MAINTAINING PROCEDURAL KNOWLEDGE: RIPPLE-DOWN-FUNCTIONS

TIM MENZIES

AI Lab, Computer Science, University of NSW,
P.O. Box 1, Kensington, NSW, Australia, 2033
timm@spectrum.cs.unsw.oz.au

ABSTRACT

Research into the maintenance of procedural knowledge has focused on the detection of declarative meta-structures within the procedures. An alternative approach is described here based on the *ripple-down-rules* (RDR) formalism. To the standard RDR rule tree, we add a *functions environment* hierarchy that stores the implementation of the procedures used in the rules. This functions environment structures the evolution of functions and scopes the effect of a newer version of a procedure to the rules that require it. We describe implementation techniques for the maintenance environment using commercially- available tools. *KEYWORDS:* Knowledge acquisition, knowledge maintenance, declarative knowledge, procedural knowledge, ripple-down-rules.

1. Introduction

Despite the many advantages of a logic-based formalism for representing knowledge, the use of *procedural constructs* to represent knowledge is often required in a knowledge-based system. Such constructs exist in nearly all commercial systems used for expert systems development. The use of procedures to express knowledge is orthogonal to much of the knowledge acqusition (KA) research. KA is often viewed as the task of extracting declarative statements about the universe from a domain expert[1]. Supporters of the declarative approach argue that an appropriate analysis of seemingly-procedural knowledge can detect an underlying declarative structure. For example, Hayes-Roth *et al*[2] describe RLL, a frame based system that stores its "rules" as instances of a rule class. The slots of the rule frames represent simple descriptions of aspects of procedural knowledge formally stored, somewhat opaquely, in raw LISP code within a rule condition. Other analysis of the meta-structure of seemingly spaghetti-like knowledge can be found in [3] [4].

Here, we take an alternative approach for three reasons. Firstly, we are interesting in extending the utility of the RDR formalism[5] [6] RDR is based on a sophisticated knowledge acquisition model, but lacks a strong understanding of the meta-structure of the experts knowledge. We note that RDR is the only knowledge maintenance environment we know of where maintenance time remains constant. Given this finding, we hypothesis that meta-knowledge of expert tasks or inferencing structure is less important than meta-knowledge of the *context* of knowledge acquisition (see section Three for more details on RDR).

Secondly, we suspect that RDR's unique approach to non-monotonicity may be relevant to procedural knowledge. RDR permits the simple and continuous correction of incorrect knowledge. Further, once a knowledge base has been initialised by (e.g.) a machine learning

program, it can then be extended using RDR[7] In domains where (i) the knowledge to be learnt is procedural and (ii) machine learning techniques exist for learning some of the domain[8] it would be useful to be able to combine human and automatic learning of procedural knowledge using RDF.

Thirdly, research into the detection of declarative meta-structures within procedural knowledge has yet to reach a definite conclusion. We note that Clancey's research in this area has been proceeding for at least a decade and does not appear to be terminating on techniques that are commercially mature[9]. Commercial practioners would prefer a general-purpose formalism that can cope with a variety of programs.

This paper is structured as follows. Section two discusses the benefits and complications of procedural constructs. Section three introduces RDF. This section includes a tutorial on RDR. Section four discusses pragmatic implementation details and section five discusses common objections to the RDF formalism.

Note that portions of this paper have appeared previously[10].

2. Procedural Constructs

Bustany advocates a knowledge engineering methodology called *application languages* based on the construction domain-specific procedural constructs[11]. *PIGE* is a Prolog-based application language developed for a farm management expert system. *PIGE* has been in the field now for two years[12]. At the time of this writing, the rule base comprises some 486 rules of varying complexity. The domain experts report that maintaining the system is not a problem[1]. A large part of this ease of maintenance was the procedural-constructs approach. The domain experts wrote the rules with occasional support from the knowledge engineers. Whenever the knowledge engineers noted that the experts had to write a particularly inelegant, clumsy, or confusing rule, they would extend the library of procedural constructs such that the rule condition could be expressed succinctly and elegantly. The acid test of each such extension was the domain experts. If the new construct made their work easier, then it remained. If they did not like it, it was junked.

