Table of Contents

List of Figures

Abstract

PSRL is a production-rule interpreter based on the primitives and knowledge representation of the SRL/1.5 language. It adds a procedural capability to SRL's declarative features, and may become the basis for a complete control environment for SRL users. The current version includes a facility for converting OPS5 rules into PSRL schemata. They can then be run with an SRL database context as the working memory. Many OPS5 debugging and tracing functions have been implemented in PSRL. Being based on SRL gives PSRL advantages over OPS5 in allowing relations among data elements, user-defined inheritance of values, default values, automatic type-restriction testing on values, and the availability of meta-information on schemata, slots and values. In addition, the user has SRL demons, a large database capability and SRL contexts for considering alternative models. PSRL encourages hierarchical organization of small modular sets of rules, each with its own control strategy. SRL demons can be used to control the execution of rule sets. PSRL is implemented so that its facilities can be reconfigured in the service of a variety of problem-solving architectures.

Acknowledgments

Mark Fox, Pat Langley, and Mark Wright were co-authors of the Preliminary Design of PSRL, and this paper borrows from some sections of that document. Digital Equipment Corporation is the primary supporter of this research. Michael Greenberg wrote the first version of the PSRL matcher code. Elaine Kant provided useful comments on the introductory overview of PSRL. The users whose projects are mentioned here have provided valuable feedback on the system, well before the manual was finished; in particular, Mary Lou Maher, Joe Mattis and Arvind Sathi have helped to make PSRL more useful. The last three have also supplied written descriptions that were used to construct the chapter on applications. I am grateful to Mary Lou Maher, Arvind Sathi, Duvvuru Sriram and Stephen Smith for many helpful comments on the manual. Jill Fain, Duvvuru Sriram, Mark Fox, and Arvind Sathi provided crucial assistance in the comparisons of PSRL to ROSIE and LOOPS.

1. Introduction and Summary

PSRL combines the production system approach of data-directed control [13, 14, 5, 15] with the features of SRL, the Schema Representation Language of Fox, et al [8, 16]. SRL is a Lisp-based declarative formalism for representing complex, structured objects and their interrelations. Rules in PSRL are expressed in an extension of Forgy's OPS5 [6] language syntax, $\mathrm{PSRL}_{\mathrm{OPS8}}$, and many of the OPS5 user commands are available, so that the running environment of PSRL is similar to that of OPS5.[1] Since rules are translated into schemata, and operate with respect to schemata, the PSRL environment also includes the full set of SRL commands and utilities. PSRL is much less efficient than OPS5 (it is currently run interpretively, with an optimizing compiler still being planned), so that it is impractical to deal with large rule sets. Instead of executing rules in large sets, they are partitioned into small packets that are stored in schemata, organized using SRL relations, and triggered by SRL demons.

Given the variety of ways of operating within SRL (including a version of OPS5 with an interface for transferring data to and from SRL schemata), PSRL seems most appropriate for monitoring and manipulations that involve bringing together and combining information in an associative, pattern-directed way, from a number of schemata. PSRL is less appropriate for single-schema operations and operations where specific schemata (as opposed to classes or sets with similar properties) are used. It is currently being used in a high-rise building design system, in a project management system with heuristic rules, and in a knowledge-based graphics system (these are described in more detail below).

1.1. Data Memory

PSRL rules access and manipulate schemata within the scope of entire SRL database contexts, inheriting considerable representational power from SRL. An SRL user can define new relations among objects, allowing various kinds of information to be inherited automatically and according to user-defined constraints. SRL contains facilities for automatically checking the validity of values, and for executing procedures (demons) as a result of schema operations. The SRL database system provides automatic swapping of schemata to disk files, so that very large collections of them are manageable. Thus PSRL rules can deal with a much larger and more permanent database than is possible in OPS5. Databases are usually built up by other means, with PSRL applied to the resulting mass of data to perform bookkeeping, maintenance, and communication operations.

1.2. Production Memory

PSRL interprets rules whose syntax closely resembles that of OPS5. A translator takes the OPS5-like rules and builds schemata to represent them internally. Rules are grouped into sets, and stored in "production-system" schemata, whose names can be assigned to global variables or stored inside a user's schemata and demons. Features have been added to extend the syntax beyond OPS5: allowing specific, named schemata to be matched; allowing schemata to specify direct relationships to other schemata (OPS5 only allows indirect pointers); and allowing restrictions to be placed on class names and on element variables. Element variables (the ones in braces surrounding condition elements) can be used freely inside elements, for matching and modification purposes. The user can specify that inheritance of values is to be allowed in matching and that matching is to be done with respect to particular contexts within an SRL database.

