RIF Production Rule Dialect

W3C Editor's Draft 18 December 2008

This version:
http://www.w3.org/2005/rules/wg/draft/ED-rif-prd-20081218/
Latest editor's draft:
http://www.w3.org/2005/rules/wg/draft/rif-prd/
Previous version:
http://www.w3.org/2005/rules/wg/draft/ED-rif-prd-20081125/ (color-coded diff)

Editors:
Christian de Sainte Marie, ILOG
Adrian Paschke, Free University Berlin
Gary Hallmark, Oracle

This document is also available in these non-normative formats: PDF version.

Abstract

This document specifies RIF-PRD, a Rule Interchange Format (RIF) dialect to enable the interchange of production rules.

Status of this Document

May Be Superseded

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

Set of Documents

This document is being published as one of a set of 5 documents:

1. RIF Use Cases and Requirements
2. RIF Core
3. RIF Datatypes and Built-Ins 1.0
4. RIF Production Rule Dialect (this document)
5. RIF Test Cases

Please Comment By 23 January 2009

The Rule Interchange Format (RIF) Working Group seeks public feedback on these Working Drafts. Please send your comments to public-rif-comments@w3.org (public archive). If possible, please offer specific changes to the text that would address your concern. You may also wish to check the Wiki Version of this document for internal-review comments and changes being drafted which may address your concerns.

No Endorsement

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Patents

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Contents

1 Overview

This document specifies the production rule dialect of the W3C rule interchange format (RIF-PRD), a standard XML serialization format for many production rule languages.

Production rules are rule statements defined in terms of both individual facts or objects, and groups of facts or classes of objects. They have an *if* part, or *condition*, and a *then* part, or *action*. The condition is like the condition part of logic rules (as covered by the basic logic dialect of the W3C rule interchange format, RIF-BLD). The *then* part contains actions, which is different to the conclusion part of logic rules which contains only a logical statement. Actions can add, delete, or modify facts in the knowledge base, and have other side-effects.

Example 1.1. *«A customer becomes a "Gold" customer as soon as his cumulative purchases during the current year top $5000»*; *«Customers that become "Gold" customers must be notified immediately, and a golden customer card will be printed and sent to them within one week»*; *«For shopping carts worth more than $1000, "Gold" customers receive an additional discount of 10% of the total amount»*, are all examples of production rules. □

As a production rule interchange format, RIF-PRD specifies an abstract syntax that shares most features with many concrete production rule languages, and it associates each abstract construct with normative semantics and a normative XML concrete syntax.

Production rules are statements of programming logic that specify the execution of one or more actions in the case that their conditions are satisfied. Production rules therefore have an operational semantic (formalizing state changes, e.g., on the basis of a state transition system formalism). The OMG Production Rule Representation specification [PRR] summarizes it as follows:

1. *Match:* the rules are instantiated based on the definition of the rule conditions and the current state of the data source;
2. *Conflict resolution:* a decision algorithm, often called *the conflict resolution strategy*, is applied to select the rule instances to be executed;
3. *Act:* the state of the data source is changed, by executing the selected rule instances' actions. If a terminal state has not been reached, the control loops back to the first step (*Match*).

In the section Operational semantics of rules and rule sets, the semantics for rules and rule sets is specified, accordingly, as a labeled terminal transition system (PLO04), where state stransitions result from executing the action part of instantiated rules. When several rules are deemed able to be executed during the rule execution process, a *conflict resolution strategy* is used to determine the order of rules to execute Sub-section Instance Selection specifies how an intended conflict resolution strategy can be attached to a rule set interchanged with RIF-PRD, and defines a default conflict resolution strategy.

However, as a RIF dialect, RIF-PRD has also been designed to allow interoperability between rule languages over the World Wide Web. In RIF, this is achieved by sharing the same syntax for constructs that have the same semantics across multiple dialects. As a consequence, RIF-PRD shares most of the syntax for rule conditions with RIF-BLD [RIF-BLD], and the semantics associated to the syntactic constructs used for representing the condition part of rules in RIF-PRD is specified, in Section Semantics of condition formulas, in terms of a model theory, as it is in the specification of RIF-BLD as well. In addition to exploiting similarities between the two dialects, it allows them to share the same RIF definitions for data types and built-ins [RIF-DTB].

In the section Operational semantics of actions, the semantics associated with the constructs used to represent the action part of rules in RIF-PRD is specified in terms of a transition relation between successive states of the data source, as defined by the condition formulas that they entail, thus making the link between the model-theoretic semantics of conditions and the operational semantics of rules and rulesets.

The abstract syntax is specified in mathematical english, and the abstract syntactic constructs defined in the sections Abstract Syntax of Conditions, Abstract Syntax of Actions and Abstract Syntax of Rules and Rulesets, are mapped one to one onto the concrete XML syntax in Section XML syntax. A lightweight notation is also defined along with the abstract syntax, to allow for a human-friendlier specification of the semantics. A more complete presentation syntax is specified using an EBNF in Section Presentation Syntax. However, only the XML syntax and the associated semantics are normative. A normative XML schema will also be provided in future versions of the document.

Example 1.2. In RIF-PRD presentation syntax, the first rule in example 1.1. might be represented as follows:

```
Prefix(ex1 http://rif.example.com/2008/prd#)
(* ex1:rule_1 *)
Forall ?customer ?purchasesYTD (
 If      And( ?customer#ex1:Customer
                ?customer[ex1:purchasesYTD->?purchasesYTD]
                External(pred:numeric-greater-than(?purchasesYTD 5000)))
 Then ex1:Gold(?customer) )
```

The condition languages of RIF-PRD and RIF-BLD have much in common, including much of their semantics. Although their abstract syntax and rule semantics are different, due to the operational nature of the actions in production rules, there is a subset for which they are equivalent: essentially, rules with no negation and no uninterpreted functions in the condition, and with only assertions in the action part. For that subset, the XML syntax is the same, so many XML documents are valid in both dialects and have the same meaning. The correspondence between RIF-PRD and RIF-BLD is detailed in Appendix Compatibility with RIF-BLD.

This document is mostly intended for the designers and developers of RIF-PRD implementations, that is, applications that serialize production rules as RIF-PRD XML (producer applications) and/or that deserialize RIF-PRD XML documents into production rules (consumer applications).

2 Conditions

This section specifies the language of the rule conditions that can be serialized using RIF-PRD, by specifying:

- the abstract syntax that all production rule languages interchanging rules using RIF-PRD must have in common for expressing conditions;
- and the intended semantics of the condition formulas in a RIF-PRD document.

Note to the reader: this section depends on Section Constants, Symbol Spaces, and Datatypes of RIF data types and builtins [RIF-DTB].

2.1 Abstract syntax

For a production rule language to be able to interchange rules using RIF-PRD, its alphabet for expressing the condition parts of a rule must, at the abstract syntax level, consist of:

- a countably infinite set of ***constant symbols*** `Const`;
- a countably infinite set of ***variable symbols*** `Var` (disjoint from `Const`);
- a countably infinite set of argument names, `ArgNames` (disjoint from `Const` and `Var`);
- syntactic constructs to denote:
  - Function calls;
  - Relations, including equality, class membership and subclass relations;
  - conjunction, disjunction and negation;
  - existential conditions.

For the sake of readibility and simplicity, this specification introduces a notation for these constructs. That notation is not intended to be a concrete syntax, so it leaves out many details: the only concrete syntax for RIF-PRD is the XML syntax.

Notice that the production rule systems for which RIF-PRD aims to provide a common XML serialization use only externally specified functions, e.g. builtins. RIF-BLD specifies, in addition, a construct to denote uninterpreted function symbols, which RIF-PRD does not require: this is one of two differences between the alphabets used in the condition languages of RIF-PRD and RIF-BLD.

The second point of difference is that RIF-PRD does support a form of negation. RIF-BLD does not support negation because logic rule languages use many different kinds of negations, none of them prevalent enough to justify inclusion in the basic logic dialect of RIF (see also the RIF framework for logic dialects).

2.1.1 Terms

The most basic construct that can be serialized using RIF-PRD is the *term*. RIF-PRD provides for the representation and interchange of several kinds of terms: *constants*, *variables*, *positional* terms and terms with *named arguments*

Definition (Term).

1. *Constants and variables*. If $\mathtt{t} \in \mathtt{Const}$ or $\mathtt{t} \in \mathtt{Var}$ then `t` is a ***simple term***.
2. *Positional terms*. If $\mathtt{t} \in \mathtt{Const}$ and $\mathtt{t}_1$, ..., $\mathtt{t}_n$, $n \geq 0$, are terms then $\mathtt{t(t_1\ ...\ t_n)}$ is a ***positional term***.
   Here, the constant `t` represents a function and $\mathtt{t}_1$, ..., $\mathtt{t}_n$ represent argument values.
3. *Terms with named arguments*. A ***term with named arguments*** is of the form $\mathtt{t(s_1\text{->}v_1\ ...\ s_n\text{->}v_n)}$, where $n \geq 0$, $\mathtt{t} \in \mathtt{Const}$ and $\mathtt{v}_1$, ..., $\mathtt{v}_n$ are

terms and s_1, ..., s_n are pairwise distinct symbols from the set `ArgNames`. The constant `t` here represents a function; s_1, ..., s_n represent argument names; and v_1, ..., v_n represent argument values. The argument names, s_1, ..., s_n, are required to be pairwise distinct. Terms with named arguments are like positional terms except that the arguments are named and their order is immaterial. Note that a term of the form `f()` is, trivially, both a positional term and a term with named arguments. ☐

2.1.2 Atomic formulas

The atomic truth-valued constructs that can be serialized using RIF-PRD are called *atomic formulas*.

Definition (Atomic formula). An ***atomic formula*** can have several different forms and is defined as follows:

1. *Positional atomic formulas*. If $t \in$ `Const` and t_1, ..., t_n, $n \geq 0$, are terms then $t(t_1\ \ldots\ t_n)$ is a ***positional atomic formula*** (or simply a ***positional atom***).
2. *Atomic formulas with named arguments*. An ***atomic formula with named arguments*** (or simply a ***atom with named arguments***) is of the form $t(s_1\text{->}v_1\ \ldots\ s_n\text{->}v_n)$, where $n \geq 0$, $t \in$ `Const` and v_1, ..., v_n are terms and s_1, ..., s_n are pairwise distinct symbols from the set `ArgNames`. The constant `t` here represents a predicate; s_1, ..., s_n represent argument names; and v_1, ..., v_n represent argument values. The argument names, s_1, ..., s_n, are required to be pairwise distinct. Atoms with named arguments are like positional Atoms except that the arguments are named and their order is immaterial. Note that an atom of the form `t()` is, trivially, both a positional atom and an atom with named arguments.
3. *Equality atomic formulas*. `t = s` is an ***equality atomic formula*** (or, simply, an ***equality***), if `t` and `s` are terms.
4. *Class membership atomic formulas* (or just *membership*). `t#s` is a ***membership atomic formula*** if `t` and `s` are terms. the term `t` is the *object* and the term `s` is the *class*.
5. *Subclass atomic formulas*. `t##s` is a ***subclass atomic formula*** if `t` and `s` are terms.
6. *Frame atomic formulas*. $t[p_1\text{->}v_1\ \ldots\ p_n\text{->}v_n]$ is a ***frame atomic formula*** (or simply a ***frame***) if t, p_1, ..., p_n, v_1, ..., v_n, $n \geq 0$, are terms. The term `t` is the *object* of the frame; the p_i are the *property* or *attribute* names; and the v_i are the property or attribute *values*. In this document, an attribute/value pair is sometimes called a *slot*.
   Membership, subclass, and frame atomic formulas are used to describe objects, classifications and class hierarchies.
7. *Externally defined atomic formulas.* If `t` is a positional, named-argument, or a frame atomic formula then `External(t)` is an ***externally defined atomic formula***. Such atomic formulas are used for representing built-in predicates. ☐

Note that not only predicates, but also frame atomic formulas can be externally defined. Therefore, external information sources can be modeled in an object-oriented way via frames.

Editor's Note: Objects are commonly used in PR systems. In this draft, we reuse frame, membership, and subclass formulas (from RIF-BLD) to model objects. We are aware of current limits, such as difficulty expressing datatype and cardinality constraints. Future drafts will address that problem. We are interested in feedback on the merits and limitations of this approach.

Observe that the argument names of frames, p_1, ..., p_n, are terms and so, as a special case, can be variables. In contrast, atoms with named arguments can use only the symbols from `ArgNames` to represent their argument names, which can neither be constants from `Const` nor variables from `Var`.

Note that atomic formulas are sometimes also called *terms*, e.g. in the realm of logic languages: the specification of RIF-BLD, in particular, follows that usage. The abstract syntactic elements that are called *terms* in this specification, are called *basic terms* in the specification of RIF-BLD.

