1. INTRODUCTION TO RULE-ORIENTED PROGRAMMING IN LOOPS

The core of decision-making expertise in many kinds of problem solving can be expressed succinctly in terms of rules. The following sections describe facilities in LOOPS for representing rules, and for organizing knowledge-based systems with rule-oriented programming. The LOOPS rule language provides an experimental framework for developing knowledge-based systems. The rule language and programming environment are integrated with the object-oriented, data-oriented, and procedure-oriented parts of LOOPS.

Rules in LOOPS are organized into production systems (called RuleSets) with specified control structures for selecting and executing the rules. The work space for RuleSets is an arbitrary LOOPS object.

Decision knowledge can be factored from control knowledge to enhance the perspicuity of rules. The rule language separates decision knowledge from meta-knowledge such as control information, rule descriptions, debugging instructions, and audit trail descriptions. An audit trail records inferential support in terms of the rules and data that were used. Such trails are important for knowledge-based systems that must be able to account for their results. They are also essential for guiding belief revision in programs that need to reason with incomplete information.

1.1 Introduction

Production rules have been used in expert systems to represent decision-making knowledge for many kinds of problem-solving. Such rules (also called *if-then* rules) specify actions to be taken when certain conditions are satisfied. Several rule languages have been developed in the past few years and used for building expert systems. The following sections describe the concepts and facilities for rule-oriented programming in LOOPS.

LOOPS has the following major features for rule-oriented programming:

(1) Rules in LOOPS are organized into ordered sets of rules (called RuleSets) with specified control structures for selecting and executing the rules. Like subroutines, RuleSets are building blocks for organizing programs hierarchically.

(2) The work space for rules in LOOPS is an arbitrary LOOPS object. The names of the instance variables provide a name space for variables in the rules.

(3) Rule-oriented programming is integrated with object-oriented, data-oriented, and procedure-oriented programming in LOOPS.

(4) RuleSets can be invoked in several ways: In the object-oriented paradigm, they can be invoked as methods by sending messages to objects. In the data-oriented paradigm, they can be invoked

as a side-effect of fetching or storing data in active values. They can also be invoked directly from Lisp programs. This integration makes it convenient to use the other paradigms to organize the interactions between RuleSets.

(5) RuleSets can also be invoked from rules either as predicates on the LHS of rules, or as actions on the RHS of rules. This provides a way for RuleSets to control the execution of other RuleSets.

(6) Rules can automatically leave an audit trail. An audit trail is a record of inferential support in terms of rules and data that were used. Such trails are important for programs that must be able to account for their results. They can also be used to guide belief revision in programs that must reason with incomplete information.

(7) Decision knowledge can be separated from control knowledge to enhance the perspicuity of rules. The rule language separates decision knowledge from meta-knowledge such as control information, rule descriptions, debugging instructions, and audit trail descriptions.

(8) The rule language provides a concise syntax for the most common operations.

(9) There is a fast and efficient compiler for translating RuleSets into Interlisp functions.

(10) LOOPS provides facilities for debugging rule-oriented programs.

The following sections are organized as follows: Section 1.2, "Basic Concepts," outlines the basic concepts of rule-oriented programming in LOOPS. It contains many examples that illustrate techniques of rule-oriented programming. Section 1.3, "Organizing a Rule-Oriented Program," describes the rule syntax, and the remaining sections in this chapter discuss the facilities for creating, editing, and debugging RuleSets in LOOPS.

1.2 Basic Concepts

Rules express the conditional execution of actions. They are important in programming because they can capture the core of decision-making for many kinds of problem-solving. Rule-oriented programming in LOOPS is intended for applications to expert and knowledge-based systems.

The following sections outline some of the main concepts of rule-oriented programming. LOOPS provides a special language for rules because of their central role, and because special facilities can be associated with rules that are impractical for procedural programming languages. For example, LOOPS can save specialized audit trails of rule execution. Audit trails are important in knowledge systems that need to explain their conclusions in terms of the knowledge used in solving a problem. This capability is essential in the development of large knowledge-intensive systems, where a long and sustained effort is required to create and validate knowledge bases. Audit trails are also important for programs that do non-monotonic reasoning. Such programs must work with incomplete information, and must be able to revise their conclusions in response to new information.

