SweetRules Rule Format Information Guide

Introduction

This document summarizes the Commonrules 3.3 and RuleML formats that have been used extensively in SweetRules V2.0. The CommonRules 3.3 CLP grammar has also been included here.

Differences between CommonRules V2.1 and V3.3

The most prominent difference between the two is in the depiction of CLP programs. We have provided the same functionality in the two versions below, explained via the examples below.

A. The Quaker Example

The rules for this program are:

1) If a person is a Quaker, he is a pacifist
2) If the individual is a Republican, he is NOT a pacifist
3) If a person is both a Quaker and a Republican, then Rule 2 will override Rule 1. Hence Nixon is both a Quaker and a Republican, but because of the override rule is not a pacifist.

CR 2.1 format

```
/**
 * quakers and pacifists
 */

<qua> pacifist(?X) < quaker(?X).
<rep> cneg pacifist(?X) < republican(?X).
quaker(nixon).
republican(nixon).
overrides(rep,qua).
```

[Note that a period ends the statements and an arrow (<) stands for conditionals]

CR 3.3 format

```
/**
 * quakers and pacifists
 */
```

```
<qua>
  if
    quaker(?X)
  then
    pacifist(?X);

<rep>
  if
   republican(?X)
  then
   cneg pacifist(?X);

/* some facts*/
quaker(nixon);
republican(nixon);
overrides(rep,qua);
```

[Note that a semicolon ends the statements and an ifthen construct is used in conditional execution. The other major formats remain the same across both versions]

B. The Credit Report Example

The rules for this program are:

4) If a person is a Quaker, he is a pacifist
5) If the individual is a Republican, he is NOT a pacifist
6) If a person is both a Quaker and a Republican, then Rule 2 will override Rule 1. Hence Nixon is both a Quaker and a Republican, but because of the override rule is not a pacifist.

CR 2.1 format

```
<credSelf>     honest(self,?Requester)
               < creditRating(creditReportsRUs,?Requester,good).
<frauSelf> CNEG honest(self,?Requester)
               < creditRating(?BlackballService,?Requester,bad)
                   and fraudExpert(recommenderServiceD,?BlackballService).
overrides(frauSelf,credSelf).
fraudExpert(recommenderServiceD,studentLoanAgency).

creditRating(creditReportsRUs,Sue,good).

creditRating(creditReportsRUs,Joe,good).
creditRating(studentLoanAgency,Joe,bad).
```

CR 3.3 format

```
<credSelf>
  if
    creditRating(creditReportsRUs,?Requester,good)
  then
    honest(self,?Requester);

<frauSelf>
  if
    creditRating(?BlackballService,?Requester,bad) and
    fraudExpert(recommenderServiceD,?Fraudscreen.net)
  then
    CNEG honest(self,?Requester);

overrides(frauSelf,credSelf);
fraudExpert(recommenderServiceD,studentLoanAgency);
creditRating(creditReportsRUs,Sue,good);
creditRating(creditReportsRUs,Joe,good);
creditRating(studentLoanAgency,Joe,bad);
```

SCLP RuleML DTD 0.8+

This document is attached as a text file along with the distribution as:

RuleMLV0.8SCLPDTD+commentchange+include element.dtd

Also important is to refer to the new XML schema specification for RuleML 0.87:

http://www.ruleml.org/0.87/

The DTD specification of RuleML will no longer be maintained but will be available as an archive.

