Some Prolog Macros for Rule-Based Programming: Why? How?

Tim Menzies
Lane Department of Computer Science,
West Virginia University,
tim@menzies.com

Lindsay Mason
Electrical & Computer Engineering,
University of British Columbia, Canada;
lmason@interchange.ubc.ca

Abstract

The history, benefits, and drawbacks to pure rule-based programming is discussed. A simple extension to pure rule-based programming is described. The extensions are very quick to code and can be easily customized to support a range of knowledge engineering applications.

Categories and Subject Descriptors

D.1.6 [**Programming Techniques**]: Logic Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; D.2.M [**Software Engineering**]: Miscellaneous Rapid Prototyping; D.3.2 [**Programming Languages**]: Language Classifications—*Constraint And Logic Languages*; D.3.2 [**Programming Languages**]: Language Classifications—*Extensible Languages*; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized Application Languages*; D.3.2 [**Programming Languages**]: Language Classifications—*Very High-Level Languages*

General Terms

Human Factors,Languages ,Design

Keywords

Rule-Based Programming, History, Prolog

1 Introduction

At a workshop on rule-based programming (hereafter, RBP), it may be heresy to say that there is more to knowledge than just rules. However, after many years of commercial and research work on RBP, we assert that this is so.

This article reviews some of the history of RBP and the need to extend certain aspects of RBP. These extensions are simple to implement- so simple in fact that the entire source code for those extensions can be presented in this article.

2 A Dummies Guide to RBP

2.1 Origins & Early Successes

This article focuses on rule-based knowledge engineering. Hence, by "RBP", we really mean "how rules were used in classical knowledge engineering".

Much of the early 1980s hype surrounding commercial applications of artificial intelligence came from early successes with rule-based *production systems*. Such systems were rule-based systems that queried and updated objects in a *working memory* using a MATCH-SELECT-ACT cycle:

- MATCH: find the rules with conditions satisfied by the current contents of working memory;
- SELECT: pick one rule from the MATCHed set using a *conflict resolution strategy*;
- ACT: perform the action of the picked rule.

There are many advantages to pure RBP. For example, the uniformity of the RPG paradigm makes it amenable to:

- formal analysis of their reliability, e.g. [5];
- powerful learning schemes and languages, e.g. [8];
- the rapid creation of high-productivity programming environments, e.g. [7, 11, 12];
- the rapid training of business users so that they can create their own rule bases, e.g. [14, 15];
- powerful maintenance environments, e.g. [6, 19].

Further, RBP is an insightful theoretical tool for cognitive psychology. Pure RBP can replicate certain expert and erroneous behavior of experts. For example, one way to explain the difference between expert and novice performance is that novices fill their working (human) memory with an excess of active goals. This leaves no room for any intermediaries of any particular calculation. On the other hand, experts have compiled their experience into high-priority rules that select the next best action. Hence, the working memory of an expert has less active goals which means experts are free to use their memory to run computations [9].

Not only is RBP useful for cognitive theory, it is a useful tool for pragmatic software engineers. RBP enables a novel iterative and exploratory software development methodology. Iterative and exploratory software development is very useful when prototyping software. Such prototyping is not required for well-defined tasks. Such well-defined tasks can be implemented via a "waterfall" de-

ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
Third ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02) Pittsburgh, PA, October 5, 2002
Copyright 2002 ACM 1-58113-604-4/02/10 ...$5.00

velopment process; i.e.

$$waterfall = analysis \rightarrow design \rightarrow code \rightarrow test$$

For less-defined tasks, waterfall development can stagnate in the analysis stage since not enough is known about the domain. An alternative approach is to use RBP to generate an executable version of the current conceptualization of a system. Since each rule is a separate chunk of knowledge, it is easy to quickly add more rules. On execution, the interaction of these rules can lead to surprising results that prompt clarifications and extensions of domain knowledge. This approach has been called various things including "knowledge elicitation via irritation" or the RUDE model; i.e.

$$RUDE = \underline{R}un \rightarrow \underline{U}nderstand \rightarrow \underline{D}ebug \rightarrow \underline{E}dit$$

RBP methods resulted in the "AI spring" of the 1980s. Many well-documented, mature, and optimized RBP systems were developed such as ART [1], CLIPS[2], and OPS5 [3] (just to name a few). Numerous significant RBP systems were developed including the commercially successful XCON computer configuration system [13].

2.2 Problems with Rules

The blossoming of RBP in the AI spring was not followed by an RBP summer. An assumption underlying the RUDE approach was the *RUDE assumption*; i.e:

> *Rules are independent chunks of knowledge which can be easily added or changed or removed.*

This proved not to be the case. For example, once XCON grew to 10,000 rules, the developers of XCON had a RUDE[3] awakening: maintaining XCON's rules had become fiendishly complicated. To some extent, this was due to the density of knowledge within XCON:

- The expertise within XCON's rules reflected DEC's state-of-the-art knowledge in configuring computers.
- Such a rich library of knowledge will be intricate to maintain, no matter how it is expressed.

However, another factor that complicated XCON's maintenance was that its rules violated the RUDE assumption. Real-world rule bases often contained groups of rules with significant interactions. For example:

- A careful reverse engineering of XCON showed that the system executed though several *operator spaces* where methods for improving the design of a computer were carefully collected, rejected, elaborated, or assessed, before the appropriate best *operator* was finally selected [1].
- Figure 1 shows three rules that check for certain special cases of bagging groceries. These rules are not independent. Rule *b11* tries to sneak small items into grocery bags that aren't full and which don't contain bottles. If *b11* fails, then rule *b12* just places small items into any grocery bag at all. Rule *b13* creates a new bag when neither of the other two rules can find a bag for small items. Note the tacit reliance of *b12* on *b11* handling a certain special case (bags with bottles). Note also the tacit reliance of *b13* on the other two rules: creating empty grocery bags is a nonsense action *unless* some other agent tries to *first* fill those bags.

```
                         smallitems.pl
b11
in      bag_small_items
if      order=I  with `items has N  and
        grocery with `name=N with `size=small and
        bag=B  with *notFull and      % ◄...................... 5
        not (bag=B with `contents has C and
              grocery with `name=C with *type(bottle))
then    change order=I with select(N,`items,!items) and
        change bag=B with `contents takes N
since   'best to avoid bottles and small items'.

b12
in      bag_small_items
if      order=I with `items has N and
        grocery with `name=N with `size=small and
        bag=B with *notFull
then    change order=I with select(N,`items,!items) and
        change bag=B with `contents takes N
since   'sneaking a small item into a not full bag'.

newbag4small
in      bag_small_items
if      order with `items has N  and
        grocery with `name=N with `size=small
then    make bag with *nothing
since   'need a new bag'.
```

Figure 1. Three rules in the PIKE language (a STARLOG variant). These rules access the object defined in Figure 2, Figure 3, and Figure 4. Example adapted from Winston's BAGGER application [24].

```
                         grocery.pl
groceryDB(1, bread,       bag(plastic),       medium, n).
groceryDB(2, glop,        jar,                small,  n).
groceryDB(3, granola,     box(cardboard),     large,  n).
groceryDB(4, iceCream,    carton(cardboard), medium, y).
groceryDB(5, pepsi,       bottle,             large,  n).
groceryDB(6, potatoChips, bag(plastic),       medium, n).
groceryDB(7, pizza,       box(cardboard),     large,  y).

% GROCERY has five fields, none of which are indexed
grocery=groceryDB(id,name,type,size,frozen).  % ◄............. 10

% define  GROCERY types
grocery*type(T)   --> functor(`type,T,_). % ◄.................. 13

% size symbols to numbers
grocery*volumes([small/1, medium/2, large/3]).

% accessing the numeric size of a particular size symbol
grocery*volume(V) --> *volumes(Vs), `size=S, Vs has S/V.
```

Figure 2. A PIKE definition of the GROCERY object.

The use of such coordinating rules violates the RUDE assumption since every addition to the rule base has to be assessed with respect to its effect on the rest of the rules.

Another problem with pure RBP is that the paradigm can confuse, not clarify, certain types of procedural knowledge. Consider for example, the process of finding the total volume of items in a grocery bag. One *generator rule* is required for transferring pairs of grocery items from that set to a temporary space of "candidate sums". Another *intermediary rule* matches and deletes each pair, then asserts the sum of their sum as another member of the "candidate sums".