1.3 Organizing a Rule-Oriented Program

In any programming paradigm, it is important to have an organizational scheme for composing large systems from smaller ones. Stated differently, it is important to have a method for partitioning large programs into nearly-independent and manageably-sized pieces. In the procedure-oriented paradigm, programs are decomposed into procedures. In the object-oriented paradigm, programs are decomposed into objects. In the rule-oriented paradigm, programs are decomposed into *RuleSets*. A LOOPS program that uses more than one programming paradigm is factored across several of these dimensions.

There are three approaches to organizing the invocation of RuleSets in LOOPS:

Procedure-oriented Approach. This approach is analogous to the use of subroutines in procedure-oriented programming. Programs are decomposed into RuleSets that call each other and return values when they are finished. *SubRuleSets* can be invoked from multiple places. They are used to simplify the expression in rules of complex predicates, generators, and actions.

Object-oriented Approach. In this approach, RuleSets are installed as methods for objects. They are invoked as methods when messages are sent to the objects. The method RuleSets are viewed analogously to other procedures that implement object message protocols. The value computed by the RuleSet is returned as the value of the message sending operation.

Data-oriented Approach. In this approach, RuleSets are installed as access functions in active values. A RuleSet in an active value is invoked when a program gets or puts a value in the LOOPS object. As with active values with Lisp functions for the *getFn* or *putFn*, these RuleSet active values can be triggered by any LOOPS program, whether rule-oriented or not.

These approaches for organizing RuleSets can be combined to control the interactions between bodies of decision-making knowledge expressed in rules. For example, Figure 1 shows the RuleSet of consumer instructions for testing a washing machine. The work space for the ruleSet is a LOOPS object of the class **WashingMachine**. The control structure While1 loops through the rules trying an escalating sequence of actions, starting again at the beginning of some rule is applied. Some rules, called one-shot rules, are executed at most once. These rules are indicated by preceding them with a one in braces ({1}).

```
RuleSet Name: CheckWashingMachine;
WorkSpace Class:  WashingMachine;
Control Structure:  while1 ;
While Condition:  ruleApplied;

(* What a consumer should do when a washing machine failes.)

  IF .Operational THEN (STOP T);

  IF load>1.0 THEN .ReduceLoad;

  If ~pluggedInTo THEN .PlugIn;

{1}  IF pluggedInTo:voltage=0 THEN breaker.Reset;

{1}  IF pluggedInTo:voltage<110 THEN SPGE.Call;

{1}  THEN dealer.RequestService;

{1}  THEN manufacturer.Complain;

{1}  THEN $ConsumerBoard.Complain;

{1}  THEN (STOP T);
```

Figure 1. Basic RuleSet

1.4 Control Structures for Selecting Rules

RuleSets in LOOPS consist of an ordered list of rules and a control structure. Together with the contents of the rules and the data, a RuleSet control structure determines which rules are executed. Execution is determined by the contents of rules in that the conditions of a rule must be satisfied for it to be executed. Execution is also controlled by data in that different values in the data allow different rules to be satisfied. Criteria for iteration and rule selection are specified by a RuleSet control structure. There are two primitive control structures for RuleSets in LOOPS which operate as follows:

Do1 [RuleSet Control Structure]

The first rule in the RuleSet whose conditions are satisfied is executed. The value of the RuleSet is the value of the rule. If no rule is executed, the RuleSet returns **NIL**.

The **Do1** control structure is useful for specifying a set of mutually exclusive actions, since at most one rule in the RuleSet will be executed for a given invocation. When a RuleSet contains rules for specific and general situations, the specific rules should be placed before the general rules.

DoAll [RuleSet Control Structure]

Starting at the beginning of the RuleSet, every rule is executed whose conditions are satisfied. The value of the RuleSet is the value of the last rule executed. If no rule is executed, the RuleSet returns **NIL**.