CommonRules 3.3 Rule Syntax

The following topics are covered in this section

---

1. CLP grammar

The following describes the CLP grammar:

```
   clp ::= (erule | mutual_exclusion | sensorlink | effectorlink)+
   erule      ::= [rulelabel] unlabelledrule
   unlabelledrule  ::= [<if> [bodyexpr] <then>]  head <ruleenddelim>
   head  ::= clit (<and> clit)*
   clit  ::= atom | <cneg> atom
   atom  ::= predicate [argsexpr]
   argsexpr  ::= largsdelim [termseq] rargsdelim
   termseq   ::= term (, term)*
   term  ::=  <lvar> | functerm | objectterm | integerterm | doubleterm | boolean term |
datetimeterm
   functerm ::= funconst [argsexpr]
   objectterm ::= <string>
   inegerterm ::= <integertoken>
   doubleterm ::= <doubletoken>
   booleanterm ::= <booleantoken>
   datetimeterm ::= datetime(<"string">)
   mutex ::= <mutex> clit <and> clit [ <given> <impliedby> bodyexpr ]<ruleenddelim>
   sensorLink ::= [rulelabel]  <sensor> predicate <class> class <method>method
[<bindingRequirement> bindingTermList ][<path><string>]<ruleenddelim>
   effectorLink ::= [rulelabel] <effector> predicate <class>  class <method>
method<ruleenddelim>
  bodyexpr ::= fclit (<and> fclit)* | fclit (<or> fclit)*
               |   delimitedbodyexpr
               |   delimdbodyexpr (<and> delimdbodyexpr)*
               |   delimdbodyexpr (<or>  delimdbodyexpr)*
   delimitedbodyexpr ::= lbodyexprdelim bodyexpr rbodyexprdelim
   lbodyexprdelim  ::= "("
   rbodyexprdelim  ::= ")"
   fclit ::= clit | <fneg> clit
   <lvar>  ::= "?" <normaltoken>
```

```
 method ::= <normaltoken>
 class ::= <normaltoken>
 path ::= <string>
 predicate ::= <normaltoken>
 funconst ::= <normaltoken> | <string>
 rulelabel  ::= <lrulelabdelim> funconst <rrulelabdelim>
<sensorassign> ::= "Sensor:"
<effectorassign> ::= ":Effector:"
<binding> ::=  "BindingRequirement:"
<path> ::=  "path:"
<class> ::= <normaltoken>
<if> ::= "if"
<then> ::= "then"
<mutex> ::= "mutex"
<given> ::= "given"
<and> ::= "and"
<or> ::= "or"
<bindingTerm ::=  "BOUND" |"FREE">
bindingTermList ::= "(" bindingTerm (,bindingTerm)* ")"

 <string>      ::=
   "\""
   (   (~["\"","\\","\n","\r"])
     | ("\\"
         ( ["n","t","b","r","f","\\","'","\""]
         | ["0"-"7"] ( ["0"-"7"] )?
         | ["0"-"3"] ["0"-"7"] ["0"-"7"]
         )
       )
   )*

<integertoken : ((["0"-"9"])|"+"|"")+ >

<doubletoken : ((["0"-"9"])|"+"|"."|"|"$"|"%")+ >

<numerictoken : ((["0"-"9"])|"+"|"."|"|"$"|"%")+ >

<booleantoken : "TRUE" | "FALSE">

<normaltoken : ((["a"-"z", "A"-"Z", "_", "/", ":", "0", ".", "0"-"9"])|"\\." )+ >

  <impliedby>  ::= "<"
 <largsdelim>  ::= "("
 <rargsdelim>  ::= ")"
 <lrulelabdelim>  ::= "<"
 <rrulelabdelim>  ::= ">"
```

<ruleenddelim> ::= ";"
<dot> ::= "."

In addition,
<comment_begin> ::= "/*"
<comment_end> ::= "*/"

The above definition of <string> is hard to understand. The backslash character (\ ) is being used as an escape character (for specifying control characters) both in the process of defining, and in what is being defined. In English, the easiest way to understand it is as follows. One way to form a <string> is to give a sequence of ordinary keyboardtypable characters, including spaces, then enclose it in doublequote characters. E.g., "This is convenient; '1.6' ; {a,b}^ $. Go! [15% is fine] " . Beyond this, one can specify some special characters, including to control white space, by using the backslash character (\ ) as a prefix escape character before a single other character or a 3digit octal number. E.g., \n or \222 . WARNING: we have not extensively tested this facility for special characters.

We recommend that variable names should begin with upper case letters (e.g. ?X, ?Name) and all other names such as predicate and function names begin with lower case letters. Although the CommonRule SCLPEngine recognizes variable names that begin with lower case, XSB/Smodels inferencing engines regard arguments that begin with upper case letters as varaibles. To avoid confusion, the user should use variable names that begin as upper case and all other names with lower case letters.

2. CLPfile syntax; related CLP syntax, expressive restrictions, and semantics

Next, we discuss CLPfile syntax. Along the way, we discuss a number of details that are really about CLP syntax, including treatment of OR in the body, AND in the head, recommended expressive restrictions, miscellaneous terminology, and miscellaneous cautions.

