NISTIR 7363
2006

Notes on an Information Model for Production Rules Exchange

Peter Denno,
National Institute of Standards and Technology,
Gaithersburg, Maryland, 20899
peter.denno@nist.gov

ABSTRACT

Production rules are rules used in production systems, sometimes called *production rule engines.* Production rule engines are commonly used to implement *expert systems*. This report describes an information model for the exchange of production rules and for description of the relationship of the rules to a data population. The report is a precursor to a standardization effort in the ISO 10303 suite of standards. The report takes the form of a standards document generally, but departs from that form in places to provide more user-friendly explanation.

1.0 Scope

This report provides guidance on two forms of data structure mapping: (1) mapping of production rules to EXPRESS-based data [1], and; (2) mapping of instances in an EXPRESS population to structures typically found in production rule systems. The EXPRESS information model which this report describes is provided in an appendix to this report. The information model provides:

- information describing rules for their use with typical production rule software
- information identifying the entity data type instances of a given EXPRESS population, and their attributes, that are subject to the rules
- information identifying the disposition of the rules with respect to the organization that produced them
- information providing logical grouping of rules
- information identifying the production rule software, and its version, with which the rules are intended to be used
- constraints concerning the form of rules

Out of scope for the information model are:

- provisions describing the syntax of rules with respect to particular production rule software
- provisions describing the execution semantics of rules

2.0 Definitions

- **atomic formula** - a syntactic structure consisting of a predicate symbol of some arity *n*, and an *n*-tuple of terms
- **condition** - an atomic formula, or negated atomic formula, in the premise of a rule
  Note: conditions serve as patterns used in the pattern matching process of a production rule engine
- **constant** - a symbol that invariantly names a distinct object
- **function** - a term-forming relation that maps from an n-tuple of argument to at most one object
- **ground fact** - an atomic formula that is free of variables
- **interpretation** - the assignment of objects from the domain to the appropriate constructs of a formal language
  Note: These assignments are made by a human.
- **object -** the thing referred to by a symbol in an interpretation
- **population mapping** - a mapping of data governed by an EXPRESS schema to working memory of a production rule engine
- **predicate -** a sentence-forming relation among objects
  Note: A predicate is encoded as an atomic formula. A predicate (father-of henry peter) might read “Henry is the father of Peter.”
- **predicate symbol** - a name that refers to the form of a predicate
- **production memory** - the internal representation of rules in a production rule system
- **row value** - a sequence of values, none of which are row values
- **row variable** - a variable naming a row value in the context of a substitution
- **substitution** - a mapping from variables to terms
- **symbol** - an atomic lexical structure
- **term** - an expression denoting an object
- **variable** - a symbol that, in the context of a substitution, names an object
- **working memory** - the internal representation of asserted facts in a production rule system

3.0 Fundamental Concepts

3.1 Two Targets of the Mapping

As stated above, this report describes two forms of mapping: (1) mapping of production rules to EXPRESS-based data, and; (2) mapping of EXPRESS data types, and instances in an EXPRESS population, to three structures typically found in production rule systems: symbols, row values, and ground facts. The purpose of the first form of mapping is to allow production rules and related data to be exchanged by EXPRESS-based software tools. The purpose of the second form of mapping is to describe relationships between EXPRESS-based populations and the rules. The mapping defines how the extents of predicates can be populated so that there is an interpretation where all the rules are true.

Mapping of the production rules to the exchange form is described in Clause 3.2. Mapping of EXPRESS-based data to production rule system structures described in Clause 3.3.

EXAMPLE 1: A CLIPS [5] rule is described by data conforming to this report using the form of mapping (1), mapping rules to EXPRESS-based data.

EXAMPLE 2: A property defined by an attribute of an entity instance is mapped to a ground fact using the form of mapping (2), mapping EXPRESS-based data to production rule system structures.

EXAMPLE 3: An EXPRESS aggregate value is mapped to a production rule row value using form of mapping (2). The row value is a substitution for a row variable in the premise of a rule.