The **DoAll** control structure is useful when a variable number of additive actions are to be carried out, depending on which conditions are satisfied. In a single invocation of the RuleSet, all of the applicable rules are invoked.

Figure 2 illustrates the use of a **Do1** control structure to select one of three mutually exclusive actions.

```
RuleSet Name: SimulateWashingMachine;
WorkSpace Class:  WashingMachine;
Control Structure:  Do1 ;

(* Rules for controlling the wash cycle of a washing machine.)

  IF controlSetting = 'RegularFabric
  THEN .Fill .Wash .Pause .SpinAndDrain
    .SprayAndRinse .SpinAndDrain
    .Fill. DeepRinse .Pause .DampDry;

  IF controlSetting = 'PermanentPress
  THEN .Fill .Wash .Pause .SpinAndPartialDrain
    .FillCold .SpinAndPartialDrain
    .FillCold .Pause .SpinAndDrain
    .FillCold. DeepRinse .Pause .DampDry;

  IF controlSetting = 'DelicateFabric
  THEN .FillSoak1 .Agitate .Soak4 .Agitate
    .Soak1 .SpinAndDrain .SprayAndRinse
    .SpinAndDrain .Fill .DeepRinse .Pause .DampDry;
```

Figure 2. RuleSet showing Do1

There are two control structures in LOOPS that specify iteration in the execution of a RuleSet. These control structures use an explicit while-condition associated with the RuleSet. They are direct extensions of the two primitive control structures above.

While1 [RuleSet Control Structure]

This is a cyclic version of **Do1**. If the while-condition is satisfied, the first rule is executed whose conditions are satisfied. This is repeated as long as the while condition is satisfied or until a **Stop** statement or transfer call is executed (see Section 2.14, "Stop Statements"). The value of the RuleSet is the value of the last rule that was executed, or **NIL** if no rule was executed.

WhileAll [RuleSet Control Structure]

This is a cyclic version of **DoAll**. If the while-condition is satisfied, every rule is executed whose conditions are satisfied. This is repeated as long as the while condition is satisfied or until a **Stop** statement is executed. The value of the RuleSet is the value of the last rule that was executed, or **NIL** if no rule was executed.

The "while-condition" is specified in terms of the variables and constants accessible from the RuleSet. The constant **T** can be used to specify a RuleSet that iterates forever (or until a **Stop** statement or transfer is executed). The special variable **ruleApplied** is used to specify a RuleSet that continues as long as some rule was executed in the last iteration. Figure 3 illustrates a simple use of the **WhileAll** control structure to specify a sensing/acting feedback loop for controlling the filling of a washing machine tub with water.

```
RuleSet Name: FillTub;
WorkSpace Class:  WashingMachine;
Control Structure:  WhileAll ;
Temp Vars:  waterLimit;
WhileCond:  T;

(* Rules for controlling the filling of a washing tub with
water.)

{1!} IF loadSetting = 'Small THEN waterLimit_10;
{1!} IF loadSetting = 'Meduim THEN waterLimit_13.5;
{1!} IF loadSetting = 'Large THEN waterLimit_17;
{1!} IF loadSetting = 'ExtraLarge THEN waterLimit_20;

(* Respond to a change of temperature setting at any time.)

  IF termperatureSetting = 'Hot
  THEN HotWaterValve.Open ColdWaterValve.Close;

  IF termperatureSetting = 'Warm
  THEN HotWaterValve.Open ColdWaterValve.Open;

  IF termperatureSetting = 'Cold
  THEN HotWaterValve.Close ColdWaterValve.Open;

(* Stop when the water reaches its limit.)

  IF waterLevelSensor.Test >= waterLimit
  THEN HotWaterValve.Close ColdWaterValve.Close
    (Stop T);
```

Figure 3. RuleSet with WhileAll

There are two control structures in LOOPS that specify iteration over a set of elements in the execution of a RuleSet. These control structures use an explicit while-condition associated with the RuleSet. They are direct extensions of the two primitive control structures above.