A rule has the form " **If** conditions **then** conclusions " or " conclusions < conditions ". The If ..Then representation is prefered as it relates more closely to our daily use of the If ...Then.. statement.

Each rule has an optional rule label. Rule labels are handles for specifying prioritization. A rule label precedes the (unlabelled part of the) rule, and is enclosed in angle brackets: "< ... >". A rule label is simply a logical constant, i.e., a 0ary logical function symbol.

The predicate "**overrides**" is syntactically reserved as the name of the prioritization predicate. It takes two arguments, which intuitively should be rule labels as arguments. The predicate "overrides" is otherwise an ordinary predicate name, and rule labels are otherwise ordinary 0ary function symbols; both are part of the ordinary logical language.

The symbol " **?**" prefixes a logical variable.

CNEG means classical negation. Intuitively, **CNEG p** means the holder of the rule/fact believes/knows that p is false.

FNEG means negationasfailure. **FNEG p** represents a weaker belief than **CNEG p**. Intuitively, **FNEG p** means the holder of the rule/fact does not believe/know p is true.

Terminology: For brevity, we call a classical literal a "**CLITERAL**", and we call a body literal (the literals from the IF clause) a "**FCLITERAL**". In these, "C" stands for "**CNEG**" and "F" stands for "**FNEG**". In the grammar above, "clit" stands for "**CLITERAL**", and "fclit" stands for "**FCLITERAL**".

Terminology: "**erule**" means a CLPfile rule. The "e" part of "**erule**" stands for "extendedform". An "**orule**" means a rule that is "ordinaryform" in the sense that it is "**pseudoHorn**": the head is a single cliteral, and the body is a conjunction (AND) of zero or more fcliterals. Erules (beyond orules's) are permitted for the sake of syntactic convenience. An orule is simply a special case of an erule.

A (e)rule head is permitted to be not just a single cliteral, but a conunction (AND) of multiple cliteral's.

A (e)rule body is permitted to be not just a conjunction (AND) of multiple fcliteral's, but to be syntactically built up from fcliteral's by using the OR connective as well as the AND connective.

While converting a CLP (courteous logic program) to OLP (ordinary logic program), e.g., in the interlingua Translator and in the Transformer, erule's are converted to orule's, i.e., the set of erule's is converted to a semantically equivalent set of orule's. Caution: in the current version of CommonRules, this conversion has computational complexity that is worstcase exponential in time and space (output size). This was done for simplicity of implementation. In a nearfuture version of CommonRules, however, this conversion will be accomplished by a more sophisticated technique that is computationally tractable (worstcase cubic) in time and space (output size). The worstcase only arises when there are many OR'd subexpressions within a (e)rule body; otherwise this conversion is tractable even in the current version. The worstcase arises in converting the following (e)rule, for example: **If (a1 OR b1) AND (a2 OR b2) AND ... AND (ak OR bk) then C.** This results in 2^k orules, in the current version: {I**if a1 AND a2 AND ... AND ak then C. If b1 AND a2 AND ... AND ak. .... then C, If b1 AND b2 AND ... AND bk then C.** }

FNEG, CNEG, AND, OR are all caseinsensitive. White space is ignored (with the exception of lineends for singleline comments that begin with "//"; see below). Sequencing of rules does not matter semantically. Note that sequencing of rules is often not preserved by the various operations of CommonRules (e.g., Translator, Transformer, SCLPEngine steps). Sequencing of literals within a rule body, or within a rule head, does not matter semantically, in the sense that two AND'd literals, or two OR'd literals, can equivalently be swapped.

A **Fact** is just a special case of a (o)rule: one in which the head is a single ground (i.e., fully instantiated) literal and the body is empty (semantically, the body is regarded as always True).

If a rule's body is empty, the rule can be expressed as a fact. (i.e. no if or then , just a "**Cliteral**").

In order to use XSB and Smodels which like many OLP systems use the following case convention, please stick to the same case convention in CLPfile as well. Predicate and function names must start with a lowercase letter. Logical variable names must start with an uppercase letter after the "?" symbol. It is very easy to make a mistake by not obeying this case convention, especially when using the interlingua transformer to transform into XSB or SModels.

Please note that the value of the quoted string "a1234" is not equal to a1234. Please exercise caution in this regard when using quotes.