2.1.3 Formulas

Composite truth-valued constructs that can be serialized using RIF-PRD are called *formulas*.

Note that terms (constants, variables and functions) are *not* formulas.

More general formulas are constructed out of the atomic formulas with the help of logical connectives.

Definition (Condition formula). A ***condition formula*** can have several different forms and is defined as follows:

1. *Atomic formula*: If φ is an atomic formula then it is also a condition formula.
2. *Conjunction*: If φ_1, ..., φ_n, $n \geq 0$, are condition formulas then so is $\mathtt{And}(\varphi_1\ ...\ \varphi_n)$, called a *conjunctive* formula. As a special case, `And()` is allowed and is treated as a tautology, i.e., a formula that is always true.
3. *Disjunction*: If φ_1, ..., φ_n, $n \geq 0$, are condition formulas then so is $\mathtt{Or}(\varphi_1\ ...\ \varphi_n)$, called a *disjunctive* formula. As a special case, `Or()` is permitted and is treated as a contradiction, i.e., a formula that is always false.
4. *Negation*: If φ is a condition formula, then so is $\mathtt{Not}(\varphi)$, called a *negative* formula.
5. *Existentials*: If φ is a condition formula and $?V_1$, ..., $?V_n$, $n>0$, are variables then $\mathtt{Exists}\ ?V_1\ ...\ ?V_n(\varphi)$ is an *existential* formula. □

In the definition of a formula, the component formulas φ and φ_i are said to be ***subformulas*** of the respective condition formulas that are built using these components.

The function ***Var(e)*** that maps a term, atomic formula or formula *e* to the set of its free variables is defined as follows:

- if $e \in Const$, then $Var(e) = \{\}$;
- if $e \in Var$, then $Var(e) = \{e\}$;
- if p and arg_i, i = 0...n, are terms, then, $Var(p(arg_i) = Var(p) \cup_{i=0...n} Var(arg_i)$;
- if p and arg_i, i = 0...n, are terms, then, $Var(External(p(arg_i)) = Var(p) \cup_{i=0...n} Var(arg_i)$;
- if t_1 and t_2 are terms, then $Var(t_1\ [=|\#|\#\#]\ t_2)\ Var(t_1) \cup Var(t_2)$;
- if o', k_i, i = 1...n, and v_i, i = 1...n, are terms, then $Var(o[k_1\text{->}v_1 ... k_n\text{->}v_n])\ Var(o) \cup_{i=1...n} Var(k_i) \cup_{i=1...n} Var(v_i)$;
- if f_i, i = 0...n, are condition formulas, then $Var([AND|OR|NOT](f_i)) \cup_{i=0...n} Var(f_i)$;
- if f is a condition formula and $x_i \in Var$ for i = 1...n, then, $Var(Exists\ x_1 ... x_n\ (f)) = Var(f) - \{x_i \mid i = 1...n\}$.

Definition (Ground formula). A condition formula φ is a ***ground formula*** if and only if $Var\varphi = \{\}$ and φ does not contain any existential subformula. ☐

In other words, a ground formula does not contain any variable term.

2.1.4 Well-formed formulas

The specification of RIF-PRD does not assign a standard meaning to all the formulas that can be serialized using its concrete XML syntax: formulas that can be meaningfully serialized are called *well-formed*. Not all formulas are well-formed with respect to RIF-PRD: it is required that no constant appear in more than one context. What this means precisely is explained below.

The set of all constant symbols, `Const`, is partitioned into several subsets as follows:

- A subset of individuals.
  The symbols in `Const` that belong to the primitive datatypes are required to be individuals.
- A number of subsets for predicate symbols such that there is one subset per symbol arity (defined below) for externally defined predicates and one for non-external predicates.
  Note that this implies that symbols used for external predicate names cannot be used for other predicates. Also, the definition of arity, below, implies that the arities for positional predicate symbols and for predicate symbols with named arguments are distinct even if the numbers of arguments are the same. Therefore, symbols that are used for positional

predicates cannot be used for predicates with named arguments, and vice versa.
- A number of subsets of function symbols (only externally defined function can be serialized using RIF-PRD). As with predicate symbols, there are separate subsets for symbols with different arities; function symbols with named arguments are in their own subsets. The only exception is the case of nullary symbols, which take zero arguments as in `f()`, since they are considered to be both positional and named-argument symbols.

Each predicate and function symbol that take at least one argument has precisely one ***arity***. For positional predicate and function symbols, an arity is a non-negative integer that tells how many arguments the symbol can take. For symbols that take named arguments, an arity is a set $\{s_1 \ \ldots \ s_k\}$ of argument names ($s_i \in$ `ArgNames`) that are allowed for that symbol. Nullary symbols (which take zero arguments) are said to have the arity 0.

An important point is that neither the above partitioning of constant symbols nor the arity are specified explicitly. Instead, the arity of a symbol and its type is determined by the context in which the symbol is used.

Definition (Context of a symbol). The ***context of an occurrence*** of a symbol, `s∈Const`, in a formula, φ, is determined as follows:

- If `s` occurs as a function symbol in a term of the form `s(...)` with arity α then `s` occurs in the *context of an external function symbol with arity* α (or simply the *context of a function symbol with arity* α, since RIF-PRD knows only external functions);
- If `s` occurs as a predicate in an atomic subformula of the form `s(...)` with arity α then `s` occurs in the *context of a predicate symbol with arity* α;
- If `s` occurs as a predicate in an atomic subformula `External(s(...))` with arity α then `s` occurs in the *context of an external predicate symbol with arity* α;
- If `s` occurs in any other context (in a frame: `s[...]`, `...[s->...]`, or `...[...->s]`; or in a positional/named argument term: `p(...s...)`, `q(...->s...)`), it is said to occur as an *individual*. □

Definition (Well-formed formula). A formula φ is ***well-formed*** iff:

- every constant symbol mentioned in φ occurs in exactly one context.
- Whenever a formula contains a function term `t` or an external atomic formula `External(t)`, `t` must be an instance of the coherent set of external schemas (Section Schemas for Externally Defined Terms of RIF data types and builtins [RIF-DTB]) associated with the language of RIF-PRD.
- If `t` is an instance of the coherent set of external schemas associated with the language then `t` can occur only as a function term `t` or as an external atomic formula `External(t)`, i.e., as an external term or atomic formula. □

Definition (RIF-PRD condition language). The ***RIF-PRD condition language*** consists of the set of all well-formed formulas. ☐

2.2 Semantics of condition formulas

This section specifies the intended semantics of the condition formulas in a RIF-PRD document.

For compatibility with other RIF specifications (in particular, RIF data types and builtins), and to make the interoperability with RIF logic dialects (in particular RIF Core [RIF-Core] and RIF-BLD), the intended semantics for RIF-PRD condition formulas is specified in terms of a model theory.

2.2.1 Semantic structures

Definition (Semantic structure). A ***semantic structure***, ***I***, is a tuple of the form <***TV***, ***DTS***, ***D***, $\boldsymbol{D}_{\mathrm{ind}}$, $\boldsymbol{D}_{\mathrm{func}}$, $\boldsymbol{I}_{\mathrm{C}}$, $\boldsymbol{I}_{\mathrm{V}}$, $\boldsymbol{I}_{\mathrm{F}}$, $\boldsymbol{I}_{\mathrm{frame}}$, $\boldsymbol{I}_{\mathrm{NF}}$, $\boldsymbol{I}_{\mathrm{sub}}$, $\boldsymbol{I}_{\mathrm{isa}}$, $\boldsymbol{I}_{=}$, $\boldsymbol{I}_{\mathrm{external}}$, $\boldsymbol{I}_{\mathrm{truth}}$>. Here ***D*** is a non-empty set of elements called the ***Herbrand domain*** of '***I***, i.e., the set of all ground terms which can be formed by using the elements of `Const`. $\boldsymbol{D}_{\mathrm{ind}}$, $\boldsymbol{D}_{\mathrm{func}}$ are nonempty subsets of ***D***. $\boldsymbol{D}_{\mathrm{ind}}$ is used to interpret the elements of `Const` that are individuals and $\boldsymbol{D}_{\mathrm{func}}$ is used to interpret the elements of `Const` that are function symbols. `Const` denotes the set of all constant symbols and `Var` the set of all variable symbols. ***TV*** denotes the set of truth values that the semantic structure uses and ***DTS*** is a set of identifiers for primitive datatypes (please refer to Section Datatypes of RIF data types and builtins [RIF-DTB] for the semantics of datatypes). The set of all ground (positional|named|frame|external) formulas which can be formed by using the function symbols with the ground terms in the Herbrand domain is the ***Herbrand base***, $\boldsymbol{H}_{\mathrm{B}}$. A semantic structure ***I*** is a ***Herbrand interpretation***, $\boldsymbol{I}_{\mathrm{H}}$, if the corresponding subset of $\boldsymbol{H}_{\mathrm{B}}$ is the set of all ground formulas which are true with respect to ***I***. ☐

As far as the assignment of a standard meaning to formulas in the RIF-PRD condition language is concerned, the set ***TV*** of truth values consists of just two values, **t** and **f**.

The other components of ***I*** are *total* mappings defined as follows:

1. $\boldsymbol{I}_{\mathrm{C}}$ maps `Const` to ***D***.

   This mapping interprets constant symbols. In addition:

   - If a constant, `c` ∈ `Const`, is an *individual* then it is required that $\boldsymbol{I}_{\mathrm{C}}$(`c`) ∈ $\boldsymbol{D}_{\mathrm{ind}}$.
   - If `c` ∈ `Const`, is a *function symbol* (positional or with named arguments) then it is required that $\boldsymbol{I}_{\mathrm{C}}$(`c`) ∈ $\boldsymbol{D}_{\mathrm{func}}$.
2. $\boldsymbol{I}_{\mathrm{V}}$ maps `Var` to $\boldsymbol{D}_{\mathrm{ind}}$.

This mapping interprets variable symbols.

3. $\boldsymbol{I}_{\mathrm{F}}$ maps $\boldsymbol{D}$ to functions $\boldsymbol{D}^*_{\mathrm{ind}} \rightarrow \boldsymbol{D}$ (here $\boldsymbol{D}^*_{\mathrm{ind}}$ is a set of all sequences of any finite length over the domain $\boldsymbol{D}_{\mathrm{ind}}$).

   This mapping interprets positional terms and gives meaning to positional predicate function.

4. $\boldsymbol{I}_{\mathrm{NF}}$ maps $\boldsymbol{D}$ to the set of total functions of the form `SetOfFiniteSets(ArgNames` × $\boldsymbol{D}_{\mathrm{ind}}$) → $\boldsymbol{D}$.

   This mapping interprets function symbols with named arguments and gives meaning to named argument functions. This is analogous to the interpretation of positional terms with two differences:

   - Each pair <`s,v`> ∈ `ArgNames` × $\boldsymbol{D}_{\mathrm{ind}}$ represents an argument/value pair instead of just a value in the case of a positional term.
   - The arguments of a term with named arguments constitute a finite set of argument/value pairs rather than a finite ordered sequence of simple elements. So, the order of the arguments does not matter.

5. $\boldsymbol{I}_{\mathrm{frame}}$ maps $\boldsymbol{D}_{\mathrm{ind}}$ to total functions of the form `SetOfFiniteBags`($\boldsymbol{D}_{\mathrm{ind}}$ × $\boldsymbol{D}_{\mathrm{ind}}$) → $\boldsymbol{D}$.

   This mapping interprets frame terms and gives meaning to frame functions. An argument, `d` ∈ $\boldsymbol{D}_{\mathrm{ind}}$, to $\boldsymbol{I}_{\mathrm{frame}}$ represents an object and the finite bag {<`a1,v1`>, ..., <`ak,vk`>} represents a bag of attribute-value pairs for `d`. $\boldsymbol{I}_{\mathrm{frame}}$ is used to determine the truth valuation of frame terms.

   Bags (multi-sets) are used here because the order of the attribute/value pairs in a frame is immaterial and pairs may repeat: `o[a->b a->b]`. Such repetitions arise naturally when variables are instantiated with constants. For instance, `o[?A->?B ?C->?D]` becomes `o[a->b a->b]` if variables `?A` and `?C` are instantiated with the symbol `a` and `?B`, `?D` with `b`.

6. $\boldsymbol{I}_{\mathrm{sub}}$ gives meaning to the subclass relationship. It is a mapping of the form $\boldsymbol{D}_{\mathrm{ind}}$ × $\boldsymbol{D}_{\mathrm{ind}}$ → $\boldsymbol{D}$.

