FRulekit: A Frame-Based Production System

User's Manual

4 November 1993

Peter Shell
Jaime Carbonell

1. Introduction

FRulekit is an experimental production system written in Commonlisp. At its core is an augmented implementation of the OPS Rete pattern matcher.[1] As well as the standard OPS5 conflict resolution strategies of LEX and MEA and an extended strategy, it also allows users to write their own conflict strategies, and incorporates an agenda system for controlling the ordering of rules. It has a trace package which makes it possible to build rule-learning strategies, and supports reason maintenance.

FRulekit stands for Frame-Rulekit since working memory elements are frames, and matching against frames are supported. It uses the **Parmenides** frame system, so it is assumed that the reader is also familiar with this frame program. See the Parmenides manual (available as /afs/cs/user/pshell/parmenides/parmenides.ps) for documentation.

It is assumed that the reader is familiar with production systems. Since **FRulekit** goes beyond OPS-5 in many ways, the OPS-5 syntax is not used. Knowing lisp, especially Common Lisp, is useful but not essential.

2. Working Memory Elements

Working memory elements (WMEs) look vaguely like the OPS-5 vector representation. WMEs are an instance of one and only one class, and each class has a set of *attributes*, or slots. Classes and their slots are defined by **literalize**. This is where the similarity ends however. **Literalize** defines classes through the Parmenides **def-frame** function (see below), so that all WME classes are subclasses of the predefined **WME** class. This implies that slots may have *facets*, which are like slots of a slot and which have associated data (*fillers*); or slots may simply be directly associated with fillers. Fillers may be any CommonLisp type. This also implies that demons may be associated with WME slots and classes.

The syntax of a WME is:

```
[<class> <slot> <value> <slot> <value> ...]
```

Slot names are preceded by colons. For example, if **goal** was a class which had the slots **name**, **parent** and **action**, then the following would be a legal WME:

```
[goal :name get-cookie :parent get-food :action look-for-cookie]
```

WMEs are implemented as Parmenides frames, and should only be made or altered by a Rulekit command (such as **$make** or **$modify**), or the Parmenides **set-facet-demons** command (note that if a class which is a WME is modified through **set-facet** or **setf**, Rulekit will not see the change and the production system may become inconsistent). Working Memory is simply a set of Working Memory Elements, and is usually thought of as a declarative representation of knowledge.

FRulekit uses a *freelist* in order to more efficiently manage memory. Whenever a WME or frame is removed, it is put on the freelist, and when a new WME or frame is needed, if there is one of the appropriate type already on the freelist, then that one is recycled. The functions **$remove** and **$remove-keep**, described below, release WMEs to the freelist when *RECORD-LEVEL* is 0, which means that you shouldn't expect to be able to use the WME after calling them.

[1] The original Rete pattern matching algorithm and data structure was developed by Charles Forgy at CMU around 1975.

3. The Left Hand Side of a Rule

The left-hand-side (LHS) is a list of tests, or *Condition Elements*, which test something about the state of working memory elements. If all of the tests are true, then the rule *matches* and is a candidate for *firing* (see Conflict Resolution, below). There are two basic kinds of operations allowed on the LHS of a rule: *testing* operations and *binding* operations. In each LHS command, either something is bound to a **FRulekit** variable, or something about a **FRulekit** variable or WME is tested.

3.1. Matching Against Working Memory

Working Memory tests check to see if there are any WMEs in Working Memory which match a certain *pattern*, or *Condition Element* (abbreviated *CE*). Constants in the value field of LHS patterns do not need to be quoted, although constants in lisp CHECKS and BINDS (see below) do (unless of course if the lisp function in the CHECK doesn't evaluate its arguments).

Patterns have the same form as WME's, with the following extensions. First, in addition to having constants in the value field, variables may also be used. Variables are of the form =<varname>; for example, =x.[2]The first time a variable occurs in the LHS, it is bound to the given slot. The next time it occurs in the LHS, it means that the indicated slot must have the same value as the first slot did. For example, the following LHS checks to see if any two goals have the same action:

```
((goal :name =n1 :action =a)
 (goal :name =n2 :action =a))
```

This LHS would match every pair of goals whose actions were the same.

FRulekit will construct different tests and access functions in the LHS depending on whether a slot has facets. The default facet accessed in left hand side tests is **value** for slots with facets.

3.2. The ABSence test

Another extension is that one may check for the *absence* of WMEs from working memory. This is done by putting the symbol <ABS> before the pattern. Example:

```
(<ABS> (goal :name find-cookie))
```

These are also referred to as *negation tests*. The degree of sophistication of absence tests varies from prod. system to prod. system, so the exact capabilities of the current **FRulekit** absence tests will be described:

- In Rulekit, arbitrary lisp *checks* are allowed within absence tests.
- The label command inside the absence test refers to the WME being tested and, like all variables bound inside absence tests, only has the scope of the absence test.
- Nested absence testing is not allowed (i.e., an absence test within an absence test).
- Currently, only one condition element may appear in each absence test (any number of absence tests are allowed in a LHS however).
- Absence tests of nested CE's is not allowed since nested CE's are expanded into multiple CE's by the FRulekit parser (see *nested CE's*, below).

---

[2]Note: because of the special syntax of the '=' character, be sure to quote it with #\ if you use it somewhere other than in a Rulekit variable.

3.3. The <OR> test

In addition to testing for the absence of a pattern, one may test for the *disjunction* of a set of patterns. The semantics of the <OR> command is that if any one of the given set of Condition Elements is satisfied, then the <OR> clause is satisfied. Further, while an instantiation containing one disjunct is in the conflict set, **FRulekit** will not attempt to match other disjuncts of the same rule, thus gaining efficiency. The syntax of the <OR> command is:

```
(<OR> (<list-of-CE's>)
      (<list-of-CE's>) ...)
```

Each *<list-of-CE's>* is a list of one or more (possible <ABS>) CE's. Each disjunct may be thought of as a seperate production, in that the variables in the disjuncts cannot be referenced by other disjuncts.

Notice the difference between the <OR> statement, which allows what we call *external disjunction*, and *internal disjunction*. *External disjunction* allows one to find a WME which meets one of a number of WME specifications (CE's); *internal disjunction*, which is performed in **FRulekit** by a lisp **or** inside of a **CHECK**, allows a slot to take a number of different values, but the rest of the specification of the WME is the same.

The following limitations currently apply to the <OR> command; some will be lifted as we find efficient ways to ease them:

- Nested <OR>'s are not allowed.
- Each disjunct must contain the same number of positive CE's; they may contain any number of <ABS> CE's however. For example the following is not allowed because the first disjunct has 1 positive CE and the second has none:

  ```
  :LHS ((...)
     (<OR> ((boy :age 2))
        ((<ABS> (girl :age 1)))))
  ```
- The first CE in a LHS cannot be an <OR>.
- Multiple <OR> statements *are* allowed, but subject to the following restriction: they may not occur directly after each other. I.e., there must be a non-<OR> CE between two <OR> statements.

Example:

```
(RULE find-boy-or-girl
 :LHS
  ((person :age =a)
   (<OR> ((boy :wt =w1 (LABEL =bg)))
         ((girl :wt 4 (LABEL =bg))))
   )
 :RHS (
       (if ($disj 1) (format T "First disjunct is true~%")
           (format T "Second disjunct is true~%"))
       (format T "Found a boy/girl: ")
       (pp-wme =bg)))
```

The **$disj** function is explained in section, *General Commands*; see also **$disj-nums** in the same section.

3.4. Lisp Tests

Sometimes it is useful to call LISP directly from the LHS to make additional checks on the variables. A lisp *CHECK* is allowed to appear anywhere in a condition element where a slot would appear. The syntax is:

```
(CHECK <expression>).
```

<expression> is any lisp expression and should include Rulekit variables bound up to that point. For example, in the above test for two goals with the same action, there is nothing saying that the two goals can't be the same. If we want to eliminate this possibility, we could add the following CHECK:

```
(CHECK (not (equal =n1 =n2)))
```

this could also be written:

```
(CHECK (<> =n1 =n2))
```

to make it more reminiscent of the OPS-5 syntax. The user is free to call any lisp function, including user-defined functions. Lisp checks should always occur after the Working Memory tests which contain the variables that they mention. In this example the (<> =n1 =n2) should occur after the bindings of n1 and n2 have been established by the two working memory element tests. Thus, the entire LHS would be:

```
((goal :name =n1 :action =a)
 (goal :name =n2 :action =a (CHECK (<> =n1 =n2))))
```

Lisp checks may also be combined with <ABS> tests, and may call macros as well.

Extreme care must be taken with LHS LISP checks. They should have no side-effects, because it isn't obvious when they will be evaluated. If a lisp condition doesn't check anything about the working memory elements, but rather tests a user data structure, it is not guaranteed to always work. This is because the lisp checks are put into the internal Rulekit Rete net, and so are not tested until the working memory element that they are associated with changes. If a LISP test that doesn't refer to anything in working memory is desired, then the second kind of lisp test, the *EXTRA-TEST*, should be used.

3.5. The BIND commands

When specifying slots in a LHS pattern, Rulekit variables are implicitly bound to the value of the slot accessed. However, the user may bind variables to other values, through the *BIND* and *MBIND* LHS commands. The syntax is:

```
(BIND <variable> <expression>)
```

or:

```
(MBIND (<variable>*) <expression>),
```

where <expression> is the same as in the CHECK command, and each <variable> is a Rulekit variable which hasn't yet been bound to anything. In the case of MBIND, the given <expression> should return at least as many multiple values as there are variables in the variable list.