The predicates such as "equals", "notEquals", "lessThan", "greaterThan", "greaterThanOrEquals" and "equalsOrLessThan" are syntactically reserved predicate names. (Click here for a description of other CommonRules reserved predicates) They take two arguments. These predicates are treated specially in inferencing in SCLPEngine and XSB: they are evaluated in the manner of Prolog builtins. Thus, one need not explicitly give a bunch of facts of the form "equals(a,a)" and "notEquals(a,b)" as rules in the CLP; rather these facts are automatically computed, i.e., implicitly available in SCLPEngine and XSB. For now, we call such predicates "builtin" predicates. **Caution: Intuitively, builtin predicates should not be used in the heads of rules, because it will cause confusion and may be rejected by many existing rule systems. (Note that this caution is not yet enforced by the current version of CommonRules.)**

A **mutex (mutual exclusion)** has the form **"MUTEX clit1 AND clit2 [GIVEN body]** .", where clit1 and clit2 are classical literals and body is an expression similar to a rule body. As the square brackets indicate, the **GIVEN** is optional. The mutex specifies that clit1 and clit2 are opposers, given that body is true. Intuitively, the mutex specifies that at most one of these opposers should be inferred for any particular ground instance of the mutex.

A rule or mutex must be terminated by a period (";"). It is easy to inadvertently omit this and get a parsing error!

A **sensorLink** which specifies the link between a predicate and a java class takes the form: **[rulelabel] <sensorassign> predicate <classmethod> class <ruleenddelim> method [<binding> bindingTermList] [<aprocpath> path]** where **class** is the name of the java class and **method** is the method within the class to be executed when the predicate is encountered during rule execution. **BindingTermList** is a sequence of **"BOUND" or "FREE"** constants which indicate if the variables in the argument list of the predicates where the sensor is linked are BOUND or FREE with respect to the location of each term in the argument list of the predicate. The sequence of BOUND/FREE constants must be enclosed with "(" and ")". A BOUND constant indicates that the variable must be a constant or is bound to some values before passing to the sensor. A FREE constant indicates that the variable does not need to be bound before passing to the sensor. SCLPEngine will varify the above binding requirements are met for each sensor before execution. An error message will be displayed if the sensor specification fails the binding requirement test. If NO binding requirement is specified in the sensorLink statement for the sensor, the default setting is that ALL vairables in the argument list of the sensor predicate are FREE. **Path** is a string that specifies the location of the directory of the sensor classes relative to the CLASSPATH. The path must begin and end with " " " .

An **effectorLink** which specifies the link between a predicate and a java class takes the form: **[rulelabel] <effectorassign> predicate <classmethod>class <ruleenddelim> method [<aprocpath> path]** where **class** is the name of the java class and method is the **method** within the class to be executed when the predicate as a conclusion is encountered during rule execution. Path is a string that specifies the location of the directory of the effector classes relative to the CLASSPATH. The path must begin and end with " "".

A Comment begins with "/*" and ends with "*/"; no nesting of such comments is permitted. A Singleline Comment begins with a "//" and ends at the line end. The content of a comment or singleline comment is ignored during parsing. These comments are similar to those in Java and C++.

Note: The CommonRules Transformer introduces new predicates formed by adding certain substrings as suffixes or prefixes to the originalinput CLP's predicate names. Next is a list of these substrings. These substrings should not be used as part of the input CLP's predicate names: " n_, _o, _c, _r, _s, _u"; you should treat these as syntactically reserved. (In future versions of CommonRules, this will probably be made more flexible.) In addition, the logical function name "emptyLabel" is syntactically reserved: it is introduced by the Transformer (and elsewhere) to stand for an empty rule label. Please refer to the the Transformer section for more information about the transformer output.

Caution on predicate and function names: The same predicate name, or function name, cannot be used with two (or more) different arities. No warning is issued about this, in the current version of CommonRules. It is very easy to inadvertently omit an argument and get apparently mystifying noninferences!

More details on the CLPfile syntax are specified in the formal grammar specification above, but we endeavored to capture the most important aspects of that grammar in the prose points above.