FOR1 [RuleSet Control Structure]

This is a cyclic version of **Do1**. If the iteration-condition (or while-condition) is satisfied, the first rule is executed whose conditions are satisfied or until a **Stop** statement is executed. This is repeated as long as the iteration condition is satisfied. The value of the RuleSet is the value of the last rule that was executed, or **NIL** if no rule was executed.

FORALL [RuleSet Control Structure]

This is a cyclic version of **DoAll**. If the iteration-condition is satisfied, every rule is executed whose conditions are satisfied. This is repeated as long as the iteration condition is satisfied or until a **Stop** statement is executed. The value of the RuleSet is the value of the last rule that was executed, or **NIL** if no rule was executed.

The "iteration-condition" is specified in terms of the variables and constants accessible from the RuleSet. The simplest condition is

(FOR <iterVar> IN <setExpr> DO ruleSet) ;

The **setExpr** will be parsed with the RuleSet parser. The symbol **ruleSet** is a reserved word, and must be spelled as shown (no changes in capitalization).

Here is an example of iteration:

Control Structure: FORALL;
Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)) DO ruleSet) ;

For each buyer in the list produced by RoadStops, the ruleSet will be run. In a **FOR1**, the iteration will go on to the next buyer as soon as one rule executes. In a **FORALL**, all rules in the RuleSet will be tried.

For nested iteration one can use a slightly more complicated form, as illustrated by the following example:

Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)) DO
(FOR seller in (RoadStops ($ Producer)) DO ruleSet)) ;

An experienced Lisp user can see that this resembles the CLISP iteration construct. In fact, except that you can (must) use the RuleSet syntax in the construct, it is the CLISP construct, and any such construct can be used. A DO1 or DOALL ruleSet will be substituted for the occurrence of the atom ruleSet, depending on whether the Control Structure is a FOR1 or FORALL.

As an abbreviation, if the construct does not contain the atom ruleSet, then (DO ruleSet) is appended to the Iteration Condition for a **FOR1** or **FORALL**. Thus one could write the first example as:

Iteration Condition: (FOR buyer IN (RoadStops ($ Consumer)))

1.5 One-Shot Rules

One of the design objectives of LOOPS is to clarify the rules by factoring out control information whenever possible. This objective is met in part by the declaration of a control structure for RuleSets.

Another important case arises in cyclic control structures in which some of the rules should be executed only once. This was illustrated in the Washing Machine example in Figure 1 where we wanted to prevent the RuleSet from going into an infinite loop of resetting the breaker, when there was a short circuit in the Washing Machine. Such rules are also useful for initializing data for RuleSets as in the example in Figure 3.

In the absence of special syntax, it would be possible to encode the information that a rule is to be executed only once as follows:

Control Structure: While1
Temporary Vars: triedRule3;

...

IF ~triedRule3 *condition*$_1$ *condition*$_2$ **THEN triedRule3_T** *action*$_1$**;**

In this example, the variable **triedRule3** is used to control the rule so that it will be executed at most once in an invocation of a RuleSet. However, the prolific use of rules with such control clauses in large systems has led to the common complaint that control clauses in rule languages defeat the expressiveness and conciseness of the rules. For the case above, LOOPS provides a shorthand notation as follows:

{1} **IF** *condition*$_1$ *condition*$_2$ **THEN** *action*$_1$**;**

The brace notation means exactly the same thing in the example above, but it more concisely and clearly indicates that the rule executes only once. These rules are called "one shot" or "execute-once" rules.

In some cases, it is desired not only that a rule be executed at most once, but that it be tested at most once. This corresponds to the following:

Control Structure: While1
Temporary Vars: triedRule3;

...

IF ~triedRule3 triedRule3_T *condition*$_1$ *condition*$_2$ **THEN** *action*$_1$**;**

In this case, the rule will not be tried more than once even if some of the conditions fail the first time that it is tested. The LOOPS shorthand for these rules (pronounced "one shot bang") is

{1!} **IF** *condition*$_1$ *condition*$_2$ **THEN** *action*$_1$**;**

These rules are called "try-once" rules.