[1] From Inference Corporation

[2] The "C" Language Integrated Production System, developed by NASA [18]

[3] Pun. Function: noun. Etymology: perhaps from Italian "puntiglio" which means fine point or quibble. Definition: the usually humorous use of a word in such a way as to suggest two or more of its meanings or the meaning of another word similar in sound.

```
                          order.pl
% ORDER has two fields and the first one is indexed
order=orderDB(+id,     % "+" denotes indexing  ◀................ 2
              items ).

order*size(20). % max number of items in an order
order*active.   % ORDER is "active";
                % i.e. delete all at reset

% Accessing the GROCERY term with a certain Name.
% GROCERY defined in Figure 2
order*item(Name,X) --> grocery with `name=Name
                       with `self=X.

% backtracks through all GROCERY items that are items
% in this ORDER
order*item(Item) --> `items  has Name, *item(Name,Item).
```

Figure 3. A PIKE definition of the ORDER object.

```
                          bag.pl
bag=bagDB(+id,contents).    %◀.................................... 1

bag*active.
bag*capacity(20).
bag*empty --> `contents=[].

bag*newBag(Id,Contents) --> flag(ids,Id,Id+1),
                            !id=Id,  %◀.......................... 8
                            !contents=Contents.
bag*nothing --> *newBag(_,[]).

bag*largeItem(I) --> grocery with `name=I
                            with `size=large.

bag*largeItems1(Item,1) -->  *largeItem(Item),!.
bag*largeItems1(_,0).

bag*largeItems([H|T],N0+N) --> *largeItems1(H,N0),
                               *largeItems(T,N).
bag*largeItems([],0).

bag*largeItems(N) -->  `contents=Items,
                       *largeItems(Items,N0),
                       N is N0.

bag*volume([Item|Items],V0+V) -->             %◀.............. 26
    grocery with `name=Item with *volume(V0),
    *volume(Items,V).
bag*volume([],0).                             %◀.............. 29

bag*volume(V) -->
    `contents=Items, *volume(Items,V0), V is V0.

bag*full --> * volume(V), *capacity(S), V >= S.    %◀......... 34

bag*notFull --> not (* full).      %◀.......................... 36
```

Figure 4. A PIKE definition of the BAG object.

A final *report rule* waits till the generator and intermediary rule stop firing, then accesses the surviving "candidate sum" as the total volume of the grocery bag. A more succinct representation of this procedural summation knowledge that does not use rules is shown in the list summation procedure in Figure 4 between line 26 and line 29.

Many other researchers argued that rules were not the appropriate primitive construct of knowledge engineering. Despite careful attempts to generalize the early RBP knowledge engineering work (e.g. [23]), the construction of knowledge-based systems remained a somewhat hit-and-miss process. By the end of the 1980s, it was recognized that design concepts such as RBP were incomplete [4]. For example, Bobrow's reverse engineering of real-world knowledge-based systems [2] found that numerous paradigms were being employed including rule-based, logic-based,

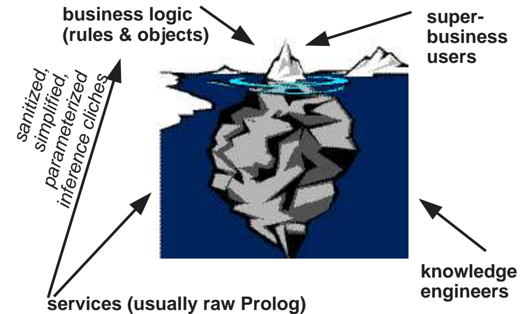

Figure 5. The "iceberg model" of knowledge engineering.

functional, object-oriented, and "access-based" (which, these days, we might call implicit invocation [22]). The 1990s was characterized by an extensive search for *higher-level reusable patterns of inference* such as:

- Propose-and-revise (e.g. as done by [21]).
- Recursive descent of "problem spaces" (e.g. as done by [25]).

2.3 Beyond RBP

The above problems, and our own commercial knowledge engineering (e.g. [14, 15]), lead us to extent RBP. As done in many other shells (e.g. ART, CLIPS), the need to use both procedural and declarative rule knowledge made us combine RBP with a simple object-oriented approach. Rule conditions and actions could use verbs defined in the object-language. For example, rule *b12* on line 5 of Figure 1 checks that a bag is *notFull*, where *notFull* is a procedure defined at the end of *bag* on line 36 of Figure 4.

Also, for a while, we tried coding up knowledge engineering languages based on the supposedly reusable *higher-level reusable patterns of inference*. However, there was a problem. Our repeated experience was that while small communities of experts might reuse an inference pattern, that pattern was not widely endorsed elsewhere. That is, while designing a rule-base around a certain inference pattern was useful, each new application needed a new inference pattern (an effect reported elsewhere [10]). More generally, while many higher-level inference patterns have been identified (e.g. propose-and-revise, heuristic classification, recursive descent of "problem spaces'), the reusability of these patterns is questionable since there never was widespread and stable agreement of the internal structure of these patterns [16, 17].

Even though inference patterns may not be reusable between domains, they may be useful within a particular domain. Our default architecture for a new knowledge based system was the *iceberg model* of Figure 5. In that architecture, knowledge engineers work "under the waterline" to build infrastructure to support the "in view" knowledge bases created by advanced business users. Our role as knowledge engineers was to:

- Identify cliches in the expert's approach to different problems. Such cliches may include the supposedly reusable inference pattern.
- Craft support code for each such cliches.

Where possible, the support code was heavily parameterized so that

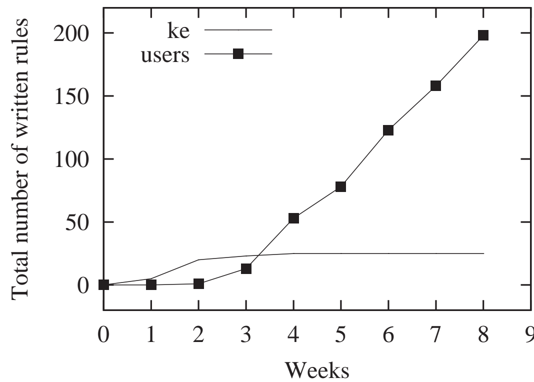

Figure 6. Patterns of rule growth. KE= knowledge engineers

it could be extensively customized. These customization parameters then became the “tip of the iceberg” that was visible to our business users. These users then used these upper-most drivers in their rules and objects. Our cliches included low-level idioms such as summing all items in a list as well as domain-specific high-level inference patterns.

Figure 6 shows a typical pattern of authoring rules using this iceberg approach. Note that the knowledge engineers write some rules in the initial stages while, by the end of the development, the users have written most of the rules. This pattern of rule authoring arises from the following development methodology:

Language development: Initially, the knowledge engineers struggle to understand the domain and identify the relevant cliches. After a week or two, some of these cliches are found and implemented as support code.

Transition: The knowledge engineer then builds a few sample rules to demonstrate the usage of the support code. These sample rules are then used to train the business users.

Development: Business users go on to write most of the knowledge base.

Language elaboration: The knowledge engineer watches their progress to identify common inference cliches that are awkward or clumsy or error prone to encode. The knowledge engineers then (i) augment the support code and (ii) show the business users how to simplify their rules using the augmented support code. As a result, the business users learn how to encode and update their own rule base using a knowledge language that has been heavily customized to their domain.

Maintenance: Maintenance in this approach is relatively simple since business users can update their own knowledge even when the knowledge engineer is unavailable.

3 STARLOG

The iceberg model is only possible when the practitioner can quickly craft a new set of inference cliches. The rest of this paper discusses STARLOG, a customization tool kit for knowledge engineering.

The STARLOG system is a set of *load-time macros* that convert sentences in some domain-specific terminology into a simple clause-based logic. Since these macros are called at load time then, in many cases, the overheads of interpretation is incurred once at load time and never at runtime.

STARLOG is a Prolog-based framework for building different languages for knowledge engineering. Prolog was chosen as the underlying implementation language for several reasons. Firstly, the iceberg model demands that the high-level knowledge structure be implemented on top of routines written in a general purpose language. Prolog is a general programming langauge with facilities for defining user-friendly structures in a pseudo-English style; e.g. the rules shown in Figure 1. That is it can handle both the low-level routines and the high-level structures. Also, in the authors' experience, Prolog is easier to customize than the other languages they know well such as Lisp, Scheme, C, Pascal, Smalltalk, Perl, Awk, or JAVA. Further, Prolog's built-in search and match simplifies much of the implementation of a RBP shell.

This paper presents 367 lines of Prolog that implements a simple, but useful rule/object interpreter/optimizer. Of this code, 127 lines are support utilities; 153 implement an optimized simple object language and 87 lines implement a forward chaining rule-based system. Such a small set of utilities can easily be customized for new domains. One such customization is the PIKE language[4] used for the rules and objects shown in Figure 1, Figure 2, Figure 3 and Figure 4.

PIKE supports three main constructs shown: objects, methods, and rules. These constructs are discussed below, after introducing a sample PIKE application and reviewing some Prolog technology.