To get the intended semantic behavior that is called "courteous" in the theoretical papers about courteous LP's, we recommend that you obey the following additional expressive restrictions.
1) The inferrable prioritization should be a strict partial order among the rule pairs that actually conflict.
2) The mutex's should specify opposition that is "oneof", by which we mean the following. The opposition specified by the mutex's, after groundinstantiating the mutex's (including any classical mutex's) should be equivalent to a disjoint collection of "oppositionlocales". Each oppositionlocale consists of two or more ground classical literals {p1,...,pk}, such that at most one of those literals is permitted (by the mutexspecified opposition constraints) to be concluded. All of the examples we give, except for prescription.clp, obey this oneof restriction.
3) In addition, it is advisable to be careful about cyclic dependence, as always in LP's. Note that, for now, CommonRules is not doing any checking or enforcement of these expressive restrictions (1) and (2) .

The full expressive power of conditional mutex's (i.e., beyond specifying oneof opposition) is provided as an experimental feature, for now. It is as yet unclear theoretically how useful (or conceptually intuitive) this full power is. See the subsection below about expressive restrictions required by SCLPEngine: rangerestrictedness and predicateacyclicity.

Missing in CLP is any concept of rules/premises that are firstorder/logicallymonotonic/"forsure", as distinguished from defaultflavor.
Our philosophy is that it is better to represent such rules/premises instead as having very high priority, i.e., maximal priority. Then if it so happens that the veryhighpriority rules conflict with each other, inconsistent conclusions do not result.

3. CLP Situated Programming

CommonRules version provides a framework and rule format for the flexible creation of platform independent, reuseable, runtime ynamically loadable attached procedures which allows userdefined rules to collect the necessary information, e.g. from a database or from an internet web site (sensor) during rule execution and take appropriate actions (effector) when conclusions are reached. No recompilation of modules other than the attached procedures and no link libraries are needed. The link of a predicate or a conclusion to an attached procedure is simply expressed by a simple statement in the rule set. For example, rules can be incorporated into a personal/business agent to act as one's

automatic agent to browse the web and select the product and take the necessary actions (e.g. make bids, ask for more information etc.,) based on the requirements specified by the rule set. Such feature has the advantage of separating the business logic representated by the rules from the mechanism of data collection and subsequent actions which allows the business logic to be exchanged by various business entities. Under the CommonRules framework, such rule set can be created by nonprogrammer with a simple step by step instruction. The fact that attached procedures can be used by different rule sets without modification is an important element in the rapid development and deployment of business rules.

CommonRules provides the mechanism to link a predicate or conclusion in a rule to an external executable module by using the dynamic class loading functions of the Java Development Kit (JDK). The user specifies the link of a predicate or conclusion of a rule to an attached procedure in a rule set. A dynmaic registration step registers and verifies all required attached procedures at the time the ruleset is loaded. The sensor attached procedure is loaded and executed when the rule engine encounters a predicate with the attached procedure link. Similary, the effector attached procedure is loaded and executed with the information (in the form of instantiated variable) concluded when a conclusion is generated with the attached procedure link.

For more information, please refer to sections CLP Grammar, CLP Syntax, sensorLink and effectorLink, CLP Builtin Sensors and How to use the CLP Sensor and Effector Attached Procedure. For a list of sample sensors and effectors which come with the CommonRules package, please refer to the directory CommonRules30\com\ibm\commonrules\built_in_aprocs and the tutorial examples.

4. CLP Builtin Sensors and Arithmetic Predicates

CommonRules provides a set of common logical, arithmatic and list operators. Please refer to the Programmer Guide, Appendix D Build in Functions and Build in Predicates.

Please note that this version of CommonRules rule engine support nested arithmatic functions.

For example:

if a(?X, ?Y, ?Z) and equals(add(?X,?Y), ?Z) then b(?Z);

5. CLPfile Examples on Rules, Facts, Various Terms, Mutexes and Overrides

1. Rule.

<rule1> if husband(?X, ?Y) AND wife(?Y, ?X) then married(?X,?Y);

<rule1> is the optional rule label.
married(?X, ?Y) is the head (consequent) CLITERAL of the rule with two variable arguments ?X and ?Y.

husband(?X, ?Y) is the first FCLITERAL of the body of the rule with two variable arguments ?X and ?Y.
AND is the the AND logical operator
wife(?Y, ?X) is the second FCLITERAL of the body of the rule with two variable arguments ?X and ?Y.