Procedural constructs create significant problems with knowledge maintenance. Consider a procedural construct used in many rules. Experience with a certain case could prompt a redefinition of that construct. This could have adverse effects on other rules. The new definition of the construct could patch one case while causing errors in other cases. This problem is particularly acute in application language systems. Imagine a domain expert hacking a application language knowledge base. Since they are working with succinct, high-level drivers, they can very quickly create *spaghetti knowledge*; i.e. convoluted knowledge that gets more convoluted every time it is maintained. It is easy to test for spaghetti knowledge. In a *good maintenance environment*, the change time remains constant. A *bad maintenance environment* causes change time to increase as the spaghetti grows more and more entangled (as illustrated in Figure One).

1. To be accurate, the domain experts simply did not understand the question "how hard is it to change the rules?". They looked blank then said "Sorry? What do you mean? We just change things."

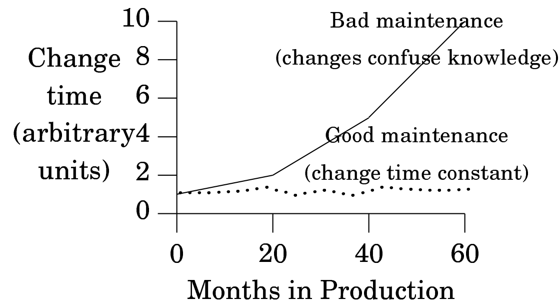

Figure 1. Good vs bad maintenance

Sadly, most knowledge is maintained in bad maintenance environments. Software development continues and the time taken to process each change request lengthens. Eventually, the change time proves prohibitive and the system is redesigned. The process repeats since the new system is usually another bad maintenance environment.

There are very few examples of good maintenance environments. One such good environment is RDR. This system will be used as the basis for our procedural maintenance environment.

3. Ripple-Down-Functions

This section describes RDF, an extension to RDR. RDF enforces the disciplined evolution of procedural constructs. Using an RDR tree, the scope of the change to a construct is limited to the rules written *after* the revision of a construct at a particular rule. The definition of the construct in existing knowledge does not change.

Suppose that experience suggested that the definition of some construct was incomplete. After an examination of several cases, it was proposed to add another conjunction (*tomorrowIsMyDayOf*) to this procedure; for example:

```
iWantToGoHome  :- late, working,lunchALongTimeAgo,tomorrowIsMyDayOff.
tomorrowIsMyDayOff :- day(friday).
```

Suppose further that the expert system was in production and that we already had rules that used the construct *iWantToGoHome*. Since the system currently works adequately, we should be reluctant to change working knowledge. We may wish to "freeze" knowledge that has proved satisfactory and only use this new definition in any additional knowledge[2].

The RDR scheme adopts the frozen knowledge principal for propositional systems. In RDR, whenever a case results in an inappropriate conclusion, the patch knowledge is entered in as a *unless* test beneath the rule that resulted in error. As the knowledge base develops, it grows into a binary tree with knowledge patches stored at every node (see Figure Two).

---

2. This is an application of the heuristic: "if it ain't broke, don't fix it".

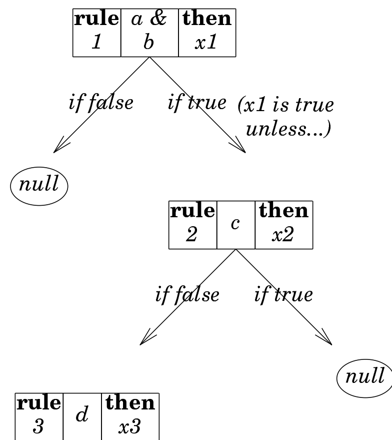

Figure 2. An example RDR tree

At runtime, the final conclusion is the conclusion of the last satisfied node. At maintenance time, when fixing deficient knowledge, the *unless* logic is added beneath the incorrectly last-satisfied node. Only the *logic delta* is added in the new node since the system can not get to this node without first satisfying the logic from the root to the node. So, if *x2* is the correct conclusion when *a & b & c* is true, but incorrect when *c* is false, then we add the logic delta *c* to a new node on the *if true* branch beneath *rule 1*.

The frozen knowledge principal simplifies maintenance. To see this, consider how most expert systems would encode the knowledge in the above RDR tree. Most probably, they would enumerate all the logical paths in the RDR tree and write one rule for each path. Assuming the RDR interpreter, then the logical paths for Figure Two are shown in Figure Three.