---

[1] OPS5 is well known; OPS6 existed as a design only; OPS7 is currently only an experimental implementation in Pascal, for the PERQ.

A direct link to Lisp functions is provided in the 'eval' action function, which takes a Lisp S-expression, substitutes values for OPS variables, and then evaluates it. The OPS5 Lisp interface is not implemented, since SRL functions provide equivalent power, in combination with the eval action function. Demons can be attached to classes of schemata to provide another sort of procedural escape. Using either the eval function or demons, a user can readily have rule-sets executed recursively from within the actions of rules.

1.3. Control

Rule sets can be invoked by demons attached to schemata, or by Lisp function calls. Each rule set has associated with it a control function that determines what style of conflict resolution is to be used during its execution. A number of options are supplied, and the user can code others, as needed. Some examples of existing control functions are: cycling repeatedly through a list, going back to the beginning after each successful rule firing; sequencing through a list, executing all possible matches of each rule, then going on to the next; stopping execution after one rule has fired; and cycling through the list of productions in reverse order.

1.4. The Interpreter

Most of the tracing and debugging facilities available in OPS5 are also provided to PSRL users. In addition, SRL-proficient users can attach special-purpose demon functions for monitoring and tracing purposes, as needed.

1.5. Extensions

Many problem-solving architectures other than the basic production system recognize-act cycle are possible, using the basic modules of PSRL. PSRL also compares favorably with other similar systems such as LOOPS [1] and ROSIE [4], especially in regard to making full use of the power of SRL.

1.6. Prerequisites

Some familiarity with the OPS5 language [6] will be helpful, but not required, here. The basic prerequisite is familiarity with AI programming, pattern matching and production systems [2, 3, 15], and with the SRL/1.5 language and concepts [16]. Examples, especially the extended one in Chapter 3, are intended to be helpful to production-system novices in reviewing the central concepts.

1.7. Manual Organization

Chapter 2 discusses PSRL from the standpoint of its representation as SRL schemata. Most system aspects are described declaratively, and the schemata precisely delineate the full power of expressions in PSRL. Dealing exclusively with PSRL at the SRL level, however, seems cumbersome in comparison to the $\text{PSRL}_{\text{OPS8}}$ notational level, as described in the later chapters. Chapter 3 introduces the $\text{PSRL}_{\text{OPS8}}$ notation, first presenting an extended example of how an OPS8 rule is represented as SRL schemata, and then proceeding with full details on various declarations and on the options available for expressing rules in OPS8. Chapter 4 gives details on the user commands available in PSRL. The first few sections fill in details on loading and running the system, and on setting up the control of rule set execution. Then come details on tracing, debugging and user interface.

Chapter 5 briefly sketches several developing systems that are using PSRL, in order to illustrate some of the possibilities. It also compares PSRL to several other rule-based representation systems. Chapter 6 indicates some directions for further development of PSRL.

Chapter 7 contains several illustrative runs of PSRL, showing examples of execution of most of the important PSRL commands. These appendices are intended to present selected features of the system in a natural user-oriented ordering, in contrast to the main chapters whose organization is along logical, structural, or taxonomic lines.

2. Overview of PSRL Representation

The main concepts involved in production systems are:

- Production system: a set of rules that are considered together as a procedure for performing a data-directed computation;
- Production, or Rule: a condition-action (or IF-THEN) pair, specifying a data condition under which the action of the rule is performed;
- Condition, or left-hand-side (LHS): a pattern or abstraction specifying a data state or a class of possible data states;
- Action, or right-hand-side (RHS): a sequence of data manipulations to be performed;
- Architecture: the combination of a pattern matcher for conditions, procedures for executing actions, a conflict resolution or other control procedure, and various parameters that affect the operation of the interpreter; and
- Environment: the combination of architecture, production system, working memory, and other database parameters that are involved with the execution of a production system.

In PSRL, SRL schemata are used to specify all of the above. The current implementation does not make use of the last two schemata while it is running, since only one value is available for each slot. Later versions, however, are expected to use them for a variety of special-purpose configurations.

This chapter will sketch the concepts of PSRL at the SRL level. The progression is generally from high-level schemata, such as production system architecture, to low-level ones, such as templates. As mentioned above, users of PSRL will probably prefer to do most manipulation at the $\text{PSRL}_{\text{OPS8}}$ level (see Chapter 3), due to its greater convenience - many schemata can be involved in the SRL definition of even a single rule, making it cumbersome to work at that level. So the purpose of this chapter is to provide some insight as to how rules and systems can be represented internally, as a basis for understanding how the $\text{PSRL}_{\text{OPS8}}$ notation is interpreted. Some readers may prefer to skim this and refer back to it at later points for clarification.