For example, if for some reason it is desired to bind the names of the two goals matched in the previous example to a Rulekit variable, then a BIND command could be added:

```
((goal :name =n1 :action =a)
 (goal :name =n2 :action =a (CHECK (<> =n1 =n2))
       (BIND =both (list =n1 =n2))))
```

This would result in the Rulekit variable **=both** being bound to the list containing the values of =n1 and =n2. This variable may subsequently be used just like other Rulekit variables, on both the LHS and RHS. The BIND commands are particularly useful when the <expression> is an expensive one. Rulekit usually only executes the expression once and stores the result for all subsequent accesses. Due to efficiency issues in the current implementation however, expressions in a BIND command will be re-evaluated if the BIND command occurs in the first condition element, or in an absence test. Up to 16 BIND variables are allowed per production.

3.6. Specifying Slots

In traditional OPS-style production systems, the slot to be matched against has to be specified by a constant symbol at rule-writing time. In **FRulekit**, a special slot-specification operator, called the *bracketed* syntax, allows the user to refer to slots not only by name, but also by an expression which is dependent on other variables in the LHS. Thus, names of slots may be computed by Lisp functions and/or by storing them as the values of slots in other WMEs. It also allows rules to access facets of slots besides the default *value* facet.

The full bracketed syntax is:

```
[<slot-expression> {<facet-expression>}].
```

<slot-expression> and <facet-expression> are the same as in **CHECK** and **BIND**, or may be constant slot or facet names. <facet-expression> is optional and defaults to :value if the slot is known to be faceted. If the slot specification is dynamic and the facet is not specified, then **FRulekit** will at run time retrieve either the slot or the **value** facet of the slot, depending on whether the slot is faceted.

The dynamic slot specification occurs in the left hand side where an ordinary (static) slot specification would occur. As with ordinary slot specifications, if the expression in the value position is either a constant or a previously-bound **FRulekit** variable, then a check will be made that the value of the given slot is equal to the given value; otherwise a BIND command will be generated to bind the given **FRulekit** variable to the value of the specified slot. If the computed slot doesn't appear in the frame, then the associated condition element won't match.

For example, the following two condition elements cause **FRulekit** to find the value of a slot named in the **cur-slot** field (assuming that the proper **literalizes** had been made):

```
((goal :name print-slot :cur-slot =slotname)
 (data [=slotname] =slotvalue))
```

Thus, if the value of the **cur-slot** slot was **a**, then **=slotvalue** would be bound to the value of the **a** slot in a **data** frame. If the variable **slotvalue** was previously bound in the production, then the second condition element shown here would not match unless the value of the **a** slot was equal to the previous value of **slotvalue**.

The bracketed slot syntax need not always dynamically specify slots and facets. For example, if the **a** slot had a facet named **time**, then to refer to this facet statically, bracketed syntax would be needed:

```
((goal :name print-slot :cur-slot a)
 (data [:a :time] =slotvalue))
```

Thus, the bracket syntax is used whenever dynamic specification of slots is needed, or when access to facets is needed. Dynamic slot access is powerful, but slower than static slot access, since **FRulekit** is forced to use a slower Parmenides function (**get-instance-facet** or **get-instance-slot**) in order to retrieve the correct value.

3.7. Labeling Condition Elements

Condition elements may be *labeled* (i.e., bound to a Rulekit variable) by using another special command, the LABEL command, within the condition element. For example, to label the first test above as cond1, one could write:

```
(goal (LABEL =cond1) :name =n1 :action =a)
```

The LABEL command may be placed anywhere after the class specification. Variables bound to condition elements are just like variables bound to slot values; thus the two major uses of labeling are:

- Removing and modifying WMEs. To remove or modify a WME on the RHS, it must be referred to. Label the condition element corresponding to the WME to do this. See the section on Right Hand Side Actions for more details.
- Nested working memory elements. Since slots may take any value, they can even take on another WME as a value! They may also be matched against in the LHS. This will be explained next.

3.8. Nested WMEs

In order to assign a WME to a slot of another WME, both the parent and the nested WME must be bound to a Rulekit variable (usually through the LABEL LHS command.) Then, the appropriate Right-hand-side action is executed which connects the two WMEs. Again, see the section on Right Hand Side Actions for more details.

To specify on the LHS that the value of a slot be a certain WME, simply write the condition element matching that type of WME in the value position of the outer condition element. To match against a woman who likes a girl who likes a dog, for example, one would write:

```
(woman :likes (girl :likes dog))
```

Nested WME matching automatically LABELS the nested WME with the same name as the class of the WME (or a unique label name if there is already a label with that name in the rule), unless there is a label statement in the nested CE. For example, the above CE would be the same as writing,

```
(woman :likes =girl)
(girl (LABEL =girl) :likes dog)
```

Thus, it binds the variable girl to the WME that is nested in the woman WME. If the original CE had been,

```
(woman :likes (girl (LABEL =g1) :likes dog))
```

then it would be equivalent to writing,

```
(woman :likes =g1) (girl (LABEL =g1) :likes dog)
```

Note that since nested WME matching involves matching more than one WME, it can not be put in an absence test. See Appendix III for examples of creating and matching nested WMEs.

3.9. The =! Syntax

The =! operator is like =x except it matches any frame which *isa-instance* of x in the Parmenides semantic net, and binds that frame to =x. For example, =!person in a slot description means:

```
=person (CHECK (isa-instance =person ’person))
```

isa-instance is a Parmenides function which returns T iff the first argument is an instance of the second argument (or one of its subclasses in the is-a hierarchy).

=!x may also be used in the class description (i.e., where the class name usually goes in a condition element). When it appears there, it means to match against the class **X**, or any class under **X** in the is-a hierarchy. For example, if boy is-a person, then the test:

```
(=!person ...)
```

would match boys as well as people. Note that this can be used to match any wme, by simply saying, =!WME. However, it is inefficient since it forces **FRulekit** to test each WME whenever it is created or removed. =! in the class description position doesn’t bind any variables. The =! syntax may also be used in nested Ce’s.

3.10. The =< Syntax

=< is similar to =! except it uses *isa-or-instance* instead of *isa-instance*. Thus it is useful when the value of a slot is either a class or an instance. It has no meaning when in the class description position.

Reader-macro note: the =x syntax is implemented by a read macro which converts =x to (frk::var x), =!x gets read as (frk::any x), and =<x gets read as (frk::anysub x).

4. Extra Lisp Test

The Extra Lisp Test is so called because it tests about something in addition to working memory. The Extra Lisp Test makes tests about data structures independent of the Working Memory, and can't refer to LHS variables. Also, whereas Lisp Checks are only evaluated when the associated WME is put into working memory, Extra Lisp Tests are evaluated every time the rule is considered in the Agenda. (As an efficiency measure, Extra Tests are actually not evaluated until it is also found that the rule is in the conflict set.) Since Extra Lisp Tests are only evaluated when they are up for consideration in the Agenda, it doesn't make sense to have Extra Lisp Tests when the Conflict Resolution Strategy is not AGENDA.

Extra Lisp Tests go under the :EXTRA-TEST slot of rules. Since extra-lisp-tests must be evaluated on every cycle when the associated rule is being considered by the Agenda module, they are not very efficient and so should be used sparingly.

5. Right Hand Side Actions

When a rule fires, it executes its right-hand-side (RHS) actions with the variable bindings being given by the WMEs that it matched. The RHS is essentially a progn, i.e., a list of forms to be evaluated by LISP. Any lisp function may be called in the RHS, but the most important ones to call are:

5.1. $MAKE

This is the main way to make a new working memory element. Its syntax is: ($MAKE <pattern>), where all the variables mentioned in the pattern have been bound by the LHS of the production. It evaluates all of its arguments, so be sure to quote atoms. For example, to make a subgoal of the goal find-cookie, one would write:

```
($make 'goal :name 'find-cookie-jar :parent 'find-cookie)
```

For an example with variables, suppose we wanted to note the fact that there were two goals with the same action by making a new goal with the name *same-action*. If the LHS of our production was the goal tests above, then the RHS could be:

```
($make 'goal :name 'same-action :action =a)
```

Note that if the **name** slot of the goal WME had facets, then the above $make would be:

```
($make 'goal :name '(value same-action) :action (list 'value =a))
```

5.2. $REMOVE

Remove deletes the given WMEs from Working Memory, and if any production had previously matched but doesn't without the WME, then it will take that production out of the set of candidates (the *conflict set*). Remove usually takes the *labels* of the WMEs that matched in the LHS of the rule. WME labels are established by the *label* tag in condition elements. For example, to delete the two goals which had the same action in the above LHS, you could rewrite the condition elements to label them as =c1 and =c2:

```
  ((goal :name =n1 :action =a (LABEL =c1))
```

```
(goal :name =n2 :action =a (LABEL =c2) (CHECK (<> =n1 =n2))))
```

Next $remove would refer to labels =c1 and =c2:

```
($remove =c1 =c2)
```

$Remove also tells Parmenides to delete the frame representing the WME. See also **$remove-keep**, in the Command Summary, which doesn't delete the frame.

5.3. $MODIFY

This is actually a combination of $REMOVE and $MAKE. The syntax is ($MODIFY <wme-label> slots>), where <wme-label> is the same as it is in $REMOVE, and <slots> are those used in the <pattern> of $MAKE. For example, to change the action field of a goal, one would write,

```
($MODIFY =c1 :action 'drink)
```

The fields not specified in the $MODIFY are left unchanged. Doesn't fire any demons associated with the frame; **$modify-demons**, in the Command Summary, does this. See also **$modify-in-place** in the Command Summary.