   The operator `##` is required to be transitive, i.e., `c1 ## c2` and `c2 ## c3` must imply `c1 ## c3`. TThis is ensured by a restriction in Section Interpretation of condition formulas;

7. $\boldsymbol{I}_{\mathrm{isa}}$ gives meaning to class membership. It is a mapping of the form $\boldsymbol{D}_{\mathrm{ind}}$ × $\boldsymbol{D}_{\mathrm{ind}}$ → $\boldsymbol{D}$.

   The relationships `#` and `##` are required to have the usual property that all members of a subclass are also members of the superclass, i.e., `o # cl` and cl ## scl must imply o # scl. This is ensured by a restriction in Section Interpretation of condition formulas;

8. $\boldsymbol{I}_{=}$ is a mapping of the form $\boldsymbol{D}_{\text{ind}} \times \boldsymbol{D}_{\text{ind}} \rightarrow \boldsymbol{D}$.

   It gives meaning to the equality operator.

9. $\boldsymbol{I}_{\text{truth}}$ is a mapping of the form $\boldsymbol{D} \rightarrow \boldsymbol{TV}$.

   It is used to define truth valuation for formulas.

10. $\boldsymbol{I}_{\text{external}}$ is a mapping from the coherent set of schemas for externally defined functions to total functions $\boldsymbol{D}^* \rightarrow \boldsymbol{D}$. For each external schema σ = (?X$_1$ ... ?X$_n$; τ) in the coherent set of external schemas associated with the language, $\boldsymbol{I}_{\text{external}}(\sigma)$ is a function of the form $\boldsymbol{D}^n \rightarrow \boldsymbol{D}$.

    For every external schema, σ, associated with the language, $\boldsymbol{I}_{\text{external}}(\sigma)$ is assumed to be specified externally in some document (hence the name *external schema*). In particular, if σ is a schema of a RIF built-in predicate or function, $\boldsymbol{I}_{\text{external}}(\sigma)$ is specified in [RIF-DTB] so that:

    - If σ is a schema of a built-in function then $\boldsymbol{I}_{\text{external}}(\sigma)$ must be the function defined in the aforesaid document.
    - If σ is a schema of a built-in predicate then $\boldsymbol{I}_{\text{truth}} \circ (\boldsymbol{I}_{\text{external}}(\sigma))$ (the composition of $\boldsymbol{I}_{\text{truth}}$ and $\boldsymbol{I}_{\text{external}}(\sigma)$, a truth-valued function) must be as specified in [RIF-DTB].

For convenience, we also define the following mapping ***I*** from terms to ***D***:

- $\boldsymbol{I}$(k) = $\boldsymbol{I}_{\text{C}}$(k), if k is a symbol in Const;
- $\boldsymbol{I}$(?v) = $\boldsymbol{I}_{\text{V}}$(?v), if ?v is a variable in Var;
- $\boldsymbol{I}$(p(t$_1$ ... t$_n$)) = $\boldsymbol{I}_{\text{P}}$($\boldsymbol{I}$(p))($\boldsymbol{I}$(t$_1$),...,$\boldsymbol{I}$(t$_n$));
- $\boldsymbol{I}$(p(s$_1$->v$_1$ ... s$_n$->v$_n$)) = $\boldsymbol{I}_{\text{NF}}$($\boldsymbol{I}$(p))({<s$_1$,$\boldsymbol{I}$(v$_1$)>,...,<s$_n$,$\boldsymbol{I}$(v$_n$)>})
  Here we use {...} to denote a set of argument/value pairs;
- $\boldsymbol{I}$(o[a$_1$->v$_1$ ... a$_k$->v$_k$]) = $\boldsymbol{I}_{\text{frame}}$($\boldsymbol{I}$(o))({<$\boldsymbol{I}$(a$_1$),$\boldsymbol{I}$(v$_1$)>, ..., <$\boldsymbol{I}$(a$_n$),$\boldsymbol{I}$(v$_n$)>})
  Here {...} denotes a bag of attribute/value pairs.
- $\boldsymbol{I}$(c1##c2) = $\boldsymbol{I}_{\text{sub}}$($\boldsymbol{I}$(c1), $\boldsymbol{I}$(c2));
- $\boldsymbol{I}$(o#c) = $\boldsymbol{I}_{\text{isa}}$($\boldsymbol{I}$(o), $\boldsymbol{I}$(c));
- $\boldsymbol{I}$(x=y) = $\boldsymbol{I}_{=}$($\boldsymbol{I}$(x), $\boldsymbol{I}$(y));
- $\boldsymbol{I}$(External(t)) = $\boldsymbol{I}_{\text{external}}(\sigma)$($\boldsymbol{I}$(s$_1$), ..., $\boldsymbol{I}$(s$_n$)), if t is an instance of the external schema σ = (?X$_1$ ... ?X$_n$; τ) by substitution ?X$_1$/s$_1$ ... ?X$_n$/s$_1$.
  Note that, by definition, External(t) is well-formed only if t is an instance of an external schema. Furthermore, by the definition of coherent sets of external schemas, t can be an instance of at most one such schema, so $\boldsymbol{I}$(External(t)) is well-defined.

The effect of datatypes. The set ***DTS*** must include the datatypes described in Section Primitive Datatypes of RIF data types and builtins [RIF-DTB].

The datatype identifiers in ***DTS*** impose the following restrictions. Given `dt` $\in$ ***DTS***, let $\boldsymbol{LS}_{\text{dt}}$ denote the lexical space of `dt`, $\boldsymbol{VS}_{\text{dt}}$ denote its value space, and $\boldsymbol{L}_{\text{dt}}$: $\boldsymbol{LS}_{\text{dt}} \rightarrow \boldsymbol{VS}_{\text{dt}}$ the lexical-to-value-space mapping (for the definitions of these concepts, see Section Primitive Datatypes of RIF data types and builtins [RIF-DTB]. Then the following must hold:

- $\boldsymbol{VS}_{\text{dt}} \subseteq \boldsymbol{D}_{\text{ind}}$; and
- For each constant `"lit"^^dt` such that `lit` $\in \boldsymbol{LS}_{\text{dt}}$, $\boldsymbol{I}_{\text{C}}$(`"lit"^^dt`) = $\boldsymbol{L}_{\text{dt}}$(`lit`).

That is, $\boldsymbol{I}_{\text{C}}$ must map the constants of a datatype `dt` in accordance with $\boldsymbol{L}_{\text{dt}}$.

RIF-PRD does not impose restrictions on $\boldsymbol{I}_{\text{C}}$ for constants in symbol spaces that are not datatypes included in ***DTS***.

2.2.2 Interpretation of condition formulas

This section defines how a semantic structure, ***I***, determines the truth value $TVal_{\text{I}}(\varphi)$ of a condition formula, φ. In PRD a semantic structure is represented as a Herbrand interpretation.

We define a mapping, $TVal_{\text{I}}$, from the set of all condition formulas to ***TV***. Note that the definition implies that $TVal_{\text{I}}(\varphi)$ is defined *only if* the set ***DTS*** of the datatypes of ***I*** includes all the datatypes mentioned in φ and $\boldsymbol{I}_{\text{external}}$ is defined on all externally defined functions and predicates in φ.

Definition (Truth valuation). ***Truth valuation*** for well-formed condition formulas in RIF-PRD is determined using the following function, denoted $TVal_{\text{I}}$:

- *Positional atomic formulas*: $TVal_{\text{I}}(\texttt{r}(\texttt{t}_1\ \ldots\ \texttt{t}_n)) = \boldsymbol{I}_{\text{truth}}(\boldsymbol{I}(\texttt{r}(\texttt{t}_1\ \ldots\ \texttt{t}_n)))$;
- *Atomic formulas with named arguments*: $TVal_{\text{I}}(\texttt{p}(\texttt{s}_1\texttt{->}\texttt{v}_1\ \ldots\ \texttt{s}_k\texttt{->}\texttt{v}_k)) = \boldsymbol{I}_{\text{truth}}(\boldsymbol{I}(\texttt{p}(\texttt{s}_1\texttt{->}\texttt{v}_1\ \ldots\ \texttt{s}_k\texttt{->}\texttt{v}_k)))$;
- *Equality*: $TVal_{\text{I}}(\texttt{x = y}) = \boldsymbol{I}_{\text{truth}}(\boldsymbol{I}(\texttt{x = y}))$.
  To ensure that equality has precisely the expected properties, it is required that:
    - $\boldsymbol{I}_{\text{truth}}(\boldsymbol{I}(\texttt{x = y})) = \mathbf{t}$ if $\boldsymbol{I}(\texttt{x}) = \boldsymbol{I}(\texttt{y})$ and that $\boldsymbol{I}_{\text{truth}}(\boldsymbol{I}(\texttt{x = y})) = \mathbf{f}$ otherwise. This is tantamount to saying that $TVal_{\text{I}}(\texttt{x = y}) = \mathbf{t}$ iff $\boldsymbol{I}(\text{x}) = \boldsymbol{I}(\text{y})$;
- *Subclass*: $TVal_{\text{I}}(\texttt{sc \#\# cl}) = \boldsymbol{I}_{\text{truth}}(\boldsymbol{I}(\texttt{sc \#\# cl}))$.
  To ensure that the operator `##` is transitive, i.e., `c1 ## c2` and `c2 ## c3` imply `c1 ## c3`, the following is required:
    - For all `c1`, `c2`, `c3` $\in \boldsymbol{D}$, if $TVal_{\text{I}}(\texttt{c1 \#\# c2}) = TVal_{\text{I}}(\texttt{c2 \#\# c3}) = \mathbf{t}$ then $TVal_{\text{I}}(\texttt{c1 \#\# c3}) = \mathbf{t}$;

- *Membership*: $TVal_{I}(\texttt{o \# cl}) = \boldsymbol{I}_{truth}(\boldsymbol{I}(\texttt{o \# cl}))$.
  To ensure that all members of a subclass are also members of the superclass, i.e., `o # cl` and `cl ## scl` implies `o # scl`, the following is required:
    - For all $\texttt{o}, \texttt{cl}, \texttt{scl} \in \boldsymbol{D}$, if $TVal_{I}(\texttt{o \# cl}) = TVal_{I}(\texttt{cl \#\# scl}) = \mathbf{t}$ then $TVal_{I}(\texttt{o \# scl}) = \mathbf{t}$;
- *Frame*: $TVal_{I}(\texttt{o[}\texttt{a}_1\texttt{->}\texttt{v}_1 \ \ldots \ \texttt{a}_k\texttt{->}\texttt{v}_k\texttt{]}) = \boldsymbol{I}_{truth}(\boldsymbol{I}(\texttt{o[}\texttt{a}_1\texttt{->}\texttt{v}_1 \ \ldots \ \texttt{a}_k\texttt{->}\texttt{v}_k\texttt{]}))$.
  Since the bag of attribute/value pairs represents the conjunctions of all the pairs, the following is required, if `k > 0`:
    - $TVal_{I}(\texttt{o[}\texttt{a}_1\texttt{->}\texttt{v}_1 \ \ldots \ \texttt{a}_k\texttt{->}\texttt{v}_k\texttt{]}) = \mathbf{t}$ if and only if $TVal_{I}(\texttt{o[}\texttt{a}_1\texttt{->}\texttt{v}_1\texttt{]}) = \ldots = TVal_{I}(\texttt{o[}\texttt{a}_k\texttt{->}\texttt{v}_k\texttt{]}) = \mathbf{t}$;
- *Externally defined atomic formula*: $TVal_{I}(\texttt{External(t)}) = \boldsymbol{I}_{truth}(\boldsymbol{I}_{external}(\sigma)(\boldsymbol{I}(\texttt{s}_1), \ldots, \boldsymbol{I}(\texttt{s}_n)))$, if `t` is an atomic formula that is an instance of the external schema $\sigma = (\texttt{?X}_1 \ \ldots \ \texttt{?X}_n\texttt{;} \ \tau)$ by substitution $\texttt{?X}_1\texttt{/s}_1 \ \ldots \ \texttt{?X}_n\texttt{/s}_1$.
  Note that, by definition, `External(t)` is well-formed only if `t` is an instance of an external schema. Furthermore, by the definition of coherent sets of external schemas, `t` can be an instance of at most one such schema, so $\boldsymbol{I}(\texttt{External(t)})$ is well-defined;
- *Conjunction*: $TVal_{I}(\texttt{And(}c_1 \ldots c_n\texttt{)}) = \mathbf{t}$ if and only if $TVal_{I}(c_1) = \ldots = TVal_{I}(c_n) = \mathbf{t}$. Otherwise, $TVal_{I}(\texttt{And(}c_1 \ldots c_n\texttt{)}) = \mathbf{f}$.
  The empty conjunction is treated as a tautology, so $TVal_{I}(\texttt{And()}) = \mathbf{t}$;
- *Disjunction*: $TVal_{I}(\texttt{Or(}c_1 \ldots c_n\texttt{)}) = \mathbf{f}$ if and only if $TVal_{I}(c_1) = \ldots = TVal_{I}(c_n) = \mathbf{f}$. Otherwise, $TVal_{I}(\texttt{Or(}c_1 \ldots c_n\texttt{)}) = \mathbf{t}$.
  The empty disjunction is treated as a contradiction, so $TVal_{I}(\texttt{Or()}) = \mathbf{f}$;
- *Negation*: $TVal_{I}(\texttt{Not(}c\texttt{)}) = \mathbf{f}$ if and only if $TVal_{I}(c) = \mathbf{t}$. Otherwise, $TVal_{I}(\texttt{Not(}c\texttt{)}) = \mathbf{t}$;
- *Existence*: $TVal_{I}(\texttt{Exists ?v}_1 \ \ldots \ \texttt{?v}_n \ (\varphi)) = \mathbf{t}$ if and only if for some $\boldsymbol{I}^*$, described below, $TVal_{I^*}(\varphi) = \mathbf{t}$.
  Here $\boldsymbol{I}^*$ is a semantic structure of the form $<\boldsymbol{TV}, \boldsymbol{DTS}, \boldsymbol{D}, \boldsymbol{D}_{ind}, \boldsymbol{D}_{pred}, \boldsymbol{I}_C, \boldsymbol{I}^*_V, \boldsymbol{I}_P, \boldsymbol{I}_{frame}, \boldsymbol{I}_{NP}, \boldsymbol{I}_{sub}, \boldsymbol{I}_{isa}, \boldsymbol{I}_{=}, \boldsymbol{I}_{external}, \boldsymbol{I}_{truth}>$, which is exactly like $\boldsymbol{I}$, except that the mapping $\boldsymbol{I}^*_V$, is used instead of $\boldsymbol{I}_V$. $\boldsymbol{I}^*_V$ is defined to coincide with $\boldsymbol{I}_V$ on all variables except, possibly, on $\texttt{?v}_1,\ldots,\texttt{?v}_n$. □