4 A Sample Application

This paper contains full source code for a PIKE rule/object system that extends Patrick Winston's BAGGER problem [24]. BAGGER is Winston's allegory for XCON: XCON configures computers by checking the right components are combined together while BAGGER checks that the right grocery orders are combined together in grocery bags. Our extension includes rules, objects and methods for groceries, bags, and orders.

PIKE's BAGGER is loaded in Figure 7. The system contains five rule groups:

`Global`: This is the initial group. It creates a sample order; see Figure 7. The current group is then changed to...

`Check_order`: Checks if any items are missing from the order; see Figure 8. The current group is then changed to...

`Bag_large_items`: Handles the bagging of the bulky items; see Figure 9. The current group is then changed to...

`Bag_medium_items`: Handles the bagging of the mid-sized items; see Figure 10. The current group is then changed to...

`Bag_small_items`: Tries to sneak the small items into the bags created above; see Figure 1.

Figure 11 shows what happens when the whole system is loaded

[4]Why “Pike”? For Star Trek aficionados, we offer the following notes (and others need not read any further). STARLOG variants should be named, in order, after the captains of the Rodenberry-class star ships: i.e. PIKE KIRK, SPOCK, PICARD, SISKO, JANEWAY, DA'AN and ARCHER. The names STOCKER, DECKER, KAHN and ZO'OR are reserved for throw-away crazy prototypes, for obvious reasons.

ruleseg.pl

```
:- [starlog].     % see Figure 32

:- [grocery       % see Figure 2
   ,order         % see Figure 3
   ,bag           % see Figure 4
   ].

:- [checkrules    % see Figure 8
   ,largeitems   % see Figure 9
   ,mediumitems  % see Figure 10
   ,smallitems   % see Figure 1
   ].

startup
in    global
if    true
then  make bag with *nothing and
      make order
         with 'id=1
         with 'items= [bread,glop, pizza,
                       granola,iceCream,
                       potatoChips] and
      goto check_order
since 'BAGGER v3.0 is up and running!!'.

ruleseg :- time(fchain),
           listing(orderDB),
           listing(bagDB).

:- demos(ruleseg).
```

Figure 7.

checkrules.pl

```
b1
in     check_order
if     order=I with 'items has potatoChips and
       not (order=I  with 'items  has  N and
            grocery  with 'name=N  with *type(bottle)
            )
then   change order=I  with 'items takes pepsi
since 'order ' and I and ' has chips, but needs pepsi'.

b2
in    check_order
if    true
then  goto bag_large_items
since 'all done with checking orders'.
```

Figure 8.

and executed. Given the ORDER

```
[bread, glop, pizza, granola,
 iceCream, potatoChips]
```

PIKE's BAGGER generates two bags:

```
bagDB(1, [glop, potatoChips, iceCream, bread]).
bagDB(0, [granola, pizza, pepsi]).
```

5 Inside Prolog and PIKE

This section discusses some details of Prolog and PIKE. For space reasons, this discussion misses certain details. A longer version of this paper is under preparation and will contain more information.

5.1 Prolog

5.1.1 Terms

The basic building block of Prolog is a *term*. Terms have functors and arguments. For example, the term

```
carried(variables(who,age,cheer)).
```

has a functor `carried` with one argument which, in turn, has the functor `variables` with three arguments. Terms with only one or two arguments need not use brackets if the functor is declared to be an *operator*.

largeitems.pl

```
bottles
in    bag_large_items % ◀........................................ 2
if    order=I with 'items has N and
      grocery with 'name=N
              with 'size=large
              with *type(bottle) and
      bag = B with *largeItems(L) and
      L < 6
then  change order=I with select(N,'items,!items) and
      change bag = B with 'contents takes N
since 'there''s room in bag ' and B and
      ' for a large bottle'.

largeitems
in     bag_large_items
if     order=I  with 'items has N  and
       grocery with 'name=N
               with 'size=large    and
       bag = B with *largeItems(L) and
       L < 6
then   change order=I with select(N,'items,!items) and
       change bag = B with 'contents takes N
since 'there''s room in bag ' and B and
      ' for one ' and N.

newbag
in      bag_large_items
if      order with 'items has N  and
        grocery with 'name=N
                with 'size=large
then    make bag with *nothing
since   'need a new bag'.

endlarge
in      bag_large_items
if      true
then    goto bag_medium_items
since   'all done with large items'.
```

Figure 9.

mediumitems.pl

```
b8
in     bag_medium_items
if     order=I  with 'items has N and
       grocery with 'name=N with 'size=medium and
        (bag=B   with *empty   or
         (bag=B with 'contents has C and
          grocery with 'name=C with 'size=medium)
        )
then  change order=I with select(N,'items,!items) and
      change bag=B with 'contents takes N
since 'bag ' and B and ' can hold item ' and N.

newbag4medium
in      bag_medium_items
if      order with 'items has N  and
        grocery with 'name=N
                with 'size=medium
then    make bag with *nothing
since   'need a new bag'.

endmedium
in      bag_medium_items
if      true
then    goto bag_small_items
since   'all done with small items'.
```

Figure 10.

```
ruleseg.out
% output from ruleseg.pl

[global::startup]
   BAGGER v3.0 is up and running!!
[check_order::b1]
   order [1] has chips, but needs pepsi
[check_order::b2]
   all done with checking orders
[bag_large_items::bottles]
   there's room in bag [0] for a large bottle
[bag_large_items::largeitems]
   there's room in bag [0] for one pizza
[bag_large_items::largeitems]
   there's room in bag [0] for one granola
[bag_large_items::endlarge]
   all done with large items
[bag_medium_items::newbag4medium]
   need a new bag
[bag_medium_items::b8]
   bag [1] can hold item bread
[bag_medium_items::b8]
   bag [1] can hold item iceCream
[bag_medium_items::b8]
   bag [1] can hold item potatoChips
[bag_medium_items::endmedium]
   all done with small items
[bag_small_items::b11]
   best to avoid bottles and small items

:- dynamic orderDB/2.

orderDB(1, []).

:- dynamic bagDB/2.

bagDB(1, [glop, potatoChips, iceCream, bread]).
bagDB(0, [granola, pizza, pepsi]).

% runtime = 0.04 sec(s)
```

Figure 11.

```
ops.pl
:- op(999, xfx , if).
:- op(998, xfx , then).
:- op(997, xfx,  since).
:- op(996,  fx,  say).
:- op(990, xfy , or).
:- op(989, xfy , and).
:- op(988,  fy , not).
:- op(980,  fx,  [make,change,zap]).
:- op(970, xfy , with).
:- op(969, xfx , takes).
:- op(968, xfx , has).
:- op(1,    fx , goto).
:- op(1,   xfy , [at,in]).
:- op(1,    fx , ['`,!,*]).
```

Figure 12.

Figure 12 defines some operators which lets the Prolog reader input pseudo-English statements without requiring tedious; e.g.

```
a if b and c or d then l and m.
```

Internally, these operators become standard terms:

```
:- [ops].
:- Rule = ((a if b and c or d then l and m)),write_canonical(Rule).

if(a, then(  or(  and(  b, c),
                  d),
             and( l, m)))
```

This term seems fearsomely complex but can easily be processed by Prolog's pattern matching facility. For example, the query

```
Label if OR1 or OR2 then Action
```

will search for any rule and, if the above example had been asserted into the Prolog database, it would yield

```
LABEL=  a
OR1=    b and c
OR2=    d
ACTIOn= l and m
```

5.1.2 *Clauses*

A Prolog program is a set of clauses of the form

```
Head :- SubGoal1, SubGoal2,...
```

where `Head, Subgoalx` are terms. To prove a `Head`, Prolog seeks clauses whose head match `SubGoali`. The subgoals are explored left-to-right in the order in which they appear in the source code. Prolog is hence often called a backward-chaining language since the Prolog interpreter from heads back to subgoals that might prove that head. Nevertheless, it is simple to implement in Prolog forward-chaining rules. For example, if we rewrite

```
RuleId if Condition then Action
```

as

```
rule(RuleId) :- Condition, Action
```

then the call

```
:- rule(X).
```

will forward chaining through the rules trying the conditions before the actions.

PIKE will use a small variant of the above scheme: rule conditions and rule actions will be separated into two clauses that share variables in the head; e.g.:

```
cond(RuleId) :- Condition.
act(RuleId) :- Action.
```

This "two clause" trick enables MATCH-SELECT-ACT since it lets us test conditions without necessarily firing any actions.

5.1.3 `Term_expansion`

The "two clause" trick is implemented by an adjustment to the Prolog reader that performs certain processing at load time. The following code

```
term_expansion(X if Y then Z,[(cond(X) :- Y), (act(X) :- Z)]).
```

tells Prolog that if ever a rule is read, the appropriate `cond,act` clauses should be asserted *instead of* the rule.