2.1. Production System Architecture

The slots of a **ps-architecture** schema, Figure 2-1,[2] specify the functions that the PSRL interpreter should use to interpret rules according to that architecture. The LHS-MATCHER (left-hand-side-matcher) is applied to the 'if' or 'condition' parts of rules to determine which conditions are satisfied by the current state of the database. The RHS-EXECUTER is a function to perform the right-hand-side actions of rules. The CONFLICT-RESOLUTION slot has a function for choosing among the set of matched rules, for a subset of rules whose actions are to be performed. The repeated application of the functions specified in the **ps-architecture** as just described is the traditional recognize-act cycle of production systems.

[2]The fonts used here are as in other SRL reports: **bold** for schema names, SMALL CAPITALS for slot names, and *italics* for facet names.

```
{{ ps-architecture
      LHS-MATCHER:
      RHS-EXECUTER:
      CONFLICT-RESOLUTION: }}
```

Figure 2-1: **ps-architecture** Schema

As mentioned above, the current PSRL$_{OPS8}$ interpreter does not make use of the **ps-architecture** schema, but simply operates according to **psrl-ops8**, shown in Figure 2-2. The above description of the recognize-act cycle is only an approximation, as regards the current implementation.

```
{{ psrl-ops8
      IS-A: "ps-architecture"
      LHS-MATCHER: fire-m
      RHS-EXECUTER: fire-x
      CONFLICT-RESOLUTION: update-conflicts}}
```

Figure 2-2: **psrl-ops8** Schema

Each set of PSRL rules (each 'method' or 'PS module') is organized by a schema that is an instance of the one shown in Figure 2-3.

```
{{ production-system
      RULE-SET:
          restriction: (set (or (type "instance" "rule")
                                (type "instance" "production-system")))
      NUMBERS:
      CONTROL: }}
```

Figure 2-3: **production-system** Schema

The value of the RULE-SET slot names the rules that are contained in the specific system; the set of elements is taken to be ordered. If an element of a RULE-SET is a production system rather than a rule, then that production system is interpreted (its entire RULE-SET is processed according to its CONTROL slot) whenever a rule would be, in that position in the set. The NUMBERS slot specifies numerical parameters controlling the interpretation of the rules in the **production-system**, as in the Langley's PRISM system [9]. It is unused at present. The CONTROL slot specifies a function for the recognize-act cycle for the given **production-system**. A number of different possibilities are currently available, described in Section 4.6. A detailed example of a production system is given in Section 3.1.

2.2. Multiple environments for production system execution

PSRL will eventually allow multiple environments in which rules are evaluated (Figure 2-4). An environment would include all the 'dynamic' aspects of PS execution, in contrast to production-system, rule, etc., which define relatively 'static' aspects. In fact, the environment is the only place that such dynamic information is placed, in order to cleanly separate it from rules and other static entities. Think of rule sets as functions, while environments specify the data to which the functions are applied - there could obviously be more than one such application going on at the same time in a computation. Thus, separate activations of the same set of production rules would be specified by different environments. Each environment would maintain its own rule set, in its **production-system** schema, and its own conflict set. The user would define environments, activate and deactivate them, and delete them. (A later version may allow parallel execution.) In the current version of PSRL$_{OPS8}$, only one environment is active, and it is specified internally rather than using a schema.

```
{{ environment
      PS-ARCHITECTURE:
      PRODUCTION-SYSTEM:
      CONTEXT:
      AGENDA:
      STATUS: }}
```

Figure 2-4: environment Schema

The slots in an environment would indicate how one would start up and run a PS. One slot would contain an instance of the **ps-architecture** schema, described above. Another would contain the name of a production-system, which includes a set of rules. The CONTEXT slot would indicate the SRL database context in which interpretation of the rules would take place.[3] The AGENDA and STATUS slots would hold other information regarding the dynamic execution within an environment, namely currently uncompleted actions within the rule-set (used, for example, when a rule-set's execution is pre-empted or suspended by that of another rule-set) and the current execution status (e.g., active, suspended, or completed).