2.2.3 Satisfaction of a condition

We now define what it means for a set of ground formulas to satisfy a condition formula. The satisfaction of condition formulas by a set of ground formulas provides formal underpinning to the operational semantics of rulesets interchanged using RIF-PRD.

Definition (State). A state $\boldsymbol{S}$ is a Herbrand Interpretation $\boldsymbol{I}_H$. □

Definition (Condition Satisfaction). A condition formula, $\boldsymbol{\varphi}$ is satisfied under variable assignment $\boldsymbol{\sigma}$ in a state $\boldsymbol{S}$, written as $\boldsymbol{S} \models \boldsymbol{\varphi}[\boldsymbol{\sigma}]$, iff $\boldsymbol{TVal}_S(\boldsymbol{\varphi}[\boldsymbol{\sigma}]) = \mathbf{t}$. □

2.2.4 Matching substitution

At the syntactic level, the interpretation of the variables by a valuation function I_V is realized by a *substitution*. The *matching substitution* of constants to variables, as defined below, provides the formal link between the model-theoretic semantics of condition formulas and the operational semantics of rule sets in RIF-PRD.

Let `Term` be the set of the terms in the RIF-PRD condition language (as defined in section Terms).

Definition (Substitution). A ***substitution*** is a finitely non-identical assignment of terms to variables; i.e., a function σ from `Var` to `Term` such that the set $\{x \in \mathtt{Var} \mid x \neq \sigma(x)\}$ is finite. This set is called the domain of σ and denoted by $Dom(\sigma)$. Such a substitution is also written as a set such as $\sigma = \{t_i/x_i\}_{i=0..n}$ where $Dom(\sigma) = \{x_i\}_{i=0..n}$ and $\sigma(x_i) = t_i$, $i = 0..,n$. □

Definition (Ground Substitution). A *ground substitution* is a substitution σ that assigns only ground terms to the variables in $Dom(\sigma)$: $\forall\ x \in Dom(\sigma),\ Var(\sigma(x)) = \emptyset$ □

Notice that since RIF-PRD covers only externally defined interpreted functions, a ground term can always be replaced by a constant. In the remainder of this document, it will always be assumed that a ground substitution assigns only constants to the variables in its domain.

Definition (Matching Substitution). Let ψ be a condition formula, and φ be a set of ground formulas that satisfies ψ. We say that ψ matches φ with substitution σ : Var -> Terms if and only if there is a syntactic interpretation I such that for all $?x_i$ in $Var(\sigma)$, $I(?x_i) = I(\sigma(?x_i))$. □

3 Actions

This section specifies the *action* part of the rules that can be serialized using RIF-PRD (the conclusion of a production rule is often called the *action* part, or, simply, the *action*; the *then* part, with reference to the *if-then* form of a rule statement; or the *right-hand side*, or *RHS*. In the latter case, the condition is usually called the *left-hand side* of the rule, or *LHS*). Specifically, this section specifies:

- the abstract syntax that all production rule languages interchanging rules using RIF-PRD must have in common for expressing actions;
- and the intended semantics of the individual action formulas in a RIF-PRD document.

In production rule systems, the action part of the rules is used, in particular, to add, delete or modify facts in the data source with respect to which the condition of rules are evaluated and the rules instantiated. As a rule interchange format, RIF-PRD does not make any assumption regarding the nature of the data source that the producer or the consumer of a RIF-PRD document uses (e.g. a rule engine's working memory, an external data base, etc). As a consequence, the syntax of the actions that RIF-PRD supports are defined with respect to the RIF-PRD condition formulas that represents the facts that the actions are intended to affect. In the same way, the semantics of the actions is specified in terms of how the effects of their execution are intended to affect the evaluation of rule condition.

Editor's Note: This version of the draft specifies only a very limited set of basic atomic actions. Future draft will extend that set, in particular to support actions whose effect is not, or not only, to modify the fact base.

3.1 Abstract syntax

For a production rule language to be able to interchange rules using RIF-PRD, its alphabet for expressing the action part of a rule must, at the abstract syntax level, consist of syntactic constructs to denote:

- the assertion of a positional atom, an atom with named arguments, or a frame, membership, or subclass atomic formula;
- the retraction of a positional atom, an atom with named arguments, or a frame;
- the addition of a new frame object;
- the removal of a frame object and the retraction of the corresponding frame and class atomic formulas;
- a sequence of these actions, including local variables and a mechanism to bind a local variable to a frame slot value or a new frame object.

Editor's Note: These actions may seem foreign to a reader who is familiar with typical production rule language actions, such as assert an object, modify/update a field of an object, and retract/remove an object. As noted in Section Atomic formulas, in this draft, objects are modeled using frame, membership, and subclass formulas that are re-used from RIF-BLD and RIF-Core. Therefore, the object-oriented actions are defined to act upon frame, membership, and subclass relations. Mappings from a typical Java-like object model to RIF-PRD maps "instanceof" to membership, "extends" and "implements" to subclass, and object properties/fields to frame slots. To assert an object requires asserting both its class membership and its frame slots. To modify a slot, e.g. change color from red to blue, requires retracting the old frame slot and asserting the new frame slot. Indeed, frame slots are multi-valued and, therefore, merely asserting a frame slot does not overwrite a prior value, it adds to the set of values. Frame slots do not inherently constrain either the datatype or the cardinality of their values. Future drafts will address the issue of object representation in RIF-PRD. We are open to suggestions.

3.1.1 Atomic actions

Atomic action constructs take constructs from the RIF-PRD condition language as their arguments.

Definition (Atomic action). An ***atomic action*** can have several different forms and is defined as follows:

1. *Assert*: If φ is a positional atom, an atom with named arguments, a frame, a membership atomic formula, or a subclass atomic formula in the RIF-PRD condition language, then `Assert(φ)` is an atomic action. φ is called the *target* of the action.
2. *Retract*: If φ is a positional atom, an atom with named arguments, or a frame in the RIF-PRD condition language, then `Retract(φ)` is an atomic action. φ is called the *target* of the action.
3. *Retract object*: If `t` is a term in the RIF-PRD condition language, then `Retract(t)` is an atomic action. `t` is called the *target* of the action. ☐

Editor's Note: Whether and under what restrictions, if any, membership and subclass atomic formulas are allowable targets for atomic assert actions is still under discussion in the working group. We welcome feedback on that issue.

Definition (Ground atomic action). An atomic action with target `t` is a ***ground atomic action*** if and only if *Var*(`t`) = Ø. ☐

3.1.2 Action blocks

The *action block* is the top level construct to represent the conclusions of the production rules that can be serialized using RIF-PRD. An action block contains a non-empty sequence of atomic actions. It may also include *action variable declarations*.

The *action variable declaration* construct is used to declare variables that are local to the action block, called *action variables*, and to assign them a value within the action block.

Editor's Note: This version of RIF-PRD supports only a limited mechanism to initialize local action variables. Action variables may be bound to newly created frame objects or to slot values of frames. Future versions may support different or more elaborate mechanisms.

Definition (Action variable declaration). An ***action variable declaration*** is a pair, *(v p)* made of an *action variable*, *v*, and an *action variable binding* (or, simply, *binding*), *p*, where *p* has one of two forms:

1. *frame object declaration*: if the action variable, *v*, is to be assigned the identifier of a new frame, then the action variable binding is a *frame object declaration*: New(*v*). In that case, the notation for the action variable declaration is: (?o New(?o));
2. *frame slot value*: if the action variable, *v*, is to be assigned the value of a slot of a ground frame, then the action variable binding is a frame: *p* = o[s->*v*], where o is a term that represents the identifier of the ground frame and s is a term that represents the name of the slot. The associaed notation is: (?value o[s->?value]). □

Definition (Action block). If $(v_1\ p_1)$, ..., $(v_n\ p_n)$, $n \geq 0$, are action variable declarations, and if a_1, ..., a_m, $m \geq 1$, are atomic actions, then Do($(v_1\ p_1)$, ..., $(v_n\ p_n)$ a_1 ... a_l) denotes an **action block**. □

3.1.3 Well-formed action blocks

The specification fo RIF-PRD does not assign a standard meaning to all the action blocks that can be standardized using its concrete XML syntax. Action blocks that can be meaningfully serialized are called *well-formed*. The notion of well-formedness, already defined for condition formulas, is extended to atomic actions, action variable declarations and action blocks.

The main restrictions are that one and only one action variable bindings can assign a value to each action variable binding, and that the assertion of a membership atomic formula is meaningful only if for a new frame object.

Definition (Well-formed atomic action). An atomic action is ***well-formed*** if and only if one of the following is true:

- it is an Assert and its target is a well-formed atom (positional or with named arguments), or a well-formed frame, membership or subclass atomic formula;
- it is a Retract and its target is a well-formed term or a well-formed atom (positional or with named arguments), or a well-formed frame atomic formula. □

Definition (Well-formed action variable declaration). An action variable declaration (?v *p*) is ***well-formed*** if and only if one of the following is true:

- the action variable binding, *p*, is the declaration of a new frame object: *p* = New(?v), and its argument is the action variable that is declared in the same action variable declaration, ?v;
- the action variable binding, *p*, is a well formed frame atomic formula, *p* = o[a_1->t_1...a_n->t_n], $n \geq 1$, and the action variable, v occurs in the position of a slot value, and nowhere else, that is: $v \notin Var(o) \cup Var(a_1) \cup \ldots \cup Var(a_n)$ and $v \in \{t_1\ \ldots\ t_n\}$. □

For the definition of a well-formed action block, the function *Var(f)*, that has been defined for condition formulas, is extended to atomic actions and frame object declarations as follows:

- if *f* is an atomic action with target *t*, then *Var(f)* = *Var(t)*;
- if *f* is a frame object declaration, `New(?v)`, then *Var(f)* = *{?v}*.

Definition (Well-formed action block). An action block is ***well-formed*** if and only if all of the following is true:

- all the action variable declarations, if any, are well-formed;
- each action variable, if any, is assigned a value by one and only one action binding, that is: if b_1 = $(v_1\ p_1)$ and b_2 = $(v_2\ p_2)$ are two action variable declarations in the action block, then $v_2 \notin Var(p_1)$ if $v_1 \in Var(p_2)$, and, reciprocally, $v_1 \notin Var(p_2)$ if $v_2 \in Var(p_1)$;
- all the actions in the action block are well-formed atomic actions;
- if an atomic action in the action block, `a`, asserts a membership atomic formula, $a = \mathrm{Assert}(t_1\ \#\ t_2)$, then the object term in the membership atomic formula, t_1, is an action variable that is declared in the action block and the action variable binding is the declaration of a new frame object. □

Definition (RIF-PRD action language). The ***RIF-PRD action language*** consists of the set of all the well-formed action blocks. □

3.2 Operational semantics of atomic actions

This section specifies the intended semantics of the atomic actions in a RIF-PRD document.