Rule	**If**	**Then**
1	a & b & not c	x1
2	a & b & c	x2
3	a & b & not c & d	x3

Figure 3. The logic implicit in the example RDR tree

If the knowledge of the system is patched, then in a conventional rule-based expert system, this patch could extend over many rules. Repeating our above example, the patch on the *x1* error requires an edit to one rule (*rule1*) and the creation of another (*rule 2*). Further, the new logic refers to *c* which is a new concept that must be propagated down to all related rules (*rule 3*). The more related rules, the more edits. As the knowledge base grows, so to does the number of edits. Hence the time taken to make a change increases and we have a bad maintenance environment.

In RDR, existing knowledge is frozen and we only *extend* the knowledge base. For this reason, RDR is a good maintenance environment. The time taken to change the knowledge does not increase as the knowledge grows since the knowledge base author does not have to tour all the knowledge to make a patch. Instead, at patch time, the system presents the author with a list of candidate delta logic and the author selects item(s) off that list. These items are added beneath the incorrect node. This is the action at *every* knowledge patch time. Maintenance time hence remains constant.

Note that the RDR formalism makes no commitment to tree structures that are optimal. An RDR tree can contain repeated tests, redundant knowledge, and its sub-trees can overlap each other semantically. While this is less than optimal in a computational sense, it is somewhat misguided to attempt to optimise an RDR tree to (e.g.) remove the redundancies or separate out the overlaps. The important feature of an RDR tree is that it is optimised for maintenance. Alternative knowledge representation schemas *may* run faster[3] but incur the penalty of non-constant maintenance time[4].

RDR is being used to develop *PIERS*, an expert system for biochemistry at the St. Vincents Hospital, Sydney. The current system comprises 1250 rules and is in routine use. Maintenance time remains constant and the system is maintained by the domain experts without the need for knowledge engineers[15]. This represents somewhat of a triumph for the RDR approach. Previous attempts at building intelligent software for the *PIERS* domain required sophisticated inferencing procedures and experimental causal modelling techniques (e.g. the *ABEL* system[16]). Further these experimental systems never went into routine production. We can conjecture that these domains can be tamed via the use of sophisticated knowledge acquisition modules rather than sophisticated inferencing modules.

If we adopt the RDR frozen knowledge principal, then it follows that the definition of *iWantToGoHome* should be split each time a refinement is made. In terms of RDR, we should add a *rule chronology* list that stores the order in which new nodes were added to the system. Each entry in the list contains a *procedure environment* that stores the definitions of procedural constructs. When a rule needs the definition of a procedure, it finds its own reference in the rule chronology and searches back towards *rule 1*. At each entry in the chronology, it checks the current environment for a definition of the required procedure. If found, the search stops and the procedure is executed.

We can optimise this representation. We need only add a new procedure environment when a definition changes. Whenever a new rule is created, a pointer is added from this rule to the latest environment. If the maintenance cycle produces a change in the procedure definition, then:

1. A new environment is pushed with the new definition.
2. A new rule is added under the current node on the *if false* branch. The procedure environment pointer of this new rule points to the new environment.

---

3. Techniques for automatically optimising propositional systems such as RDR are discussed by Colomb[13] and Forgy[14]
4. It should be noted that even the seemingly inefficient RDR trees have never proven to be too slow in practice.

3. The logic in the old node is copied along with the logic delta. When the new rule executes, it will now execute using the new definition.

Continuing the previous RDR example of Figure Three, suppose the *b* is found to be faulty in the case of *a and b and e*. A new definition *b1* is written. A new rule *rule 4* is added to our RDR tree. The resultant tree (and the pointers to the functions environments) is shown in Figure Four. (It is assumed that working definitions of *a, b, c,* and *d* were defined during some analysis phase and created as the first procedure environment at the same time as *rule1*.)

