',Y') & … Then …

If there is no positive recursion arc inside of this subset, then this subset of rules comes down to selecting the objects (x,y) such that P(x,y), and then to checking the remaining conditions of the rules.

Recursive classification rules: The same as the previous one, except that some rules mention in their action part the pattern P(X,Y).

Loop: A subset of rules where the first rule selects an object or a couple of objects to be studied, where the internal rules act on this object or couple of objects, and where the last rule triggers the selection of another object or couple of objects.

Of course, a global pattern can contain another global pattern. All these subsets can be compiled (see Section 3.7).

The suspected subsets having a particular pattern are extracted from the KB by means of ad hoc heuristic rules. For instance, a subset of consecutive rules all having the same premise and/or action pattern is strongly suspected to be a classification pattern (recursive or not). Then, these suspected patterns must be checked.

3.6 Selecting interesting subproblems

Usually, there are thousands of potential recursions in a knowledge base. Our experiments showed that Shal can prove that a large amount is fictitious only by using simple symbolic propagation. However, thousands remain, and it would be unrealistic to perform full symbolic propagation between all the couples of rules linked by a recursion arc.

It turns out that the few global patterns mentioned above are very frequent in knowledge bases. Otherwise, understanding the behavior of KB would be intractable, even for a human being. However, performing full symbolic constraint propagation in a pure deductive way is impossible, and it is impossible as well to perfectly recognize the global patterns at first glance.

Hence, the heuristics used for suspecting the global patterns are crucial. Once a subset of rules suspected of having a global pattern is selected, one can run full symbolic propagation on this subset. Indeed, the internal-most suspected global patterns are examined first. Then, if it turns out that there actually is a global pattern, a description of its behavior can be substituted for the corresponding subset. Eventually, the whole KB can be analyzed by successively contracting it into nested global patterns.

3.7 Compiling and improving interpretation

As mentioned above, when a global pattern is recognized, it is possible to compile the subset of rules into a sequence of simple procedures. However, this is not always possible, for various reasons: either there is no global pattern matching KB, or Shal's knowledge is not sufficient to recognize the patterns - some lack of knowledge in symbolic propagation -, or compiling a recognized pattern turns out to be out of reach. In any case, it is not always possible to completely compile a given KB.

However, when this happens - and this happens -, the deductions drawn by Shal can be useful for improving the KB interpretation. For instance, it is very important for the inference engine to know that a given potential recursion is fictitious. This can avoid it to propagate working memory elements into part of the rules network.

There are some links between Shal and the inference engine which normally should run the knowledge base. When a subset of rule is fully compiled, the inference engine calls the compiled procedures instead of interpreting the rules. Otherwise, the inference engine is capable of taking advantage of the pieces of information given by Shal, in order to improve its efficiency. In this case, Shal simply acts as an optimizer.

3.8 Contradiction discovery

This aspect is very important, though not yet fully developed. As mentioned in Section 2.4, Shal can exhibit some contradictions by confronting KB and already-acquired knowledge about KB. That enables the system to focus the KB designer's attention on the differences between what he thinks the system does and what it actually does. Our experiments showed that this was extremely useful in a practical way.
However, there is no logical checking mechanism in our system, and of course no completeness property of the knowledge kernel for contradiction discovery. Indeed, we think that this would be a goal beyond reach. Performing complete logical checking is at least as difficult as processing full symbolic propagation between all the couples of rules linked by a recursion arc.

4 The bootstrapping process, and current results

Shal is a knowledge base. Thus, Shal can apply to itself. In fact, it is not only necessary, it is also one of the intended features of this work. Indeed, it would be intractable to design a completed version of Shal, and then to run it on various knowledge bases. There would be many undetected errors in Shal, and above all that running Shal by means of the inference engine would require too many resources. This is extremely frustrating, since we can reasonably expect that, once a completed version of Shal can examine itself, it can work using reasonable amount of resources. Then, we had to design Shal through a sequence of bootstrapping steps. The basic principle is: "As soon as you know something, use it". To date, we have designed and experimented four versions of Shal (and discarded many more), each one being processed by the previous one.

4.1 The bootstrapping bottleneck

The difficulty lies in finding out a path towards the next version. We first designed a simple version of Shal for proving that some recursion arcs were fictitious. $Shal_1$ used very simple symbolic propagation and implicit knowledge. Then, we designed a second version for eliminating more recursion arcs, in particular those which were crucial in $Shal_1$. It turned out that this version required too many resources, and we could not reasonably run it on itself. The reason was that, for implementing more sophisticated elimination of recursion arcs, we designed a knowledge base with many recursion arcs which could not be eliminated. In other words, in order to eliminate recursion, one must first be able to process recursion properly.
What happened then is the main bottleneck of our approach. When designing a piece of knowledge K_1, one must be able to reasonably process K_1. If K_1 cannot process itself, then one must design K_2 for processing K_1. This loop might go on. However, it

must stop quickly, otherwise the knowledge base becomes too intricated and simply too large. Moreover, Shal's growth is not linear in some sense. Suppose a new chunk of knowledge K is added to Shal. Then, when running the extended Shal system on itself, K is present not only in the production memory, but also in the working memory. This is at least a quadratic process. Hence, we must find a trade-off, but this trade-off must be good enough to enable the extended system to process itself.
The second version of Shal was designed to improve the treatment of positive recursion, which was the main obstacle. Roughly speaking, all the positive recursion arcs which could not be eliminated were considered as attachment arcs. Then, we gave Shal3 the ability to use symbolic propagation and simple theorem proving methods for eliminating crucial recursion arcs. Shal4 is able to recognize some global patterns and to compile them in simple cases. We are now facing a new difficulty: improving global pattern recognition and compiling requires a sophisticated pattern recognizer and the corresponding compiler.