The effect intended of the ground atomic actions in the RIF-PRD action language is to modify the state of the fact base, in such a way that it changes the set of conditions that are satisfied before and after each atomic action is performed.

As a consequence, the intended semantics of the ground atomic actions in the RIF-PRD action language determines a relation, called the *RIF-PRD transition relation*: $\rightarrow_{\text{RIF-PRD}} \subseteq W \times L \times W$, where *W* denotes the set of all the states of the fact base, and where *L* denotes the set of all the ground atomic actions in the RIF-PRD action language.

Since the satisfaction of condition formulas is defined with respect to the Herbrand interpretation of ground formulas (Section Satisfaction of a condition), we will assume in the following that the states of the fact base are represented by such sets, for the purpose of specifying the intended operational semantics of atomic actions, or rules and of rule sets serialized using RIF-PRD.

Definition (RIF-PRD transition relation). The intended semantics of RIF-PRD atomic actions is completely specified by the **transition relation** $\rightarrow_{\textbf{RIF-PRD}} \subseteq W \times L$

× *W*. *(w, α, w')* ∈ $\rightarrow_{RIF\text{-}PRD}$ if and only if *w* ∈ *W*, *w'* ∈ *W*, α is a ground atomic action, and one of the following is true:

1. α is `Assert(φ)`, where φ is a ground atomic formula, and *w'* = *w* + *φ*;
2. α is `Retract(φ)`, where φ is a ground atomic formula, and *w'* = *w* - *φ*;
3. α is `Retract(o)`, where `o` is a constant, and *w'* = *w* - *{*`o[s->v]` | *for all the values of terms* `s` *and* `v`*}* - *{*`o#c` | *for all the values of term* `c`*}*. □

Rule 1 says that all the condition formulas that were satisfied before an assertion will be satisfied after, and that, in addition, the condition formulas that are satisfied by the asserted ground formula will be satisfied after the assertion.

Rule 2 says that all the condition formulas that were satisfied before a retraction will be satisfied after, except if they are satisfied only by the retracted fact.

Rule 3 says that all the condition formulas that were satisfied before the removal of a frame object will be satisfied after, except if they are satisfied only by one of the frame or membership formulas about the removed object or a conjunction of such formulas.

4 Production rules and rulesets

This section specifies the rules and rulesets that can be serialized using RIF-PRD, by specifying:

- the abstract syntax that all production rule languages interchanging rules using RIF-PRD must have in common for rules and rule sets;
- and the intended semantics of the rules and ruleset in a RIF-PRD document.

4.1 Abstract syntax

For a production rule language to be able to interchange rules using RIF-PRD, in addition to the RIF-PRD condition and action languages, its alphabet must, at the abstract syntax level, contain syntactic constructs:

- to associate a condition and an action block into a rule;
- to declare the variables that are free in a rule, to specify their bindings, and to associate them with that rule into a rule with less free variables;
- to group rules and to associate specific operational semantics to groups of rules.

4.1.1 Rules

Definition (Rule). A ***rule*** can be either:

- an *unconditional action block*;

- a *conditional action block*: if condition is a formula in the RIF-PRD condition language, and if action is a well-formed action block, then If condition, Then action is a conditional action;
- a *rule with bound variables*: if $?v_1 \; ... \; ?v_n$, *n > 0*, are variables; $p_1 \; ... \; p_m$, *m ≥ 0*, are condition formulas (called *binding patterns*), and rule is a rule, then Forall $?v_1 ... ?v_n \; (p_1 ... p_m)$ (rule) is a rule. □

RIF-BLD compatibility. A rule If condition, Then action can be equivalently written action :- condition, that is, using RIF-BLD notation. Indeed, the normative XML syntax is the same for a conditional assertion in RIF-BLD and for a conditional action in RIF-PRD. The use of RIF-BLD notation is especially useful if the condition formula, condition, contains no negation, and if the action block, action, contains only Assert atomic actions. The use of the same notation emphasizes that such a rule has the same semantics in RIF-PRD and RIF-BLD.

Editor's Note: At this stage, the above assertion regarding the equivalence of a specific fragment of RIF-PRD and RIF-BLD is mostly the statement of an objective of the working group. That issue will be addressed more completely in a future version of this document.

To emphasize that equivalence even further, an action block can be written as simply And($\varphi_1 \; ... \; \varphi_n$), if it contains only atomic assert actions: Do(Assert(φ_1) ... Assert(φ_n)), *n ≥ 1*. If the action block consists of a single atomic assert action: Do(Assert(φ)), then it can be written as simply φ.

Notice that the notation for a rule with bound variables uses the keyword Forall for the same reasons, that is, to emphasize the overlap with RIF-BLD. Indeed, Forall does not indicate the universal quantification of the declared variables, in RIF-PRD, but merely that the execution of the rule must be considered for all their bindings as constrained by the binding patterns. However, when no negation is used in the conditions and only assertions in the actions, the XML serialization of a RIF-PRD rule with bound variables is exactly the same as the XML serialization of a RIF-BLD universally quantified rule, and their semantics coincide.

4.1.2 Groups

As was already mentioned in Section Overview, production rules have an operational semantics that can be described in terms of matching rules against states of the fact base, selecting rule instances to be executed, and executing rule instances' actions to transition to new states of the fact base.

When production rules are interchanged, the intended rule instance selection strategy, often called the *conflict resolution strategy*, need be interchanged along with the rules : in RIF-PRD, the *group* is the construct that is used to group sets of rules and to associate them with a conflict resolution strategy. Many production rule systems use priorities associated with rules as part of their conflict resolution strategy: in RIF-PRD, the group is also used to carry the priority information that may be associated to the interchanged rules.

Definition (Group). If `strategy` is an IRI that denotes a conflict resolution strategy, if `priority` is an integer, and if each rg_j, $0 \le j \le n$, is either a rule or a group, then any of the following is a ***group***:

- `Group` rg_0 ... rg_n, $n \ge 0$;
- `Group strategy` rg_0 ... rg_n, $n \ge 0$;
- `Group priority` rg_0 ... rg_n, $n \ge 0$;
- `Group strategy priority` rg_0 ... rg_n, $n \ge 0$. □

4.1.3 Well-formed rules and groups

The function *Var(f)*, that has been defined for condition formulas and extended to actions, is further extended to rules, as follows:

- if *f* is an action block that declares action variables $?v_1$ ... $?v_n$, $n \ge 0$, and that contains actions a_1 ... a_m, $m \ge 1$, then $Var(f) = \bigcup_{1 \le i \le m} Var(a_i) - \{?v_1 \; ... \; ?v_n\}$;
- if *f* is an conditional action where `c` is the condition formula and `a` is the action, then $Var(f) = Var(c) \cup Var(a)$;
- if *f* is a quantified rule where $?v_1$ ... $?v_n$, $n > 0$, are the declared variables; p_1 ... p_m, $m \ge 0$, are the binding patterns, and `r` is the rule, then $Var(f) = (Var(r) \cup Var(p_1) \cup ... \cup Var(p_m)) - \{?v_1 \; ... \; ?v_n\}$.

Definition (Well-formed rule). A rule, `r`, is a ***well-formed rule*** if and only if it contains no free variable, that is, $Var(r) = \emptyset$, and either:

- it is an unconditional well-formed action block, `a`;
- or it is a conditional action where the condition formula, `c`, is a well-formed condition formula, and the action block, `a`, as a well-formed action block, and no atomic action in `a` has a subclass atomic formula as its target;
- or it is a quantified rule, `Forall V (P) (r)`, and the quantified rule, `r` is a well-formed rule, and each of the declared variables in $V = \{?v_i\}_{0 \le i \le n}$ is free in some of the binding patterns in $P = \{p_j\}_{0 \le j \le m}$ or in the quantified rule, `r`; that is, $V \subseteq Var(r) \cup Var(p_1) \cup ... \cup Var(p_m)$, $m \ge 0$. □

Definition (Well-formed group). A ***well-formed group*** is either a group that contains only well-formed rules and well-formed groups, or a group that contains no rule or group (an empty group). □

The set of the well-formed groups contains all the production rulesets that can be meaningfully interchanged using RIF-PRD.

4.2 Operational semantics of rules and rule sets

4.2.1 Motivation and example

As already mentioned in Section Overview, the description of a production rule system as a transition system can be used to specify the intended semantics that is associated with production rules and rulesets interchanged using RIF-PRD.

The intuition of describing a production rule system as a transition system is that, given a set of production rules *RS* and a fact base w_0, the rules in *RS* that are satisfied, in some sense, in w_0 determine an action a_1, whose execution results in a new fact base w_1; the rules in *RS* that are satisfied in w_1 determine an action a_2 to execute in w_1, and so on, until the system reaches a final state and stops. The result is the fact base w_n when the system stops.

Example 3.1. Judicael, a chicken and potato farmer, uses a rule based system to decide on the daily grain allowance for each of her chicken. Currently, Judicael's rule base contains one single rule, the *chicken and mashed potatoes* rule:

```
(* ex:ChickenAndMashedPotatoes *)
Forall ?chicken ?potato ?weight
      (And(?chicken#ex:Chicken
           (Exists ?age
                   And(?chicken[ex:age->?age]
                       External(pred:numeric-greater-than(?age, 8)))))
       And(?potato#ex:Potato
           ex:owns(?chicken ?potato)
           (Exists ?weight
                   And(?potato[ex:weight->?weight]
                       External(pred:numeric-greater-than(?weight External(
  If And(External(pred:string-not-equals(External(ex:today()), "Tuesday"))
         Not(External(ex:foxAlarm())))
  Then Do( (?allowance ?chicken[ex:allowance->?allowance])
           Execute(ex:mash(?potato))
           Retract(?potato)
           Retract(ex:owns(?chicken ?potato))
           Retract(?chicken[ex:allowance->?allowance])
           Assert(?chicken[ex:allowance->External(func:numeric-multiply(?al
```

Judicael has four chickens, Jim, Jack, Joe and Julia, that own three potatoes (BigPotato, SmallPotato, UglyPotato) among them:

- Jim (daily grain allowance = 10) is 12 months old, Jack (daily grain allowance = 12) is 9 months old, Joe (daily grain allowance = 6) is 6 months old and Julia (daily grain allowance = 14) is 10 months old;
- BigPotato weights 70g, SmallPotato weights 10g, UglyPotato weights 50g;

- Jim owns BigPotato, Jack and Woof own SmallPotato jointly (Woof is the farm's dog. It inherited joint ownership of SmallPotato from its aunt Georgette) and Joe owns UglyPotato.

That is the initial set of facts w_0.

When the rule is applied to w_0:

- the first pattern selects *{Jim/?chicken, Jack/?chicken, Julia/?chicken}* as possible values for variable *?chicken* (Joe is too young);
- the second pattern selects *{(Jim/?chicken, BigPotato/?potato)}* as the only possible substitution for the variables *?chicken* and *?potato* (UglyPotato does not belong to either Joe, Jack or Julia and SmallPotato is too small);

Suppose that Judicael's implementation of *today()* returns *Monday* and that the *foxAlarm()* is *false* when the *Chicken and mashed potatoes* rule is applied: the condition is satisfied, and the actions in the conclusion are executed with *BigPotato* substituted for *?potato*, *Jim* substituted for *?chicken*, and *10* substituted for *?allowance*. This results in the following changes in the set of facts:

- *BigPotato* is mashed (and removed from the list of potatoes to be considered in future applications of the *Chicken and mashed potatoes* rule);
- the daily grain allowance of *Jim* is now 11;
- *Jim* does not own a potato anymore.

The resulting set of facts w_1 is thus:

- Jim (daily grain allowance = 11) is 12 months old, Jack (daily grain allowance = 12) is 9 months old, Joe (daily grain allowance = 6) is 6 months old and Julia (daily grain allowance = 14) is 10 months old;
- SmallPotato weights 10g, UglyPotato weights 50g;
- Jack and Woof own SmallPotato jointly and Joe owns UglyPotato.

When the *Chicken and mashed potatoes rule* in applied to w_1, the first pattern still selects *{Jim/?chicken, Jack/?chicken, Julia/?chicken}* as possible values for variable *?chicken*, but the second pattern does not select any possible substitution for the couple *(?chicken, ?potato)* anymore: the rule cannot be satisfied, and the system, having detected a final state, stops.

The result of the execution of the system is w_1. □

4.2.2 Definitions and notational conventions

More precisely, a production rule system is defined as a labeled terminal transition system (e.g. PLO04), for the purpose of specifying the intended semantics of a RIF-PRD rule or group of rules.