2.3. Rule Structure

A rule is specified as an instance of the **rule** schema, Figure 2-5. The PSRL-LHS slot contains a set of elements that are interpreted as conditions, being instances of **lhs-element**. The PSRL-RHS slot contains a set of elements, instances of **rhs-element**, that are interpreted as actions to be executed - an "<action>" is one of a set of predefined schemata (including user-defined ones). Details on these appear in the next sections. A third slot is added by the current interpreter, PSRL-VARIABLES, whose value is a list of the match variables in the rule, alphabetized.

[3] In version 1.3, it will be the name of the working memory context, as described in the $PSRL-CONTEXT global variable, described below; rules will be stored in contexts according to the variable $PSRL-RULE-CONTEXT.

```
{{ rule
     PSRL-LHS:
          restriction: (set (type "instance" "lhs-element"))
     PSRL-RHS:
          restriction: (set (type "psrl-action" "<action>")) }}
```

Figure 2-5: rule Schema

2.4. Left-hand-side element schemata

The PSRL-LHS slot of a **rule** is filled by an ordered set of instances of the **lhs-element** schema (Figure 2-6).

```
{{ lhs-element
     TEMPLATE:
     NEGATION:
          restriction: (or t nil)
          default: nil
     SCHEMA-VARIABLE:
     INDEX:
          restriction: (or t nil)
          default: t
     MATCH-SET: }}
```

Figure 2-6: lhs-element Schema

The TEMPLATE slot contains a schema that is to be matched against all schemata in the current PSRL context. Slots and values in a template act as a pattern, successfully matching all schemata that have similar slots and values, along with others not mentioned in the template. The NEGATION slot can reverse the logical sense of the match, if filled with a 't' value: the lhs-element will succeed in matching whenever the template does NOT have any matches, and vice versa. When a match succeeds, the schemata that matched are bound to the value of the schema-variable slot, if it is an instance of **match-variable** (described in Section 2.5). That binding can then be used in later templates within the same left-hand-side. If the variable was bound previously, its binding, a schema name, is checked to ensure that it matches the present template. This allows a schema to be obtained in a relational slot in one schema and then matched in detail in a later template, e.g., to force it to have some further properties. Other match features exist, and will be described in Section 3.4. (The example in Section 3.1 may also be helpful.)

The INDEX slot, unused at present, may be used in a future PSRL version to improve matching efficiency, by enabling the recording of possible matches. The list of matching schemata would be stored in the MATCH-SET slot.

A template is an image of a schema, with variables allowed in some or all of its slot values, and with different restrictions, defaults, etc. specifiable on the slots. A sketch of a template is given in 2-7. In real applications, the "<xxx>" schema names would be replaced by names of actual corresponding types of schemata. Templates, for instance, have unique names generated by the PSRL$_{OPS8}$ translator.

```
{{<template>
    TEMPLATE+INV: "<lhs-element-containing-this>"
    PSRL-INSTANCE: "<element-class-to-be-matched>"
    <SLOT1>: "<match-variable>"
        range: <range-spec>
    <SLOT2>: <value>
    etc.}}
```

Figure 2-7: <template> Schema

A relation must be included that links a template with the class of schemata that it can match; the default relation to serve this purpose is **psrl-instance**, shown in 2-8. (The user can change the relation to be used, by setting the variable $psrl-instance, as described in Section 4.5.) The **psrl-instance** relation has a limited transitivity and is used internally in PSRL in such a way as to limit the search paths associated with it, in the hope of making **psrl-instance** efficient enough to be used in its role as a syntactic relation. The <element-class-to-be-matched> in a template can be an existing SRL schema, or it can be specified with the PSRL$_{OPS8}$ literalize declaration.

```
{{psrl-instance
    IS-A: "relation"
    INCLUSION: "is-a-inclusion-spec"
    FUNCTION: is-a-fn
    TRANSITIVITY: (step "psrl-instance" all t)
    INVERSE: "psrl-instance+inv"}}
```

Figure 2-8: psrl-instance Schema

Two important features that templates may have are: a <range-spec> that restricts the values to be matched in the slot; and a <match-variable> to be bound to the matched value, or whose binding (if it is already bound) serves to restrict any further bindings. A slot with empty value specifies to the matcher that the slot must be defined, but may not yet have a value. The presence of a <match-variable> or other expression or <range-spec> implies that the slot must have a value in order to match.

A schema matches a template if the schema has all of the relations specified in the template and if values of other slots in the template match the corresponding ones in the schema. Values match if they are identical or if a variable from the template is either unbound (it will be bound after matching) or bound to the identical value. If there is a *range* attached to any of the slots, those must also be satisfied by the value in the schema (which is the value bound to the match variable, if any). There is a special value, '{}', that allows a slot in a template to match any value at all in the schema, including the case where a slot is defined but has no value.