4.2 The bootstrapping positive feedback

This approach provides us with a positive feedback. Each time we implement methods for using new concepts, these methods and concepts automatically are incorporated into the system's knowledge repository. For instance, we can now speak of sets and operations on sets: the system has some knowledge about it. This is a feedback on the expressiveness of the language. There is another feedback: once a new concept is defined, one can use it to write again the knowledge base, thus providing us with a shorter and easier-to-analyse knowledge base. We are seeking efficiency through generality. We hope in the next future that this positive feedback will be very significant.

4.3 Current results

Besides the other aspects which can only be qualitatively assessed, the efficiency gain can be quantitatively measured. The average gain ratio when compared to the initial inference engine performance is 3.25. In particular, this is the ratio when Shal4 is run onto itself. However, it can be significantly higher when global patterns can be recognized.

5 Related work

A priori analyzing a knowledge base by means of a meta-knowledge base was already proposed by Porcheron [1988]. In particular, he proposed to study the recursion arcs between rules. The main difference with our work is that he used graph-theoretical concepts to analyze these two graphs, such as connectivity or simple connectivity. Instead, we use simple propagation and heuristics in order to get a reasonably efficient system. Another major difference is that, though foreseeing it, he did not try to apply his system to itself.
Parchemal describes in his PdD dissertation [1987] a system, called SEPIAR, for analyzing how a production system runs. The way SEPIAR works is very different from ours, since it does not perform a priori deductions. However, it could draw

conclusions similar to ours, in particular that a positive recursion hardly ever happens, or that a particular index on the working memory would be useful for improving interpretation.
Another meta-level system is Maciste, being designed by Pitrat [1986, 1988]. His system is far more ambitious than ours. It was the first system to systematically apply bootstrapping to knowledge bases.
Other meta-level systems are well-known, such as EURISKO [Lenat, 1983] and SOAR [Laird, Newell & Rosenbloom, 1987]. However, their goal is very different from Shal's, since they aim at discovering or learning new knowledge. Shal does not discover new concepts, and it learns nothing new. Instead, it applies meta-knowledge designed for analyzing knowledge. We think that Shal lies somewhere in between simple knowledge processing and learning.

6 Conclusion

We showed in this paper how a knowledge base can understand some key features of the behavior of a knowledge base. This can help achieving different knowledge engineering tasks: compiling and/or improving interpretation, contradiction discovery, and knowledge acquisition. As it is a regular knowledge base, our system can take advantage from this approach, by applying onto itself. In particular, this makes our system an indefinitely extendable system.

References

[Dormoy & Raiman, 1988]. J-L Dormoy and O. Raiman. Assembling a device. AAAI 88.

[Dormoy, 1988]. J-L Dormoy. Controlling qualitative resolution. AAAI 88.

[Dormoy, 1989]. J-L Dormoy. Amélioration de l'efficacité du pattern matching dans le langage à base de règles BOOJUM. (*Improving pattern matching efficiency in production systems*). Convention IA 1989, January 1989, Paris, France.

[Dormoy, 1989]. J-L Dormoy. SHAL: a rule-based rule compiler which applies to itself. IFS conference on Artificial Intelligence and Expert Systems in Manufacturing, October 1989, Berlin, FRG.

[Forgy, 1979] C.L. Forgy. On the efficient implementation of production systems. Carnegie-Mellon University, PhD, 1979.

[Laird, Newell & Rosenbloom, 1987] SOAR: an architecture for general intelligence. Artificial Intelligence 33 (1987), pp. 1-64.

[Lenat, 1983] D.B. Lenat. EURISKO: A program that learns new heuristics and domain concepts. Artificial Intelligence 21 (1983), pp. 61-98.

[Mackworth & Freuder, 1985] A.K. Mackworth, E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artificial Intelligence 25 (1985), pp. 65-74.

[Mohr & Henderson, 1986] R. Mohr and T.C. Henderson. Arc and path consistency revisited. Artificial Intelligence 28 (1986), pp. 225-233.

[Parchemal, 1987] Y. Parchemal. Spying: an approach for determining Methods to efficiently interprete rule-based systems. Internal report n° 87/5 of the LAFORIA, University Paris 6, April 1987.

[Pitrat, 1986] J. Pitrat. Le problème du bootstrap. (*The bootstrapping problem*). Internal report "Cahiers du Laforia". Proceedings of the LAFORIA workshop on Meta-Knowledge, Strasbourg, France.

[Pitrat, 1988] J. Pitrat. Ceci n'est pas un article. (*This is not a paper*). Internal report "Cahiers du Laforia" n° 70. Proceedings of the French-Spanish workshop on Meta-Knowledge, Areny de Mar, Spain.

[Porcheron, 1988] M. Porcheron. MILORE, a meta-level knowledge-based architecture for production system execution. ECAI 1988, Munich, FRG.

[Wellman & Simmons, 1988] M.P. Wellman & R.G. Simmons. Mechanisms for Reasoning about St Paul, Minnesota.