"rule1" is the rule label for the rule that follows. Rule label is ordinary 0ary function symbols which is part
of the ordinary logical language.

2. Fact. Fact is treated as a rule with no conditions attached.

wife(Nancy, Joe).

3. Rule involving NEGATION AS FAILURE and CLASSICAL NEGATION.

FNEG CNEG b(?X) AND CNEG c(?X) AND FNEG d(?X) then CNEG a(?X);
Reminder: CNEG is CLASSICAL NEGATION. **FNEG** is NEGATION AS FAILURE
4. Rule with different types of terms.
Rule with *functional term*
If b(g(?X)) AND c(m, j) then a(f(?X));
g(?X) and f(?X) are functional terms; they are arguments of the predicates **a** and **b**, respectively. **c(m, j)** is a ground literal.
Rule with *string term*
If Customer(?Name,?ID) AND equals(?Name, "John") then sendmailTo(?Name);
"John" is a string term.
Rule with *integer term*
If Customer(?Name,?PuchaseAmount) AND greaterThan(?PurchaseAmount, 230) then goldClub(?Name);
230 is an integer term
Rule with *double term*
If Customer(?Name,?PurchaseAmount) AND equals(?PurchaseAmount, 350.00) then sendmailTo(?Name);
350.00 is a double term

Rule with *boolean term*
If Customer(?Name,?greaterThan40) AND equals(?greaterThan40, TRUE) then goldClub(?Name);
230 is an integer term
Rule with *dateTime term*
If Customer(?Name,?PurchaseDate) AND equals(?PurchaseDate, datetime("Tue Mar 12 14:56:01 EST 2002")**) then sendmailTo(?Name);**
datetime("Tue Mar 12 14:56:01 EST 2002") **is a dateTime term**
5. Mutex.

MUTEX
loyalCategory(?Customer) and premiereCategory(?Customer) .

loyalCategory(?Customer) and **premiereCategory(?Customer)** are the two opposer CLiteral's.

MUTEX
discount(?Cust, ?YPercent) and discount(?Cust, ?ZPercent) GIVEN notEquals(?YPercent, ?XPercent);.

discount(?Cust,?YPercent) and **discount(?Cust, ?ZPercent)** are the two opposer CLiteral's.
notEquals(?YPercent,?ZPercent) is the condition under which **discount(?Cust,?YPercent)** and **discount(?Cust, ?ZPercent)** oppose each other.

6. Overrides.

<r1> If b(?X) then a(?X);
<r2> If f(?X) then cneg a(?X);
overrides(r1, r2).

"**overrides**" is the reserved prioritization predicate, it takes a pair of rule labels as arguments.
"**overrides (r1, r2)**" indicates that a rule labelled by "r1" has higher priority than a rule labelled by "r2". "overrides" is otherwise an ordinary predicate symbol which is part of the ordinary logical language.

6. Expressive Restrictions Required by SCLPEngine

Before a rule set is truly digested for inferencing by the OLP inferencing engine portion of the CLPEngine, two expressive restrictions on the OLP rule set are checked in order to guarantee that the OLP rule set is sufficiently wellformed to perform inferencing using CLPEngine's OLP inferencing algorithm. The two expressive restrictions are:

1. RangeRestrictedness Restriction:

All logical variables that appear somewhere in the rule must (also) appear in the following subset of the rule's literals: the FNEGpositive nonbuiltin body literals.
By a "FNEGpositive" literal, we mean that FNEG does not appear in (front of) the literal; i.e., the literal is classical. By "nonbuiltin" literal, we mean that the literal's predicate is not a reserved predicate evaluated in the manner of a builtin.

The rationale for the rangerestrictedness restriction is to ensure that sufficient bindings (i.e., instantiations) for logical variables are available during the course of inferencing to keep from enumerating & testing large or infinite domains for those variables. Many rule inferencing systems employ some form of rangerestrictedness restriction.

For the sake of informativeness, there are three different Exceptions that are reported in the course of checking the rangerestrictedness restriction.