3.2 Mapping Rule System Objects to EXPRESS-based Data

Objects are those things that a rules engine allows to play roles in the terms of predicates. The range of such things may exceed the scope of provisions for object identification described in this report. **`Term_select`** comprises types of objects that may serve as terms in predicates.

```
TYPE Term_select = SELECT (Symbol, Constants, Func, Rule_variable,
Row_Variable); END_TYPE;
```

3.2.1 Primitive Types

String, number, and logical values of the rule system (rule system ‘constants’) are mapped to their respective EXPRESS types, **`STRING`**, **`NUMBER`**, and **`LOGICAL`**.

```
TYPE PR_NUMBER = NUMBER; END_TYPE;
TYPE PR_STRING = STRING; END_TYPE;
TYPE PR_LOGICAL = LOGICAL; END_TYPE;
TYPE PR_BINARY = BINARY; END_TYPE;

TYPE Constants = SELECT (PR_NUMBER, PR_STRING, PR_LOGICAL, PR_BINARY);
END_TYPE;
```

3.2.2 Symbols

Symbols are used by the rules system to name the things on which assertions are made. Symbols are mapped to the entity type **`Symbol`**.

```
ENTITY Symbol;
  name : STRING;
END_ENTITY;
```

Example: EXPRESS entity instances could be represented by symbols. An entity instance encoded using the clear text encoding (ISO 10303-21) [6] **`#34=POINT(1.0,2.0,1.0);`** might be represented in the production system as a **`Symbol`** with **`name '#34'`**.

3.2.3 Row values

Row values are available in some production rule systems. Row values are mapped to **`Row_value`**.

```
ENTITY Row_value;
  values : LIST OF Term_Select;
 WHERE
  WR1: SIZEOF(QUERY(v <* SELF.values | contains_variable(v))) = 0;
  WR2: SIZEOF(QUERY(v <* SELF.values | 'PRODUCTION_RULE_ARM.ROW_VALUE'
      IN TYPEOF(v))) = 0;
END_ENTITY;
```

3.2.4 Functions

The entity data type **`Func`** is used to represent the notion of function in the production rule system.

```
ENTITY Func;
```

```
 func_sym : Function_Symbol;
 terms : LIST OF Term_select;
END_ENTITY;
```

3.2.5 Variables

Kinds of variables are distinguished by their scope and the kind of value that they may name. Production rule systems vary with respect to their provisions for grouping rules for execution. This specification provides for grouping by **`Rule_set`**, which identifies a collection of rules, and **`Rule_set_group,`** which identifies a collection of rule sets. Variable may be scoped to individual rules, a rule set, or a rule group. A *row variable* is a variable naming a row value in the context of a substitution.

```
TYPE Scope_select = SELECT (Rule_definition, Rule_set, Rule_set_group);
END_TYPE;

ENTITY Abstract_variable
  ABSTRACT SUPERTYPE OF (ONEOF (Scalar_variable, Row_variable));
  name : STRING;
  scope : Scope_select;
 UNIQUE
  UR1 : name, scope;
END_ENTITY;

ENTITY Scalar_variable
 SUBTYPE OF (Abstract_variable);
END_ENTITY;

ENTITY Row_variable
 SUBTYPE OF (Abstract_variable);
END_ENTITY;
```

3.2.6 Atomic Formulas and Conditions

An *atomic formula* is a syntactic structure consisting of a predicate symbol of some arity *n*, and an *n*-tuple of terms. An atomic formula represents a predicate, a sentence-forming relation among the values represented by the *n* terms. The subtype **`Rule_condition`** of entity data type **`Atomic_formula`** is used to specify a condition of a rule.

```
ENTITY Atomic_formula;
  pred_sym : Predicate_symbol;
  terms : LIST OF Term_select;
END_ENTITY;

SUBTYPE_CONSTRAINT Atomic_formula_sc FOR Atomic_formula;
  ABSTRACT SUPERTYPE;
  ONEOF (Rule_condition, Ground_fact);
```

```
END_SUBTYPE_CONSTRAINT;

ENTITY Rule_condition
 SUBTYPE OF (Atomic_formula);
  positive : BOOLEAN;
END_ENTITY;
```

The attribute **positive** indicates the relationship of working memory to the satisfaction of the condition. When **positive** is **TRUE**, the condition is satisfied if a working memory element is found that matches the pattern in the context of some variable substitutions. When **positive** is **FALSE**, the condition is satisfied if no working memory element is found that matches the pattern in the context of the variable substitutions. Note that the matching is presumed to be performed in the context of variable substitutions that are consistent across all conditions of the rule.