The two kinds of one-shot rules are our first examples of the use of meta-descriptions preceding the rule body in braces. See Section 1.7, "Saving an Audit Trail of Rule Invocation," for information on using meta-descriptions for describing the creation of audit trails.

1.6 First/Last Rules

It is sometimes useful to have rules which fire before or after the ordinary part of the RuleSet is invoked, independent of the form of the control structure. For example, in a DO1, such "FIRST " rules could be used for initialization. These now exist, and are notated by putting a {F} for a first rule in the MetaDescription field, and a {L} for a last rule. If a RuleSet has L rules which execute, the value of the RuleSet is the value of the last rule which executed.

1.7 Saving an Audit Trail of Rule Invocation

A basic property of knowledge-based systems is that they use knowledge to infer new facts from older ones. (Here we use the word "facts" as a neutral term, meaning any information derived or given, that is used by a reasoning system.) Over the past few years, it has become evident that reasoning systems need to keep track not only of their conclusions, but also of their reasoning steps. Consequently, the design of such systems has become an active research area in AI. The audit trail facilities of LOOPS support experimentation with systems that can not only use rules to make inferences, but also keep records of the inferential process itself.

1.7.1 Motivations and Applications

Debugging. In most expert systems, knowledge bases are developed over time and are the major investment. This places a premium on the use of tools and methods for identifying and correcting bugs in knowledge bases. By connecting a system's conclusions with the knowledge that it uses to derive them, audit trails can provide a substantial debugging aid. Audit trails provide a focused means of identifying potentially errorful knowledge in a problem solving context.

Explanation Facilities. Expert systems are often intended for use by people other than their creators, or by a group of people *pooling* their knowledge. An important consideration in validating expert systems is that reasoning should be *transparent*, that is, that a system should be able to give an account of its reasoning process. Facilities for doing this are sometimes called *explanation systems*

and the creation of powerful explanation systems is an active research area in AI and cognitive science. The audit trail mechanism provides an essential computational prerequisite for building such systems.

Belief Revision. Another active research area is the development of systems that can "change their minds". This characteristic is critical for systems that must reason from incomplete or errorful information. Such systems get leverage from their ability to make assumptions, and then to recover from bad assumptions by efficiently reorganizing their beliefs as new information is obtained. Research in this area ranges from work on non-monotonic logics, to a variety of approaches to belief revision. The facilities in the rule language make it convenient to use a user-defined calculus of belief revision, at whatever level of abstraction is appropriate for an application.

1.7.2 Overview of Audit Trail Implementation

When *audit mode* is specified for a RuleSet, the compilation of assignment statements on the right-hand sides of rules is altered so that audit records are created as a side-effect of the assignment of values to instance variables. Audit records are LOOPS objects, whose class is specified in RuleSet declarations. The audit records are connected with associated instance variables through the value of the **reason** properties of the variables.

Audit descriptions can be associated with a RuleSet as a whole, or with specific rules. Rule-specific audit information is specified in a property-list format in the meta-description associated with a rule. For example, this can include *certainty factor* information, categories of inference, or categories of support. Rule-specific information overrides RuleSet information.

During rule execution in audit mode, the audit information is evaluated after the rule's LHS has been satisfied and before the rule's RHS is applied. For each rule applied, a single audit record is created and then the audit information from the property list in the rule's meta-description is put into the corresponding instance variables of the audit record. The audit record is then linked to each of the instance variables that have been set on the RHS of the rule by way of the **reason** property of the instance variable.

Additional computations can be triggered by associating active values with either the audit record class or with the instance variables. For example, active values can be specified in the audit record classes in order to define a uniform set of side-effects for rules of the same category. In the following example, such an active value is used to carry out a "certainty factor" calculation.