The process of matching and the ways of specifying rule elements will be clarified in later chapters, by way of examples (see, e.g., Sections 3.1 and 3.4).

A *range-spec* is an expression specified by:[4]

```
range-spec              ::= (AND range-spec*)
                        ::= (OR range-spec+)
                        ::= (NOT range-spec+)
                        ::= (QUOTE atomic-value)
                        ::= (= atomic-value)
                        ::= (<> atomic-value)
                        ::= (<=> atomic-value)
                        ::= (< atomic-value)
                        ::= (> atomic-value)
                        ::= (<= atomic-value)
                        ::= (>= atomic-value)
                        ::= (FUNCTION Lisp-function-name)
                        ::= (FUNCTION Lisp-lambda-expression)
                        ::= (TYPE schema-name schema-name)
                        ::= atomic-value

atomic-value            ::= constant-symbolic-atom
                        ::= variable
                        ::= number

schema-name             ::= SRL/1.5-double-quoted-schema-name

constant-symbolic-atom  ::= SRL/1.5-schema-name-without-double-quotes
```

These terms are not defined formally here: *Lisp-function-name*, *Lisp-lambda-expression*, *number*, *variable*, *SRL/1.5-double-quoted-schema-name* and *SRL/1.5-schema-name-without-double-quotes*. A *number* is a Franz Lisp or Common Lisp number. A *variable* is an atom beginning with '<' and ending with '>', as in OPS5.

Range-spec expressions serve the same purpose as corresponding elements of the OPS5 language, testing matched values with various logical (AND, OR, NOT) predicates, as in SRL restrictions, and arithmetic (=, <>, <=>, <, >, <=, >=) predicates as in OPS5, which are described more fully in Section 3.4.2. When match-variables occur in the expressions, their bound values are substituted before the predicates are evaluated, except inside a QUOTE list. The FUNCTION option allows a one-argument function to be evaluated, given as its argument the current value being matched. If the function returns a non-nil value, the match succeeds and continues. (There are user-callable functions for accessing values of match-variables, as described in Section 3.4.2, so that predicates can obtain other arguments from the matching context.) The TYPE option uses the SRL function **r-test**.

[4] These grammar conventions (cf. OPS5 manual) are used here: an *italics typefont* is used for non-terminals in the grammar; **this typefont** is used for literal symbols; superscript asterisk (*) means 0 or more of the item; superscript plus (+) means one or more.

2.5. Match variable schema

Schemata that are to be considered variables during pattern matching must be instances of the **match-variable** schema, Figure 2-9. (There is a function to declare such schemata, described in Section 4.11, but PSRL$_{OPS8}$ uses its grammar to declare most variables implicitly, so the user doesn't usually need to be concerned about declaring them.) The VALUE slot is used to store a variable's binding. The TYPE slot is unused, but may be used later for specifying other semantics of variable matching. More on the process of matching and the use of variables is given in Section 3.1. See also Section 3.4.

```
{{ match-variable
     VALUE:
     TYPE: }}
```

Figure 2-9: match-variable Schema

2.6. Right-hand-side element schemata

The PSRL-RHS slot of a **rule** is filled by an ordered set of instances of the **rhs-element** schema (Figure 2-10). The PSRL-ACTION slot has the name of the action to be done, e.g., "make" or "modify". There is a predefined set of such 'generic' actions that can fill the PSRL-ACTION slot, and the user can define others by creating corresponding schemata. Many actions include a PSRL-INSTANCE slot for various purposes, usually to name a schema class. The PSRL-INTERP-FN slot names a function used to interpret the action - interp-action interprets an arbitrary action (by dispatching to other interpretation functions).

```
{{ rhs-element
     PSRL-ACTION:
     PSRL-INSTANCE:
     PSRL-INTERP-FN: interp-action}}
```

Figure 2-10: rhs-element Schema

An example of a generic action schema is in Figure 2-11. It has two of the slots described for the **rhs-element** schema, and one new one, PSRL-MAKE-NAME, which indicates the name of the schema to be made (an option of the PSRL$_{OPS8}$ syntax, to be described in Section 3.5.1).

```
{{ make
     PSRL-INSTANCE:
     PSRL-MAKE-NAME:
     PSRL-INTERP-FN: interp-make}}
```

Figure 2-11: make Schema

Notice that conditions and actions in PSRL are not parallel in their definitions. Conditions are two-level structures (lhs-element and template), while actions are a single level. Actions however must fit into a set of pre-defined classes, while conditions express patterns on arbitrary schemata.