3.2.7 Ground Facts

A ground fact is an atomic formula containing no variables. The entity data type **Ground_fact** is used to represent an item that shall be made present in the working memory of the production rules engine before the execution of rules commences.

```
ENTITY Atomic_formula;
  pred_sym : Predicate_Symbol;
  terms : LIST OF Term_select;
END_ENTITY;

SUBTYPE_CONSTRAINT Atomic_formula_sc FOR Atomic_formula;
  ABSTRACT SUPERTYPE;
  ONEOF (Rule_condition, Ground_fact);
END_SUBTYPE_CONSTRAINT;

ENTITY Ground_fact
 SUBTYPE OF (Atomic_formula);
 WHERE
  WR1: SIZEOF(QUERY(r <* SELF\Atomic_formula.terms |
                         contains_variable(r))) = 0;
END_ENTITY;
```

3.2.8 Rules

A **Forward_chaining_rule** represents an assertion stating that for any set of substitutions under which the premise is satisfied, the ground facts represented by the conclusion under those substitutions also hold.

A **Back_chaining_rule** represents an assertion stating that for any set of substitutions under which the body is satisfied, the ground facts represented by the head under those substitutions also hold.

Forward_chaining_rule is distinguished from **Back_chaining_rule** by the expressions allowed in the head and body of each.

```
ENTITY Forward_chaining_rule
  SUBTYPE OF (Rule_definition);
  premise : Clause_Select;
  conclusion : Literal_conjunction;
  WHERE
   WR1 (local_vars_of(SELF.conclusion) <=
        local_vars_of(SELF.premise));
END_ENTITY;

TYPE Clause_select = SELECT (Simple_clause, Complex_clause);
END_TYPE;

ENTITY Simple_clause;
  formulas : LIST [1:?] OF Rule_condition;
END_ENTITY;

ENTITY Literal_conjunction
  SUBTYPE OF (Simple_clause);
END_ENTITY;
```

Example: The following depicts a CLIPS rule and its encoding using the clear text encoding (ISO 10303-21) [6]. The premise of the rule uses the **Simple_clause** syntax:

```
(defrule rule1
  (rep_context.kind ?x 'Systems Engineering')
  (view_def_context.ctx_name ?x ?view)
 =>
  (syseng.view ?x ?view))

#1=FORWARD_CHAINING_RULE('fcr1','rule1',
'test rule',#100,(),#101,(),$,#2,#10);
#2=LITERAL_CONJUNCTION((#3,#7));
#3=RULE_CONDITION('rep_context.kind',(#4,'Systems Engineering'),.T.);
#4=RULE_VARIABLE('?x',.F.);
#7=RULE_CONDITION('view_def_context.ctx_name',(#4,#8),.T.);
#8=RULE_VARIABLE('?view',.F.);
#9=RULE_CONDITION('syseng.view',(#4,#8),.T.);
#10=LITERAL_CONJUNCTION((#9));

#100=VIEW_DEFINITION_CONTEXT('rule demo','spec writing',
```

```
'need example for spec');
#101=RULE_VERSION('ver 1','Version 1',#102);
#102=RULE_PRODUCT('rp1','rule product 1','A rule product');
```

In the context of a substitution, a variable names an object. The object is the value of the variable under the substitution. When no substitution applies, the variable is said to be *unbound*. All local variables occurring in the conclusion of a rule shall also appear in the premise of a rule. The rule **WR1** of **Forward_chaining_rule** enforces this constraint against unbound variables in the conclusion.

3.2.9 Common Syntactic Features of Rules

Production rule systems commonly provide syntactic features that, though they have no effect on the execution model of the system, make the formulation of rules less verbose. An example of such a syntactic convenience is the nesting of conjunctions and disjunctions of conditions in the premise of a rule. This report provides for this syntactic feature using the **Complex_clause** entity data type.

```
TYPE Clause_select = SELECT (Simple_clause, Complex_clause);
END_TYPE;
ENTITY Complex_Clause;
  clauses : LIST [2:?] OF Clause_select;
END_ENTITY;

ENTITY Complex_conjunctive_clause
  SUBTYPE OF (Complex_Clause);
END_ENTITY;

ENTITY Complex_disjunctive_clause
  SUBTYPE OF (Complex_Clause);
END_ENTITY;
```