1a. If a variable appears in the head of the rule but does not appear in any FNEGpositive nonbuiltin body literal, then the "**Consequent Variable Binding Check Fails** " Exception is thrown (reported). E.g.,

If b(?Y) AND c(?Y) then a(?X);

1b. If a variable appears in a FNEG'd body literal but does not appear in any FNEGpositive nonbuiltin body literal, then the "**FNEG Variable Binding Check Fails**" Exception is thrown (reported). E.g.,

If b(?X) AND FNEG c(?X, ?Y) then a(?X) ;

1c. If a variable appears in a builtin body literal, but not in any FNEGpositive nonbuiltin body literal, then the "**Function Variable Binding Check Fails**" (sic; a better name would be "Builtin Variable Binding Check Fails") Exception is thrown (reported). E.g.,

If b(?X) AND equals(?X, ?Y) then a(?X) ;

2. PredicateAcyclicity Restriction:

The predicate dependency graph must be acyclic, i.e., not have any (directed) loops. This prevents recursive dependence of a predicate upon itself, through rules.

If a rule set fails to obey this restriction, then the "**ORules Cyclic Check Fails**" Exception is thrown (reported). E.g.,

If b(?X) and c(?Y) then a(?X);
If a(?X) and d(?Z) then b(?X);

The cycle detection algorithm is adapted from one in the book "Telecommunication Design" by Aaron Kershenbaum.

Workaround: XSB and Smodels do not require predicateacyclicity; indeed, they can handle quite general (and difficult) cases of recursive dependence through rules. So if you have a cyclically dependent rule set, you can just try doing inferencing in XSB or Smodels, e.g., **after doing courteous compilation** using the Transformer.

Appendix: Reference Material: Pointers

1. CommonRules overall

http://www.research.ibm.com/rules/ , navigate to papers.

See especially: May 1999 Research Reports / papers.

2. Courteous Logic Programs

http://www.research.ibm.com/rules/papers.html

See especially: Benjamin Grosof, 1997. IBM Research Report RC 20836.
"Courteous Logic Programs: Prioritized Conflict Handling for Rules".

3. Interlingua

http://www.research.ibm.com/rules/papers.html
See especially: May1999 Research Reports / papers, including:
Benjamin Grosof and Yannis Labrou, 1999.
"An Approach to Using XML and a Rulebased Content Language
With an Agent Communication Language".

4. EECOMS

EECOMS is a large project on interenterprise supply chain integration in the domain of manufacturing. http://ciimplex.bocaraton.ibm.com/

5. KIF (Knowledge Interchange format)

http://logic.stanford.edu/kif/dpans.html

6. XSB

http://www.cs.sunysb.edu/~sbprolog/xsbpage.html

7. Smodels

http://saturn.hut.fi/html/staff/ilkka.html

Note that we currently use version 1 of Smodels, not version 2.

8. Java JDK 1.2

http://www.javasoft.com

9. XML and IBM's xml4j package

http://www.alphaworks.ibm.com/tech/xml4j

10. Skij

http://www.alphaworks.ibm.com/tech/Skij

11. Logic programming literature review of standard definitions and results

Chitta Baral and Michael Gelfond. Logic programming and representation. Journal of Logic Programming, 19,20:73148, 1994. Includes extensive review of literature.

RuleML Examples

Following are some examples of RuleML representations.

1. An Authentication example for W3C

```
 <?xml version="1.0" standalone="no" ?>
   <!DOCTYPE rulebase (View Source for full doctype...)>
<rulebase direction="bidirectional">
      <!
       Tim BernersLee's authentication rule (without URI inspection):
      Any person who was some time in the last 2 months an employee of
      an organization which was some time in the last 2 months a W3C member
      may register
   >
<if>
<atom>
 <rel>may register</rel>
 <var>any</var>
   </atom>
<and>
<atom>
 <rel>person</rel>
 <var>any</var>
   </atom>
<atom>
 <rel>organization</rel>
 <var>org</var>
   </atom>
<atom>
 <rel>employee in</rel>
 <var>any</var>
 <var>org</var>
<cterm>
 <ctor>last</ctor>
<cterm>
 <ctor>month</ctor>
 <ind>2</ind>
   </cterm>
   </cterm>
   </atom>
<atom>
 <rel>member in</rel>
 <var>org</var>
 <ur label="W3C">http://www.w3.org/</ur>
<cterm>
 <ctor>last</ctor>
<cterm>
```

```
 <ctor>month</ctor>
 <ind>2</ind>
   </cterm>
   </cterm>
   </atom>
   </and>
   </if>
   </rulebase>
```

