**3rd ACM SIGPLAN Workshop on Rule-Based Programming (submitted); October 5, 2002 manuscript No.**
(will be inserted by the editor)

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

**Tim Menzies**[1], **Lindsay Mason**[2]

[1] Lane Department of Computer Science, University of West Virginia, PO Box 6109, Morgantown, WV, 26506-6109, USA; `http://tim.menzies.com`; e-mail: `tim@menzies.com`
[2] Electrical & Computer Engineering, University of British Columbia, Canada; e-mail: `lmason@interchange.ubc.ca`

WP ref: 02/starlog/starlog June 5, 2002

**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 supports a range of knowledge engineering applications.

## 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 update 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 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. [16, 17];
- 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 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" development 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. This rule base will, most probably, be incomplete. However, 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 that 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 and such a rich library of knowledge will be intricate to maintain, no matter how it is expressed. However, another factor complicated XCON's maintenance was that it's 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 or elaborated, 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 can't be understood in isolation with the others. 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 bad 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.

Such coordinating rules violate 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. For example, the process of finding the volumes of different items in a grocery bag. One *generator rule* is required of

[1] From Inference Corporation

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

[3] Pun. Function: noun. Etymology: perhaps from Italian puntiglio fine point, 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. Sorry.

*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
        not (bag=B with 'contents has C and
             grocery with 'name=C with *type(bottle))
then    change order=I with delete('items,N,!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 delete('items,N,!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'.
```

**Fig. 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 [23].

*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).

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

% define  GROCERY types
grocery*type(T)   --> functor('type,T,_). % ◀..... 12

% 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.
```

**Fig. 2** A PIKE definition of the GROCERY object.

*order.pl*

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

**Fig. 3** A PIKE definition of the ORDER object.

*bag.pl*

```
bag=bagDB(+id,contents).

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) -->              %◀... 25
    grocery with 'name=Item with *volume(V0),
    *volume(Items,V).
bag*volume([],0).                              %◀... 28

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

bag*full --> * volume(V), *capacity(S), V >= S.    %◀ 33

bag*notFull --> not (* full).
```

**Fig. 4** A PIKE definition of the BAG object.

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". 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 is the list summation procedure shown in Figure 4 between line 25 and line 28.

Many other researchers argued that rules were not the appropriate primitive construct of knowledge engineering. Despite careful attempts to generalize the early knowledge engineering work (e.g. [22]), 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, functional, object-oriented, and "access-based" (which, these days, we might call implicit invocation [21]). 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 [20] or a recursive descent of "problem spaces" (e.g. as done by [24]).

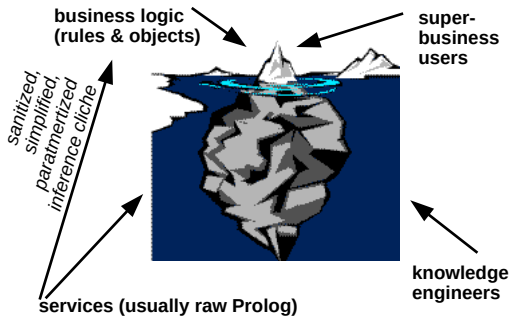


**Fig. 5** The "iceberg model" of knowledge engineering.

### *2.3 Beyond RBP*

The above problems, and our own commercial knowledge engineering (e.g. [16, 17]), lead us to extent RBP. Like many others (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* in Figure 1 checks that a bag is *notFull* and *notFull* is a procedural defined at the end of *bag* in 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-based 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 [14, 15].

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.
- Craft support code for each such cliche.

Where possible, the support code was heavily parameterized so that it could be extensively customized. These customization parameters then became the "tip of the iceberg" that was in view to our business users. These users then used these upper-most drivers in their rules and objects. Our cliches in-

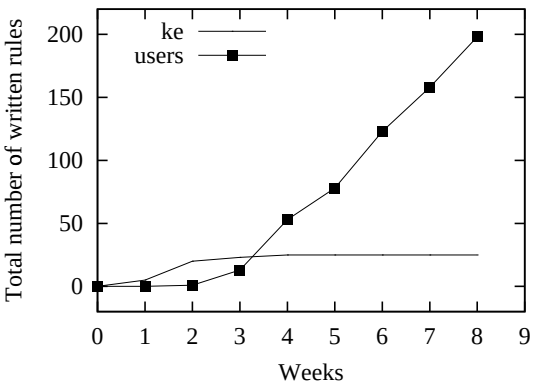


**Fig. 6** Patterns of rule growth. KE= knowledge engineers

cluded low-level idioms such as summing all items is 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 from the following development methodology:

*Language development:* Initially, the knowledge engineer(s) 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.