EXAMPLE: The following depicts a CLIPS rule and its encoding using the clear text encoding (ISO 10303-21) [6]. The premise of the rule uses the **Complex_clause** syntax:

```
(defrule rule2
 (OR
  (AND (foo ?x 7)
       (bar ?x ?y))
  (AND (baz ?z)
       (not (foo ?x 7)))
  (AND
       (taz ?z 2)
       (OR (taz ?z 3) (gaz ?z 3))
       (OR (taz ?z 6) (gaz ?z 6))))
 =>
  (foobar ?x ?y ?z))
```

```
#11=FORWARD_CHAINING_RULE('fcr2','rule2',
'test rule',#100,(),#101,(),$,#12,#50);
#12=COMPLEX_DISJUNCTIVE_CLAUSE((#13,#14,#15));
#13=LITERAL_CONJUNCTION((#31,#33));
#14=LITERAL_CONJUNCTION((#35,#37));
#15=COMPLEX_CONJUNCTIVE_CLAUSE((#16,#17,#18));
#16=LITERAL_CONJUNCTION((#38));
#17=LITERAL_DISJUNCTION((#40,#41));
#18=LITERAL_DISJUNCTION((#42,#43));
#31=RULE_CONDITION('foo',(#32,7),.T.);
#32=RULE_VARIABLE('?x',.F.);
#33=RULE_CONDITION('bar',(#32,#34),.T.);
#34=RULE_VARIABLE('?y',.F.);
#35=RULE_CONDITION('baz',(#36),.T.);
#36=RULE_VARIABLE('?z',.F.);
#37=RULE_CONDITION('foo',(#32,7),.F.);
#38=RULE_CONDITION('taz',(#36,2),.T.);
#40=RULE_CONDITION('taz',(#36,3),.T.);
#41=RULE_CONDITION('gaz',(#36,3),.T.);
#42=RULE_CONDITION('taz',(#36,6),.T.);
#43=RULE_CONDITION('gaz',(#36,6),.T.);
#50=RULE_CONDITION('foobar',(#32,#34,#36),.T.);
```

3.3 Mapping EXPRESS-based Data to Rule System Objects

3.3.1 Enumeration values

Enumeration values are mapped to symbols, where the suffix of the name of the symbol is identical with the name of the enumeration item, and the prefix of the name of the symbol are the characters specified by the **`prefix`** attribute of an instance of **`Enum_reference_prefix`**, if such an instance exists in the population. If there is no such instance, the prefix is the empty string, that is, no characters are prefixed to the name.

```
ENTITY Enum_reference_prefix;
 prefix : STRING;
END_ENTITY;

RULE max_one_entity_prefix FOR (Enum_reference_prefix);
 WHERE
   SIZEOF(QUERY(x <* Enum_reference_prefix | TRUE)) <= 1;
END_RULE;
```

EXAMPLE: If an instance of **`Enum_reference_prefix`** is present in the population, and the value of its **`prefix`** attribute is **`':'`**, then an EXPRESS enumeration value **`GREEN`** is encoded as **`:GREEN`**

3.3.2 Aggregates

EXPRESS aggregate values, where the base data type is not itself an aggregate, are mapped using **Row_value**. This report contains no provisions for mapping EXPRESS aggregates where the base data type is itself an aggregate.

3.3.3 Selection from an EXPRESS-based population

A population mapping is expressed through provisions that (1) identify instances involved in the mapping (described in this clause) and (2) identify what ground facts about those instances are to be asserted to working memory (described in Clause 3.3.4). Population mapping occurs before the execution of rules commences.

Instances involved in the mapping are identified as though a query were applied to a specified entity extent. The entity data type **Extent** provides for the expression of this.

```
ENTITY Extent;
  source : STRING;
  variable_id : OPTIONAL STRING;
  query_expression : STRING;
  syntax : OPTIONAL Expression_syntax;
END_ENTITY;
```

The attribute **Extent.syntax** indicates the syntax in which the query is specified. If the value of this attribute is **EXPRESS**, then instances are identified as though the EXPRESS query expression (10303-11v2 clause 12.6.7) [1] were applied to an aggregate containing a specified entity extent. The details of the mapping of the query expression to the attributes of the **Extent** entity type are described below.

The syntax of the EXPRESS query expression is:

QUERY ’(’ variable_id ’<*’ aggregate_source ’|’ logical_expression ’)’

Though this syntax is not used in the mapping, it correlates with the attributes of the **Extent** entity:

(1) The value of **Extent.variable_id** performs the role of variable_id above. The value shall be a string conforming to the syntax of an EXPRESS identifier.

(2) The value of **Extent.query_expression** performs the role of logical_expression above. The value shall be a string conforming to the syntax of an EXPRESS expression. The expression shall either be the EXPRESS boolean value **TRUE,** or it shall refer to the variable identifier supplied by **Extent.variable_id**.

(3) The value of **Extent.source** performs the role of aggregate_source above. The value shall be the string **'GENERIC_ENTITY'** or of the form **'SCHEMA.TYPE'** (in upper case) where **TYPE** names an entity type and **SCHEMA** is the name of the schema that contains the definition of the type.

(4) A value shall be supplied for **Extent.variable_id** unless the value of **Extent.query_expression** is **TRUE**.