6. Conflict Resolution

On any given cycle, there may be more than one candidate productions for firing. For example, if there were two pairs of goals which had the same action, then there would be two possible firings of that production: one for each of the pairs. (These different firings of the same production are called *instantiations*.) Before any production is fired, all candidates are put into a set called the *conflict set*. Conflict resolution is a procedure for choosing from the conflict set which instantiation(s) should fire.

Rulekit currently provides three conflict resolution strategies, the OPS LEX and MEA methods, and a new one called FULL-CR. The Agenda mechanism may be thought of as a conflict resolution strategy but is given its own section because it's not implemented that way. **FRulekit** also allows the user to tailor the conflict resolution strategies at various levels of detail.

6.1. LEX

A brief summary of the OPS5 LEX strategy is presented here. For more detail see the OPS5 User's Manual by Lanny Forgy. The **FRulekit** version of LEX has an added step, step 4, as well as a fix to include the intra-element tests in step 3.

1. Refract by deleting instantiations that have already fired.
2. **Narrow-down-lex**: Select the production whose condition elements are most recent.
3. **Num-tests**: In the case of a tie, select the production whose LHS conditions have the greatest number of tests. A test here is defined is a test to see if the value of a slot is some constant, or to see if a variable is consistently bound (including intra-element tests and label tests).
4. **Num-checks**: In the case of a tie, select the production whose conditions have the greatest number of variables involved in lisp checks.
5. If there is still a tie, choose one arbitrarily.

6.2. MEA

MEA, or means-ends analysis, is the same as **LEX**, with the **narrow-down-mea** step added after refraction:

Narrow-down-mea: Compare the recencies of the working-memory elements matching the first condition elements of the instantiation. The instantiations using the most recent WMEs dominate.

6.3. FULL-CR

FULL-CR is the same as MEA, with the following additional steps at the end: **Narrow-down-wme-class** and **Narrown-down-test-class**. The first compares hierarchical positions of the the classes of the matched WMEs. A WME whose class is *is-a* another, is more specific than that one. For example, the dog class is more specific than the mammal class. The second compares hierarchical positions of the the classes of the rules' tests.

6.4. User-defined C.R. Strategies

There are many ways to tailor the conflict-resolution behavior of **FRulekit**, depending on what level of control is desired. The usual way is to set the global switch *CR-STRATEGY* to one of the pre-defined lisp variables, *MEA*, *LEX* or *FULL-CR*.

These three lisp variables are bound to an ordered list of filters which are names of functions which act on successive lists of instantiations. For example, *LEX* is bound to: (NARROW-DOWN-LEX NUM-TESTS NUM-CHECKS), which correspond to the 2nd through 5th process in section 6.1 (refraction is controlled by the flag *REFRACT* which is by default on). The **FRulekit** conflict-resolver calls the first filter on the entire conflict set, and calls each of the remaining ones on the result of the previous filter. It stops when there is only one remaining candidate. If there are no more filters left and there is more than one instantiation, then it arbitrarily chooses one of the candidates. In order to change the order of these filters, simply set *CR-STRATEGY* to a different permutation of them. Alternatively, you can write your own filter and put the name of it in the filter list.

The usual way to define one's own filter is through the two filtering-defining macros, which is how all of the current **FRulekit** filters are defined. The first one, **Def-cr-strategy**, defines filters which compare only one attribute of the instantiations. **Narrow-down-mea**, **num-tests** and **num-checks** are defined with this filter, since the number of tests, checks and recency of the first matched WME is only one attribute. **Def-complex-cr-strategy** defines filters which iterate through the list of matched WMEs in the instantiations, comparing them. **Narrow-down-lex** and **narrow-down-wme-class** are defined with this macro, since they need to compare all of the WMEs in the instantiations.

The method that the complex filter functions compare instantiations is as follows: First, initialize the list of candidates to the entire conflict-set. Iterate through the list of WMEs in each instantiation. On each iteration, reduce the candidate list to the instantiations whose WMEs are ranked highest by the comparer function. If the candidate set is reduced to one instantiation, then return that instantiation, else continue. When there are no more WMEs to compare, return the remaining candidates.

See the command description section for details of how to use **Def-cr-strategy** and **Def-complex-cr-strategy**. Set the switch *TRACE-CR* to T in order to trace the action of the conflict resolver.

7. Agenda

Agendas are, in their simplest form, ordered and dynamically alterable lists of tasks to perform. Rulekit Agendas are currently nested two deep, which means that the lists of tasks are themselves put into an ordered list. This top-level list is called the *Agenda*, and the list of rules is called a *bucket*. A bucket may be thought of as a task, a context, or a goal. The order of the rules in a bucket and the order of buckets in an agenda are changed by the rules themselves. This change is done by a rule adding or deleting another rule or bucket. The Agenda is started with the **r-agenda** or **cont-agenda** command (see the Command Summary).

One advantage of Rulekit over OPS-5 is the integration of Agendas with OPS. There are many ways that one can combine agendas and an OPS type rule base, but the one we will use gives the Agenda the top-level control so that OPS is called from the Agenda. Thus, the Agenda system can be thought of as an alternative to the OPS recognize-act cycle. It is more like an iterate-act-recognize cycle. The rules and buckets in the agenda will be iterated over as they were before, but now the test to see if they should fire will not be only a lisp test in the rule field, but also whether the rule is in the conflict set. When it is found that a rule is in the conflict set, it will be fired (i.e., the instantiation of its RHS evaluated). To resolve the conflict of there being more than one possible instantiation of the same rule, the list of filters in *CR-STRATEGY* will be successively applied to the conflict set.

Note that since the Agenda system calls OPS as a subroutine when firing a rule, when OPS returns, the Agenda might still be in the same bucket as it was before, depending on the bucket and agenda control strategies. Because of the actions of the last rule, OPS may cause a rule to match which is in a higher-priority bucket than the current one. It won't fire even though it probably should. Agenda control strategy *:priority* should be used if the user wants to be guaranteed that the highest-priority rule in the highest-priority bucket always fires when it should. We will now go into more detail about the Agenda and Bucket strategies.

7.1. Bucket Control Strategies

Common to all bucket traversal strategies is the fact that whenever a rule returns :halt,[3] execution of the bucket stops.

- **:linear**
  This is the simplest of the bucket strategies. The list of rules in the bucket is simply iterated over once. Each rule is tested and fired if the test is true. Execution is halted when the end of the bucket is reached (i.e., there is no cycling).

- **:cyclic**
  This is a multi-pass linear control strategy. The list of rules in the bucket is iterated over. Each rule is tested and fired if the test is true. When the end of the bucket is reached, Rulekit checks to see if any rule in the bucket has fired. If so, execution is restarted at the beginning of the bucket. If not, execution is halted.

- **:priority**
  This is the priority cycle strategy. The list of rules is iterated over until a rule fires. At this point, control returns to the beginning of the bucket. When the end of the bucket is reached and no rule in the bucket has fired, execution is halted.

[3]note: 'halt is not the same as :halt!

7.2. Agenda Control Strategies

Since an agenda is essentially a bucket of buckets, the possible control strategies for agendas are more complex. Rulekit provides four of the most useful control strategies.

Just as there is a special control flag in the bucket control strategies, there are special control flags in the agenda control which override any agenda control strategy.

- When a rule returns :halt, execution of the agenda stops.
- When a rule returns :bucket-halt, execution of the bucket stops and control moves on to the next bucket.
- When a rule returns :recycle, control goes to the top of the agenda (which is the first rule in the first bucket).

The available agenda control strategies are as follows:

- **:linear**
  FRulekit simply traverses each bucket in a linear manner. When the end of one bucket is reached, it goes to the next bucket. Execution stops when the end of the last bucket is reached.
- **:bucket**
  Upon firing any rule, control goes back to the first rule in the current bucket. When the end of a bucket is reached, control proceeds with the next bucket.
- **:bucket-priority**
  Control returns to the beginning of the current bucket whenever a rule fires. When the end of a bucket is reached, FRulekit checks to see if any rule in the bucket has fired. If so, control is passed to the top of the agenda. Otherwise, execution continues with the next bucket.
- **:linear-cycle**
  Linear with respect to the agenda but cyclic inside the buckets. FRulekit iterates through the buckets one after the other. At the end of a bucket, if any rule had fired in that bucket, then control is returned to the beginning of that bucket again. Otherwise, the next bucket is chosen.
- **:priority**
  Control returns to the top of the agenda whenever any rule fires.

8. Trace Package

The important trace flag is *RECORD-LEVEL*, which determines whether Rulekit should be tracing itself while it runs. Since tracing slows down the system slightly, it is by default set to 1 (only enough tracing for the **back** function), but is set to 2 when the trace package (trace.slisp) is loaded. **RE-INIT-TRACE** may be called to re-initialize.

8.1. Purpose

There are two different uses for the trace module. The first is that the trace functions described here can be used as tools for learning and analogy algorithms. The knowledge compilation techniques composition and proceduralization, as well as goal-subsumption, analogy and generalization, will be possible using this package and Rulekit. The second use of a trace package is for user debugging. Many of the trace functions that are used for learning are also useful for debugging productions. While the main goal of this package is to provide tools for implementors of learning algorithms, some of the trace functions will be used for writing debugging functions.

8.2. Data Types

WME. Working Memory Element.