One way to recognize the user-level constructs in a Prolog program is to look for the `term_expansion` definitions. PIKE's definitions are shown in Figure 13. The constructs shown there will be discussed below, after we see a little more Prolog and PIKE.

```
hooks.pl
% METHODS:
% Implemented in Figure 19
term_expansion((Helper*Head --> Body),  X) :-
    ecg((Helper*Head --> Body),X).
term_expansion(Helper*Head, X) :-
    ecg((Helper*Head --> true),X).

% OBJECTS:
% Implemented in Figure 21
term_expansion(Helper=Spec, X) :-
    spec(Helper=Spec,X).

% RULES:
% Implemented in Figure 23
term_expansion(Label if Condition then Action,X) :-
    rules(Label if Condition then Action,X).
```

Figure 13.

```
sharedvars.pl
sharedVars(T1,T2,V) :- vars(T1,V1), vars(T2,V2),
                       sharedVars1(V1,V2,V),!.
sharedVars(_,_,[]).

sharedVars1([],_,[]).
sharedVars1([H|T0],L,[H|T]) :- member(X,L),H == X,!,
                              sharedVars1(T0,L,T).
sharedVars1([_|T0],L,   T ) :- sharedVars1(T0,L,T).

vars(Term,All) :- setof(One,vars1(Term,One),All).

vars1(Term,V) :- subterm(Term,V), var(V).

subterm(X,X).
subterm(In, X) :- compound(In), arg(_,In,Arg),
                  subterm(Arg,X).
```

Figure 14.

5.1.4 Variables

The two clauses cond and act require some extension in order to support the conflict resolution of RBP. For example, it may be required that actions are never fired twice in the same situation. It is therefore useful to check that the bindings passed to the action are unique.

Figure 14 shows code that can find the variables shared by two terms: this will be used later by PIKE to ensure that actions are only fired on new bindings. These shared variables are computed once at load time, then cached. More specifically, PIKE stores rules internally not as cond/2 and act/2 but as lhs/4 and rhs1/4:

```
lhs(Group,Priority,Id,Memory) :- Condition
rhs1(Group,Id,Memory) :- Action
```

where Memory are the shared variables found by Figure 14, and the other variables locate a rule within a particular rule Group at a Priority set by the user.

5.1.5 DCGs

Another useful Prolog trick, used extensively by PIKE, are the *carry variables* of the *definite clause grammars*, or DCGs. DCGs are special clauses of the form:

```
head --> subgoal1, subgoal2,... subgoalN.
```

which, at load time, are converted by term_expansion into the standard format; e.g.

```
oldNew.pl
oldNew(Field,Old,New,C0,C1) :-
    carried(Term),
    functor(Term,F,A),  Term =.. [F|Fields],
    functor(C0,  F,A),  C0   =.. [F|L0],
    functor(C1,  F,A),  C1   =.. [F|L1],
    oldNew1(Fields,Field,Old,New,L0,L1).

oldNew1([Field|_],Field,Old,New,[Old|T],[New|T]) :- !.
oldNew1([_|Fields],Field,Old,New,[H|T1],[H|T2]) :-
    oldNew1(Fields,Field,Old,New,T1,T2).
```

Figure 15.

```
head(         In,          Out) :-
     subgoal1(In,C1),
     subgoal2(   C1,C2),
     subgoal3(      C2,C3),
     ...
     subgoalN(          Cn,Out).
```

where the variables In,C1,C2... Out are *carry variables* that are passed from head to subgoals, then back to the head (note that the last subgoal contains Out which is also in the head of the clause). These carry variables can be used to carry around modifiable state information. For example, subgoal2 can generate a new C2 from the C1 variable passed to it.

PIKE uses DCGs to simplify accessing named data fields. For example, in the following code, we are performing calculations using carry variables held by a term with three named fields who, age and cheer:

```
carried(variables(who,age,cheer)).
birthday --> +age, -cheer.
```

That is, every birthday we get a little older and a little less cheerful. Internally, the DCG definition of birthday becomes:

```
birthday(  In,     Out) :- +(age, In, C1), -(cheer,   C1, Out).
```

This clause requires a implementation of + and - which increment and decrement (respectively), the named fields age and cheer. To handle that process, we create a new term and copy over nearly all the variables from the old term to the new term. The only variable not copied verbatim is the field referenced in the code (e.g. age) which we modify before inserting into the new term.

```
+(Field,C0,C) :- oldNew(Field,Old,New,C0,C), New is Old + 1.
-(Field,C0,C) :- oldNew(Field,Old,New,C0,C), New is Old - 1.
```

OldNew is defined in Figure 15 and, reading through it, the reader might protest “That’s a lot of work just to add one plus one!”. PIKE uses a much faster method to access named variables (see later, in §5.2.4) but for pedagogical reasons lets assume that we must optimize oldNew.

5.1.6 Goal_expansion

Recall from the above that term_expansion pre-processes an entire clause. A simpler expansion facility is goal_expansion which only works on individual subgoals.

Using goal_expansion it is trivial to perform (e.g.) all the oldNew processing *once* at load time and *never* at runtime. Suppose the following goal_expansions were asserted *before* loading our birthday definition:

```
goal_expansion(+(F,C0,C), New is Old + 1) :- oldNew(F,Old,New,C0,C).
goal_expansion(-(F,C0,C), New is Old - 1) :- oldNew(F,Old,New,C0,C).
```

If so, then the definition of birthday is highly optimized since its internal form now becomes:

```
birthday(variables(A, B, C), variables(A, D, E)) :- D is B+1, E is C-1.
```

Note in the above form, the who variable (A) is copied over unmodified to the output term.

While this scheme seems overkill for adding one plus one, it has certain advantages. For example, if the number of the carried variables every grows, then the rest of the code will adjust automatically. Also, as we shall see, this simple scheme can be extended to implement an interesting object-based system.

5.2 PIKE

This section describes the high-level details of PIKE (and the lower-level details are shown in the appendix).

5.2.1 Named Fields

The discussion above offered a simple introductory example of using named fields within a term. PIKE's usage extends that simple example in several ways. Firstly, PIKE's definition of named fields will allow multiple name spaces. To make the user's life easier, we call each name space a *class*. For example, the example shown in this file defines one class called GROCERY with named fields id, name, type, size, frozen and another class called ORDER with named fields id, items:

```
grocery=groceryDB(id,name,type,size,frozen).
order=orderDB(+id, items ).
```

The syntax for defining the named fields is a little different to the carried fact used above. In the PIKE syntax, classes are defined using:

```
Handle=Functor(Field1,Field2,...)
```

where Handle is how user code refers to the class and Functor is an internal name used by PIKE. Fields can optionally be marked with an + symbol denoting that it is an indexed term.

5.2.2 Get References and Starred Clauses

Another way PIKE extends the above example about birthday is that named fields can appear *anywhere* in a term. For example:

```
who=person(height,weight).
who*bodySurfaceArea(A) --> A is 0.20247 * 'height^0.725 * 'weight^0.425.
```

Here, 'X (e.g. 'height) is a *get reference* and are shorthand for "go get the value of the field named X *before* running this subgoal".

Note the use of who*bodySurfaceArea(A). These starred clauses denotes a clause that should be called assuming that the carried variables contain the named fields of the who class.

5.2.3 Set References

Apart from 'X, another useful PIKE trick is the *set reference* !X. Set references are shorthand for "go set the value of the field named X to the variable *after* running this subgoal". So, in PIKE, it is possible to write

```
..., append('siblings,'newBornBabies,!siblings), ...
```

That is, the new value of siblings will be the old value appended to newBornBabies. Hence, in PIKE-Prolog, it is finally possible to write $x = x + 1$ as follows:

```
..., !x is 'x + 1, ...
```

5.2.4 Accessors

Get and set methods need knowledge of how to reach variables within a term. This is implemented via *accessor predicates* which take the form

```
Handle(Field, OldValue, NewValue, OldTerm, NewTerm).
```

These accessors copy every member of OldTerm to NewTerm *except* for the variable for the field named Field. The value of that binding in OldTerm is OldValue and the value of that binding in NewTerm is NewValue. For example, if PIKE reads

```
who=person(name,age, height,jobs).
```

then it would use term_expansion to generate:

```
who(name,  Old, New, person(Old,X,Y,Z), person(New,X,Y,Z)).
who(age,   Old, New, person(X,Old,Y,Z), person(X,New,Y,Z)).
who(height,Old, New, person(X,Y,Old,Z), person(X,Y,New,Z)).
who(jobs  ,Old, New, person(X,Y,Z,Old), person(X,Y,Z,New)).
```

These accessors can be used as follows. The call

```
... , who(age, OldValue, OldValue, OldTerm, OldTerm), ...
```

accesses age without changing it. Also, the call

```
... , who(age, _, NewValue, OldTerm, NewTerm), ...
```

throws away the OldValue of age and replaces it with NewValue. Further, the call

```
... , who(age,OldValue, NewValue, OldTerm, NewTerm),
      NewValue is OldValue+1, ...
```

places a value one more than the OldValue of age into NewValue. Lists can be updated the same way. For example, the following call pushes x onto the current list of jobs:

```
... , who(job,OldValue, [x | OldValue], OldTerm, NewTerm), ...
```

5.2.5 ECGs: extended clause grammars

PIKE's extension to DCGs are called ECGs (extended clause grammars) and include classes, starred clauses, set references and get references.