Under this mapping, evaluation of the query expression identifies a subset of the entity population that may be referenced as **Entity_assertion.source** or **Attribute_assertion.source** for assertion of ground facts as described in Clause 3.3.4. Elements taken one by one from the source aggregate provide a substitution for variable_id in the logical_expression. The logical_expression is then evaluated. If logical_expression evaluates to true, the element is added to the result; otherwise, it is not.

Example 1: If **syntax** is **EXPRESS**, an **Extent** instance with **source 'GENERIC_ENTITY'** and **query_expression 'TRUE'** represents an aggregate consisting of all instances in the population.

Example 2: If **syntax** is **EXPRESS**, an Extent instance with **source 'my_schema.component_2d_location'**, **variable 'p'** and **query_expression '(p.x = 0) AND (p.y = 0)'** represents an aggregate of all instances of type **component_2d_location** that are located at the origin.

Example 3: If **syntax** is XPATH, then **query_expression** might contain a W3C XPath path expression [7] that identifies a node sequence representing the intended instances.

3.3.4 Assertion of ground facts

Extent instances are referenced by instances of type **Fact_type** to represent the relationship between each element of the aggregate represented by the **Extent** instance and a ground fact.

```
ENTITY Fact_type;
 ABSTRACT SUPERTYPE OF (ONEOF (Entity_assertion, Attribute_assertion));
  source : Extent;
  predicate_symbol : STRING;
END_ENTITY;

ENTITY Entity_assertion
  SUBTYPE OF (Fact_type);
END_ENTITY;

ENTITY Attribute_assertion
```

```
  SUBTYPE OF (Fact_type);
  group_qualifier : OPTIONAL STRING;
  attribute : STRING;
END_ENTITY;
```

Entity_assertion represents an instruction to add one unary ground fact to working memory for each element of the aggregate represented by **source**. The predicate symbol of the facts asserted is that specified in attribute **predicate_symbol**. A ground fact is asserted for each element of the aggregate represented by **source**. The argument to the ground fact is a **Symbol** instance representing the subject element.

Attribute_assertion represents an instruction to add one binary ground fact to working memory for each element of the aggregate represented by **source**. The predicate symbol of the facts asserted is that specified in attribute **predicate_symbol**. A ground fact is asserted for each element of the aggregate represented by **source**. The argument to the ground fact is an ordered tuple consisting of a **Symbol** instance representing subject element and the value of the attribute of the instance specified by **attribute**, an upper case string. The value of **group_qualifier** shall be an upper case string naming the entity data type containing the attribute. It shall be supplied if the source of the named attribute would otherwise be ambiguous (because the complex entity instance contains multiple attributes with the name specified by **attribute**).

4.0 Built-in Functions

The **Function_symbol** **ATTR_VAL** refers a function of two arguments. The first argument shall be an entity instance and the second a string of form **ENTITY.ATTRIBUTE** (upper case) naming an attribute of the argument entity instance. The value of the function is the value of the named attribute of the argument instance.

EXAMPLE: Though the **kind** attribute of an entity **Representation_context** may not have been asserted as a ground fact (such as might be assumed from the example in Clause 3.2.8), a predicate may still be expressed for such a fact:

```
#3=RULE_CONDITION('rep_context.kind’,(#4,#5),.T.);
#4=RULE_VARIABLE('?x',.F.);
#5=FUNC(.ATTR_VAL.,(#4,'REPRESENTATION_CONTEXT.KIND'));
```