Definition (labeled terminal transition system): A labeled terminal transition system is a structure {**C**, **L**, →, **T**}, where

- **C** is a set of elements, *c*, called configurations, or states;
- **L** is a set of elements, *a*, called labels, or actions;
- $\rightarrow \subseteq C \times L \times C$ is the transition relation, that is: $(c, a, c') \in \rightarrow$ iff there is a transition labeled *a* from the state *c* to the state *c'* or, more appropriately in the case of a production rule system, the execution of action *a* in the state *c* causes a transition to state *c'* ;
- **T** $\subseteq$ C is the set of final states, that is, the set of all the states *c* from which there are no transitions: $T = \{c \in C \mid \forall\, a \in L, \forall\, c' \in C, (c, a, c') \notin \rightarrow\}$. □

For many purposes, a representation of the states of the fact base is an appropriate representation of the states a production rule system seen as a transition system. However, the most widely used conflict resolution strategies require information about the history of the system, in particular with respect to the rule instances that have been selected for execution in previous states. Therefore, each state of the transition system used to represent a production rule system must keep a memory of the previous states and the rule instances that where selected and triggered the transition in those states.

To avoid the confusion between the states of the fact base and the states of the transition system, the latter will be called *production rule system states*.

Definition (Production rule system state). A ***production rule system state*** (or, simply, a ***system state***), *s*, is characterized by

- a state of the fact base, *facts(s)*;
- if *s* is not the initial state: a previous system state, *previous(s)*, such that, given two system states s_1 and s_2, $s_1 = previous(s_2)$ if and only if the production rule system the sequential execution of the action parts of the rule instances in *picked*(s_1) transitioned the system from system state s_1 to system state s_2;
- if *s* is not the current state: the ordered set of rule instances, *picked(s)*, that the conflict resolution strategy picked, among the all the rule instances that matched *facts(s)*. □

In the following, we will write *previous(s) = NIL* to denote that a system state *s* is the initial state.

Here, a *rule instance* is defined as the result of the substitution of constants for all the rule variables in a rule.

Let *R* denote the set of all the rules in the rule language under consideration.

Definition (Rule instance). Given a rule, $r \in R$ and a ground substitution, σ, such that $Var(r) \subseteq Dom(\sigma)$, where *Var(r)* denotes the set of the rule variables in *r*, the result, $ri = \sigma(r)$, of the substitution of the constant *σ(`?x`)* for each variable `?x` $\in Var(r)$ is a ***rule instance*** (or, simply, an ***instance***) of *r*. □

Given a rule instance *ri*, let *rule(ri)* identify the rule from which *ri* is derived by substitution of constants for the rule variables, and let *substitution(ri)* denote the substitution by which *ri* is derived from *rule(ri)*.

In the following, two rule instances ri_1 and ri_2 of a same rule *r* will be considered *different* if and only if *substitution*(ri_1) and *substitution*(ri_2) substitute a different constant for at least one of the rule variables in *Var(r)*.

In the definition of a production rule system state, a rule instance, *ri*, is said to match a state of a fact base, *w*, if its defining substitution, *substitution(ri)*, matches the RIF-PRD condition formula that represents the condition of the instantiated rule, *rule(ri)*, to the ground formula that represents the state of facts *w*.

Let *W* denote the set of all the possible states of a fact base.

Definition (Matching rule instance). Given a rule, *ri*, and a state of the fact base, *w* ∈ *W*, *ri* is said to ***match*** *w* if and only if one of the following is true:

- *rule(ri)* is an unconditional action block;
- *rule(ri)* is a conditional action block: `If condition, Then action`, and *substitution(ri)* matches the condition formula `condition` to the ground condition formula that represents *w*;
- *rule(ri)* is a rule with bound variables: Forall $?v_1...?v_n$ $(p_1...p_n)$ (r'), $n \geq 0$, $m \geq 0$, and *substitution(ri)* matches each of the condition formulas p_i, $0 \leq i \leq m$, to the ground condition formula that represents *w*, and the rule instance *ri'* matches *w*, where *ri'* is the instance of rule *r'* such that *substitution(ri')* = *substitution(ri)*. □

Definition (Conflict set). Given a rule set, *RS* ⊆ *R*, and a system, *s*, the set, *conflictSet(RS, s)* of all the different instances of the rules in *RS* that match the state of the fact base, *facts(s)* ∈ *W* is called the ***conflict set*** determined by *RS* in *s*. □

In each non-final state, *s*, of a production rule system, a subset, *picked(s)*, of the rule instances in the conflict set is selected and ordered; their action parts are instantiated, and they are executed. This is sometimes called: *firing* the selected instances.

Definition (Action instance). Given a system state, *s*; given a rule instance, *ri*, of a rule in a rule set, *RS*; and given the action block in the action part of the rule *rule(ri)*: Do$((v_1\ p_1)...(v_n\ p_n)\ a_1...a_m)$, $n \geq 0$, $m \geq 1$, where the $(v_1\ p_1)$, $0 \leq i \leq n$, represent the action variable declarations and the a_j, $1 \leq j \leq m$, represent the sequence of atomic actions in the action block; if *ri* is a matching instance in the conflict set determined by *RS* in system state *s*: *ri* ∈ *conflictSet(RS, s)*, the substitution σ = *substitution(ri)* is extended to the action variables $v_1...v_n$, $n \geq 0$, in the following way:

- if v_i is assigned the identifier of a new frame by the action variable declaration: (v_i New(v_i)), then $\sigma(v_i) = c_{new}$, where c_{new} *is a constant of* *type* `rif:IRI` *that does not occur in any subformula of the ground formula that represents the state of the fact base that is associated to* s, facts(s)*;*

- if v_i is assigned the value of a frame's slot by the action variable declaration: $(v_i\ o[s\text{->}v_i])$, then *σ(v_i)* is a constant such that the frame formula $o[s\text{->}v_i])$ matches the state of the fact base *facts(s)* with subtitution σ.

The sequence of ground atomic actions that is the result of substituting a constant for each variable in the atomic actions of the action block of the rule instance, *ri*, according to the extended substitution, is the ***action instance*** associated to *ri*. □

Let *actions(ri)* denote the action instance that is associated to a rule instance *ri*. By extension, given an ordered set of rule instances, *ori*, *actions(ori)* denotes the sequence of ground atomic actions that is the concatenation, preserving the order in *ori*, of the action instances associated to the rule instances in *ori*.

4.2.3 Operational semantics of a production rule system

All the elements that are required to define a production rule system as a labeled terminal transition system have now been defined.

Definition (RIF-PRD Production Rule System). A ***RIF-PRD production rule system*** is defined as a labeled terminal transition system $PRS = \{S, A, \rightarrow_{PRS}, T\}$, where :

- S is a set of system states;
- A is a set of transition labels, where each transition label is a sequence of ground RIF-PRD atomic actions;
- The transition relation $\rightarrow_{PRS} \subseteq S \times A \times S$, is defined as follows: $\forall\ (s, a, s') \in S \times A \times S$, $(s, a, s') \in \rightarrow_{PRS}$ if and only if all of the following hold:
    1. *(facts(s), a, facts(s'))* $\in \rightarrow^{*}_{\text{RIF-PRD}}$, where $\rightarrow^{*}_{\text{RIF-PRD}}$ denotes the transitive closure of the transition relation $\rightarrow_{\text{RIF-PRD}}$ that is determined by the specification of the semantics of the atomic actions supported by RIF-PRD;
    2. *a = actions(picked(s))*;
- $T \subseteq S$, a set of final system states. □

Intuitively, the first condition in the definition of the transition relation $\rightarrow_{\text{PRS}}$ states that a production rule system can transition from one system state to another only if the state of facts in the latter system state can be reached from the state of facts in the former by performing a sequence of ground atomic actions supported by RIF-PRD, according to the semantics of the atomic actions.

The second condition states that the allowed paths out of any given system state are determined only by how rules instances are *picked* from the conflict set for execution by the conflict resolution strategy.

Given a ruleset $RS \subseteq R$, the associated conflict resolution strategy *LS*, and an initial state of the fact base, $w \in W$, the input function to a RIF-PRD production rule system is defined as:

$Eval(RS, LS, w) \rightarrow_{PRS} s \in S$, such that $facts(s) = w$ and $previous(s) = NIL$.

Given a set *T* of final system states, the output function is defined as:

$$\forall\, s' \in T,\ s' \rightarrow_{PRS} w' = facts(s')$$

Or, using $\rightarrow^{*}_{PRS}$ to denote the transitive closure of the transition relation $\rightarrow_{PRS}$:

$$\forall\, w \in W,\ \exists\, s' \in T,\ \exists\, w' \in W,\ w' = facts(s') \text{ and } Eval(RS, LS, w) \rightarrow^{*}_{PRS} w'$$

Therefore, the exact behavior of a RIF-PRD production rule system depends on:

1. the conflict resolution strategy, that is, how rule instances are, precisely, selected for execution from the rule instances that match a given state of the fact base;
2. and how the set *T* of final system states is, precisely, defined.

4.2.4 Conflict resolution

The process of selecting one or more rule instances from the conflict set for firing is often called: *conflict resolution*.

In RIF-PRD the conflict resolution algorithm (or conflict resolution *strategy*) that is intended for a set of rules is denoted by a keyword or a set of keywords that is attached to the rule set. In this version of the RIF-PRD specification, a single conflict resolution strategy is specified normatively: it is denoted by the keyword `rif:forwardChaining` (a constant of type *rif:IRI*), for it accounts for a common conflict resolution strategy used in most forward-chaining production rule systems.

Future versions of the RIF-PRD specification may specify normatively the intended conflict resolution strategies to be attached to additional keywords. In addition, RIF-PRD documents may include non-standard keywords: it is the responsability of the producers and consumers of such document to agree on the intended conflict resolution strategies that are denoted by such non-standard keywords.

Conflict resolution strategy: `rif:forwardChaining`

Most existing production rule systems implement conflict resolution algorithms that are a combination of the following elements (under these or other, idiosyncratic names; and possibly combined with additional, idiosyncratic rules):

- *Refraction.* The essential idea of *refraction* is that a given instance of a rule must not be fired more than once as long as the reasons that made it eligible for firing hold. In other terms, if an instance has been fired in a given state of the system, it is no longer eligible for firing as long as it satisfies the states of facts associated to all the subsequent system states;
- *Priority.* The rule instances are ordered by priority of the instantiated rules, and only the rule instances with the highest priority are eligible for firing;

- *Recency.* the rule instances are ordered by how long a rule instance has been continuously satisfied in the states of facts associated to previous system states, and only the most recent ones are eligible for firing.

The RIF-PRD keyword `rif:forwardChaining` denotes the common conflict resolution strategy that can be summarized as follows: given a conflict set

1. Refraction is applied to the conflict set, that is, all the refracted rule instances are removed from the conflict set;
2. The remaining rule instances are ordered by decreasing priority, and only the rule instances with the highest priority are kept in the conflict set;
3. The remaining rule instances are ordered by decreasing recency, and only the most recent rule instances are kept in the conflict set;
4. Any remaining tie is broken arbitrarily, and a single rule instance is kept for firing.

As specified earlier, *picked(s)* denotes the ordered list of the rule instances that were picked in a system state, *s*. Under the conflict resolution strategy denoted by `rif:forwardChaining`, the list denoted by *picked(s)* contains a single rule instance, for any given system state, *s*.

Given a system state, *s*, a rule set, *RS*, and a rule instance, *ri* ∈ *conflictSet(RS, s)*, let *recency(ri, s)* denote the number of system states before *s*, in which *ri* has been continuously a matching instance: if *s* is the current system state, *recency(ri, s)* provides a measure of the recency of the rule instance *ri*. *recency(ri, s)* is specified recursively as follows:

- if *previous(s) = NIL*, then *recency(ri, s) = 1*;
- else if *ri* ∈ *conflictSet(RS, previous(s))*, then *recency(ri, s) = 1 + recency(ri, previous(s))*;
- else, *recency(ri, s) = 1*.

In the same way, given an rule instance, *ri*, and a system state, *s*, let *lastPicked(ri, s)* denote the number of system states before *s*, since *ri* has been last fired. *lastPicked(ri, s)* is specified recursively as follows:

- if *previous(s) = NIL*, then *lastPicked(ri, s) = 1*;
- else if *ri* ∈ *picked(previous(s))*, then *lastPicked(ri, s) = 1*;
- else, *lastPicked(ri, s) = 1 + lastPicked(ri, previous(s))*.

Finally, given a rule instance, *ri*, let *priority(ri)* denote the priority that is associated to *rule(ri)*, or zero, if no priority is associated to *rule(ri)*. If *rule(ri)* is inside nested `Group`s, *priority(ri)* denotes the priority that is associated with the innermost `Group` to which a priority is explicitely associated, or zero.