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

*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 , [‘,!,*]).
```

**Fig. 7**

*starlog.pl*

```
:- [flags].  % see Figure 23

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


:- @[%%% standard start up files
     ops        % see Figure 7
    ,hooks      % see Figure 9
    ,hacks      % see Figure 27

     %%% some general library routines
    ,show       % see Figure 29
    ,tidy       % see Figure 18
    ,demos      % see Figure 30
    ,singleton  % see Figure 28
    ,misc       % see Figure 31
    ,sharedvars % see Figure 17

    %%% code specific to rules and objects in Prolog
    % the object system:
    ,spec       % see Figure 12,  & Figure 10
    ,wrapper    % see Figure 24
    ,ecg        % see Figure 13
    ,verbs      % see Figure 25
    % the rule system:
    ,rules      % see Figure 15, Figure 19 & Figure 22
    ,fchain     % see Figure 16
    %,egs       % see Figure 26 (uncomment to see demos)
    ].
```

**Fig. 8** 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”.

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 due to its ease of customization. For example, Figure 7 changes the Prolog interpreter such that certain user-friendly structures can be created in a pseudo-English style; e.g. the rules shown in Figure 1.

This paper presents 367 lines of Prolog that implements a simple, but useful rule/object interpreter/optimizer. The main load file of STARLOG is shown in Figure 8. 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

*hooks.pl*

```
% OBJECTS:
term_expansion(Helper=Spec, X) :-
    spec(Helper=Spec,X).

% METHODS:
term_expansion((Helper*Head --> Body),  X) :-
    ecg((Helper*Head --> Body),X).
term_expansion(Helper*Head, X) :-
    ecg((Helper*Head --> true),X).

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

**Fig. 9**

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 in Figure 9: objects, methods, and rules. These constructs are discussed below Due to space restrictions, this discussion will be quite terse. A longer version of this article is under preparation which explains the system in more detail.

### *3.1 Objects*

Knowledge base authors (hereafter, authors) can define objects using the idiom `Helper=Spec` where `Spec` describes the structure of a working memory object. Each PIKE object contains a set of named fields, some of which are marked by a `+` if they are index fields. GROCERY items have no indexes (see Figure 2) but ORDERs and BAGs have one indexed field each (see Figure 3 and Figure 4). `Helper` is the name of a predicate that can access parts of an object. For example, the Figure 10 shows the internal Prolog representation of Figure 4. For example, the `grocery/5` predicates starting at line 14 allow the access and update of fields within a GROCERY object.

The `Helper=Spec` idiom are handled by the `spec/2` predicate defined in Figure 12. Currently, PIKE's objects support encapsulation and polymorphism, but not inheritance.

### *3.2 Methods*

The idiom `Helper*Head --> Body` is PIKE's method syntax. These methods are an extension to the standard Prolog definite clause grammar and so are called *ECG*s. ECGs can contain named references to objects. Within ECGs, the idiom `'X` accesses the current value of field `X` and the idiom `!X` updates field `X`. For example:

[4] For Star Trek aficionados, we offer the following notes. 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.

*speceg.out*

```
% output from speceg.pl

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

grocery(type(A), groceryDB(B,C,D,E,F),  % ◀..... 6
                 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)) :-
   [small/1, medium/2, large/3]has E/A.

grocery(self,   A, B, A, B). % ◀....................... 14
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(B, C, D, E, F)) :-
   write(groceryDB/5).

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

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

sets(grocery, groceryDB(A, B, C, D, E), []) :-  % ◀ 41
   bretract(groceryDB(B, C, D, E, F)),
   bassert(groceryDB(A, B, C, D, E)).

makes(grocery, groceryDB(B, C, D, E, F), []) :- % ◀ 45
   bassert(groceryDB(B, C, D, E, F)).

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

goal_expansion(grocery(A, B, C), D) :-  % ◀..... 51
   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)).
```

**Fig. 10** Output from Figure 11.

– Line 12 of Figure 2 has the code `functor('type,T,_-)`. STARLOG's load-time macros expand this into the access to the third field of `groceryDB/5` (i.e. the `type` field) shown in Line 8 of Figure 10.
– Line 8 of Figure 4 has the code `!id=Id`. This idiom forces an update of the first field of a `bagDB/2` fact to a value computed from the `flag/3` predicate[5]; i.e.:

```
bag(newBag(Id, Contents), _, bagDB(Id, Contents)) :-  flag(ids,Id,Id+1)
```

The `Helper*Head --> Body` idiom is handled by `ecg/2` defined in Figure 13.

