3. USING RULES IN LOOPS

The LOOPS rules language is supported by an integrated programming environment for creating, editing, compiling, and debugging RuleSets. This section describes how to use that environment.

3.1 Creating RuleSets

RuleSets are named LOOPS objects and are created by sending the class **RuleSet** a **New** message as follows:

(_ ($ RuleSet) New)

After entering this form, the user will be prompted for a LOOPS name as

RuleSet name: *RuleSetName*

Afterwards, the RuleSet can be referenced using LOOPS dollar sign notation as usual. It is also possible to include the RuleSet name in the **New** message as follows:

(_ ($ RuleSet) New NIL *RuleSetName***)**

3.2 Editing RuleSets

A RuleSet is created empty of rules. The RuleSet editor is used to enter and modify rules. The editor can be invoked with an **EditRules** message (or **ER** shorthand message) as follows:

(_ *RuleSet* **EditRules)**
(_ *RuleSet* **ER)**

If a RuleSet is installed as a method of a class, it can be edited conveniently by selecting the **EditMethod** option from a browser containing the class. Alternatively, the **EditMethod** message can be used:

(_ *ClassName* **EditMethod** *selector***)** [Message]

Both approaches to editing retrieve the source of the RuleSet and put the user into the TTYIN or TEdit editor, treating the rule source as text.

Initially, the source is a template for RuleSets as shown in Figure 7. The rules are entered after the comment at the bottom. The declarations at the beginning are filled in as needed and superfluous declarations can be discarded.

```
RuleSet Name: RuleSetName;
WorkSpace Class:  ClassName;
Control Structure:  doAll;
While Condition:  ;
Audit Class:  StandardAuditRecord;
Rule Class:  Rule;
Task Class:  ;
Meta Assignments:  ;
Temporary Vars:  ;
Lisp Vars:  ;
Debug Vars:  ;
Compiler Options:  ;

  (* Rules for whatever.  Comment goes here.)
```

Figure 7. Initial Template for a RuleSet

You can then edit this template to enter rules and set the declarations at the beginning. In the current version of the rule editor, most of these declarations are left out. If you choose the **EditAllDecls** option in the RuleSet editor menu, the declarations and default values will be printed in full.

The template is only a guide. Declarations that are not needed can be deleted. For example, if there are no temporary variables for this RuleSet, the **Temporary Vars** declaration can be deleted. If the control structure is not one of the **while** control structures, then the **While Condition** declaration can be deleted. If the compiler option **A** is not chosen, then the **Audit Class** declaration can be deleted.

When you leave the editor, the RuleSet is compiled automatically into a Lisp function.

If a syntax error is detected during compilation, an error message is printed and you are given another opportunity to edit the RuleSet.

3.3 Copying RuleSets

Sometimes it is convenient to create new RuleSets by editing a copy of an existing RuleSet. For this purpose, the method **CopyRules** is provided as follows:

(_ *oldRuleSet* **CopyRules** *newRuleSetName***)** [Message]

This creates a new RuleSet by some of the information from the pespectives of the old RuleSet. It also updates the source text of the new RuleSet to contain the new name.

3.4 Saving RuleSets on Lisp Files

RuleSets can be saved on Lisp files just like other LOOPS objects. In addition, it is usually useful to save the Lisp functions that result from RuleSet compilation. In the current implementation, these functions have the same names as the RuleSets themselves. To save RuleSets on a file, it is necessary to add two statements to the file commands for the file as follows:

(FNS * MyRuleSetNames)
(INSTANCES * MyRuleSetNames)

where **MyRuleSetNames** is a Lisp variable whose value is a list of the names of the RuleSets to be saved.

If RuleSets are methods associated with a class, and they are saved by using (FILES?), then the file package saves the appropriate entries. The user does not have to be concerned with editing the filecoms of the file being made.

3.5 Printing RuleSets

To print a RuleSet without editing it, one can send a **PPRules** or **PPR** message as follows:

(_ *RuleSet* **PPRules)** [Message]

(_ *RuleSet* **PPR)** [Message]

A convenient way to make hardcopy listings of RuleSets is to use the function **ListRuleSets**. The files will be printed on the **DEFAULTPRINTINGHOST** as is standard in Interlisp-D. **ListRuleSets** can be given four kinds of arguments as follows:

(ListRuleSets *RuleSetName***)**
(ListRuleSets *ListOfRuleSetNames***)**
(ListRuleSets *ClassName***)**
(ListRuleSets *FileName***)**

In the *ClassName* case, all of the RuleSets that have been installed as methods of the class will be printed. In the last case, all of the RuleSets stored in the file will be printed.