5.0 Appendix -- EXPRESS Short Form

```
SCHEMA PRODUCTION_RULE;

USE FROM Specification_document_arm;
```

```
USE FROM Software_arm;
USE FROM Activity_arm;
USE FROM Product_identification_arm;
USE FROM Date_time_assignment_arm;
TYPE PR_NUMBER = NUMBER; END_TYPE;
TYPE PR_STRING = STRING; END_TYPE;
TYPE PR_LOGICAL = LOGICAL; END_TYPE;
TYPE PR_BINARY = BINARY; END_TYPE;

TYPE Constants = SELECT (PR_NUMBER, PR_STRING, PR_LOGICAL, PR_BINARY);
END_TYPE;

TYPE Function_symbol = SELECT (PR_STRING, Built_in_functions);
END_TYPE;

TYPE Built_in_functions = EXTENSIBLE ENUMERATION OF (ATTR_VAL);
END_TYPE;

TYPE Expression_syntax = EXTENSIBLE ENUMERATION OF (EXPRESS);
END_TYPE;

TYPE Predicate_symbol = STRING;
END_TYPE;

TYPE Scope_select = SELECT (Rule_definition, Rule_set, Rule_set_group);
END_TYPE;

TYPE Term_select = SELECT (Symbol, Constants, Func, Scalar_variable,
Row_value, Row_variable);
END_TYPE;

ENTITY Enum_reference_prefix;
 prefix : STRING;
END_ENTITY;

RULE max_one_entity_prefix FOR (Enum_reference_prefix);
 WHERE
   SIZEOF(QUERY(x <* Enum_reference_prefix | TRUE)) <= 1;
END_RULE;

ENTITY Rule_definition
  ABSTRACT SUPERTYPE OF (ONEOF (Forward_chaining_rule, Back_chaining_rule))
  SUBTYPE OF (Rule_software_definition);
END_ENTITY;

ENTITY Forward_chaining_rule
  SUBTYPE OF (Rule_definition);
  premise : Clause_Select;
  conclusion : Literal_conjunction;
  WHERE
```

```
  WR1:  local_vars_of(SELF.conclusion) <= local_vars_of(SELF.premise);
END_ENTITY;

ENTITY Back_chaining_rule
  SUBTYPE OF (Rule_definition);
  head : Rule_condition;
  body : LIST OF Rule_condition;
 WHERE
  WR1: SELF.head.positive = TRUE;
  WR2:  local_vars_of(SELF.head) <= local_vars_of(SELF.body);
END_ENTITY;

ENTITY Simple_clause;
  formulas : LIST [1:?] OF Rule_condition;
END_ENTITY;

SUBTYPE_CONSTRAINT Simple_clause_sc FOR Simple_Clause;
  ABSTRACT SUPERTYPE;
  ONEOF (Literal_conjunction, Literal_disjunction);
END_SUBTYPE_CONSTRAINT;

ENTITY Literal_conjunction
  SUBTYPE OF (Simple_clause);
END_ENTITY;

ENTITY Literal_disjunction
  SUBTYPE OF (Simple_clause);
END_ENTITY;

ENTITY Atomic_formula;
  pred_sym : Predicate_Symbol;
  terms : LIST OF Term_select;
END_ENTITY;

SUBTYPE_CONSTRAINT Atomic_formula_sc FOR Atomic_formula;
  ABSTRACT SUPERTYPE;
  ONEOF (Rule_condition, Ground_fact);
END_SUBTYPE_CONSTRAINT;

ENTITY Rule_condition
 SUBTYPE OF (Atomic_formula);
  positive : BOOLEAN;
END_ENTITY;

ENTITY Ground_fact
 SUBTYPE OF (Atomic_formula);
 WHERE
  WR1: SIZEOF(QUERY(r <* SELF\Atomic_formula.terms | contains_variable(r))) =
0;
END_ENTITY;
```

```
SUBTYPE_CONSTRAINT Complex_clause_sc FOR Complex_Clause;
  ABSTRACT SUPERTYPE;
  ONEOF (Complex_conjunctive_clause, Complex_disjunctive_clause);
END_SUBTYPE_CONSTRAINT;

ENTITY Complex_Clause;
  clauses : LIST [2:?] OF Clause_select;
END_ENTITY;

ENTITY Complex_conjunctive_clause
  SUBTYPE OF (Complex_Clause);
END_ENTITY;

ENTITY Complex_disjunctive_clause
  SUBTYPE OF (Complex_Clause);
END_ENTITY;

ENTITY Symbol;
  name : STRING;
END_ENTITY;

ENTITY Abstract_variable
  ABSTRACT SUPERTYPE OF (ONEOF (Scalar_variable, Row_variable));
  name : STRING;
  scope : Scope_select;
 UNIQUE
  UR1 : name, scope;
END_ENTITY;

ENTITY Scalar_variable
 SUBTYPE OF (Abstract_variable);
END_ENTITY;

ENTITY Row_variable
 SUBTYPE OF (Abstract_variable);
END_ENTITY;

ENTITY Row_value;
  values : LIST OF Term_select;
 WHERE
  WR1: SIZEOF(QUERY(v <* SELF.values | contains_variable(v))) = 0;
  WR2: SIZEOF(QUERY(v <* SELF.values | 'PRODUCTION_RULE_ARM.ROW_VALUE' IN
TYPEOF(v))) = 0;
END_ENTITY;

ENTITY Func;
 func_sym : Function_Symbol;
 terms : LIST OF Term_select;
END_ENTITY;