Rulekit provides three useful predefined slots for WMEs:

- **%time**
  Accessed by calling **wme-%time**, this stores the time tag of when the WME was created.
- **%created**
  Accessed by **wme-%created**, this stores a pointer to the instantiation of the production which created the WME (see *instantiations*, below).
- **%class**
  Accessed by **wme-%class**, this is a symbol which is the name of the class of the WME.

Other slots, defined by literalize, are accessed in a similar way.

?WME. A negated or positive Working Memory Element. A negated WME can mean the absence of a WME in some contexts, or the deletion of a WME from WM in others, depending on what function is being applied to it. A ?WME is a potted-pair whose car is either ABS or POS, and whose cdr is a WME data structure.

WMES. Plural Working Memory Elements. These are just lists of WMEs. Note: this is the convention for all pluralized data types.

PATTERN. These are like WMEs except they are lists and may contain variables instead of constants in their value field. Variables are preceded by an '=' sign.

```
Example:  (polygon :name =x :size-of 3 :perimeter-of =y)
```

Note: this is only what patterns look like in the source production system file. The '=' sign is actually a read macro which, when seen by the production reader, expands into (var x). Thus, a trace function which returns a PATTERN would actually return (polygon :size-of 3 :perimeter-of (var x))

CONDE. Condition Element, LHS pattern. They may additionally contain negation tests, which would be of the form (ABS <PATTERN>)

WM. Working Memory, a set of WMEs.

Production Instantiation. These are productions which have matched. Thus all of their variables (except ones which only appear in <ABS> tests) are instantiated with values. The access functions for instantiations are:

- **var-names-of <instant>**
  returns: list of variable names <instant> is an instantiation. This returns the names of the variables in the instantiation.
- **var-in-instant <varname> <instant>**
  Returns the value of <varname> in instantiation <instant>.
- **instant-wmes-of <instant>**
  Returns a list of WMEs which matched the given instantiation, in the order of the corresponding condition elements of the production.
- **prod-of <instant>**
  Production from which the instantiation came. The production is also a Parmenides frame. Access functions for the various fields of the production are described next.
- **give-additions-on-cycle <N>**
  Returns a list of the WMEs which were made on cycle N.

- **give-deletions-on-cycle <N>**
  Returns a list of the WMEs which were removed on cycle N.

Productions. The access functions for productions are formed by concatenating the symbol rk-rule- to the slot name. For example, the function rk-rule-lhs returns the lhs of a production. The useful slots are:

- **LHS**
  The Left Hand Side, an ordered list of condition elements.
- **RHS**
  The Right Hand Side, an ordered list of actions.
- **PNAME**
  The name of the production.

Other slots are described in the command summary under the RULE section.

8.3. Tracing Productions

8.3.1. give-matches <?WME>

returns: <production-matches> Give the production instantiations which were ever placed in the conflict set (they could have since been removed from the c.s.) because the ?WME directly helped it's conditions to match. This is different from the first function because with the first function, the production doesn't have to be completely matched and so only the production, and not the instantiation, can be returned.

8.3.2. give-firings <?WME>

returns: <production-firings> Returns all productions which *fired* as a result of ?WME being added to WM. [Note that a negative WME here would mean a WME being deleted from WM.] A subset of **give-matches** since not all of the matches will have necessarily fired.

8.3.3. give-next-matches <?WME>

returns: <production-matches> Give the production instantiations which were placed in the conflict set on the next cycle after the given WME was. Note that ?WME may not have been responsible for the instantiation, since more than one instaniation may be created on each cycle.

8.3.4. give-next-firings <?WME>

returns: <production-firings> Returns all production which *fired* on the next cycle after ?WME was added to/deleted from WM. A subset of both **give-firings** and **give-next-matches**.

8.3.5. give-matches-on-cycle N

returns: <production-matches> Return all the matches on cycle N.

8.3.6. give-firings-on-cycle N

returns: <production-firings> Return all the firings on cycle N. This will be a singleton list for conflict resolution strategies which don't allow parallelism.

8.3.7. give-named-matches <pname> N

returns: <production-matches> Returns a list of the instantiations of production <pname> which fired on cycle N.

8.3.8. give-named-firings <pname> N

returns: <production-firings> Returns a list of the firings of production <pname> which fired on cycle N.

8.3.9. lhs-of <pname>

returns: CONDs Return the lhs condition patterns given a production name.

8.3.10. rhs-of <pname>

returns: PATTERNs Return the rhs patterns given a production name.

8.4. WME Creation

8.4.1. prod-responsible-for <?WME>

returns: <production-firing> Give the production firing which last added ?WME to WM if ?WME is positive, and give the production firing which last deleted ?WME from WM if ?WME is negative.

8.4.2. possible-prods-responsible-for <?WME>

returns: <production-matches> *not implemented*. Give the production matches which could possibly add/delete ?WME to/from WM.

8.4.3. in-wmp <CONDE>

returns: List of WMEs Return a list of WMEs in WM that match the given pattern. If CONDE is an absence test, then it returns all WMEs that *don't* match the pattern. Ignores variables; thus intra-condition-element testing won't work.

8.4.4. give-negs <production-firing>

returns: <PATTERNS> *Note: not implemented*. Give the most specific negation tests (i.e., if a negation test has a variable in it, give the negation test with the variable replaced by it's instantiation for rule firing).

8.5. Tracing Goals

At the moment there is no single representation of goal trees in Rulekit. However, the following functions are useful for tracing changes of goals' states, regardless of representation. Note that the goal trace data structure is independent of the OPS trace data structures. The goal trace is stored in the string, *TRACE-TREE*.

- **trace-pop-success <goalname>**
  Records a successful termination of a goal by writing into the goal trace data structure.
- **trace-pop-failure <goalname>**
  Records an un-successful termination of a goal by writing into the goal trace data structure.
- **trace-interrupt**
  Records that an interrupt from a goal tree has occurred.
- **trace-push-goal <parent-name> <subgoal-name>**
  Records that a subgoal under parent-name has been activated.
- **record-action <action>**
  Inserts given generic action into the goal trace tree.

9. Command Summary

This section is divided into three parts: the OPS-style commands, the Agenda commands, and the commands which can't fit into one of the first two categories.

9.1. General Commands

9.1.1. RULE

```
syntax: (RULE <rulename> . (<rule-slot> <contents>)*)
```

Defines a rule. Neither the rulename nor slot contents are evaluated. Possible slots are the standard OPS slots LHS and RHS, as well as the Agenda slots EXTRA-TEST, ADD, DEL, BUCKET-ADD, BUCKET-DEL and KTEST:

- **:LHS**
  Defines the Left Hand Side part of the rule. The LHS is a list of working-memory tests, which are themselves lists.

- **:RHS**
  Defines the Right Hand Side part of the rule. The RHS is a list of actions, which are simply lisp expressions to be evaluated. In the Agenda system, we sometimes refer to a rule action "returning" a value. The returned value is defined to be the value that the last expression returns. In addition, the OPS module itself returns a value. The top level OPS commands, such as **start** and **run**, return what the last rule returned.

- **:BELIEFS**
  Similar in syntax and meaning to the RHS. However, the :BELIEFS slot has the additional meaning that if the rule un-matches after it has fired, then the actions taken in this slot will be un-done. Thus, this provides a rudimentary reason-maintenance system. See also the function **def-inverse-action**, which tells FRulekit how to take back user-defined functions.

Important: to avoid infinite loops, don't remove a WME in the :BELIEFS slot which was matched on in the :LHS! This is equivalent to believing that if A, then not A. If you want to remove a WME which was matched on the :LHS, then do this on the :RHS. Thus, the :RHS can be thought of a set of side-effects since they are not un-done when the :LHS is no longer true.

- **:EXTRA-TEST**
  This and the remaining slots are only applicable when the AGENDA strategy is used. **:EXTRA-TEST** contains a lisp expression which must be true before the rule can fire. The lisp expression must not refer to any LHS variables; for this kind of test use lisp *checks* in the LHS. Use the :EXTRA-TEST as little as possible, for it is less efficient than working memory tests and defeats the purpose of the rete algorithm.

- **:ADD**
  If the rule fires, the add field tells Agenda to add the given set of rules to the designated bucket at the specified position in the bucket. The buckets, positions and rules are specified as:

  ```
  ((<bucket-spec-1> <bucket-position-1> . rule-names-1)
   (<bucket-spec-2> <bucket-position-2> . rule-names-2) ..)
  ```

  The notation for <bucket-spec> and <bucket-position> are described in Appendix IV. *rule-names* is simply a list of rule names.

- **:DEL**
  If the rule fires, the **del** field specifies what rules to delete from what buckets:

  ```
  ((<bucket-spec-1> . rule-names-1)
   (<bucket-spec-2> . rule-names-2) ...)
  ```

- **:KTEST**
  If the rule test is *not* true, then the ktest is checked. If the ktest is true, then the rule will delete itself from the current bucket. Ktest is essentially a suicide test for rules that have out-lived their usefulness.

- **:BUCKET-ADD**
  If the rule fires, then the bucket-add field specifies what buckets to add to the agenda:

```
((bucket-name-1 <bucket-spec-1>)
 (bucket-name-2 <bucket-spec-2>) ...)
```

```
Example: ((bucket-1 :*next) (bucket-3 bucket-1))
```

  The buckets are added *after* the given agenda positions. The notation for <bucket-spec> is also located in Appendix IV.