[5] SWI Prolog's `flag/3` predicate maintains and updates numeric counters.

*speceg.pl*

```
:- [starlog]. % see Figure 8

:- [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).
```

**Fig. 11** Starlog sample; generates Figure 10.

*3.3 Forward Chaining Rules*

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 14 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

$$\texttt{lhs}(\texttt{Group},\texttt{Priority},\texttt{Id},\texttt{Memory}) : -\texttt{Condition}$$

and

$$\texttt{rhs1}(\texttt{Group},\texttt{Id},\texttt{Memory}) : -\texttt{Action}$$

(see line 14 in Figure 15 and line 16 in Figure 15). 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 processing 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 some current group in the `group/1` fact (see line 35 in Figure 16). 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 16). At runtime, rules are explored within a group in priority ordering from priority one to lower priorities (see line 28 in Figure 16 and line 34 in Figure 16).

*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 17 which is called at line 29 in Figure 15.

*Recency:* When PIKE asserts anything, it is asserted above all older assertions (e.g. see line 41 in Figure 10 and line 45 in Figure 10). Hence, rules will fire more on newer assertions than older assertions.

*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), % ◀ 12
                                        clause(H3,Body))
         ,  goal_expansion(H4,H5a)
         ,  (goal_expansion(H5,true) :- one(H5)) % ◀ 15
         ,  (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,_,_).
```

**Fig. 12** See Figure 11 for sample usage.

*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).
% ◀..................................................... 28

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),
                               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).
```

**Fig. 13**

*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 delete('items,N,!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 delete('items,N,!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'.
```

**Fig. 14**

`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` all 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 13 and line 28 in Figure 13

### *3.4 The* `With` *Statement*

Another important idiom within PIKE is

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

PIKE expands 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

## 4 Optimizations

PIKE also includes three optimization methods: unfolding and a unification of the match/select process:

*Unfolding:* If a clause sub-goal matches to a single other clause, then the sub-goal is replaced by the body of the other clause; see line 12 in Figure 12. Sometimes, this unfolding unwraps to just a `true` sub-goal; see line 15 in Figure 12. Such `true` sub-goals are redundant and are purged via the code of Figure 18.

*Unified match/select:* The order in which PIKE's rules are to be tested can be determined via the rule priority number. If rules are tested within each group in this order, then the *first* rule with a satisfied condition would be the highest priority satisfied rule. By exploring rules in this order,

*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)).
```

**Fig. 15**

*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), % ◀..... 28
     lhs(Group,Priority,Id,Mem),
     not alreadyUsed(Group,Id,Mem),
     assert(alreadyUsed(Group,Id,Mem)),
     rhs(Group,Id,Mem).