Given a conflict set, *cs*, the conflict resolution strategy `rif:forwardChaining` can now be described with the help of four rules, where *ri* and *ri'* are rule instances:

1. **Refraction rule**: if *ri* ∈ *cs* and *lastPicked(ri, s)* ≤ *recency(ri, s)*, then *cs = cs - ri*;

2. **Priority rule**: if *ri* ∈ *cs* and *ri'* ∈ *cs* and *priority(ri)* < *priority(ri')*, then *cs* = *cs* - *ri*;
3. **Recency rule**: if *ri* ∈ *cs* and *ri'* ∈ *cs* and *recency(ri, s)* > *recency(ri', s)*, then *cs* = *cs* - *ri*;
4. **Tie-break rule**: if *ri* ∈ *cs*, then *cs* = *{ri}*.

The *refraction rule* removes the instances that have been in the conflict set in all the system states at least since they were last fired, that is, it removes the refracted instances from the current conflict set; the *priority rule* removes the instances such that there is at least one instance with a higher priority; the *recency rule* removes the instances such that there is at least one instance that is more recent; and the *tie-break rule* keeps one rule from the set, arbitrarily.

To select the singleton rule instance, *picked(s)*, to be fired in a system state, *s*, given a rule set, *RS*, the conflict resolution strategy denoted by the keyword `rif:forwardChaining` consists in the following sequence of steps:

1. start with the conflict set, *cs*, that a rule set *RS* determines in a system state *s*: *cs* = *conflictSet(RS, s)*;
2. apply the *refraction rule* to all the rule instances in *cs*;
3. then apply the *priority rule* to all the remaining instances in *cs*;
4. then apply the *recency rule* to all the remaining instances in *cs*;
5. then apply the *tie-break rule*.

4.2.5 Halting test

Editor's Note: This section is still under discussion (see ISSUE-65). This version specifies a single, default halting test: future version of this draft may specify additional halting tests, and/or a different default. The Working Group seeks feedback on which halting tests and which combinations of tests should be supported by RIF-PRD and/or required from RIF-PRD implementations; and which halting test should be the default, if any.

By default, a system state is final, given a rule set, *RS*, and a conflict resolution strategy, *LS*, if there is no rule instance available for firing after application of the conflict resolution strategy.

For the conflict resolution strategy identified by the RIF-PRD keyword `rif:forwardChaining`, a system state, *s*, is ***final*** given a rule set, *RS* if and only if the remaining conflict set is empty after application of the *refraction rule* to all the rule instances in *conflictSet(RS, s)*. In particular, all the system states, *s*, such that *conflictSet(RS, s)* = ∅ are final.

5 XML Syntax

This section specifies a common concrete XML syntax to serialize any production rule set written in a language that share the abstract syntax speicifed in section 4.1, provided that its intended semantics agrees with the semantics that is described in section 4.2.

In the following, after the notational conventions are introduced, we specify the RIF-PRD XML constructs that carry a normative semantics with respect to the intended interpretation of the interchanged rules. They are specified with respect to the abstract syntax, and their specification is structured according to the specification of the abstract syntax in sections 2.1, 3.1 and 4.1.

The root element of any RIF XML document, `Document` and other XML constructs that do not carry a normative semantics with respect to the intended interpretation of the interchanged rules are specified in the last sub-section.

5.1 Notational conventions

5.1.1 Namespaces

Throughout this document, the `xsd:` prefix stands for the XML Schema namespace URI `http://www.w3.org/2001/XMLSchema#`, the `rdf:` prefix stands for `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, and `rif:` stands for the URI of the RIF namespace, `http://www.w3.org/2007/rif#`.

Syntax such as `xsd:string` should be understood as a compact URI (CURIE) -- a macro that expands to a concatenation of the character sequence denoted by the prefix `xsd` and the string `string`. The compact URI notation is used for brevity only, and `xsd:string` should be understood, in this document, as an abbreviation for `http://www.w3.org/2001/XMLSchema#string`.

5.1.2 BNF pseudo-schemas

The XML syntax of RIF-PRD is specified for each component as a pseudo-schema, as part of the description of the component. The pseudo-schemas use BNF-style conventions for attributes and elements: "?" denotes optionality (i.e. zero or one occurrences), "*" denotes zero or more occurrences, "+" one or more occurrences, "`[`" and "`]`" are used to form groups, and "|" represents choice. Attributes are conventionally assigned a value which corresponds to their type, as defined in the normative schema. Elements are conventionally assigned a value which is the name of the syntactic class of their content, as defined in the normative schema.

```
<!-- sample pseudo-schema -->
    <defined_element
          required_attribute_of_type_string="xs:string"
          optional_attribute_of_type_int="xs:int"? >
      <required_element />
      <optional_element />?
      <one_or_more_of_these_elements />+
      [ <choice_1 /> | <choice_2 /> ]*
   </defined_element>
```

5.1.3 Syntactic components

Three kinds of syntactic components are used to specify RIF-PRD:

- ***Abstract classes*** are defined only by their subclasses: they not visible in the XML markup and can be thought of as extension points. In this document, abstract constructs will be denoted with all-uppercase names;
- ***Concrete classes*** have a concrete definition and they are associated with specific XML markup. In this document, concrete constructs will be denoted with CamelCase names with leading capital letter;
- ***Properties***, or *roles*, define how two classes relate to each other. They have concrete definitions and are associated with specific XML markup. In this document, properties will be denoted with camelCase names with leading smallcase letter.

5.2 Conditions

This section specifies the XML constructs that are used in RIF-PRD to serialize condition formulas.

5.2.1 TERM

The `TERM` class of constructs is used to serialize terms, be they simple terms, that is, constants and variables; or positional terms or terms with named arguments, both being, per the definition of a well-formed formula, representations of externally defined functions.

As an abstract class, `TERM` is not associated with specific XML markup in RIF-PRD instance documents.

```
   [ Const | Var | External ]
```

5.2.1.1 Const

In RIF, the `Const` element is used to serialize a constant.

The `Const` element has a required `type` attribute and an optional `xml:lang` attribute:

- The value of the `type` attribute is the identifier of the `Const` symbol space. It must belong to the type `xsd:anyURI`. In the RIF data types and builtins document, the section about [DTB#Constants.2C_Symbol_Spaces.2C_and_Datatypes|Constants, Symbol spaces and Datatypes]] lists the builtin symbol spaces and data types that all implementations of RIF-PRD must support. Rule sets that are exchanged through RIF-PRD can use additional, user-defined symbol spaces;

- The `xml:lang` attribute, as defined by 2.12 Language Identification of XML 1.0 or its successor specifications in the W3C recommendation track, is optionally used to identify the language for the presentation of the `Const` to the user. It is allowed only in association with constants of the type `rif:text`. A compliant implementation MUST ignore the `xml:lang` attribute if the type of the `Const` is not `rif:text`.

The content of the `Const` element is the constant's litteral, which can be any Unicode character string.

```
<Const type=xsd:anyURI [xml:lang=xsd:language]? >
    Any Unicode string
</Const>
```

\{\{EdNote|text=The case of non-standard data types, that is, of constants that do not belong or cannot be cast in one of RIF builtin types for interchange purposes, is still under discussion in the WG. The WG seeks feedback on whether they should be allowed and why.\}\}

Example 2.1. In each of the examples below, a constant is first described, followed by its serialization in RIF-PRD XML syntax.

a. A constant with builtin type *xsd:integer* and value *123*:

```
<Const type="xsd:integer">123</Const>
```

b. A constant which symbol *today* is defined in Joe the Hen Public's namespace *http://rif.example.com/2008/joe#*. The type of the constant is `rif:iri`:

```
<Const type="rif:iri">
   http://rif.example.com/2008/joe#today
</Const>
```

c. A constant with symbol *BigPotato* that is local to the set of rules where it appears (e.g. a RuleSet specific to Paula's farm). The type of the constant is `rif:local`:

```
<Const type="rif:local">BigPotato</Const>
```

d. A constant with non-builtin type xsd:int and value 123:

```
<Const type="xsd:int">123</Const>
```

5.2.1.2 Var

In RIF, the `Var` element is used to serialize a variable.

The content of the `Var` element is the variable's name, serialized as an Unicode character string.

```
        <Var> any Unicode string </Var>
```

Example 2.2. The example below shows the XML serialization of a reference to a variable named: `?chicken`.

```
    <Var> chicken <Var>
```

5.2.1.3 External

As a `TERM`, the `External` element is used to serialize a positional term or a term with named arguments. In RIF-PRD, a positional or a named-argument term represents always a call to an externally specified function, e.g. a builtin, a user-defined function, a query to an external data source...

The `External` element contains one `content` element, which in turn contains one `Expr` element that contains one `op` element, followed zero or one `args` element or zero of more `slot` elements:

- The `content` and `Expr` element ensure compatibility with the RIF Basic Logic Dialect [RIF-BLD] that allows non-evaluated (that is, logic) functions to be serialized using an `Expr` element;
- The content of the `op` element must be a `Const`. When the `External` is a `TERM`, the content of the `op` element serializes a constant symbol of type `rif:iri` that must uniquely identify the evaluated function to be applied to the `args` `TERM`s. In the RIF data types and builtins document, the section List of RIF Builtin Predicates and Functions lists the builtin functions that all implementations of RIF-PRD must support. The content of the `op` element can also identify an user-defined function: it is the responsibility of the producers and consumers of RIF-PRD rulesets that reference non-builtin functions to agree on their semantics;
- The optional `args` element contains zero or more constructs from the `TERM` abstract class. The `args` element is used to serialize the arguments of a positional term. The order of the `args` sub-elements is, therefore, significant and MUST be preserved. This is emphasized by the required value `"yes"` of the required attribute `rif:ordered`;
- Each optional `slot` element contains one required `Name` sub-element, that contains an Unicode string that serializes the slot key, and a required `TERM` that serializes its value. The `slot` element is used to serialize an argument name-value pair in a term with named arguments. The order of the `slot` elements is, therefore, not significant;

```
<External>
   <content>
      <Expr>
         <op> Const </op>
         [ <args rif:ordered="yes"> TERM* </args>?
           |
           <slot rif:ordered="yes">
              <Name> Any Unicode string </Name>
              TERM
           <slot>* ]
      </Expr>
   </content>
</External>
```

Editor's Note: The slotted, or named arguments form of the `External TERM` construct is still under discussion (see also ISSUE-68). The working group seeks feedback on whether or not it should be included in PRD.

Example 2.3.

a. The first example below shows one way to serialize, in RIF-PRD, the sum of integer 1 and a variable `?x`, where the addition conforms to the specification of the builtin `fn:numeric-add`.

The prefix `fn` is associated with the namespace `http://www.w3.org/2007/rif-builtin-function#`.

```
<External>
   <content>
      <Expr>
         <op> <Const type="rif:iri"> fn:numeric-add </Const> </op>
         <args rif:ordered="yes">
            <Const type="xsd:integer"> 1 </Const>
            <Var> x </Var>
         </args>
      </Expr>
   </content>
</External>
```

b. Another example, that shows the RIF XML serialization of a call to the application-specific nullary function *today()*, which symbol is defined in the example's namespace *http://rif.example.com/2008/joe#*:

```
<External>
   <content>
      <Expr>
         <op>
            <Const type="rif:iri">
```

```
               http://rif.example.com/2008/joe#today
            </Const>
         </op>
      </Expr>
   </content>
</External>
```

5.2.2 ATOMIC

The ATOMIC class is used to serialize atomic formulas: positional and named-arguments atoms, equality, membership and subclass atomic formulas, frame atomic formulas and externally defined atomic formulas.

As an abstract class, ATOMIC is not associated with specific XML markup in RIF-PRD instance documents.

```
[ Atom | Equal | Member | Subclass | Frame | External ]
```

5.2.2.1 Atom

In RIF, the Atom element is used to serialize a positional atomic formula or an atomic formula with named arguments.

The Atom element contains one op element, followed by zero or one args element or zero or more slot arguments:

- The content of the op element must be a Const. It serializes the predicate symbol (the name of a relation);
- The optional args element contains zero or more constructs from the TERM abstract class. The args element is used to serialize the arguments of a positional atomic formula. The order of the arg's sub-elements is, therefore, significant and MUST be preserved. This emphasized by the required value "yes" of the required attribute rif:ordered;
- Each optional slot element contains one required Name sub-element, that contains an Unicode string that serializes the slot key, and a required TERM that serializes its value. The slot element is used to serialize an argument name-value pair in an atomic formula with named arguments. The order of the slot elements is, therefore, not significant;

```
<Atom>
   <op> Const </op>
   [ <args rif:ordered="yes"> TERM* </args>?
     |
     <slot rif:ordered="yes">
        <Name> Any Unicode string </Name>
        TERM
```

```
          <slot>* ]
       </Atom>
```

Editor's Note: The slotted, or named arguments form of the `Atom` construct is still under discussion (see also ISSUE-68). The working group seeks feedback on whether or not it should be included in PRD.