To implement the set and get references of ECGs, PIKE has to parse a term and generate two lists: named fields to access before calling a goal, and named fields to change after calling a goal. This parser is shown in Figure 16. This code generates as a side-effect the actually callable term with the named fields changed for the get/set references. For example:

```
wrapper.pl
wrapper(X,F,Out) :-
    wrap(X,F,Before,[],After,[],Goal),
    append(Before,[call(Goal)|After],Out).

wrap(X,F,B0,B,A0,A,Y) :-
    once(wrap0(X,Z)),
    wrap1(Z,F,B0,B,A0,A,Y).

wrap0(X,        leaf(X) ) :- var(X).
wrap0(X,        leaf(X) ) :- atomic(X).
wrap0([],    leaf(true) ).
wrap0([H|T],      [H|T] ).
wrap0(`X,            `X ).
wrap0(!X,            !X ).
wrap0(X,        term(X) ).

wrap1(leaf(X), _,B, B, A, A, X).
wrap1([H0|T0], F,B0,B, A0,A, [H|T]) :-
    wrap(H0,   F,B0,B1,A0,A1,H),
    wrap(T0,   F,B1,B, A1,A, T).
wrap1(term(X), F,B0,B, A0,A, Y) :-
    X =.. L0,
    wrap(L0,F,B0,B,A0,A,L),
    Y =.. L.
wrap1(`X, F,[H|B],B,A,A,Y) :- H=..[F,X,Y,Y].
wrap1(!X, F,B,B,[H|A],A,Y) :- H=..[F,X,_,Y].
```

Figure 16.

```
?- wrapper(append(`siblings,`newBornBabies,!siblings),who,W).

W = [ who(siblings, A, A)
    , who(newBornBabies, B, B)
    , call(append(A, B, C))
    , who(siblings, _, C)
    ]
```

These wrapped calls are then placed into a DCG clause; e.g.

```
xx --> who(siblings, A, A), who(newBornBabies, B, B),
       call(append(A, B, C)), who(siblings, _, C).
```

which means that, internally, these become

```
xx(                     In,          Out) :-
    who(siblings, A, A, In,   C1)
    who(newBornBabies, B, B,  C1,C2)
    call(append(A, B, C)),
    who(siblings, _, C,          C2, Out).
```

Note that the `who/3` facts have been expanded to `who/5` facts; i.e. they can now access using the accessors defined in the previous section.

5.2.6 *Unfolding*

ECGs also include a simple unfolding optimizer where subgoals are sometimes replaced by the body of the clause whose head matches the subgoal. That's quite a mouthful but is simple to show:

```
a(X) :- b,c(X),d(X).

c(X) :- X is pi* 2.

d(X) :- X > 10, print(big).
d(X) :- X <= 10, print(small).
```

If we unfold `a(X)`by replacing subgoals with other clause bodies, we get:

```
a(X) :- b, X is pi*2,d(X).
```

In this example, `d(X)` was not unfolded since there are two clauses for `d(X)` (these could be added as a disjunction, but there seems little advantage in doing so). Figure 17 shows code that detects subgoals that only match one head in the Prolog database . PIKE calls these solo clauses *singleton*s and only singletons are unfolded.

```
singleton.pl
singleton(X) :-
    copy_term(X,Y),
    clause(X,_,Ref1),
    not((
        clause(Y,_,Ref2),
        not(Ref1 =:= Ref2))).

one(X) :- singleton(X),X.
```

Figure 17.

```
tidy.pl
tidy(A,C) :-
    option(brave)
    -> once(tidy1(A,B)),once(tidy1(B,C))
    ;  once(tidy1(A,C)).

tidy1(A,             A) :- var(A).
tidy1(X=X,        true) :- option(brave).
tidy1(X is Y,     true) :- option(brave), ground(Y), % ◀....... 8
                           X is Y.
tidy1((A :- true),   A).
tidy1((A :- B),      R) :- tidy1(B,TB),
                           (TB=true -> R=A; R=(A:-TB)).
tidy1((A,B),    (A,TB)) :- var(A), tidy1(B,TB).
tidy1((A,B),    (TA,B)) :- var(B), tidy1(A,TA).
tidy1(((A,B),C),     R) :- tidy1((A,B,C), R).
tidy1((true,A),      R) :- tidy1(A,R).
tidy1((A,true),      R) :- tidy1(A,R).
tidy1((A,B),         R) :- tidy1(A,TA), tidy1(B,TB),
                           (TB=true -> R=TA; R=(TA,TB)).
tidy1((A;B),   (TA;TB)) :- tidy1(A,TA), tidy1(B,TB).
tidy1((A->B), (TA->TB)) :- tidy1(A,TA), tidy1(B,TB).
tidy1(not(A),  not(TA)) :- tidy1(A,TA).
tidy1(A,             A).
```

Figure 18. Remove redundant trues.

Note also that the subgoal `X is pi*2` could also be executed a compile time. Line 8 in Figure 18 shows code that looks for such compile-time executable statements. These variables bound by this compile-time call spread to the rest of the clause and the called subgoal is replaced with `true`. The rest of Figure 18 seeks out redundant `true`s and removes them. Without this removal of redundant trues, our definition of `a(X)` would be:

```
a(6.28319) :- b, true, d(6.28319).
```

With this removal, the clause becomes

```
a(6.28319) :- b, d(6.28319).
```

One danger with unfolding is *left propagation*; i.e. variables bound in $subgoal_i$ inappropriately effecting $subgoal_{j<i}$. For example, the following code is meant to print some value before checking if it satisfies some property:

```
e(X) :- print(X), c(X),X>10.
```

If we unfold `c(X)` as above, we get

```
e(6.28319) :- print(6.28319),  6.28319>10.
```

Note now that calling `e(1)` results in a different behavior depending on whether or not we call the original or the unfolded version: the original version prints a number, then fails, while the unfolded version fails before printing.

Left propagation is a well-studied problem [20]. Some of the solutions are complex so, for simplicity sake, PIKE just has one flag `brave` (see line 8 in Figure 18) that can disable this kind of unfolding.

5.2.7 *Objects, Methods, and Rules in PIKE*

Giving the above tools, it is simple to now implement objects, methods and rules for PIKE. For example, PIKE methods are ECG clauses, which are implemented by `ecg/2` defined in Figure 19. Also, PIKE's classes were described above and are implemented by Figure 21 (currently, PIKE's objects support encapsulation and polymorphism, but not inheritance). Example object files shown in the article are the GROCERY class of Figure 2, the ORDER class of Figure 3, and the BAG class of Figure 4. Figure 22 shows the internal Prolog representation of Figure 20. The accessor predicated of `grocery/5` start at line 15 in Figure 22.

PIKE's rules can access the above described classes and methods. The idiom `Label if Condition then Action` is PIKE's way of defining forward chaining rules. Rules have *priorities* and *groups* which can be specified within the *Label* The default group and priority is *global* and 10, respectively. For example, line 2 in Figure 9 shows a rule being entered into the *bag_large_items* rule group.

Internally, the rule

```
Id in Group at Priority if Condition
then Action
```

is converted into two Prolog predicates

```
lhs(Group,Priority,Id,Memory) :- Condition
rhs1(Group,Id,Memory) :- Action
```

(see line 14 in Figure 23 and line 16 in Figure 23). As described above, this separation permits the extensive customization of the forward chainer since rule conditions can be tested without triggering the rule action.

The variables `Group,Priority,Id,Memory` are used by PIKE's *conflict resolution strategies*. Recall that conflict resolution is the process of selecting one rule from the space of rules of satisfied conditions. PIKE employs the following conflict resolution strategies:

Rule groups: PIKE maintains a pointer to the current group in the `group/1` fact (see line 36 in Figure 24). Only rules within the current group are tested.

Priority ordering: Prior to forward chaining, PIKE gathers together a list of all the unique group names and rule priorities within each group (see line 20 in Figure 24). At runtime, rules are explored within a group in priority ordering starting with priority one and continuing to lower priorities (see line 29 in Figure 24 and line 35 in Figure 24).