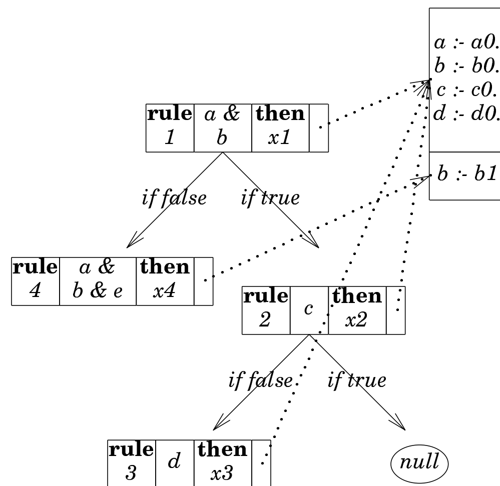

Figure 4. RDR tree with functions environments

4. Pragmatics

The functions environment stack consists of functions scoped in a hierarchy that override the definition of functions higher-up in the hierarchy. This design maps naturally into a message-passing paradigm such as object-oriented languages or Hypercard[17].

RDF is simplest to implement in an interpreted environment. At runtime, the program flow is re-directed based on the knowledge acquisition process. A natural development cycle would be to allow experts to test out their changes as soon as they have made them. The compile-link stage of traditional compiled languages, we argue, is hence inappropriate for RDF.

One class of compiled languages that could support RDF are those languages that support functions as variables. For example, C supports pointers to compiled functions. Such pointers can be stored in a look-up table and the runtime behaviour of the system can be modified by changing the contents of this table. Nevertheless, we would be reluctant to implement RDF in C. The functions still need to be compiled. We suspect that a C-based RDF would become a front end to a set of make files. If so, then the turn-around time between conceiving, making, and testing a change could be prohibitive.

RDF would be simpler in an interpreted message-passing system like Hypercard or an incrementally compiled object system like Smalltalk. Consider the Smalltalk case. The functions

environment would be a a Smalltalk class hierarchy. At runtime, functions could be browsed and edited using some specialisation of the *ClassBrowser* class[5]. The class being edited would be a subclass of the abstract class *Environment*. The methods of this abstract class would then be the implementation of *environment_zero* (see below). Every "save" command checks for a difference between the old and new function. If found, then the current subclass would be subclassed.

An instance of this new class would be created and stored in a class variable *LatestFunctions*. Rules are instances of the class *Rule*. *Rule* classes have instance variables for pointers to the true and false branch, some *Context* instance that stores the condition, as well a pointer to the environment instance. The *Context* block would have been compiled within the environment of the rule instance. Hence it would access to the environment instance's pointer. When this *Context* executes, its boolean tests are implemented as messages to the environment pointer. When a new rule is created, this *LatestFunctions* would be installed as the rule's environment pointer.

One last point: the class compiler methods within the *Behaviour* hierarchy should be customised as to forbid a re-compilation of an existing class in the *Environment* hierarchy.

We argue that this style of implementation is accessible to commercial practioners. We suggest that researchers implement RDF using a logic programming paradigm. We suspect that RDF trees contain sufficient information to drive intelligent partial evaluators that could optimise the RDF functions.

5. Discussion

This section discusses certain objections to the RDF formalism.

5.1 Excessive Change

One objection to RDF is that if the system pushes one new functions environment for *every* changed function, then the environment would get too big to be processed efficiently. Fortunately, our experience with application language development and RDR-style systems suggest that, after the initial analysis stage is over, function revision occurs at a pace at least two orders of magnitude slower than rule revision. Typically, new functions are merely *added* rather than changed. Since additions do not necessitate a new environment, the additions can be inserted without extending the size of the functions environment.

We concede that this observation is specific to the domains we have worked in (biochemistry, farm management, process control) and would require monitoring in other domains.

5.2 Change Everything?

Critics of the RDF proposal have argued that some functions should not be open to the maintenance process. They argue that functions such as (e.g.) member and subset have such a well-defined, well-elaborate definition that it would be nonsensical to permit the user to customise them.

---

5. Our example will be passed on on Digitalk's Smalltalk/V.

An alternative view is that a domain expert's understanding of a procedure may be evolving and, hence, should be maintainable. Even seemingly well-defined functions may have context-dependant properties that require modification. Consider, for example, the standard Prolog list processing predicate *member/2*. *member/2* has at least two definitions: one that backtracks to find all solutions and one that succeeds only once:

```
% memberA: backtracks
member1(X,[X|_]).
member1(X,[_|T]) :- memberA(X,Y).

% memberB: only succeeds once
member2(X,[X|_]) :- !.
member2(X,[_T|T]) :- member2(X,Y).
```

The semantics of a rule could be crucially dependant on which member is chosen. The backtracking *member2* could permit a single rule to activate multiple sub-goals while a non-backtracking member could usefully cull the search space of solutions. If an expert's initial understanding was that of (e.g.) *member1*, and they wrote rules based on that understanding, then a global conversion to *member2* might have strange side-effects.