1.7.3 An Example of Using Audit Trails

The following example illustrates one way to use the audit trail facilities. Figure 4 illustrates a RuleSet which is intended to capture the decisions for evaluating the potential purchase of a washing machine. As with any purchasing situation, this one includes the difficulty of incomplete information about the product. For example in this RuleSet, the reliability of the washing machine is estimated to be 0.5 in the absence of specific information from *Consumer Reports*. The meta-descriptions for the rules, which appear in braces, categorize them in terms of the *basis of belief* (the category *basis* is either a fact or estimate) and a *certainty factor (cf)* that is supposed to measure the "implication power" of the

rule. Within the braces, the variable on the left of the assignment statement is always interpreted as meaning a variable in the audit record, and the variables on the right are always interpreted as variables accessible within the RuleSet. This makes it straightforward to experiment with user-defined audit trails and experimental methods of belief revision. (Realistic belief revision systems are usually more sophisticated than this example.)

The result of running the RuleSet is an evaluation report for each candidate machine. Since the RuleSet was run in audit mode, each entry in the evaluation report is tagged with a reason that points to an audit record. Figure 5 illustrates the evaluation report for one machine and one of its audit records. In this example, each of the entries in the report has a reason and a cumulative certainty (cc) property in addition to the value. The value of the reason properties are audit records created as a side effect of running the RuleSet. The auditing process records the meta-description information of each rule in its audit record. This information can be used later for generating explanations or as a basis for belief revision. The auditing process can have side effects. For example, the active in the **cf** variable or the audit record performs a computation to maintain a calculated cumulative certainty in the reliability variable of the evaluation report.

The meta-descriptions for **basis** and **cf** are saved directly in the audit record. The *certainty factor* calculation in this combines information from the audit description with other information already associated with the object. To do this, the **cf** description triggers an active value inherited by the audit record from its class. This active value computes a *cumulative certainty* in the evaluation report. (Other variations on this idea would include certainty information descriptive of the premises of the rule.)

```
RuleSet Name: EvaluateWashingMachine;
WorkSpace Class:  EvaluationReport;
Control Structure:  doAll ;
Audit Class:  CFAuditRecord ;
Compiler Options:  A;

(* Rules for evaluating a potential washing machine for a
purchase.)

  .
  .
  .
  {(basics_Fact cf_1)}
  IF buyer:familySize>2 machine:capacity<20
  THEN suitability_'Poor;

  {(basics_Fact cf_.8)}
  reliability_(_($ ConsumerReports) GetFacts machine);

  {(basics_Estimate cf_.4)}
  IF 'reliability THEN reliability_.5;
  .
  .
  .
```

Figure 4. RuleSet Showing Evaluation

```
EvaluationReport "uid1"
expense:  510
```

```
suitability:  Poor cc 1 reason ...
reliability:  .5 cc .6 reason "uid2"
.
.
.
AuditRec "uid2"
rule:  "uid3"
basis:  Estimate;
cf:  #(.4 NIL PutCumulativeCertainity)
```

Figure 5. Example of an Audit Trail

1.8 Comparison with Other Rule Languages

This section considers the rationale behind the design of the LOOPS rule language, focusing on ways that it diverges from other rule languages. In general, this divergence was driven by the following observation:

When a rule is heavy with control information, it obscures the domain knowledge that the rule is intended to convey.

Rules are harder to create, understand, and modify when they contain too much control information. This observation led us to find ways to factor control information out of the rules.

1.8.1 The Rationale for Factoring Meta-Level Syntax

One of the most striking features of the syntax of the LOOPS rule language is the factored syntax for meta-descriptions, which provides information about the rules themselves. Traditional rule languages only factor rules into conditions on the left hand side (LHS) and actions on the right hand side (RHS), without general provisions for meta-descriptions.

Decision knowledge expressed in rules is most perspicuous when it is not mixed with other kinds knowledge, such as control knowledge. For example, the following rule:

IF ~triedRule4 pluggedInTo:voltage=0
THEN triedRule4_T breaker.Reset;

is more obscure than the corresponding one-shot rule from Figure 1:

{1} IF pluggedInTo:voltage=0 THEN breaker.Reset;

which factors the control information (that the rule is to be applied at most once) from the domain knowledge (about voltages and breakers). In the LOOPS rule language, a meta-description (MD) is specified in braces in front of the LHS of a rule. For another example, the following rule from Figure 4:

{(basis_Fact cf_.8)}

IF buyer:familySize>2 machine:capacity<20
THEN suitability_'Poor;

uses an MD to indicate that the rule has a particular **cf** ("certainty factor") and **basis** category for belief support. The MD in this example factors the description of the inference category of the rule from the action knowledge in the rule.