Refraction: PIKE never fires the same rule action twice on the same set of variable bindings. The `Memory` argument of `lhs/4` and `rhs1/3` contains all the variables passed from the `Condition` to the `Action`. These shared variables are found via `sharedVars/3` shown in Figure 14 which is called at line 29 in Figure 23.

Recency: When PIKE asserts anything, it is asserted above all older assertions (e.g. see line 42 in Figure 22 and line 46 in Figure 22). Hence, rules will fire more on newer assertions than older assertions.

ecg.pl

```
ecg((H*X0 --> Y0),Out) :-
    ecg1(Y0,H,Y,W0,W),
    X =.. [H,X0,W0,W],
    expand_term((X :- Y),Temp),
    tidy(Temp,Out).

ecg1(X,H,Y,W0,W) :- once(ecg0(X,Z)), ecg2(Z,H,Y,W0,W).

ecg0(X           ,         leaf(X)) :- var(X).
ecg0(!           ,               !).
ecg0((X -> Y)    ,   two((->),X,Y)).
ecg0((X and  Y)  ,   two((, ),X,Y)).
ecg0((X ,  Y)    ,   two((, ),X,Y)).
ecg0((X or  Y)   ,   two((; ),X,Y)).
ecg0((X ;  Y)    ,   two((; ),X,Y)).
ecg0(not X       ,    one((not),X)).
ecg0(* Call0     ,     local(Call)) :- c2l(Call0,Call).
ecg0(H*Call0     , foriegn(Call,H)) :- c2l(Call0,Call).
ecg0(`X takes  Y,       `X takes Y).

% ◀............................................................. 21
ecg0(change X=Id with Y0, with(X,Id,Y,gets,sets)):-
    w2c(Y0,Y).
ecg0(make   X    with Y0,with(X,_,Y,blank,makes)):-
    w2c(Y0,Y).
ecg0(zap    X=Id with Y0, with(X,Id,Y,gets,zaps)):-
    w2c(Y0,Y).
ecg0(zap    X=Id,         with(X,Id,true,gets,zaps)).
ecg0(X=Id        with Y0, with(X,Id,Y,gets,noop)):-
    w2c(Y0,Y).
ecg0(X           with Y0, with(X,_ ,Y,gets,noop)):-
    w2c(Y0,Y).
% ◀............................................................. 33

ecg0(X,                                  wrap(X)).

ecg2(!,               _,!) --> [].
ecg2(`List takes Item,H,X) --> ecg1(!List=[Item|`List],
                                     H,X).
ecg2(one(O,A0),   H,X)     --> ecg1(A0,H,A), X=..[O,A].
ecg2(two(O,A0,B0),H,X)     --> ecg1(A0,H,A),
                               ecg1(B0,H,B),
                               X =.. [O,A,B] .
ecg2(leaf(X),     _,X)     --> [].
ecg2(wrap(X0),    H,X,W0,W) :- wrapper(X0,H,X1),  % ◀........ 45
                               ecg3(X1,X,W0,W).
ecg2(local(L),    H,X)     --> calls(L,H,X).
ecg2(foriegn(L,H ),_,X,W, W) :- calls(L,H,X,_,_).
ecg2(with(H,Id,X0,Pre,Post),_,(One,Two,Three),W1,W1) :-
                     One   =..[Pre, H,W0,Id],
                     Three =..[Post,H,W,Id],
                     ecg1(X0,H,Two,W0,W).
noop(_,_,_).

ecg3([X0],X)            --> add2(X0,X).
ecg3([X0,Y|Z],(X,Rest)) --> add2(X0,X),
                            ecg3([Y|Z],Rest).

add2(call(X),X,W,W) :- !.
add2(T0,T,W0,W) :- T0 =.. L0, append(L0,[W0,W],L1),
                   T =.. L1.

calls([Call0],H,Call,W0,W) :- Call =.. [H,Call0,W0,W].
calls([Call0,Call1|Calls0],H,(Call,Calls),W0,W) :-
    Call =.. [H,Call0,W0,W1],
    calls([Call1|Calls0],H,Calls,W1,W).

w2c(A with B,(A,C)) :- !,w2c(B,C).
w2c(A,A).
```

Figure 19.

PIKE's MATCH-SELECT process assumes that the order in which PIKE's rules are to be tested can be determined via the rule priority number. If rules are tested in this order, then the *first* rule with a satisfied condition would be the highest priority satisfied rule. By

```
                      speceg.pl
:- [starlog]. % see Figure 32

:- [grocery].

portray(':spec'(A, B, C, D, E, F)) :-
    format(''':spec''(~p, ~p, ~p',[A,B,C]),nl,
    format('         ~p, ~p, ~p).',[D,E,F]).

speceg :-
    listing(grocery),
    spec(grocery=groceryDB(id,name,type,size,frozen),
         List0),
    none(List0,grocery,List),
    show(List).

none([],_,[]).
none([(H:-_)|T],F,    Rest) :- functor(H,F,_), !,
                               none(T,F,Rest).
none([H|T],     F,    Rest) :- functor(H,F,_), !,
                               none(T,F,Rest).
none([H|T],     F,[H|Rest]) :- none(T,F,Rest).

:- demos(speceg).
```

Figure 20. Starlog sample; generates Figure 22.

exploring rules in this order, PIKE avoids a computationally expensive MATCH process.

Rules are operations that can move across class boundaries. Hence, the PIKE programmer needs a way to easily access and switch between the name spaces of different classes. The `with` statement enables this accessing and switching. The idiom

```
Class=Id with Method1 with Method2 with ...
```

is expanded these to multiple method calls invoked over the same object. Important variants of this idiom are:

- `change Class=Id with Method1 with ...`
  The *Methods* are prefixed by a match to the object and are followed by an update to the object.
- `make Class with Method1 with ...`
  The `Methods` are run on a new object of the specified `Class` and are followed by an assertion of the resulting object.
- `zap Class=Id with Method1 with ...`
  The `Methods` are prefixed by a match to the object and are followed by deletion of the object. The `Methods` are called prior to deletion.
- `zap Class=Id`
  The `Methods` are prefixed by a match to the object and are followed by deletion of the object.

The prefix and following code is added between line 21 in Figure 19 and line 33 in Figure 19

```
                      spec.pl
spec(Helper=Spec,
        [ ':spec'(Helper,F,Term1,Ids,Indicies,Names)
        , (:- index(Index))
        , (:- dynamic F/Arity)
        , (portray(Term1) :- write(F/Arity))
        , (touch(Touched,Com1,Final):- Toucher)
        , (gets(Helper,Term1,Ids)   :- Term1)
        , (sets(Helper,Term2,Ids)   :- bretract(Term1),
                                       bassert(Term2) )
        , (makes(Helper,Term1,Ids)  :- bassert(Term1))
        , (zaps(Helper,Term1,Ids)   :- bretract(Term1))
        , (goal_expansion(H3,Body)  :-
               singleton(H3), % ◄.............................. 13
               clause(H3,Body))
        ,  goal_expansion(H4,H5a)
        ,  (goal_expansion(H5,true) :- one(H5)) % ◄.......... 16
        ,  (H1 :- H3)
        ,  Self
        | Rest
        ]) :-
    Spec   =.. [F|Fields],
    length(Fields,Arity),
    makeIndex(Fields,1,Indicies,Ids,Args,Names),
    H1    =.. [Helper,Com],
    H3    =.. [Helper,Com,_,_],
  %  H3a  =.. [Helper,Com,Initial,Final],
    H4    =.. [Helper,Field,Old    ,In,In],
    H5a   =.. [Helper,Field,Old,Old,In,In],
    H5    =.. [Helper,_,_,_,_,_],
    functor(Touched,F,Arity),
    Toucher=..[Helper,Com1,Touched,Final],
    Index =.. [F|Indicies],
    Term1 =.. [F|Args],
    Self  =.. [Helper,self,In,Out,In,Out],
    copy_term(Term1/Ids,Term2/Ids),
    findall(One,spec1(Helper,Names,F,Arity,One),Rest).

spec1(Helper,Names,F,Arity,One) :-
    nth1(Pos,Names,Item),
    joinArgs(F,Arity,Pos,Old,New,T1,T2),
    One =.. [Helper,Item,Old,New,T1,T2].

joinArgs(F,Arity,Pos,Old,New,Term1,Term2) :-
    length(L1,Arity),
    Pos0 is Pos - 1,
    length(Before,Pos0),
    append(Before,[Old|After],L1),
    append(Before,[New|After],L2),
    Term1 =.. [F|L1],
    Term2 =.. [F|L2].

makeIndex([],_,[],[],[],[]).
makeIndex([+H|L],N,[1|Pos],[Arg|Ids],[Arg|Args],[H|T]):-
    N1 is N + 1,
    makeIndex(L,N1,Pos,Ids,Args,T).
makeIndex([H|L],N,[0|Pos],Ids,[_|Args],[H|T]) :-
    atomic(H),
    N1 is N + 1,
    makeIndex(L,N1,Pos,Ids,Args,T).

blank(H,B,Id) :-  ':spec'(H,_,B,Id,_,_).
```

Figure 21. See Figure 20 for sample usage.