- **:BUCKET-DEL**
  If the rule fires, then the bucket-del field specifies what buckets to delete from the agenda:

```
(bucket-name-1 bucket-name-2 ...)
```

```
  example:
(RULE find-redundant-actions
  :LHS ((goal :name =n1 :action =a (LABEL =c1))
        (goal :name =n2 :action =a (CHECK (<> =n1 =n2)) (LABEL =c2)))
  :RHS (($remove =c1 =c2)
        ($make 'goal :name 'redundant-action :action =a))
        (princ "Found redundant rules with action: ") (princ =a) (terpri)))
```

9.1.2. RULE*

```
syntax: (RULE* <rulename> <lhs> <rhs>)
```

This is the non-macro form of **RULE**. It evaluates all of its arguments. <lhs> and <rhs> are lists of condition elements and action elements, respectively.

9.1.3. BUILD-RULE

```
syntax: (BUILD-RULE <rulename> . (<rule-slot> <contents>)*)
```

With the same syntax as RULE, BUILD-RULE does the same thing RULE does except it also works when there is already something in working memory. This is because it "pushes" the appropriate working-memory elements through the tests corresponding to the new conditions in the rule. Most efficient for rules which don't test classes which haven't been tested by other rules.

9.1.4. BUILD-RULE*

```
syntax: (BUILD-RULE* <rulename> <lhs> <rhs>)
```

This is the non-macro form of **BUILD-RULE**. It has the same syntax as **RULE*** and also evaluates all of its arguments. <lhs> and <rhs> are lists of condition elements and action elements, respectively.

9.1.5. PP-RULE

```
syntax: (pp-rule <rulename>)
```

Pretty-prints a production. Note: this used to be called pp.

9.1.6. PP-WME

```
syntax: (pp-wme <wme> &optional <stream>)
```

Pretty-prints a working memory element. <stream> defaults to the standard output (e.g., the terminal) for all printing functions. Writes all slots.

9.1.7. PP-WMES

```
syntax: (pp-wmes <list-of-wmes> &optional <stream>)
example: (pp-wmes (list <wme1> <wme2>))
```

Pretty-prints a list of working memory elements. <stream> defaults to the standard output (e.g., the terminal) for all printing functions. Writes all slots.

9.1.8. PP-WM

```
syntax: (pp-wm &optional <class> <stream>)
```

Pretty-prints working memory. If <class> is supplied, then it prints only those working memory elements belonging to <class>.

9.1.9. PP-ITH-WME

```
syntax: (pp-ith-wme <I> &optional <class> <stream>)
```

Pretty-prints a selected WME. If <class> is provided then the I^{th} WME of the given class is printed, otherwise the I^{th} WME is printed. The WME numbers can be seen by setting the flag ***NUMBER-WMES*** before using **pp-wm** or **save-wm**.

9.1.10. SAVE-WME

```
syntax: (save-wme <wme> &optional <stream> <all-slots>)
```

Like **pp-wme**, but prints <WME> in a form which FRulekit can read. *<all-slots>* has a default value of NIL, which means that only the slots which differ from the parent class are written. If it is true then all slots will be written.

9.1.11. SAVE-WM

```
syntax: (save-wm &optional <class> <stream>)
```

Like **pp-wm**, but prints the specified WME's in a form which FRulekit can read.

9.1.12. SAVE-ITH-WME

```
syntax: (save-ith-wme <I> &optional <class> <stream>)
```

Saves a selected WME. If <class> is provided then the I^{th} WME of the given class is saved, otherwise the I^{th} WME is saved. The WME numbers can be by setting the flag ***NUMBER-WMES*** before using **pp-wm** or **save-wm**.

9.1.13. WM

```
syntax: (wm &optional <class>)
```

Like **pp-wm**, excepts returns the working memory elements specified in a list instead of printing them. If

<class> is supplied, then it returns only those working memory elements belonging to <class>. Use this sparingly, since it must cons up a new list every time it is called.

9.1.14. SET-WM

```
syntax: (set-wm {WME}*)

example: (set-wm (goal :name 'foo :action 'bar)
                 (goal :name 'biz :action 'bang))
```

Clears Working Memory then adds the given WMEs to Working Memory and possibly changes the conflict set. Similar to **start** except **start** additionally runs the recognize-act cycle. Useful for setting up working memory when the Agenda control structure is used.

9.1.15. ADD-TO-WM

```
syntax: (add-to-wm {WME}*)
```

Adds the given WMEs to Working Memory and possibly changes the conflict set. Different from $make in that $make takes *patterns* rather than straight WMEs. Thus $make is only used on the RHS of productions, whereas **add-to-wm** is usually used from the top-level. Similar to **cont** except **cont** additionally runs the recognize-act-cycle.

9.1.16. Pp-instant

```
syntax: (pp-instant <instantiation>)

example: (pp-instant (car *CONFLICT-SET*))
```

Pretty-prints the given rule instantiation. Shows the WME's which matched the rule.

9.1.17. Def-cr-strategy

```
syntax: (def-cr-strategy <strategy-name> <extractor> <comparer>)

example:
(def-cr-strategy num-tests
      (lambda (ins) (car (rk-rule-num-tests (instant-prod ins))))
      >)
```

Defines a simple conflict-resolving filter (see the Conflict Resolution section). <strategy-name> is the name which will be given to the function which does the filtering. <extractor> should be a function name or lambda expression which returns some measure of an instantiation's appropriateness for firing given the instantiation. <comparer> should be a function which, given two measures of instantiation appropriateness, returns non-NIL iff the first one is more appropriate than the second. In the **num-tests** example, the number of tests in the rule is the appropriateness rating, and the greater-than function, ">" is the comparer.

9.1.18. Def-complex-cr-strategy

```
syntax: (def-complex-cr-strategy <strategy-name> <extractor> <comparer>)

example: (def-complex-cr-strategy narrow-down-lex wme-%time >)
```

Defines a more complex conflict-resolving filter (see the Conflict Resolution section). The <extractor> should be the name of a function which, when given a WME in an instantiation, returns a measure of appropriateness for firing. <comparer> should be a function which, given two measures of WME appropriateness, returns non-NIL iff the first one is more appropriate than the second.

Note that since **FRulekit** implements CR strategies by defining functions for each one, efficiency can be gained by compiling the **Def-cr-strategy** and **Def-complex-cr-strategy** statements.

9.1.19. Def-inverse-action

```
syntax: (def-inverse-action <action> <inverse-action>)

example: (def-inverse-action $remove-keep
                  #'(lambda (action instant)
                       (add-frame (var-in-instant (cadr action) instant))))
```

Tells FRulekit how to take back an action performed in the :BELIEFS slot. When a rule no longer matches, if it had previously fired, then the actions performed in the :BELIEFS list will be taken back (but *not* the actions in the :RHS slot). When the inverse action is called, it is given two arguments: the text of the action, e.g., '*($remove-keep =frame)*', and the instantiation of the rule which no longer matches. This can be used to identify the specific WME's and other data structures involved in the action.

9.1.20. PR-SWITCHES

```
syntax: (pr-switches)
```

Prints out the system switches, such as *TRACE-ADD*, etc. (See the next section.)

9.1.21. RE-INIT-TRACE

```
syntax: (re-init-trace)
```

Re-initializes the trace package. The trace package is initialized automatically when it is loaded, but it may be useful to re-initialize it later.

9.1.22. $DISJ

```
syntax:  ($disj <n> &optional <or-number>)
```

This RHS predicate is only meaningful in rules which contain <OR> commands. Returns T if and only if the <n>th disjunct of the <or-number>th <OR> statement in the current instantiation is true. The first disjunct is numbered 1. If <or-number> is not supplied, then the first one is assumed.

9.1.23. $DISJ-NUMS

```
syntax:  ($DISJ-NUMS)
```

This RHS function is also only meaningful in rules which contain <OR> commands. It returns a list of the disjunct numbers of the disjuncts in the <OR> statements of the current instantiation. The first number in the list corresponds to the first <OR>; the second number to the second <OR>, etc. The first disjunct is numbered 1.

Note the relationship between **$disj** and **$disj-nums**: `($disj <n> <ornum>) == (= <n> (nth (1- <ornum>) ($disj-nums)))`, although using **$disj** when possible is more efficient. **$disj-nums** is useful inside a **case** statement, since otherwise one would have to call **$disj** repeatedly.

9.2. OPS style Commands

9.2.1. START

```
Syntax: (start {WME}*)
```

This top-level command clears working memory, puts the given WMEs into Working Memory then starts the **FRulekit** system on its way. Usually, a production is written to really start the system, and its left hand side checks for something like (goal :start yes) which is added with the START command. If the Agenda control structure is desired instead of the recognize-act cycle, use *R-Agenda*, below, and use *ADD-TO-WM* to set up working memory.

9.2.2. LITERALIZE

```
Syntax: (literalize <class> <cplist> <slots>)
```

```
Example: (literalize goal (:extendable nil)
       :parent NIL
       :action (value NIL)
       :name NIL)
```

Defines a WME class by calling the Parmenides **def-frame** function, and automatically makes it **is-a** WME, which is a predefined class. The WME class is not propagatable, an attribute which will be inherited by all user-defined classes unless otherwise specified.

Note that **literalize** is very similar to the Parmenides **def-frame** function in both syntax and semantics; see the documentation of that function for more details. In particular, when any list (except NIL or "()") is given as the default value for a slot, that slot is defined to have facets. In order to declare a slot to not have facets, supply a symbol (such as T or NIL) as the default value. The main difference between **literalize** and **def-frmae** is that **literalize** makes the given class a sub-class of the WME class, and gives FRulekit certain information for printing the class as a WME.