A compromise position would be to initialise the stack of procedure environments with an *environment_zero*. Into *environment_zero*, we would install the functions that seem to have a semantics that is commonly understood (such as the set functions described above). If in fact, these functions are understood by the community of experts writing the rules, then they would need no maintenance. If, however, they require modification, then they can be altered as desired using RDF.

REFERENCES

1. Debenham J., *Knowledge Systems Design*, 1989, Prentice Hall, Sydney.
2. Hayes-Roth F., Waterman D.A., & Lenat D.B. (eds) **Building Expert Systems**, Addison-Wesley, Reading Massachusetts, 1983, pp314-321.
3. Aikins J.S. *Prototypical Knowledge for Expert Systems* in **Artifical Intelligence**, 20 (1983) pp163-210.
4. Steels L. *Components of Expertise* in *AI Magazine*, Summer 1990, pp29-49.
5. Compton P. & Jansen R. *Knowledge in Context: A Strategy for Expert System Maintenance* in Barter C. & Brooks M. (eds) **Proc AI 88**, Lecture Notes in Artificial Intelligence, Vol. 406, Berlin, Springer-Verlag, 1990.
6. Compton P., Edwards G., Kang B., Lazarus L., Malor R., Menzies T., Preston P., Srinivasan A. & Sammut C. *Ripple Down Rules: Possibilities and Limitations* Proceedings of the 6th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff 1991, pp6.1 to 6.18 (to appear in the **Knowledge Acqusition Journal**).
7. Gaines B., Compton P. *Induction of Ripple-Down-Rules*, **Proceedings of AI '92**, submitted.
8. Sammut C. Hurst S., Kedzer D. *Learning to Fly* in **Proceedings of the 9th International Conference on Machine Learning**, Aberdeen, 1992 (in press).
9. Clancey W.J. *Model Construction Operators* in **Artificial Intelligence**, 53, (1992) pp1-115.
10. Menzies T.J. *Concerning the Use of Procedural Constructs as a Knowledge Acqusition Technique*, **Australian Knowledge Acqusition Workshop**, IJCAI '91, Hunter Valley Australia, August 1991.
11. Bustany A. & Skingle B *Knowledge-based Development via Application Languages* in **Proceedings of the Fourth Australian Conference on Applications of Expert Systems**, May 11-13, 1988, pp277-302.

12. Menzies T.J. *An Expert System for Raising Pigs* in **Proceedings of the First International Conference on the Practical Application of Prolog**, April 2-3, 1992.
13. Colomb R.M. & Chung C.Y.C. *Very Fast Decision Table Execution of Propositional Expert Systems* in **Proceedings of AAAI-90**, Boston Mass. USA, 30 July- 3 August, 1990.
14. Forgy C.L. *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem* in **Artificial Intelligence**, 19, 1982, pp17-37.
15. Compton P., Edwards G., Srinivasan A., Malor R., Preston P., Kang B. & Lazarus L. *Ripple-down-rules: Turning Knowledge Acquisition into Knowledge Maintenance*, in Artificial Intelligence in Medicine (in press).
16. Patil R.S., Szolovitis P. & Schwartz W.B. *Causal Understanding of Patient Illness in Medical Diagnosis* in **&th IJCAI**, 1981, pp893-899.
17. Apple Technical Publications. (1987) **Hypercard Script Language Guide**, APDA.

CONTENTS

1. Introduction 1
2. Procedural Constructs 2
3. Ripple-Down-Functions 3
4. Pragmatics 6
5. Discussion 7
   5.1 Excessive Change 7
   5.2 Change Everything? 7
REFERENCES 8

LIST OF FIGURES

Figure 1. Good vs bad maintenance 3
Figure 2. An example RDR tree 4
Figure 3. The logic implicit in the example RDR tree 4
Figure 4. RDR tree with functions environments 6

--
--