6 Conclusion

Pure rule-based programming (RBP) had many proponents (such as the first author) in the early days of knowledge engineering. These proponents became fewer in number as developers found themselves forced to extend RBP. Such extensions are easy to implement. Consider Figure 23 and Figure 24. These two files are all that is required to convert a Prolog-based OO system written into a RBP system. These files are very short and are our evidence that extending other systems into RBP is very simple. We therefore caution against an over-devotion to pure rule-based programming. The spirit of rule-based systems can be captured and usefully extended very simply in other programming paradigms.

Acknowledgements

This research was conducted at West Virginia University under NASA contract NCC2-0979. The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

speceg.out

```
% output from speceg.pl

grocery(A) :-
        grocery(A, B, C).

grocery(type(A), groceryDB(B, C, D, E, F),
                 groceryDB(B, C, D, E, F)) :-
        functor(D, A, G).   %◀.................................... 8
grocery(volumes([small/1, medium/2, large/3]), A, A).
grocery(volume(A), groceryDB(B, C, D, E, F),
                   groceryDB(B, C, D, E, F)) :-
        E=G,
        [small/1, medium/2, large/3]has G/A.

grocery(self, A, B, A, B).  %◀.................................. 15
grocery(id,    A, B,  groceryDB(A, C, D, E, F),
                      groceryDB(B, C, D, E, F)).
grocery(name,  A, B,  groceryDB(C, A, D, E, F),
                      groceryDB(C, B, D, E, F)).
grocery(type,  A, B,  groceryDB(C, D, A, E, F),
                      groceryDB(C, D, B, E, F)).
grocery(size,  A, B,  groceryDB(C, D, E, A, F),
                      groceryDB(C, D, E, B, F)).
grocery(frozen, A, B, groceryDB(C, D, E, F, A),
                      groceryDB(C, D, E, F, B)).

':spec'(grocery, groceryDB, groceryDB/5, [],
    [0, 0, 0, 0, 0], [id, name, type, size, frozen]).

:-index(groceryDB/5).
:-dynamic groceryDB/5.

portray(groceryDB(A, B, C, D, E)) :-
        write(groceryDB/5).

touch(groceryDB(A, B, C, D, E), F, G) :-
        grocery(F, groceryDB(A, B, C, D, E), G).

gets(grocery, groceryDB(A, B, C, D, E), []) :-
        groceryDB(A, B, C, D, E).

sets(grocery, groceryDB(A, B, C, D, E), []) :- %◀............. 42
        bretract(groceryDB(F, G, H, I, J)),
        bassert(groceryDB(A, B, C, D, E)).

makes(grocery, groceryDB(A, B, C, D, E), []) :- %◀............ 46
        bassert(groceryDB(A, B, C, D, E)).

zaps(grocery, groceryDB(A, B, C, D, E), []) :-
        bretract(groceryDB(A, B, C, D, E)).

goal_expansion(grocery(A, B, C), D) :-
        singleton(grocery(A, B, C)),
        clause(grocery(A, B, C), D).
goal_expansion(grocery(B, C, D, D),
              grocery(B, C, C, D, D)).
goal_expansion(grocery(A, B, C, D, E), true) :-
        one(grocery(A, B, C, D, E)).
```

Figure 22. Output from Figure 20.

rules.pl

```
r=rule(group,id,wme,priority).

rules(In,Out) :- once(r(prep(In,Out),_,_)).

r*init --> !id=0,!group=global,!priority=10.

r*head(Id in G at P)--> !id = Id, !group=G, !priority=P.
r*head(Id at P in G)--> !id = Id, !group=G, !priority=P.
r*head(Id      in G)--> !id = Id, !group=G.
r*head(Id      at P)--> !id = Id, !priority=P.
r*head(Id          )--> !id=Id.

r*prep(X if Y0 then Z0 since Why0,
              [(lhs(G,P,Id,Mem) :- % ◀.......................... 14
                    Y)
              ,(rhs1(G,Id,Mem) :-  % ◀.......................... 16
                    Z,
                    say(G,Id,Why))
              ]) --> !,
    nl,print(X), write('? '),
    * init,!,
    call(ecgHack(Y0,Y)),
    call(ecgHack(Z0,Z)),
    call(c2l(Why0,Why)),
    * head(X),
    `group=G,
    `id = Id,
    `priority=P,
    call(sharedVars(Y,Z,Mem)),     % ◀.......................... 29
    write(' YES!').

r*prep(X if Y0 then Z0,Out) -->
    *prep(X if Y0 then Z0 since [],Out).

ecgHack(X0,X) :-
    ecg1(X0,_,X1,_,_),
    expand_term(X1,X2),
    tidy(X2,X).

rshow(Group,Id) :-
    clause(lhs(Group,P,Id,Mem),LHS),
    clause(rhs1(Group,Id,Mem),RHS),
    portray_clause((lhs(Group,P,Id,Mem) :- LHS)),
    portray_clause((rhs1(Group,Id,Mem) :- RHS)).
```

Figure 23.

7 References

[1] J. Bachant and J. McDermott. R1 Revisited: Four Years in the Trenches. *AI Magazine*, pages 21–32, Fall 1984.

[2] D. Bobrow. If prolog is the answer, what is the question? or what it takes to support ai programming paradigms. *IEEE Transactions on Software Engineering*, 11(11):1401–1408, November 1985.

[3] L. Brownston, R. Farell, E. Kant, and N. martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, 1985.

[4] B. Buchanan and R. Smith. Fundamentals of Expert Systems. In P. C. A. Barr and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence, Volume 4*, volume 4, pages 149–192. Addison-Wesley, 1989.

[5] I. Chen and T. Tsao. A reliability model for real-time rule-based expert systems. *IEEE Transactions on Reliability*, pages 54–62, March 1995.

[6] P. Compton, G. Edwards, A. Srinivasan, P. Malor, P. Preston, B. Kang, and L. Lazarus. Ripple-down-rules: Turning knowledge acquisition into knowledge maintenance. *Artificial Intelligence in Medicine*, 4:47–59, 1992.

[7] A. V. de Brug, J. Bachant, and J. McDermott. The Taming of R1. *IEEE Expert*, pages 33–39, Fall 1986.

[8] P. S. Laird, R. J. E., and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[9] J. Larkin, J. McDermott, D. Simon, and H. Simon. Expert and novice performance in solving physics problems. *Science*, 208:1335–1342, 20 June 1980.

[10] M. Linster and M. Musen. Use of KADS to Create a Conceptual Model of the ONCOCIN task. *Knowledge Acquisition*, 4:55–88, 1 1992.

[11] D. Lukose, S. Nechab, S. Pritchard, A. Lee, S. Hussen, J. Clawley, P. Jackson, C. Hare, T. Bayliss, M. Hawcutts, and A. Bdar. Taps: Knowledge management system. In *Proceedings of the Banff Knowledge Acquisition Workshop*, 1999. Available from `http://sern.ucalgary.ca/KSI/KAW/KAW99/papers/Lukose1/`.

[12] S. Marcus and J. McDermott. SALT: A Knowledge Acquisition Lan-

fchain.pl
```
refraction=alreadyUsed(+group,+id,mem).
refraction*active.

fchain :-
    no(silent),
    reset(X),
    run(X).

reset(Info) :-
    bagof(G/Ps,priorities(G,Ps),Info),
    goto global,
    forall(active(A),retractall(A)).

active(A) :-
     blank(_,A,_), touch(A,active,_).

groups(All) :-
    setof(One,A^B^C^D^clause(lhs(One,A,B,C),D),All).

priorities(Group,All) :- % ◀................................... 20
    groups(Groups),
    member(Group,Groups),
    setof(One,
          Group^B^C^D^clause(lhs(Group,One,B,C),D),All).

run(Info)  :- step(Info),!, run(Info). run(_).

step(Info) :-
     todo(Info,Group,Priority),  % ◀........................... 29
     lhs(Group,Priority,Id,Mem), % ◀........................... 30
     not alreadyUsed(Group,Id,Mem),
     assert(alreadyUsed(Group,Id,Mem)),
     rhs(Group,Id,Mem).          % ◀........................... 33

todo(Info,Group,Priority) :-    % ◀............................ 35
     group(Group),              % ◀............................ 36
     member(Group/Orders,Info),
     member(Priority,Orders).

rhs(Group,Id,Mem) :- rhs1(Group,Id,Mem),!.
rhs(Group,Id,_) :-
   format('% ?? failed rule action ~w in ~w',[Id,Group]),
   nl.
```

Figure 24.

egs.pl
```
:- [speceg].
:- [ecgeg].
:- [ruleseg].
```

Figure 25.

show.pl
```
show(X) :- show(X,_).

show([],_).
show(List,X) :-
    member(X,List),
    show2(X),fail;true.

show2((X :- Y)) :- !,portray_clause((X:-Y)).
show2(X)        :- numbervars(X,1,_), print(X),
                   write('.'), nl.
```

Figure 26. A simple pretty print.

guage for Propose-and-Revise Systems. *Artificial Intelligence*, 39:1–37, 1 1989.