The full set of actions is given below, in Section 3.5.1.

3. Representing Rules in PSRLOPS8 Notation

The first section of this chapter introduces the syntax of PSRL rules by an extended example that covers the main features. The second and succeeding sections will define formally and semantically the $\mathrm{PSRL}_{\mathrm{OPS8}}$ notation. Some aspects of the ways that the language can be used, namely those that are parameterized by Lisp global variables, are postponed until Chapter 4, where user commands are discussed. What is focussed on here is the external notation and its default translation and interpretation.

3.1. An example: Mapping from OPS8 to schemata

This section illustrates the structure of PSRL rules and the correspondence to the $\mathrm{PSRL}_{\mathrm{OPS8}}$ by going through a translation of a $\mathrm{PSRL}_{\mathrm{OPS8}}$ rule. Along the way, some features of the language syntax are introduced. The practical aspects of how to execute rules are postponed until Chapter 4.

The rule to be used, named **beg** (short for begin), is taken from a simple auto factory simulation written in OPS5. The rule, as it appears in this section, has been augmented slightly with a couple of other features for illustrative purposes. Appendix 7.4 includes a run with this rule and others executing in PSRL. The **production-system** containing the rule is shown in Figure 3-1. This schema is usually created automatically when a rule file is loaded (see Section 4.5), and its slots are filled as rules are defined using 'p' forms and control declarations. These declarations and commands are described in Section 4.5, but this schema can be taken as given for the purposes of this section.

```
{{ autosel
     INSTANCE: "production-system"
     RULE-SET: "buy2" "beg" "clu" "eng" "end"
     CONTROL: fire-sequence}}
```

Figure 3-1: autosel Schema

The **beg** rule, in its OPS8 format, is the following:

```
[p beg
   { (clock ↑mod going ↑timer <t>) <c>}
   - (automac ↑mod busy ↑machine engine)
     (steel ↑what scrap ↑amount > 4)
-->
     (bind <n>)
     (make auto ↑mod order ↑serial <n>)
     (modify <c> ↑timer (compute <t> + 1)) ]
```

This rule recognizes that the clock is running (first condition element) and that there is no currently busy machine making an engine (second condition element). Under these conditions, if there is enough scrap steel (third element), the rule fires, creating an order to manufacture an auto. It also updates the clock's time by adding one (this is an extra action not contained in the original **beg** rule). The rule's schema is in Figure 3-2.

```
{{beg
    INSTANCE: "rule"
    PSRL-LHS: "begd" "begc2" "begc3"
    PSRL-RHS: "begat" "bega2" "begaS"
    PSRL-VARIABLES: "<C>" "<n>" "<t>"}}
```

Figure 3-2: beg Schema

The first condition in the lhs is

 { (clock tfnod going ttimer <t>) <c>}

whose translation is shown in Figures 3-3 and 3-4. The first schema (begd) records the element variable, <c>, in the SCHEMA-VARIABLE slot, and points to the second schema (begdt), using the TEMPLATE slot. The second schema has a slot for each of the OPS5 attributes (preceded by 't'). There is also a PSRL-INSTANCE slot[5] for the class name, 'clock', and an inverse pointer from the template back to the lhs-element schema.

In the process of executing a production system containing the beg rule, this condition element will match all schemata in the current SRL database context that are related to the clock schema by the psrl-instance relation, and that have the atom 'going' in their MOD slot and any value at all in their TIMER slot After matching, the match-variable <t> will be bound (i.e., will have as the value of its VALUE slot) to the value of the TIMER slot of each matched schema. Also the variable <c> will be bound to the name of the schema matched. If more than one schema matches this first condition element, then the other condition elements in beg will be matched in turn using each distinct assignment of values to match-variables.

```
{{ begd
    INSTANCE: "lhs-element"
    TEMPLATE: "begdt"
    SCHEMA-VARIABLE: "<C>"}}
```

Figure 3-3: begd Schema

[5] As mentioned above, this relation for templates can be changed by the user.

```
{{ begc1t
      TEMPLATE+INV: "begc1"
      PSRL-INSTANCE: "clock"
      MOD: going
      TIMER: "<t>"}}
```

Figure 3-4: begc1t Schema

The second element in the lhs is,

`- (automac ↑mod busy ↑machine engine)`

whose translation is in Figures 3-5 and 3-6. The new feature in this element is the negation, indicating that the match will succeed if this element's match fails, i.e., if there are no database schemata matching it. This is indicated by the NEGATION slot with value 't'.

```
{{ begc2
      INSTANCE: "lhs-element"
      TEMPLATE: "begc2t"
      NEGATION: t}}
```