9.2.3. $MAKE

```
Syntax: ($make <class> <slot-specs)
```

```
Example: ($make ’goal :name ’get-cookie :parent ’get-food)
```

Evaluates all arguments. Adds the given WME to working memory and updates the conflict set if necessary. Returns the Rulekit representation of the working memory element that it makes.

9.2.4. $MAKE-NAMED

```
Syntax: ($make-named <class> <name> <slot-specs>)
```

```
Example: ($make ’goal ’get-1 :name ’get-cookie :parent ’get-food)
```

The same as **$make** except that it also allows the user to name the frame which is made.

9.2.5. $REMOVE

```
Syntax: ($remove {WME}*)
```

```
Example: ($remove =c1 =c2)
```

Removes the given WME(s) from working memory and updates the conflict set if necessary. Should only be executed from the RHS of a production or during a pause. The variables such as =c1 and =c2 are bound to a WME by a LABEL statement in the condition element corresponding to that WME. Note: be careful not to use remove, which is a CommonLisp function!

9.2.6. $REMOVE-KEEP

`Syntax: ($remove-keep {WME}*)`

`Example: ($remove-keep =c1 =c2)`

Just like **$remove**, except doesn't delete the Parmenides frame.

9.2.7. $MODIFY

`Syntax: ($modify <WME> <slot-specs>)`

`Example: ($modify =c1 :name 'eat-cookie)`

Modifies the specified slot in the given WME by first removing it from working memory, changing the specified slots, then adding the new WME to working memory. Updates the conflict set if necessary. Leaves the other slots unchanged. If *RECORD-LEVEL* (see the section on Global Flags) is 1 or more, then $modify creates a new copy of the given wme; otherwise it modifies the given WME destructively. Returns the new WME made. Should only be executed from the RHS of a production or during a pause. Doesn't fire any demons associated with the frame.

9.2.8. $MODIFY-IN-PLACE

`Syntax: ($modify-in-place <WME> <slot-specs>)`

`Example: ($modify-in-place =c1 :name 'eat-cookie)`

This is a 'true' modify. Instead of removing and making the WME, which **$modify** does, **$modify-in-place** modifies the specified WME directly, and updates the conflict set if necessary. This results in a faster modify, and more straightforward semantics. With **$modify**, if a rule had fired with a certain set of WME's, and then $modify is performed on one of those WME's, then the rule can re-match even if that rule didn't test any of the slots which were changed. In other words, with $modify, rules can fire again even when it doesn't seem that they should. For example, the following rule matches against people whose age is 10, and infers that they must weigh 100:

```
(RULE find-boy-or-girl
 :LHS
  ((person :age 10 (LABEL =person)))
 :RHS (
       ($modify =person :wt 100)))
```

Each time the rule fires, it matches again because the *=person* WME is $remove'd and $make'd. If $modify-in-place were used, it would not match again since changing the *:wt* doesn't affect the match.

If *RECORD-LEVEL* (see the section on Global Flags) is 1 or more, then $modify-in-place creates a new copy of the given wme; otherwise it modifies the given WME destructively. Returns the new WME made. Should only be executed from the RHS of a production or during a pause. Doesn't fire any demons associated with the frame.

Although **$modify-in-place** has cleaner semantics, the **$modify** command has been around for so long in production systems that many production sets rely on the semantics of $modify. So in changing to $modify-in-place, make sure that you are not relying on the old semantics.

9.2.9. $MODIFY-DEMONS

Syntax: ($modify-demons <WME> <slot-specs>)

Exactly like **$modify**, except it fires demons associated with the modified slots. Only fires demons associated with the first facet modified. Doesn't fire any class demons. See **demons** in the **Parmenides** manual.

9.2.10. EXCISE

Syntax: (excise <rule-name> &optional <excise-all>)

Deletes the given rule from Rulekit's rule memory. Removes all tests associated the rule (unless shared by other rules). Excise is automatically performed on rules that are re-defined. If <excise-all> is true, then it will excise the rule even if it is isomorphic to other rules.

9.2.11. ADD-FRAME

Syntax: (add-frame <frame>)

Adds an already-defined Parmenides frame instance to working memory. <frame> may be the name of a frame or just a frame.

9.2.12. WHYNOT

syntax: (whynot {<rulename>}*)

FRulekit provides a simple explanation facility for aiding user debugging of rules. It is a supplement to the **matches** function, below. The **WHYNOT** function tells you why the given rule didn't fire. There are two reasons for a rule not firing: if it didn't match, then **WHYNOT** will tell you the first condition element that didn't match. If the rule matched but wasn't chosen by the c.r. strategy, then it will explain the c.r. strategy and why it chose another rule over that rule. Currently the c.r. strategy is not explained, however.

9.2.13. MATCHES

syntax: (matches <rulename> &key :max :stream)

Example: (matches myrule :max 3)

Lists each of the WMEs that matched the condition elements of the given rule. If **max** is supplied then it only prints matches for that many condition elements, otherwise it prints matches for all condition elements. Gives more information than **WHYNOT** but is harder to decipher.

9.2.14. D-IN-INSTANTS

syntax: (d-in-instants <rulename>)

Finds all the differences between the various instantiations of the rule named **<rulename>**. This is helpful when more instantiations of a rule match than you want. All of the current instantiations may be seen by examining the value of the variable *CONFLICT-SET*.

9.2.15. VAR-IN-INSTANT

syntax: (var-in-instant <variable> <instantiation>)

Example: (var-in-instant 'foo (car *CONFLICT-SET*))

Returns the value of the given FRulekit variable in the given rule instantiation. The variable must be bound on the LHS of the rule.

9.2.16. HALT

syntax: `(HALT)`

Temporarily stops the production system and returns control to the user. Should only be executed from the RHS of a production. Not to be confused with the atom 'halt which can be returned from rules when the Agenda module is active.

9.2.17. RBREAK

syntax: `(rbreak <rname> &optional (<when> :BEFORE))`

Example: `(rbreak your-rule)`

Breakpointing facility for rules. Tells **FRulekit** to halt the system before and/or after firing the specified rule. <*when*> may be either :BEFORE, :AFTER, :ALWAYS, or NIL. If it's :ALWAYS then it halts both before and after firing the given rule. If it's NIL then it doesn't halt before or after the rule. <*when*> is optional and defaults to :BEFORE. Note that when a rule is re-defined, it will no longer be breakpointed. See also: ***PAUSE-EVERY-CYCLE***.

9.2.18. CONT

syntax: (CONT {<WME>}*)

Adds the given WMEs to working memory, then resumes execution of the production system.

9.2.19. RUN

syntax: `(RUN &optional (<number-of-cycles> *MAX-BACK*))`

Like continue but instead of taking WMEs to add, it optionally takes the number of cycles that the system should run. The default number of cycles is *MAX-BACK*, which is initially set to 50.

9.2.20. BACK

syntax: `(BACK &optional <number-of-cycles>)`

Like run excepts restores the system to previous states rather than moving it to new ones. Removes WMEs that were added, and adds WMEs that were removed. Default number-of-cycles is 1. In order to back up the system to its initial state, use the **clear-net** command. The maximum number of cycles for which changes are recorded is *MAX-BACK*.

9.2.21. INIT-RETE

Deletes all rules from the rule memory. Zeros the conflict set and working memory. Should be called before loading a new set of productions.

9.2.22. CLEAR-NET

Blanks the declarative memory by deleting all working memory elements from working memory and zeroing the conflict set. Sets *CYCLE* to 0. Although old working memory elements will no longer be seen by rules, they will still be considered frames by Parmenides, and will still be *is-a WME*. Thus they can be re-added to working memory with the *add-frame* command (above).

9.2.23. COMPILE-NODES

Compiles the lisp tests at each of the nodes in the Rete net, resulting in significant execution speedup. Note that even if there are no user-supplied lisp checks, Rulekit generates some lisp code for these tests. However, it is more effective when there are more user lisp checks.

Occasionally, the lisp compiler may issue the following warning while compiling the lisp code in the Rete nodes:

```
Warning between functions:
  RIGHT bound but not referenced.
```

This message could mean one of two things. There could be a CHECK which doesn't test any **FRulekit** variables. This is noted by **FRulekit** at rule-loading time if the switch *RULEKIT-TERSE* is NIL (the default). If you don't see this warning, then there must be a CHECK which tests FRulekit variables which were bound in previous CE's, but no variables in the current CE. This is a little less efficient than putting the test in one of the CE's which bind the variables, but will still work.

9.2.24. COMPILE-RHSES

Compiles the right hand side of each rule, resulting in some speedup and doing some type checking.

9.2.25. COMPILE-ALL

Compiles all nodes, right-hand-sides, and other data structures. To compile the *extra* lisp tests, use **compile-extra-tests**. **compile-nodes** gains the most efficiency compared to the amount of time spent compiling code, since most of the time is spent in the match. However, if most of the rules are fired many times, then it would probably be best to use **compile-all**.

9.2.26. INHIBIT-RULES

`Syntax: (inhibit-rules <rule-names>)`

`Example: (inhibit-rules rule1 rule2)`

Disallows the given rules from ever being able to enter into the conflict set again. Doesn't stop the rule from matching; use **EXCISE** (described above) to remove it from the discrimination net altogether. Temporarily removes from the conflict set any instantiations associated with the rule. Different from the delete field of a rule, which removes the names of rules from the Agenda, but doesn't affect the rule.