[13] J. McDermott. R1 ("xcon") at age 12: lessons from an elementary school achiever. *Artificial Intelligence*, 59:241–247, 1993.

[14] T. Menzies, J. Black, J. Fleming, and M. Dean. An expert system for raising pigs. In *The first Conference on Practical Applications of Prolog*, 1992. Available from `http://tim.menzies.com/pdf/ukapril92.pdf`.

[15] T. Menzies and B. Markey. A micro-computer, rule-based prolog expert-system for process control in a petrochemical plant. In *Proceedings of the Third Australian Conference on Expert Systems, May 13-15*, 1987.

[16] T. Menzies. OO patterns: Lessons from expert systems. *Software Practice & Experience*, 27(12):1457–1478, December 1997. Available from `http://tim.menzies.com/pdf/97patern.pdf`.

[17] T. Menzies. Knowledge elicitation: the state of the art. In S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering, Volume II*. World-Scientific, 2002. Available from `http://tim.menzies.com/pdf/00getknow.pdf`.

[18] NASA. CLIPS Reference Manual. Software Technology Branch, lyndon B. Johnson Space Center, 1991.

[19] P. Preston, G. Edwards, and P. Compton. A 1600 Rule Expert System Without Knowledge Engineers. In J. Leibowitz, editor, *Second World Congress on Expert Systems*, 1993.

[20] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, The Royal Institute of Technology (KTH), Stockholm, Sweden, May 1991. Available from `file://sics.se/pub/isl/papers/dan-sahlin-thesis.ps.gz`.

[21] A. T. Schreiber, B. Wielinga, J. M. Akkermans, W. V. D. Velde, and R. de Hoog. Commonkads. a comprehensive methodology for kbs development. *IEEE Expert*, 9(6):28–37, 1994.

[22] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[23] M. Stefik, J. Aikins, R. Balzer, J. Benoit, L. Birnhaum, F. Hayes-Roth, and E. Sacerdoti. The Organisation of Expert Systems, A Tutorial. *Artificial Intelligence*, 18:135–127, 1982.

[24] P. Winston. *Artificial Intelligence*. Addison-Wesley, 1984.

[25] G. Yost. Acquiring knowledge in soar. *IEEE Expert*, pages 26–34, June 1993.

Appendix: PIKE support routines

Figure 25 is a file which, if loaded, will exercise most of STARLOG's PIKE.

Figure 26 is a tool for printing lists of clauses.

Figure 27 defines some of the verbs used in rules.

Figure 28 contains certain Prolog hacks such as repair of overzealous DCG expansion.

Figure 29 is a tool for running a goal and trapping its output to a file.

Figure 30 is the first file loaded, it sets certain global flags.

Figure 31 contains miscellaneous code.

Figure 32 is the main load file of STARLOG/PIKE.

verbs.pl

```
goto Group :- retractall(group(_)),
              assert(group(Group)).

List has Item :- member(Item,List).

say(_,_,_) :-  option(silent),!.
say(_,_,[]) :- !.
say(Group,Id,Words) :- !,
    format('[~w::~w] ',[Group,Id]),nl,
    write('   '),
    forall(member(One,Words),write(One)),
    nl.
```

Figure 27.

hacks.pl

```
goal_expansion(append(A,B,C,D,D), append(A,B,C)).
goal_expansion(once(A,B,B),            once(A)).
goal_expansion(=..(A,B,C,C),          =..(A,B)).
goal_expansion(=(A,B,C,C),              =(A,B)).
goal_expansion(call(A,B,B),                 A).
goal_expansion(noop(_,_,_),              true).

prolog_listing:put_tabs(N) :-
    N > 0, !,
    write('   '),
    NN is N - 1,
    prolog_listing:put_tabs(NN).
prolog_listing:put_tabs(_).
```

Figure 28.

demos.pl

```
demos(G) :-
    sformat(Out,'~w.out',G),
    (exists_file(Out) -> delete_file(Out) ; true),
    write(Out),nl,nl,
    tell(Out),
    format('% output from ~w.pl',G),nl, nl,
    T1 is cputime,
    ignore(forall(G,true)),
    T2 is (cputime - T1),
    nl,format('% runtime = ~w sec(s)',[T2]),nl,
    told,
    format('% output from ~w.pl',G),nl,
    ignore(forall(G,true)),
    nl,format('% runtime = ~w sec(s)',[T2]).
```

Figure 29. Run a goal, trap its output to file and, also, show it on the screen.

flags.pl

```
:- Stuff=(gets/3, sets/3, makes/3, zaps/3,
          ':spec'/6,
          lhs/4, rhs1/3,touch/3),
   multifile(Stuff),
   discontiguous(Stuff),
   dynamic(Stuff).

:- index(gets( 1,1,0)).
:- index(sets( 1,1,0)).
:- index(makes(1,1,0)).
:- index(zaps( 1,1,0)).
:- index(lhs(1,1,1,0)).
:- index(rhs1(1,  1,0)).

:- dynamic group/1,
          option/1.

yes(X) :- option(X) -> true; assert(option(X)).
no(X) :- retractall(option(X)).

:- yes(brave).      % compile time evaluation
:- no(loadSlowly). % never skip unchanged files on load
:- no(silent).      % don't suppress rule 'since' text
:- yes(nervous).    % check that fields, methods exist
:- yes(unfold).     % replace sub-goals by true
```

Figure 30.

misc.pl

```
l2c([X,Y|Z],(X,Rest)) :- !, l2c([Y|Z],Rest).
l2c([X],X).

c2l(X,[X]) :- var(X),!.
 c2l((X and Y),[X|Rest]) :- !, c2l(Y,Rest).
c2l((X,Y),[X|Rest]) :- !, c2l(Y,Rest).
c2l(X,[X]).


term2list(Term0, L) :-
    Term0 =..L0,
    once(maplist(term2list1, L0, L)).

term2list1(H,H)  :- var(H).
term2list1(H,H)  :- atomic(H).
term2list1(H0,H) :- term2list(H0,H).

ensure(X) :- X,!.
ensure(X) :- bassert(X).

bassert(C) :- asserta(C).
bassert(C) :- retract(C),!, fail.

bretract(C) :- retract(C) , bretract1(C).

bretract1(_).
bretract1(C) :- asserta(C), fail.

chars(F) :- see(F), ignore(chars1(10)), seen.

chars1(-1) :- !.
chars1(X)  :- put(X), get_byte(Y), chars1(Y).
```

Figure 31.

starlog.pl

```
:- write('% *(star)log v[1.0]'),nl.
:- [flags].  % see Figure 30

:- op(1,fx,@).

@ X :- (option(loadSlowly)
        -> Options= []
        ;  Options=[silent(true), if(changed)]),
       load_files(X,Options).


:- @[%%% standard start up files
     ops        % see Figure 12
    ,hooks      % see Figure 13
    ,hacks      % see Figure 28

     %%% some general library routines
    ,show       % see Figure 26
    ,tidy       % see Figure 18
    ,demos      % see Figure 29
    ,singleton  % see Figure 17
    ,misc       % see Figure 31
    ,sharedvars % see Figure 14

    %%% code specific to rules and objects in Prolog
    % the object system:
    ,spec       % see Figure 21,  & Figure 22
    ,wrapper    % see Figure 16
    ,ecg        % see Figure 19
    ,verbs      % see Figure 27
    % the rule system:
    ,rules      % see Figure 23, Figure 7 & Figure 11
    ,fchain     % see Figure 24
    %,egs       % see Figure 25 (uncomment to see demos)
    ,aboutme    % see Figure ??
    ,license    % see §??
    ].

:- hello.
```

Figure 32. The idiom `@[File1, File2,..]` is shorthand for "don't load these files more that once unless they have not changed on disc and, if loading, don't print verbose load messages".