```

```
ENTITY Extent;
  source : STRING;
  variable_id : OPTIONAL STRING;
  query_expression : STRING;
  syntax : OPTIONAL Expression_syntax;
END_ENTITY;

ENTITY Fact_type
  ABSTRACT SUPERTYPE OF (ONEOF (Entity_assertion, Attribute_assertion));
   source : Extent;
   predicate_symbol : STRING;
END_ENTITY;

ENTITY Entity_assertion
  SUBTYPE OF (Fact_type);
END_ENTITY;

ENTITY Attribute_assertion
  SUBTYPE OF (Fact_type);
   entity_type : STRING;
   attribute : STRING;
END_ENTITY;

ENTITY Global_assignment;
  variable : Abstract_variable;
  val : Term_select;
 WHERE
  WR1: NOT(contains_variable(SELF.val));
END_ENTITY;

ENTITY Rule_software_definition
   SUPERTYPE OF (ONEOF (Rule_definition,
                        Rule_set_group,
                        Rule_set))
   SUBTYPE OF (Software_definition);
   SELF\Product_view_definition.defined_version : Rule_version;
END_ENTITY;

ENTITY Rule_set
   SUBTYPE OF (Rule_software_definition);
   engine : Language_reference_manual;
   conflict_resolution_strategy : OPTIONAL STRING;
   rule_member : SET[1:?] OF Rule_priority;

END_ENTITY;

ENTITY Rule_set_group
  SUBTYPE OF (Rule_software_definition);
  elements : SET[2:?] OF Rule_set;
```

```
END_ENTITY;

FUNCTION contains_variable (x : Term_select) : BOOLEAN;
  IF ('PRODUCTION_RULE_ARM.ABSTRACT_VARIABLE' IN TYPEOF(x)) THEN RETURN
(TRUE);
      ELSE IF (('PRODUCTION_RULE_ARM.FUNC' IN TYPEOF(X)) AND
               (SIZEOF(QUERY(y <* x.terms | contains_variable(y))) > 0))
               THEN RETURN (TRUE);
               ELSE RETURN (FALSE);
           END_IF;
  END_IF;
END_FUNCTION;

FUNCTION local_vars_of (thing : GENERIC) : SET [0:?] OF Scalar_variable;
  LOCAL
    accum : SET [0:?] OF Scalar_variable := [];
  END_LOCAL;
  RETURN (local_vars_aux(thing, accum));
END_FUNCTION;

FUNCTION local_vars_aux (thing : GENERIC; accum : SET [0:?] OF Scalar_variable)
                                                  : SET [0:?] OF Scalar_variable;
  LOCAL i,j,k : INTEGER; END_LOCAL;
  IF (('PRODUCTION_RULE_ARM.ABSTRACT_VARIABLE' IN TYPEOF(thing)) AND
      ('PRODUCTION_RULE_ARM.RULE_DEFINITION' IN (TYPEOF(thing.scope))))
   THEN accum := accum + thing;
   ELSE IF ('PRODUCTION_RULE_ARM.RULE_CONDITION' IN TYPEOF(thing))
        THEN REPEAT i := 1 TO HIINDEX(thing\Atomic_formula.terms);
                accum := local_vars_aux(thing\Atomic_formula.terms[i],accum);
             END_REPEAT;
        ELSE IF ('PRODUCTION_RULE_ARM.SIMPLE_CLAUSE' IN TYPEOF(thing))
             THEN REPEAT j := 1 TO HIINDEX(thing.formulas);
                         accum := local_vars_aux(thing.formulas[j],accum);
                  END_REPEAT;
             ELSE IF ('PRODUCTION_RULE_ARM.COMPLEX_CLAUSE' IN TYPEOF(thing))
                  THEN REPEAT k := 1 TO HIINDEX(thing.clauses);
                        accum := local_vars_aux(thing.clauses[k],accum);
                       END_REPEAT;
                  END_IF;
             END_IF;
        END_IF;
  END_IF;
  RETURN(accum);
END_FUNCTION;

-- =====================================================
-- Rule management
-- =====================================================

ENTITY Rule_action
```

```
   ABSTRACT SUPERTYPE OF (ONEOF (Rule_submission,
                                 Rule_adoption,
                                 Rule_rejection,
                                 Rule_supersedence,
                                 Rule_creation,
                                 Rule_expiration,
                                 Rule_change_request,
                                 Rule_request,
                                 Rule_modification));
   subject_rule : Rule_version;