9.2.27. UNINHIBIT-RULES

`Syntax: (uninhibit-rules <rule-names>)`

`Example: (uninhibit-rules rule1 rule2)`

Allows the given rules to match again, after being inhibited by *inhibit-rules*. Puts any associated instantiations back into the conflict set. Different from the add field of a rule, which adds the name of rules to the agenda. The functions **INHIBIT-RULE** and **UNINHIBIT-RULE** are also provided.

9.2.28. $MODIFY-NUM

`Syntax: ($modify-num <condition-element-number> <slot-specs>)`

`Example: ($modify-num 2 :name 'eat-cookie)`

Exactly like $modify, except takes a condition element *number* rather than a label. It is bad style to use $MODIFY-NUM from a rule, but it is provided for debugging purposes. Thus its main use should be during pauses.

9.3. Agenda Commands

9.3.1. (r-linear <bucket>)

<bucket> is an ordered list of rule names. This command runs the **linear** bucket strategy on the list **<bucket>**. It assumes that every rule in the given list has already been defined by the *RULE* command.

9.3.2. (r-cycle <bucket>)

Runs the **cyclic** bucket strategy on the designated rule list.

9.3.3. (r-priority <bucket>)

Runs the **priority** bucket strategy on the given bucket.

9.3.4. (r-agenda <agenda> &optional <cycles>)

Runs the given agenda. If **<cycles>** is supplied, then it limits the number of rule firings to that number. Otherwise it runs until the end of the agenda is reached.

<agenda> is an ordered list of (<bucket-name> <rule-name1> <rule-name2> ...). Example:

(r-agenda '((bucket-name-A rule-A-1 rule-A-2 ...)
(bucket-name-B rule-B-1 rule-B-2 ...) ...))

This is the top-level call to the agenda mechanism, and loosely corresponds to the *Start* command.

9.3.5. (create-new-agenda <agenda>)

<agenda> is the same as in the **r-agenda** command. Defines the agenda without running it.

9.3.6. (cont-agenda &optional <cycles>)

Continues the agenda execution after the Agenda has been defined with **r-agenda** or **create-new-agenda**. If **<cycles>** is supplied then it limits the number of rule firings to that number.

9.3.7. (add-rule <bucket-spec> <bucket-position> <rule-names>)

Adds the list of rules given in <*rule-names*> to the bucket specified by <*bucket-spec*> at the place in the bucket specified by <*bucket-position*>. <*bucket-spec*> is usually just a name of a bucket but see Appendix IV for the complete description of <bucket-spec> and <bucket-position>. If <bucket-spec> is a name but isn't a bucket, **FRulekit** makes a bucket with that name.

9.3.8. (add-rules <list-of-addrule-specs>)

Similar to :ADD of the RULE command except the arguments are evaluated. <list-of-addrule-specs> takes the form:

```
((<bucket-spec-1> <bucket-position-1> . rule-names-1)
 (<bucket-spec-2> <bucket-position-2> . rule-names-2) ..)
```

9.3.9. (make-bucket <bucket-name> <bucket-contents>)

<bucket-contents> is simply an ordered list of rule names to put into the bucket. Example:

(make-bucket 'bucket-1 '(rule-A-1 rule-A-2))

Defines a bucket, but doesn't put the bucket into the agenda. Use **r-agenda** or **add-buckets** to do this.

9.3.10. (add-buckets <bucket-specs>)

Like :BUCKET-ADD of the RULE command except the argument is evaluated. <bucket-specs> have the form:

```
'((bucket-name-1 <bucket-spec-1>)
  (bucket-name-2 <bucket-spec-2>) ...)
```

```
Example: (add-buckets '((bucket-1 :*next) (bucket-3 :*current)))
```

Note that if <bucket-spec> is a bucket name, that bucket must already be in the Agenda. The buckets are added *after* the given agenda positions.

9.3.11. (add-bucket <bucket-name> <bucket-spec>)

Adds a single bucket to the position in the Agenda specified by <bucket-spec>.

9.3.12. (add-new-bucket <bucket-name> <bucket-spec> <bucket-contents>)

A combination of **make-bucket** and **add-bucket**, this first defines the bucket <bucket-name> to have <bucket-contents>, then puts it into the Agenda.

9.3.13. (delete-rules <list-of-delelterule-specs>)

Similar to :DEL of the RULE command except the arguments are evaluated. <list-of-deleterule-specs> takes the form:

```
((<bucket-spec-1> . rule-names-1)
 (<bucket-spec-2> . rule-names-2) ...)
```

9.3.14. (delete-buckets <bucket-names>)

<bucket-names> is a list of bucket names. Example:

(delete-buckets '(bucket-name-A bucket-name-B))

Similar to the :BUCKET-DEL slot of the RULE schema, except it evaluates its argument.

9.3.15. (delete-bucket <bucket-name>)

Deletes the specified bucket from the Agenda.

9.3.16. (get-bucket <bucket-name>)

Returns the list of rule names associated with the given bucket.

9.3.17. (compile-extra-tests)

Similar to **compile-nodes**, this compiles the lisp expressions on the extra-test slot of each rule that has one. The more extra lisp tests there are, the more useful this function will be.

10. Useful Global Variables and Switches

In general, switches specific to Agenda are preceded by one or more exclamation points (!), whereas all other switches are surrounded by asterisks (*).

CR-STRATEGY An ordered list of instantiation filters which defines the conflict resolution strategy. When the Agenda module is used, it uses this to decide between different instantiations of the same rule. **FRulekit** provides three cr-strategies: *LEX* and *MEA*, and *FULL-CR*. See section 6 on conflict-resolution for the details of defining c-r strategies.

RULE-NAMES A list containing the names of all the Rulekit rules. Note: putting the name of a rule into a bucket doesn't make it a rule.

RECORD-LEVEL Determines how much recording of working memory changes Rulekit does. 0 means no recording, 1 means enough recording for the **back** command to work, and 2 means full tracing for the trace package. It is normally set to 1. When the trace package is loaded, it is set to 2. If you never intend to use the **back** command, then set *RECORD-LEVEL* to 0 to improve efficiency.

If the record level is 2, then when WMEs are modified, a new copy of them is made; otherwise the WME is destructively modified. If the record level is 1, then a record of modifications made to each WME is made, so that WMEs can be restored to their previous state.

CONFLICT-SET Type this to see the current contents of the conflict set.

PAUSE-EVERY-CYCLE If true, Rulekit will pause on every cycle with the lisp function **break**. Gives the user a read-eval-print loop in lisp, useful for debugging. Note that during a pause, one is effectively in the right hand side of the production that just fired. Thus, one can look at the bindings of the current production's variables just by typing the variable name (preceded by the '=' sign). Also, one may call the RHS actions such as $remove and $modify with these variables. See also: **RBREAK**.

TRACE-FIRE If true, then Rulekit types out the name of each rule when it is fired. If *TRACE-FIRE* is set to 'values, then Rulekit will additionally print out the values of all the production variables. Not to be confused with the trace package.

TRACE-MATCH Causes Rulekit to print out every production match, regardless of whether it actually fires.

TRACE-UNMATCH Causes Rulekit to print out whenever a rule gets taken away from the conflict set because it no longer matches.

TRACE-ADD When true, Rulekit reports every time a working memory element gets added to working memory.

TRACE-DELETE When true, Rulekit reports every time a working memory element gets deleted from working memory.

TRACE-CR If true, then Rulekit reports on the progress of the conflict resolver.

TRACE-CYCLE If true, causes Rulekit to print out the cycle number on every cycle. On by default.

TRACE-MODIFY If true, then Rulekit will report on every modify-in-place.

RULEKIT-TERSE Controls the printing of miscellaneous messages by **FRulekit**; for instance it will not tell you what rules are being defined if this flag is non-nil. By default NIL.

REFRACT If true, then refraction is enabled. Refraction simply means that when a rule fires, its instantiation is removed from the conflict set so that the same instantiation won't fire again. Since refraction is a useful way to avoid infinite loops, it is by default on.

MAX-BACK Set by default to 50, this tells Rulekit the maximum number of cycles to record WMEs for backing up. This number should be altered before **FRulekit** is loaded since certain data structures are created which depend on this number.

FULL-WMEPRINT When nested WMEs are encountered by **pp-wme**, they are not ordinarily printed, but instead only their class is printed. If *FULL-WMEPRINT* is non-nil, then nested WMEs will be printed fully.

NUMBER-WMES If this flag is non-NIL, then when pretty-printing a set of WME's, FRulekit will also print a WME number next to each of them. This number can be used with the **pp-ith-wme** and **save-ith-wme** functions to print or save a selected WME.

AGENDA The current agenda (i.e., list of bucket names). Use **get-bucket** on the bucket name to retrieve the list of rules in that bucket.

!trace-test If set to non-nil, then every time a rule is tested by the Agenda system, it is reported to the terminal.

!trace-act If set to non-nil, then every time the Agenda applies a rule, it prints out the result value of the rule. Similar to *TRACE-FIRE*.

!trace-agenda If set to non-nil, then agenda actions such as rule addition and deletion will be printed when they occur.

!!control The user sets this to the desired agenda control strategy. The possible values are **:linear, :priority, :bucket, :bucket-priority and :linear-cycle**, as described above.