3.6 Running RuleSets from LOOPS

RuleSets can be invoked from LOOPS using any of the usual protocols.

Procedure-oriented Protocol: The way to invoke a RuleSet from LOOPS is to use the **RunRS** function:

(RunRS *RuleSet workSpace arg2 ... argN***)** [Function]

workSpace is the LOOPS object to be used as the work space. This is "procedural" in the sense that the RuleSet is invoked by its name. *RuleSet* can be either a RuleSet object or its name.

Object-oriented Protocol: When RuleSets are installed as methods in LOOPS classes, they can be invoked in the usual way by sending a message to an instance of the class. For example, if **WashingMachine** is a class with a RuleSet installed for its **Simulate** method, the RuleSet is invoked as follows:

(_ washingMachineInstance Simulate)

Data-oriented Protocol: When RuleSets are installed in active values, they are invoked by side-effect as a result of accessing the variable on which they are installed.

3.7 Installing RuleSets as Methods

RuleSets can also be used as methods for classes. This is done by installing automatically-generated invocation functions that invoke the RuleSets. For example:

```
[DEFCLASS WashingMachine
  (MetaClass Class doc (* comment) ...)
  ...
  (InstanceVariables (owner ...))
  (Methods
        (Simulate RunSimulateWMRules)
        (Check RunCheckWMRules
              doc (* Rules to Check a washing machine.))
...]
```

When an instance of the class **WashingMachine** receives a **Simulate** message, the RuleSet **SimulateWMRules** will be invoked with the instance as its work space.

To simplify the definition of RuleSets intended to be used as Methods, the function **DefRSM** (for "Define Rule Set as a Method") is provided:

(DefRSM *ClassName Selector RuleSetName***)** [Function]

If the optional argument *RuleSetName* is given, **DefRSM** installs that RuleSet as a method using the *ClassName* and *Selector*. It does this by automatically generating an installation function as a method to invoke the RuleSet. **DefRSM** automatically documents the installation function and the method.
If the argument *RuleSetName* is **NIL**, then **DefRSM** creates the RuleSet object, puts the user into an Editor to enter the rules,

compiles the rules into a Lisp function, and installs the RuleSet as before.

DefRSM can be invoked with the browser as follows:

- Position the cursor over a class in a browser.
- Press the middle mouse button. A menu pops up.
- Select the Add option in this menu, and drag the mouse to the right to display the submenu that includes the "DefRSM" option. You are prompted to enter a selector name.

After a RuleSet has been installed as a method by using **DefRSM**, you can then edit that RuleSet by selecting the "EditMethod" option from the browser editing menu.

3.8 Installing RuleSets in Active Values

Note: The following section and any other references to active values within the rule documentation refer to active values as they were implemented in the Buttress release. The functionality of triggering rules from active values has not been tested using the current implementation of active values. It should work to use the **ExplicitFnActiveValue** class to implement this behavior.

RuleSets can also be used in data-oriented programming so that they are invoked when data is accessed. To use a RuleSet as a *getFn*, the function **RSGetFn** is used with the property **RSGet** as follows:

...
(InstanceVariables
(*myVar* **#(***myVal* **RSGetFn NIL) RSGet** *RuleSetName***))**
...

RSGetFn is a LOOPS system function that can be used in an active value to invoke a RuleSet in response to a LOOPS get operation (e.g., **GetValue**) is performed. It requires that the name of the RuleSet be found on the **RSGet** property of the item. **RSGetFn** activates the RuleSet using the local state as the work space. The value returned by the RuleSet is returned as the value of the get operation.

To use a RuleSet as a *putFn*, the function **RSPutFn** is used with the property **RSPut** as follows:

...
(InstanceVariables
(*myVar* **#(***myVal* **NIL RSPutFn) RSPut** *RuleSetName***))**
...

RSPutFn is a function that can be used in an active value to invoke a RuleSet in response to a LOOPS put operation (e.g., **PutValue**). It requires that the name of the RuleSet be found on the **RSPut** property of the item. **RSGetFn** activates the RuleSet using the *newValue* from the put

operation as the work space. The value returned by the RuleSet is put into the local state of the active value.

3.9 Tracing and Breaking RuleSets

LOOPS provides breaking and tracing facilities to aid in debugging RuleSets. These can be used in conjunction with the auditing facilities and the rule executive for debugging RuleSets. The following summarizes the compiler options for breaking and tracing:

T Trace if rule is satisfied. Useful for creating a running display of executed rules.

TT Trace if rule is tested.

B Break if rule is satisfied.

BT Break if rule is tested. Useful for stepping through the execution of a RuleSet.