Figure 3-5: begc2 Schema

```
{{ begc2t
      TEMPLATE+INV: "begc2"
      PSRL-INSTANCE: "automac"
      MOD: busy
      MACHINE: engine}}
```

Figure 3-6: begc2t Schema

The third element in the lhs,

`(steel ↑what scrap ↑amount > 4)`

has another new feature, the comparison of numerical values. The condition states that the instance of the steel schema to be matched must have an AMOUNT slot value greater than 4, and this expression is stored on the *range* facet of that slot in **begc3t**, Figure 3-8. Notice that the slot AMOUNT has as its value the atom '{}', which in OPS5 and PSRL$_{OPS8}$ will match to any value. In some cases, a user will want to bind the value matched here (e.g., something greater than 4) to a variable, so that the variable name would take the place of the '{}', and the pattern would be expressed as '↑amount {<x> > 4}'.

```
{{ begc3
      INSTANCE: "lds-element"
      TEMPLATE: "begcSt"}}
```

Figure 3-7: begc3 Schema

```
{{ begc3t
      TEMPLATE + INV: "begc3"
      PSRL-INSTANCE: "steel"
      WHAT: scrap
      AMOUNT: {}
          range: (> 4)}}
```

Figure 3-8: begc3t Schema

The right-hand-side or action part of the beg rule starts out,

```
--> (bind <n>)
```

which is an action that binds the variable <n> to a new 'random' schema. (It can bind other values, if they are specified as expressions following the variable name, but in this case a default expression analogous to (gensym) is assumed, since nothing is given.) The schema for this action is in Figure 3-9, and has the action name "bind" as the value of the PSRL-ACTION slot, "<n>" as the value of the VARIABLE slot, and no value for the RHS-TERMS slot These slot names are specific to the bind action - each PSRLopgg action has its own particular set of slots, though some share the same ones, and ail have the PSRL-ACTION slot naming the particular action involved.

```
{{ begai
      PSRL-ACTION: "bind"
      VARIABLE: "<n>"
      RHS-TERMS: }}
```

Figure 3-9: begat Schema

The second action of beg is

```
(make auto tiiod order tserial <n>)
```

whose translation is in Figure 3-10. The make action creates a schema with a gensym'd name that is PSRL-IMSTANCE of the value of the PSRL-INSTANCE slot in the action schema (auto in this example). Other slots in the action schema specify slots and values to be created in the new schema. When the values are match variables (<n> in this case), their bound values from the match or from bind actions are fled in.

```
{{ bega2
     PSRL-ACTION: "make"
     PSRL-INSTANCE: "auto"
     MOD: order
     SERIAL: "<n>"}}
```

Figure 3-10: bega2 Schema

The third action,

(modify <c> ttimer (compute <t> +1))

is another type, modify, and is shown in Figure 3-11. The modify action works on an existing schema, specified by the value in the PSRL-INSTANCE slot; in this case it would be the value bound to <c>, i.e., the schema that matched the first lhs-element The other slots of the modify action schema indicate the slots in the modified schema that are to be changed, and their values specify new values in the changed schema. In this case, there is a compute expression to be evaluated, whose result will become the TIMER slot in the changed schema.

```
{{ bega3
     PSRL-ACTION: "modify"
     PSRL-INSTANCE: "<C>"
     TIMER: (Compute "<t>" + 1) }}
```

Figure 3-11: bega3 Schema

3.2. OPS8 language definition overview

As mentioned several times in the above discussion, $^{P}SRL_{OPS8}$ is based directly on OPS5. The grammar presentation below, in particular, is closely modelled on that of OPS5. Thus, for those readers who are familiar with OPS5, there are accompanying summaries of the differences of detail. PSRL is intended to be a compatible extension of OPS5, with the exception of some actions and matching features that are not appropriate in the SRL environment. Thus most of the differences here are new features that have been added.

All aspects of the $PSRL_{OPS8}$ notation are precisely described in a BNF grammar. The following grammar conventions (cf. OPS5 manual) are used here: an *italics typefont* is used for non-terminals in the grammar; this typefont is used for literal symbols; superscript asterisk (*) means 0 or more of the item, separated by spaces; superscript plus (+) means one or more.