11. Multiple Language Messages

FRulekit can produce its messages in a number of languages. Although it currently supports only English and Spanish, it is easy to add new languages through message files. When **FRulekit** is loaded, it loads a message file for the appropriate language. When installing **FRulekit**, the definition of the **FRulekit** directory must be set - change the variable *FR-PATHNAME*, near the top of build.lisp, or set it (while in the FRulekit package) before loading FRulekit. The language can also be set by changing the variable *LANGUAGE*. The current language can be changed by calling **define-language** with the name of the appropriate language. "eng" defines English, and "esp" defines Spanish. The default is English.

12. Acknowledgements

Many people at CMU have helped with the development of FRulekit and its documentation.

- Daniel Borrajo, visiting from Spain, implemented the freelist, **$modify-in-place** command and the multi-lingual capability.
- Angela Hickman helped to write the documentation of the agenda module.
- My discussions with Brian Milnes have helped to solidify my understanding of the Rete net.
- I thank the CMU users for bearing with me while bugs were still being fixed and for suggesting ideas for improvement: Keith Barnett, Mark Perlin, Bernadette Kowalski, Patty Cheng, Klaus Gross, Hans Tallis, and Ben McLaren.

I. An Example: The Tower of Hanoi

```
;;; FRulekit program to solve the Tower of Hanoi problem.

(use-package 'FRulekit)

(setq *CR-STRATEGY* *MEA*)

(literalize move ()
  :stack NIL
  :from NIL
  :to NIL)

(literalize on ()
  :disk NIL
  :peg NIL)


(RULE break-down-problem
  :LHS ((move (LABEL =c3) :stack =stack :from =from :to =to
              (CHECK (listp =stack)))   ;; Must be a 'non-trivial' move
        (on :disk =d :peg =from (CHECK (subsetp =stack =d))))
  :RHS (($remove =c3)
        ($make 'move :stack (allbutlast =stack)
               :from (otherpeg =from =to) :to =to)
        ($make 'move :stack (car (last =stack)) :from =from :to =to)
        ($make 'move :stack (allbutlast =stack) :from  =from
               :to (otherpeg =from =to))))

(RULE move-disk
  :LHS ((move (LABEL =c1) :stack =stack :from =from :to =to)
        (on (LABEL =c2) :disk =d :peg =from
            (CHECK (eq =stack (car =d))))
        (on (LABEL =c3) :disk =x :peg =to
            (CHECK (or (null =x) (> (car =x) =stack)))))
  ;EXTRA-TESTS (equal 3 3)
  :RHS (($remove =c1)
        ($modify =c3 :disk (cons =stack =x))    ;;put it on other stack.
        ($modify =c2 :disk (cdr =d))            ;;take that disk off stack
        (format T "Moving disk ~A to top of peg ~A" =stack =to)))

(defun begin ()
  (start
   (on :peg 1 :disk '(1 2 3))
   (on :peg 2 :disk nil)
   (on :peg 3 :disk nil)
   (move :stack '(1 2 3) :from 1 :to 3)))

;;;Utility functions for tower-of-Hanoi
(defun allbutlast (a)
  (let ((a (nreverse (cdr (reverse a)))))
    (if (null (cdr a)) (car a) a)))

(defun otherpeg (a b)
  (car (delete a (delete b (copy-list '(1 2 3))))))
```

II. Another Example: Network propagation

```
;;; FRulekit program to demonstrate the propagation ability of
;;; Parmenides together with the pre-set and post-set demons.

(use-package 'Parmenides)
(use-package 'FRulekit)

(literalize person (:propagate T :cache :*ALL*)
  :walks (:value 'yes))

;;; Boy is a sub-class of person.
(literalize boy (is-a (person))
  :age (:value 10))

(defun init-testnet ()
  (set-facet 'person :walks :value 'yes))

(RULE find-walking-boy
  :LHS (
   (boy :age =a :walks yes (LABEL =b)))
  :RHS (
   (format T "Found walking boy ")
   (pp-wme =b)
   (set-facet-demons 'person :walks :value 'no)))

(RULE find-non-walking-boy
  :LHS (
   (boy :walks no (LABEL =b)))
  :RHS (
   (format T "Found non-walking boy ")
   (pp-wme =b)
   ($remove =b)))

(defun begin ()
  (init-testnet)
  (start
   (boy :age '(:value 10))))
```

III. Yet another Example: Nested WMEs

```
;;; Simple Monkey and Bananas in FRulekit
;;; This version demonstrates creation and matching of nested WMEs.
;;; The state slot of the goal WME is a pointer to an actual state wme.
;;; WANT-TO-MOVE-TO-THEM creates and MOVE-TO-THEM and EAT-THEM test
;;; for this.

(use-package 'FRulekit)

(literalize goal ()
  :state NIL)   ;; Any constant will do to specify that it's not faceted.

;;;Generic state of something.  Tells where something is, and other info.
(literalize stateof ()
  :what NIL
  :where NIL
  :other NIL)


(RULE want-to-move-to-them
  :LHS ((stateof :what bananas :where =w :other yummy (LABEL =state))
        (<ABS> (stateof :what me :where =w)))
  :RHS (($make 'goal :state =state)))

;;;Matches a GOAL wme whose state slot is filled by a STATEOF wme.
(RULE move-to-them
  :LHS (
        (goal (LABEL =goal) :state (stateof :what bananas :where =ban))
        (stateof :what me :where =w (LABEL =me) (CHECK (<> =w =ban))))
  :RHS (($modify =me :where =ban)))    ;;just hyperspace to the bananas

;;;Matches a GOAL wme whose state slot is filled by a STATEOF wme.
(RULE eat-them
  :LHS (
        (goal (LABEL =goal)
              :state (stateof :what bananas :where =w :other yummy))
        (<ABS> (stateof :what me :where =x (CHECK (<> =w =x)))))
  :RHS (($remove =stateof =goal)))
;;;FRulekit automatically binds =stateof to the nested wme

(defun begin ()
  (start
   (stateof :what 'me :where 'corner :other 'hungry)
   (stateof :what 'bananas :where 'ceiling :other 'yummy)))
```

IV. Specifying the position of buckets and rules

Many commands, such as rule add, rule delete, bucket add and bucket delete, refer to bucket positions within the agenda or to rule positions within a bucket. A bucket may be referred to by just giving the name or by giving one of the positional keywords:

<*agenda-pos*> = :*current, :*next, :*previous, :*first, or :*last

<*bucket-spec*> = <bucket-name> or <agenda-pos>

The position of a rule within a bucket is specified as:

<*bucket-position*> = :*first, :*last, or :*next

Note that :*next is only defined while the agenda is running.

A list of rules is given as:

rule-names = (<list-of-rule-names>) or :*ALL

V. Acquiring, Loading, Compiling and Using FRulekit

FRulekit is free to affiliates and members of the CMU community. It is available for a nominal fee to all others. To request a copy, send mail to pshell@ml.ri.cmu.edu or write to:

Peter Shell
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

If you use **FRulekit**, please send mail to the above address to get on the computer mailing-list for updates.

The Parmenides frame system, in /usr/pshell/parmenides/parmenides.lisp on the ML vax and WIENER RT, must always be loaded before any other module is loaded or compiled. (Note that Parmenides must be loaded before it is compiled.) Under CMU Lisp the compiled file, parmenides.fasl, should be loaded. The **FRulekit** sources are kept in /usr/pshell/frulekit/build.lisp and inter.lisp, on both the ML vax and WIENER RT machines. Build must be loaded before inter. They are known to work for many dialects of CommonLisp (e.g., CMU CLisp, Allegro, Lucid, Dec, Symbolics CLisp). However, only the compiled CMU lisp (.fasl) files are kept up to date. The compiled RT code is in /../wiener/usr/pshell/parmenides/ and /../wiener/usr/pshell/frulekit/. In order to load the multi-lingual message files, the variable *FR-PATHNAME*, near the top of build.lisp, must be defined. This variable defines the directory in which the FRulekit files are found. **FRulekit** is put into a package called 'FRulekit' with nickname 'FRK'. After loading **FRulekit**, one should do: *(use-package 'FRulekit)*.

The (optional) agenda package is in /usr/pshell/frulekit/agenda.lisp and agenda.fasl, and should be loaded after build and inter. The (optional) trace package is in /usr/pshell/frulekit/trace.lisp and trace.fasl, and should also be loaded after build and inter. All modules should be loaded before any rules are defined. There is a directory of sample **FRulekit** programs in /usr/pshell/frulekit/eg/. This document is /afs/cs/user/pshell/frulekit/frulekit.{mss, doc, ps}.

In order to load in the **FRulekit** system, one would type:

```
(load "/usr/pshell/parmenides/parmenides")
(use-package 'parmenides)                ;; optional

(load "/usr/pshell/frulekit/build")
(load "/usr/pshell/frulekit/inter")
(load "/usr/pshell/frulekit/agenda")     ;; optional
(load "/usr/pshell/frulekit/trace")      ;; optional
(use-package 'frulekit)
```

In order to compile **FRulekit** from scratch, one would type:

```
(load "/usr/pshell/parmenides/parmenides.lisp")
(compile-file "/usr/pshell/parmenides/parmenides.lisp")
(load "/usr/pshell/parmenides/parmenides")
                ;; Compiled version is MUCH faster!

(compile-file "/usr/pshell/frulekit/build.lisp")
(load "/usr/pshell/frulekit/build")
(compile-file "/usr/pshell/frulekit/inter.lisp")
(load "/usr/pshell/frulekit/inter")
(compile-file "/usr/pshell/frulekit/agenda.lisp")
(compile-file "/usr/pshell/frulekit/trace.lisp")
(use-package 'frulekit)
```

Table of Contents