Specifying the declaration **Compiler Options: T;** in a RuleSet indicates that tracing information should be displayed when a rule is satisfied. To specify the tracing of just an individual rule in the RuleSet, the **T** meta-descriptions should be used as follows:

{T} IF *cond* **THEN** *action***;**

This tracing specification causes LOOPS to print a message whenever the LHS of the rule is tested, or the RHS of the rule is executed. It is also possible to specify that the values of some variables (and compound literals) are to be printed when a rule is traced. This is done by listing the variables in the **Debug Vars** declaration in the RuleSet:

Debug Vars: a a:b a:b.c;

This will print the values of **a**, **a:b**, and **a:b.c** when any rule is traced or broken.

Analogous specifications are provided for breaking rules. For example, the declaration **Compiler Options: B;** indicates that LOOPS is to enter the rule executive (see Section 3.10, "The Rule Exec") after the LHS is satisfied and before the RHS is executed. The rule-specific form:

{B} IF *cond* **THEN** *action***;**

indicates that LOOPS is to break before the execution of a particular rule.

Sometimes it is convenient in debugging to display the source code of a rule when it is traced or broken. This can be effected by using the **PR** compiler option as in

Compiler Options: T PR;

which prints out the source of a rule when the LHS of the rule is tested and

Compiler Options: B PR;

which prints out the source of a rule when the LHS of a rule is satisfied, and before entering the break.

3.10 The Rule Exec

A Read-Compile-Evaluate-Print loop, called the rule Executive, is provided for the rule language. The rule Executive can be entered during a break by invoking the Lisp function **RE**. During RuleSet execution, the rule executive can be entered by typing **^f** (<control>-f) on the keyboard.

On the first invocation, **RE** prompts the user for a window. It then displays a stack of RuleSet invocations in a menu to the left of this window in a manner similar to the Interlisp-D Break Package. Using the left mouse button in this window creates an Inspector window for the work space for the RuleSet. Using the middle mouse button pretty prints the RuleSet in the default prettyprint window.

In the main rule Executive window, **RE** prompts the user with "**re:**". Anything in the rule language (other than declarations) that is typed to this Executive will be compiled and executed immediately and its value printed out. For example, you may type rules to see whether they execute or variable names to determine their values. For example:

re: trafficLight:color
Red
re:

this example shows how to get the value of the **color** variable of the **trafficLight** object. If the value of a variable was set by a RuleSet running with auditing, then a **why** question can be typed to the rule executive as follows:

re: why trafficLight:color

IF highLight:color = 'Green farmRoadSensor:cars timer.TL
THEN highLight:color _ 'Yellow timer.Start;

Rule 3 of RuleSet LightRules
Edited: Conway "13-Oct-82"

re:

The rule executive may be exited by typing **OK**.

3.11 Auditing RuleSets

Two declarations at the beginning of a RuleSet affect the auditing. Auditing is turned on by the compiler option **A**. The simplest form of this is

Compiler Options: A;

The **Audit Class** declaration indicates the class of the audit record to be used with this RuleSet if it is compiled in *audit* mode.

Audit Class: StandardAuditRecord;

A **Meta Assignments** declaration can be used to indicate the audit description to be used for the rules unless overridden by a rule-specific meta-assignment statement in a meta-descriptor.

Meta Assignments: cf_.5 support_'GroundWff;

3.12 Loading Rules

Set the variable LOOPSUSERSDIRECTORIES to include the directory where the Rules files are stored.

Load the file LOOPSRULES-ROOT.LCOM, which will load the following files from LOOPSUSERSDIRECTORIES:

- LOOPSBACKWARDS.LCOM
- LOOPSMIXIN
- LOOPSRULES.LCOM
- LOOPSRULESP.LCOM
- LOOPSRULESC.LCOM
- LOOPSRULESD.LCOM, which will load the file TTY.LCOM from LISPUSERSDIRECTORIES.

Editing rules will be easier if TEdit is loaded. Loading the Rules does not automatically load TEdit.

3.13 Known Problems

In a rule, the expression $pipe.ri..$p compiles to (RunRS (QUOTE ($ pipe)) ($ p)), which fails.

Meta-assignment statements cannot handle expressions. This means that statements like {cf _ .5} work fine, but {validity _ 'fact} fails.

A value of 1 in a meta-descriptor statement is always taken to be a one-shot designator. You cannot have a meta-descriptor statement like {cf_1}. However, the number 1.0 can be used; the meta-descriptor statement, {cf_1.0}, works.

Rules have not been tested without loading TEdit in order to edit RuleSets.

[This page intentionally left blank.]