DERIVE
   subject_action_assignment : SET[0:?] OF
Organization_or_person_in_organization_assignment :=
         QUERY(temp <* USEDIN ( SELF , 'PERSON_ORGANIZATION_ASSIGNMENT_ARM.'
+
'ORGANIZATION_OR_PERSON_IN_ORGANIZATION_ASSIGNMENT.ITEMS' )

                    | ( temp.role = 'subject action assignment'));
  UNIQUE
   UR1: SELF\Rule_action.subject_rule,
SELF\Rule_action.subject_action_assignment;
  WHERE
   WR1: EXISTS (subject_action_assignment) AND
(SIZEOF(subject_action_assignment) = 1 );
END_ENTITY;

ENTITY Rule_justification
   SUBTYPE OF (Rule_action);
   justified_action : Rule_action;
   justification_rationale : STRING;
WHERE
   WR1: SELF <> justified_action ;
END_ENTITY;

ENTITY Rule_adoption
   SUBTYPE OF (Rule_action);
END_ENTITY;

ENTITY Rule_change_request
   SUBTYPE OF (Rule_action);
   change_reason : STRING;
END_ENTITY;

ENTITY Rule_expiration
   SUBTYPE OF (Rule_action);
   expiration_rationale : STRING;
END_ENTITY;

ENTITY Rule_modification
   SUBTYPE OF (Rule_action);
```

```
   modification_rationale : Rule_change_request;
END_ENTITY;

ENTITY Rule_priority;
   priority : INTEGER;
   prioritized_rule : Rule_definition;
WHERE
   WR1: priority >= 0 ;
END_ENTITY;

ENTITY Rule_product
   SUBTYPE OF (Software);
WHERE
   WR1: SIZEOF ( [ 'rule' ] * types_of_product ( SELF ) ) = 1 ;
END_ENTITY;

ENTITY Rule_rejection
   SUBTYPE OF (Rule_action);
   rejection_reason : STRING;
END_ENTITY;

ENTITY Rule_request
   SUBTYPE OF (Rule_action);
END_ENTITY;

ENTITY Rule_submission
   SUBTYPE OF (Rule_action);
   submission_rationale : STRING;
END_ENTITY;

ENTITY Rule_supersedence
   SUBTYPE OF (Rule_action);
   superseded_rule : Rule_version;
END_ENTITY;

ENTITY Rule_version
   SUBTYPE OF (Software_version);
   SELF\Product_version.of_product : Rule_product;
INVERSE
   management_action : SET[1:?] OF Rule_action FOR subject_rule;
   product_definition : SET[1:?] OF Rule_software_definition FOR
defined_version;
END_ENTITY;

RULE rule_software_definition_constraint FOR (Product_view_definition);
WHERE
   WR1: SIZEOF (QUERY ( pvd <* Product_view_definition | (
NOT('PRODUCT_RULE_ARM.' + 'RULE_SOFTWARE_DEFINITION'
                        IN TYPEOF(pvd)))
                       AND ('PRODUCT_RULE_ARM.' + 'RULE_VERSION' IN TYPEOF (pvd
```

```
. defined_version)))) = 0 ;
END_RULE;

RULE rule_version_constraint FOR (Product_version);
WHERE
   WR1: SIZEOF (QUERY(pv <* Product_version | (NOT('PRODUCT_RULE_ARM.' +
'RULE_VERSION' IN TYPEOF(pv)))
                       AND ( 'PRODUCT_RULE_ARM.' + 'RULE_PRODUCT' IN
TYPEOF(pv.of_product)))) = 0 ;
END_RULE;

END_SCHEMA;
```

6.0 References

[1] International Organization for Standards, (ISO) *Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 11: description methods: The EXPRESS Language Reference Manual.* ISO 10303-11:2004, 2004.

[2] Robinson, J., A., and Voronkov, A., (editors) *Handbook of Automated Reasoning,* MIT Press, Cambridge, Massachusetts 2001.

[3] Church, A., *Introduction to Mathematical Logic*, Princeton University Press, Princeton, New Jersey, 1944.

[4] Doorenbos, R., B., *Production Matching for Large Learning Systems*, Ph.D. Dissertation, Carnegie Mellon University, 1995.

[5] unspecified authorship, *CLIPS Reference Manual, Vol 1: Basic Programming Guide, Version 6.23,* June 1, 2003.

[6] International Organization for Standards, (ISO) *Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 21: implementation methods: Clear text encoding of the exchange structure.* ISO 10303-11;1994.

[7] World Wide Web Consortium (W3C), *XML Path Language (XPath),* http://www.w3.org/TR/xpath, November 16, 1999.