todo(Info,Group,Priority) :-    % ◀..... 34
     group(Group),              % ◀..... 35
     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.
```

**Fig. 16**

*sharedvars.pl*

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

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

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

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

**Fig. 17**

PIKE avoids a computationally expensive MATCH process.; see line 28 in Figure 16 and line 34 in Figure 16.

## 5 A Sample Application

This paper contains full source code for a PIKE rule/object system that implement's Patrick Winston's BAGGER problem [23]. 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.

PIKE's BAGGER is loaded in Figure 19. The system contains five rule groups:

`Global:` Creates a sample order; see Figure 20. The current group is then changed to...

`Check_order:` Checks if any items are missing from the order; see Figure 20. The current group is then changed to...

`Bag_large_items:` Handles the bagging of the bulky items; see Figure 14. The current group is then changed to...

`Bag_medium_items:` Handles the bagging of the mid-sized items; see Figure 21. 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 22 shows what happens when the whole system is loaded and executed. Given the ORDER

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

PIKE's BAGGER generates two bags:

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

*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), X is Y.
tidy1((A :- true),   A).
tidy1((A :- B),      R) :- tidy(B,TB),
                           (TB=true -> R=A; R=(A :- TB)).
tidy1((A,B),    (A,TB)) :- var(A), tidy(B,TB).
tidy1((A,B),    (TA,B)) :- var(B), tidy(A,TA).
tidy1(((A,B),C),     R) :- tidy((A,B,C), R).
tidy1((true,A),      R) :- tidy(A,R).
tidy1((A,true),      R) :- tidy(A,R).
tidy1((A,B),         R) :- tidy(A,TA), tidy(B,TB),
                           (TB=true -> R=TA ; R=(TA,TB)).
tidy1((A;B),   (TA;TB)) :- tidy(A,TA), tidy(B,TB).
tidy1((A->B), (TA->TB)) :- tidy(A,TA), tidy(B,TB).
tidy1(not(A),  not(TA)) :- tidy(A,TA).
tidy1(A,             A).
```

**Fig. 18** Remove redundant trues.

*ruleseg.pl*

```
:- [starlog].     % see Figure 8

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

:- [checkrules    % see Figure 20
   ,largeitems   % see Figure 14
   ,mediumitems  % see Figure 21
   ,smallitems   % see Figure 1
   ].

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

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

:- demos(ruleseg).
```

**Fig. 19**

*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'.
```

**Fig. 20**

*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 delete('items,N,!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'.
```

**Fig. 21**

*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 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, pepsi]).
```

**Fig. 22**

## 6 The Rest of STARLOG

The rest of STARLOG is contained in the following files:

- Figure 23 is the first file loaded which sets certain global flags.
- Figure 24 shows a sub-routine to Figure 13.
- Figure 25 defines some of the rule conditions and actions.

*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
:- yes(loadSlowly). % never skippled 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
```

**Fig. 23**

*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].
```

**Fig. 24**

– Figure 26 is a file which, if loaded, will exercise most of STARLOG's PIKE.
– Figure 27 contains certain Prolog hacks such as repair of over-zealous DCG expansion.
– Figure 28 tests if only one clause matches a sub-goal.
– Figure 29 is a tool for printing lists of clauses.
– Figure 30 is a tool for running a goal and trapping its output to a file.
– Figure 31 contains miscellaneous code.

*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.
```

**Fig. 25**

*egs.pl*

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

**Fig. 26**

*hacks.pl*

```
%:- yes(brave).

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(call(A),                    A).
goal_expansion(noop(_,_,_),              true).
%goal_expansion(X=X,                       true) :- option(brave).

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

**Fig. 27**

*singleton.pl*

```
singleton(X) :-
    bagof(X/Y,clause(X,Y),[_]).

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

**Fig. 28**

*show.pl*

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

show([],_).
show([H|T],X) :-
    once(show1(H,Y,S)),
    (X=Y -> true; nl),
    show2(S),
    show(T,Y).

show1((X :- true),F/A,     X) :- functor(X,F,A).
show1((X :- Y),   F/A,(X:-Y)) :- functor(X,F,A).
show1(X,          F/A,     X) :- functor(X,F,A).

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

**Fig. 29** A simple pretty print.

*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)*1000,
    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]).
```

**Fig. 30** Run a goal, trap it's output to file and, also, show it on the screen.

*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]).

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

**Fig. 31**

## 7 Conclusion

Pure rule-based programming had many proponents in the early days of knowledge engineering. These proponents became fewer in number as more and more developers were forced to extend pure rule-based programming.

We have argued here that such extensions are necessary and simple. The Prolog code shown here is very brief and implements a object/rule interpreter/optimizer. Such a small system is easily customized and supports our preferred toolkit for knowledge engineering: the encoding of domain-specific knowledge cliches.

## 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 Language 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. OO patterns: Lessons from expert systems. *Software Practice & Experience*, 27(12):1457–1478, December 1997. Available from `http://tim.menzies.com/pdf/97patern.pdf`.
15. 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`.
16. 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`.
17. 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.
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. 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.
21. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
22. 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.
23. P. Winston. *Artificial Intelligence*. Addison-Wesley, 1984.
24. G. Yost. Acquiring knowledge in soar. *IEEE Expert*, pages 26–34, June 1993.