2. Broker Example

```
  <?xml version="1.0" standalone="no" ?>
    <!DOCTYPE rulebase (View Source for full doctype...)>
<rulebase direction="bidirectional">
   <!
   a simplified broker definition
    >
<if>
   <!
   conc: atomic formula
    >
<atom>
 <rel>buy</rel>
 <var>broker</var>
 <var>prod</var>
   </atom>
  <!
   prem: explicit and
    >
<and>
<atom>
 <rel>want</rel>
 <ind>customerofbroker</ind>
 <var>prod</var>
   </atom>
<atom>
 <rel>deal</rel>
 <var>prod</var>
<cterm>
 <ctor>cost</ctor>
 <ind>prod</ind>
   </cterm>
 <ind>benefitofprod</ind>
   </atom>
   </and>
   </if>
<if>
<atom>
```

```
 <rel>deal</rel>
 <var>prod</var>
 <var>cost</var>
 <var>benefit</var>
   </atom>
<atom>
 <rel>greaterorequal</rel>
 <var>benefit</var>
 <var>cost</var>
   </atom>
   </if>
<if>
<atom>
 <rel>want</rel>
 <ind>miller</ind>
 <ur label="current info on stock1">www.stock.com/stock1</ur>
   </atom>
 <and />
   </if>
<if>
<atom>
 <rel>want</rel>
 <ind>miller</ind>
 <ur>www.stock.com/stock2</ur>
   </atom>
 <and />
   </if>
   </rulebase>
```

3. Payment Example

```
 <?xml version="1.0" standalone="no" ?>
   <!DOCTYPE rulebase (View Source for full doctype...)>
<rulebase direction="bidirectional">
      <!
       The function symbol "owner" could be userdefined by a rule
      or 'directed equation' owner(car17) = john. Now, the first premise of
      the 'functional Datalog' rule
      buy(Person,Object) < pay(Person,owner(Object)) and
      take(Person,Object)
      is an "owner" function call nested into a "pay" relation call. With
      additional facts pay(fred,john) and take(fred,car17) also asserted,
      the "buy" relation call buy(fred,car17) would be computed thus:
      buy(fred,car17)
      pay(fred,owner(car17)) and take(fred,car17)
      pay(fred,john)         and true
      true                   and true
      true
      So, owner(car17) _computes_ the actual owner individual, rather than
      just _denoting_ it, as a cterm with a constructor symbol "owner"
```

```
      would. For this corresponding cterm in the first premise, we would
      have used square brackets like pay(Person,owner[car17]), but then one
      asserted fact would have to be the nonDatalog pay(fred,owner[car17]).
      Now, the above 'functional Datalog' rule can be flattened, obtaining
      buy(Person,Object) < Own = owner(Object) and pay(Person,Own)
                            and take(Person,Object)
      This flattened version, along with the directed equation and facts,
      can
      be marke up in the current RuleML as follows:
    >
<if>
<eq>
<nano>
 <fun>owner</fun>
 <ind>car17</ind>
   </nano>
 <ind>john</ind>
   </eq>
 <and />
   </if>
<if>
<atom>
 <rel>buy</rel>
 <var>person</var>
 <var>object</var>
   </atom>
<and>
<eq>
 <var>own</var>
<nano>
 <fun>owner</fun>
 <var>object</var>
   </nano>
   </eq>
<atom>
 <rel>pay</rel>
 <var>person</var>
 <var>own</var>
   </atom>
<atom>
 <rel>take</rel>
 <var>person</var>
 <var>object</var>
   </atom>
   </and>
   </if>
<if>
<atom>
 <rel>pay</rel>
 <ind>fred</ind>
 <ind>john</ind>
```

```
    </atom>
  <and />
    </if>
<if>
<atom>
  <rel>take</rel>
  <ind>fred</ind>
  <ind>car17</ind>
    </atom>
  <and />
    </if>
    </rulebase>
```

SweetRules V1.35 software and documentation

The SweetRules current package is available for download at:

http://www.chitro.com/sweet/Sweet1.35.zip

Technical Support on Sweetrules issues can be obtained from:

1. Shashi G. K. - shashi1@cs.umbc.edu (preferable)
2. Chitro Neogy – chitro@sloan.mit.edu
3. Benjamin Grosof – bgrosof@mit.edu