The following is the top level of PSRL^^p^, listing the types of things that can occur in a program,[6]

[6] Some QPS5 constructs arc reoognizad and Ignored: strategy external literal

```
OPS8-entity          ::= literalize
                     ::= vector-attribute
                     ::= production-rule
                     ::= production-system-name
                     ::= control
                     ::= ( setq parameter Lisp-expression )
                     ::= ( debugging-command )
                     ::= ( action-command )
                     ::= Lisp-expression
                     ::= comment

comment              ::= ; any-string-terminated-by-newline.
```

All of the above *OPS8-entity*'s except *Lisp-expression* are defined in this chapter and the next. *Lisp-expression* is not defined formally in this manual, and is included to emphasize that $\text{PSRL}_{\text{OPS8}}$ files are interpreted using the standard Lisp read conventions.

3.3. Working Memory and Production Memory declarations

Working Memory in PSRL is loosely defined: rules access and manipulate an entire SRL database context, and thus the context would correspond to conventional Working Memory (WM). But for debugging purposes, a more restricted definition of WM is used, namely those schemata that have been created by the "make" action (either at the top level or inside rules). During matching, the SRL function **r-find** is used to access schemata, and recall that **r-find** depends on the presence in schemata of SRL relations.

Several features in PSRL make WM more powerful representationally than is the case in OPS5:

- SRL schemata are fundamentally more powerful than OPS5 attribute-value lists; in particular, a wider variety of types is available as values of slots, including lists of values, attachment of meta-information and restrictions, default values, and inherited values; in most cases, the $\text{PSRL}_{\text{OPS8}}$ notation does not yet exploit the full power of SRL; in particular, the matching of more than the first element of a list value is not yet supported (and its semantics is not understood, due to SRL's own lack of semantics for list values); in some other cases, e.g., inheritance, the power is not obtained notationally but through global variables.

- Relations can directly connect schemata, with a schema name appearing as a value in a slot, whereas in OPS5, WM elements are not named and can point to each other only indirectly through a shared symbol.

- Several WMs can be defined within a single SRL database by using the context facility; rules would match one or the other depending on a global variable to be described below.

- Very large databases are possible, since SRL has automatic swapping of schemata to disk when they exceed available main memory; while efficiency would prohibit PSRL access to huge databases, judicious subdivision into contexts would make handling a large database feasible.

Classes of WM schemata (for use in templates and in actions) are declared, as in OPS5, by a literalize declaration,

literalize ::= (literal1ze *class-name slot-name*$^{\bullet}$)

Such declarations should be executed before rules and WM-initializing commands are loaded, to be described in more detail in Chapter 4. In this, *class-name* and *slot-name* are Lisp symbolic atoms or numbers. The effect of a literalize is to define a general schema and its slots to represent the class of elements. For instance,

```
(literalize clock mod timer.maker)
```

defines the schema in Figure 3-12. Notice that a PSRL$_{OpS3}$ user can modify such schemata after defining them[7] (e.g., adding slots or attaching demons that could be inherited by instances), which is not possible in OPS5. There are some internal side-effects of a literalize, on the class-name's atom (not the schema).

```
{{ clock
     PSRL-INSTANCE; "literalize"
     MOD:
     TIMER:
     MAKER: }}
```

Figure 3-12: clock Schema

Also as in OPS5, a slot whose value is to be treated as a list must be declared with a vector-attribute declaration, as follows:

vector-attribute ::* (vector-attribute *slot-name*)*

For example,

```
(vector-attribute Items)
 • • •
(literalize checker name balance Items)
```

declares the value of the .Items' slot to be considered a list of values. Such a declaration has internal Lisp effects, and no schemata are created, but schemata for the attributes declared are modified with a vector-attribute slot. The order of appearance of vector-attribute and literalize declarations is not significant; the declaration applies to all other declarations, making any use of 'items' (for example) as an attribute in a literatize have the special interpretation. The number of vector-attributes in an element is not restricted to one, as in OPS5.

General note: it is not necessary or correct for a user of PSf?L$_{Qpsa}$ fo *put double-quotes (") around names in rules and declarations that are known to he schema names, EXCEPT in the case of ordinary values of slots. Variables are always automatically translated to schemata, as are atoms that appear in other locations inside elements and in declarations. Values of relations are automatically translated to schema names.*

Chapter 4 will discuss ways to change the interpretation of WM, along with display and debugging commands.

[7]Or ihey can be set up previously in a user's database, since the iterfitke declaration *mil* not disturb ar* exising schema.

PSRL: An SRL-Based Production Rule System

Reference Manual for PSRL Version 1.2

Michael D. Rychener

CMU-RI-TR-85-7

Intelligent Systems Laboratory
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

December 1984

This research has been supported by Digital Equipment Corporation.