In a large knowledge-based system, a substantial amount of control information must be specified in order to preclude combinatorial explosions. Since earlier rule languages fail to provide a means for factoring meta-information, they must either mix it with the domain knowledge or express it outside the rule language. In the first option, intelligibility is degraded. In the second option, the transparency of the system is degraded because the knowledge is hidden.

1.8.2 The Rationale for RuleSet Hierarchy

Some advocates of production systems have praised the flatness of traditional production systems, and have resisted the imposition of any organization to the rules. The flat organization is sometimes touted as making it *easy to add rules*. The argument is that other organizations diminish the power of pattern-directed invocation and make it more complicated to add a rule.

In designing LOOPS, we have tended to discount these arguments. We observe that there is no inherent property of production systems that can make rules additive. Rather, *additivity* is a consequence of the independence of particular sets of rules. Such independence is seldom achieved in large *sets* of rules. When rules are dependent, rule invocation needs to be carefully ordered.

Advocates of a flat organization tend to organize large programs as a single very large production system. In practice, most builders of production systems have found it essential to create groups of rules.

Grouping of rules in flat systems can be achieved in part by using *context* clauses in the rules. Context clauses are clauses inserted into the rules which are used to alter the flow of control by naming the context explicitly. Rules in the same "context" all contain an extra clause in their conditions that compares the context of the rules with a current context. Other rules redirect control by switching the current context. Unfortunately, this approach does not conveniently lend itself to the reuse of groups of rules by different parts of a program. Although context clauses admit the creation of "subroutine contexts", they require you to explicitly program a stack of return locations in cases where contexts are invoked from more than one place. The decision to use an implicit calling stack for RuleSet invocation in LOOPS is another example of the our desire to simplify the rules by factoring out control information.

1.8.3 The Rationale for RuleSet Control Structures

Production languages are sometimes described as having a *recognize-act cycle*, which specifies how rules are selected for execution. An important part of this cycle is the *conflict resolution strategy*, which specifies how to choose a production rule when several rules have conditions that are satisfied. For example, the **OPS5** production language has a conflict resolution strategy (**MEA**) which prevents rules

from being invoked more than once, prioritizes rules according to the recency of a change to the data, and gives preference to production rules with the most specific conditions.

In designing the rule language for LOOPS, we have favored the use of a small number of specialized control structures to the use of a single complex conflict resolution strategy. In so doing, we have drawn on some control structures in common use in familiar programming languages. For example, **Do1** is like Lisp's **COND**, **DoAll** is like Lisp's **PROG**, **WhileAll** is similar to **WHILE** statements in many programming languages.

The specialized control structures are intended for concisely representing programs with different control relationships among the rules. For example, the **DoAll** control structure is useful for rules whose effects are intended to be additive and the **Do1** control structure is appropriate for specifying mutually exclusive actions. Without some kind of iterative control structure that allows rules to be executed more than once, it would be impossible to write a simulation program such as the washing machine simulation in Figure 1.

We have resisted a reductionist argument for having only one control structure for all programming. For example, it could be argued that the control structure **Do1** is not strictly necessary because any RuleSet that uses **Do1** could be rewritten using **DoAll**. For example, the rules

Control Structure: Do1;

IF a_1 b_1 c_1 **THEN** d_1 e_1**;**
IF a_2 b_2 c_2 **THEN** d_2 e_2**;**
IF a_3 b_3 c_3 **THEN** d_3 e_3**;**

could be written alternatively as

Control Structure: DoAll;
Task Vars: firedSomeRule;

IF a_1 b_1 c_1 **THEN firedSomeRule_T** d_1 e_1**;**
IF ~firedSomeRule a_2 b_2 c_2 **THEN firedSomeRule_T** d_2 e_2**;**
IF ~firedSomeRule a_3 b_3 c_3 **THEN firedSomeRule_T** d_3 e_3**;**

However, the **Do1** control structure admits a much more concise expression of mutually exclusive actions. In the example above, the **Do1** control structure makes it possible to abbreviate the rule conditions to reflect the assumption that earlier rules in the RuleSet were not satisfied.

For some particular sets of rules the conditions are naturally mutually exclusive. Even for these rules **Do1** can yield additional conciseness. For example, the rules:

Control Structure: Do1;

IF a_1 b_1 c_1 **THEN** d_1 e_1**;**
IF ~a_1 b_1 c_1 **THEN** d_2 e_2**;**
IF ~a_1 **~**b_1 c_1 **THEN** d_3 e_3**;**

can be written as

Control Structure: Do1;

IF a_1 b_1 c_1 **THEN** d_1 e_1;
IF b_1 c_1 **THEN** d_2 e_2;
IF c_1 **THEN** d_3 e_3;

Similarly it could be argued that the **Do1** and **DoAll** control structures are not strictly necessary because such RuleSets can always be written in terms of **While1** and **WhileAll**. Following this reductionism to its end, we can observe that every RuleSet could be re-written in terms of **WhileAll**.

1.8.4 The Rationale for an Integrated Programming Environment

RuleSets in LOOPS are integrated with procedure-oriented, object-oriented, and data-oriented programming paradigms. In contrast to single-paradigm rule systems, this integration has two major benefits. It facilitates the construction of programs which don't entirely fit the rule-oriented paradigm. Rule-oriented programming can be used selectively for representing just the appropriate decision-making knowledge in a large program. Integration also makes it convenient to use the other paradigms to help organize the interactions between RuleSets.

Using the object-oriented paradigm, RuleSets can be invoked as methods for LOOPS objects. Figure 6 illustrates the installation of the RuleSet **SimulateWashingMachineRules** to carry out the **Simulate** method for instances of the class **WashingMachine**. This definition of the class **WashingMachine** specifies that Lisp functions are to be invoked for Fill and Wash messages. For example, the Lisp function **WashingMachine.Fill** is to be applied when a Fill message is received. When a Simulate message is received, the RuleSet **SimulateWashingMachineRules** is to be invoked with the washing machine as its work space. Simulate message to invoke the RuleSet may be sent by any LOOPS program, including other RuleSets.

The use of object-oriented paradigm is facilitated by special RuleSet syntax for sending messages to objects, and for manipulating the data in LOOPS objects. In addition, RuleSets, work spaces, and tasks are implemented as LOOPS objects.

```
[DEFCLASS WashingMachine
  (MetaClass Class Edited (* "rtk: 12-Jun-87 07:57")
    doc (* Home appliance for wachine cloothes.))
  (Supers ElectricalDevice PlumbedDevice CleaningDevice)
  (ClassVariables)
  (InstanceVariables
    (controlSetting Meduim
      doc (* One of Small, Medium, Large, ExtraLarge))...)
  (Methods
    (Fill WashingMachine.Fill doc (* Fill the tub with water.))
    (Wash WashingMachine.Wash doc (* Perofrm the wash cycle.))
    (Simulate UseRuleSet RuleSet SimulateWashingMachineRules)
    .
    .
    .
]
```

Figure 6. RuleSet Invoked as a Method

Using the data-oriented paradigm, RuleSets can be installed in active values so that they are triggered by side-effect when LOOPS programs get or put data in objects. For example:

```
[DEFINST WashingMachine (StefiksMaytagWasher "uid2")
  (controlSetting RegularFabric)
  (loadSetting #(Medium NIL RSPut) RSPutFn CheckOverLoadRules)
  (waterLevelSensor "uid3")
]
```

The above code illustrates a RuleSet named **CheckOverLoadRules** which is triggered whenever a program changes the **loadSetting** variable in the **WashingMachine** instance in the figure. This data-oriented triggering can be caused by any LOOPS program when it changes the variable, whether or not that program is written in the rules language.