Example 2.4. The example below shows the RIF XML serialization of the positional atom *owns(?c ?p)*, where the predicate symbol *owns* is defined in the example namespace *http://rif.example.com/2008/joe#*.

```
<Atom>
   <op>
      <Const type="rif:iri">
         http://rif.example.com/2008/joe#owns
      </Const>
   </op>
   <args rif:ordered="yes">
      <Var> c </Var>
      <Var> p </Var>
   </args>
</Atom>
```

5.2.2.2 Equal

In RIF, the `Equal` element is used to serialize equality atomic formulas.

The `Equal` element must contain one `left` sub-element and one `right` sub-element. The content of the `left` and `right` elements must be a construct from the `TERM` abstract class. The order of the sub-elements is not significant.

```
<Equal>
   <left> TERM </left>
   <right> TERM </right>
</Equal>
```

5.2.2.3 Member

In RIF, the `Member` element is used to serialize membership atomic formulas.

The `Member` element contains two unordered sub-elements:

- the `instance` elements must be a construct from the `TERM` abstract class. It is required;
- the `class` element must be a construct from the `TERM` abstract class. It is required as well.

```
<Member>
   <instance> TERM </instance>
   <class> TERM </class>
</Member>
```

Example 2.5. The example below shows the RIF XML serialization of a boolean expression that tests whether the individual denoted by the variable *?c* is a member of the class *Chicken* that is defined in the example namespace *http://rif.example.com/2008/joe#*.

```
<Member>
   <instance> <Var> c </Var> </instance>
   <class>
      <Const type="rif:iri">
         http://rif.example.com/2008/joe#Chicken
      </Const>
   </class>
</Member>
```

5.2.2.4 Subclass

In RIF, the `Subclass` element is used to serialize subclass atomic formulas.

The `Subclass` element contains two unordered sub-elements:

- the `sub` element must be a construct from the `TERM` abstract class. It is required;
- the `super` elements must be a construct from the `TERM` abstract class. It is required.

```
<Subclass>
   <sub> TERM </sub>
   <super> TERM </super>
</Subclass>
```

5.2.2.5 Frame

In RIF, the `Frame` element is used to serialize frame atomic formulas.

Accordingly, a `Frame` element must contain:

- an `object` element, that contains an element of the `TERM` abstract class, the content of which serializes the individual;
- zero to many `slot` elements, each containing a `Prop` element that serializes an attribute-value pair as a pair of elements of the `TERM` abstract class, the first one that serializes the name of the attribute (or property); the second that serializes the attribute's value. The order of the `slot`'s sub-elements is significant and MUST be preserved. This is

emphasized by the required value `"yes"` of the required attribute `rif:ordered`.

```
<Frame>
   <object> TERM </object>
   <slot rif:ordered="yes"> TERM TERM </slot>*
</Frame>
```

Example 2.6. The example below shows the RIF XML syntax that serializes an expression that states that the object denoted by the variable *?c* has the value denoted by the variable *?a* for the property *Chicken/age* that is defined in the example namespace *http://rif.example.com/2008/joe#*.

```
<Frame>
   <object> <Var> c </Var> </object>
   <slot rif:ordered="yes">
      <Const type="rif:iri">
         ttp://rif.example.com/2008/joe#Chicken/age
      </Const>
      <Var> a </Var>
   </slot>
</Frame>
```

Editor's Note: The example uses an XPath style for the key. How externally specified data models and their elements should be referenced is still under discussion (see ISSUE-37).

5.2.2.6 External

In RIF-PRD, the `External` element is also used to serialize an externally defined atomic formula.

When it is a `ATOMIC` (as opposed to a `TERM`; that is, in particular, when it appears in a place where an `ATOMIC` is expected, and not a `TERM`), the `External` element contains one `content` element that contains one `Atom` element. The `Atom` element serializes the externally defined atom properly said:

```
<External>
   <content>
      Atom
   </content>
</External>
```

The `op Const` in the `Atom` element must be a symbol of type `rif:iri` that must uniquely identify the externally defined predicate to be applied to the `args TERM`s. It can be one of the builtin predicates specified for RIF dialects, as listed in section List of RIF Builtin Predicates and Functions of the RIF data types and builtins document, or it can be application specific. In the latter case, it is up to the producers and consumers of RIF-PRD rulesets that reference non-builtin predicates to agree on their semantics.

Example 2.7. The example below shows the RIF XML serialization of an externally defined atomic formula that tests whether the value denoted by the variable named *?a* (e.g. the age of a chicken) is greater than the integer value 8, where the test is intended to behave like the builtin predicate op:numeric-greater-than as specified in XQuery 1.0 and XPath 2.0 Functions and Operators.

In the example, the prefix `op:` is associated with the namespace http://www.w3.org/2007/rif-builtin-predicate#.

```
<External>
   <content>
      <Atom>
         <op> <Const type="rif:iri"> op:numeric-greater-than </Const> </op>
         <args rif:ordered="yes">
            <Var> ?a </Var>
            <Const type="xsd:decimal"> 8 </Const>
         </args>
      </Atom>
   </content>
</External>
```

5.2.3 FORMULA

The `FORMULA` class is used to serialize condition formulas, that is, atomic formulas, conjunctions, disjunctions, negations and existentials.

As an abstract class, `FORMULA` is not associated with specific XML markup in RIF-PRD instance documents.

```
[ ATOMIC | And | Or | NmNot | Exists ]
```

5.2.3.1 ATOMIC

An atomic formula is serialized using a single `ATOMIC` statement. See specification of ATOMIC, above.

5.2.3.2 And

A conjunction is serialized using the `And` element.

The `And` element contains zero or more `formula` sub-elements, each containing an element of the `FORMULA` group.

```
<And>
   <formula> FORMULA </formula>*
</And>
```

5.2.3.3 Or

A disjunction is serialized using the `Or` element.

The `Or` element contains zero or more `formula` sub-elements, each containing an element of the `FORMULA` group.

```
<Or>
   <formula> FORMULA </formula>*
</Or>
```

5.2.3.4 NmNot

A negation is serialized using the `NmNot` element.

The `NnNot` element contains exactly one `formula` sub-element. The `formula` element contains an element of the `FORMULA` group, that serializes the negated statement.

```
<NmNot>
   <formula> FORMULA </formula>
</NmNot>
```

Editor's Note: The name of that construct may change, including the tag of the XML element.

5.2.3.5 Exists

An existentially quantified formula is serialized using the `Exists` element.

The `Exists` element contains:

- one or more `declare` sub-elements, each containing a `Var` element that serializes one of the existentially quantified variables;
- exactly one required `formula` sub-element that contains an element from the `FORMULA` abstract class: the `FORMULA` serializes the formula in the scope of the quantifier.

```
<Exists>
   <declare> Var </declare>+
   <formula> FORMULA </formula>
</Exists>
```

Example 2.8. The example below shows the RIF XML serialization of a boolean expression that tests whether the chicken denoted by variable *?c* is older than 8 months, by testing the existence of a value, denoted by variable *?a*, that is both the age of *?c*, as serialized as a `Frame` element, as in example 2.6, and greater than 8, as serialized as an `External ATOMIC`, as in example 2.7.

```
<Exists>
   <declare> <Var> a </Var> </declare>
   <formula>
      <And>
         <Frame>
            <object> <Var> c </Var> </object>
            <slot rif:ordered="yes">
               <Const type="rif:iri">
                  http://rif.example.com/2008/joe#Chicken/age
               </Const>
               <Var> a </Var>
            </slot>
         </Frame>
         <External>
            <content>
               <Atom>
                  <op> <Const type="rif:iri"> op:numeric-greater-than </Con
                  <args rif:ordered="yes">
                     <Var> a </Var>
                     <Const type="xsd:decimal"> 8 </Const>
                  </args>
               </Atom>
            </content>
         </External>
      </And>
   </formula>
</Exists>
```

5.3 Actions

This section specifies the XML syntax that is used to serialize the action part of a rule supported by RIF-PRD.

5.3.1 ATOMIC_ACTION

The `ATOMIC_ACTION` class of elements is used to serialize the atomic actions: *assert* and *retract*.

```
        [ Assert | Retract ]
```

5.3.1.1 Assert

An atomic assertion action is serialized using the `Assert` element.

An atom (positional or with named arguments), a frame, a membership atomic formula and a subclass atomic formula can be asserted.

The `Assert` element has one `target` sub-element that contains an `Atom`, a `Frame`, a `Member` or a `Subclass` element that represents the facts to be added on performing the action.

```
        <Assert>
             <target> [ Atom | Frame | Member | Subclass ] </target>
        </Assert>
```

5.3.1.2 Retract

The `Retract` construct is used to serialize retract atomic actions, that result in removing a fact from the fact base. Only atoms (positional or with named arguments), frames and frame objects can be retracted.

The `Retract` element has one `target` sub-element that contains an `Atom`, a `Frame`, or a `TERM` construct that represents the facts or the object to be removed on performing the action.

```
        <Retract>
           <target> [ Atom | Frame | TERM ] </target>
        </Retract>
```

Example 2.10. The example below shows the RIF XML representation of an action that updates the chicken-potato ownership table by removing the predicate that states that the chicken denoted by variable *?c* owns the potato denoted by variable *?p*. The predicate is represented as in example 2.4.

```
    <Retract>
       <target>
          <Atom>
```

```
         <op>
            <Const type="rif:iri">
               http://rif.example.com/2008/joe#owns
            </Const>
         </op>
         <args rif:ordered="yes">
            <Var> c </Var>
            <Var> p </Var>
         </args>
      </Atom>
   </target>
</Retract>
```

5.3.2 INITIALIZATION

The `INITIALIZATION` class of elements is used to serialized the constructs that specify the initial value assigned an action variable, in an action variable declaration: it can be a new frame identifier or the slot value of a frame.

As an abstract class, `INITIALIZATION` is not associated with specific XML markup in RIF-PRD instance documents.

```
[ New | Frame ]
```

5.3.2.1 New

The `New` element is used to serialize the construct used to create a new frame identifer.

The `New` element has an `instance` sub-element that contains a `Var`, which serializes the action variable intended to be assigned the new frame identifier.

```
<New>
   <instance> Var </instance>?
</New>
```

5.3.2.2 Frame

The `Frame` element is used, with restrictions, to the serialize the construct used to assign an action variable the slot value of a frame.

In that position, a `Frame` must contain, in addition to its `object` sub-element, one and only one `slot` sub-element, that contains a sub-element of the `TERM` class, serializing the slot name, and a `Var` sub-element, that serializes the action variable intended to be assigned the slot value.

```
<Frame>
   <object> TERM </object>
   <slot rif:ordered="yes">
      TERM
      Var
   </slot>
</Frame>
```

5.3.3 ACTION_BLOCK

The ACTION_BLOCK class of constructs is used to represent the conclusion, or action part, of a production rule serialized using RIF-PRD.

If action variables are declared in the action part of a rule, or if some atomic actions are not assertions, the conclusion must be serialized as a full action block, using the Do element. However, simple action blocks that contain only one or more assert actions can be serialized like the conclusions of logic rules using RIF-Core or RIF-BLD, that is, as a single asserted ATOMIC or as a conjunction of the asserted ATOMICs.

As an abstract class, ACTION_BLOCK is not associated with specific XML markup in RIF-PRD instance documents.

```
[ Do | And | ATOMIC ]
```

5.3.3.1 Do

An action block is serialized using the Do element.

A Do element contains:

- zero or more actionVar sub-elements, each of them used to serialize one action variable declaration. Accordingly, an actionVar element must contain a Var sub-element, that serializes the declared variable; followed by a sub-element of the INITIALIZATION class, that serializes the initial value assigned to the declared variable;
- one actions sub-element that serializes the sequence of atomic actions in the action block, and that contains, accordingly, a sequence of one or more sub-elements of the ATOMIC_ACTION class.

```
<Do>
   <actionVar rif:ordered="yes">
      Var
      INITIALIZATION
   </actionVar>*
   <actions rif:ordered="yes">
```

```
          ATOMIC_ACTION+
       </actions>
    </Do>
```

Example 2.9. The example below shows the RIF XML representation of an action block that asserts a new 100 decigram potato.

```
<Do>
   <actionVar>
      <Var>p</Var>
      <New>
         <instance><Var>p</Var></instance>
      </New>
   </actionVar>
   <actions  rif:ordered="yes">
      <Assert>
         <target>
            <Member>
               <instance><Var>p</Var></instance>
               <class>
                  <Const type="rif:iri">http://rif.example.com/2008/joe#Pot
               </class>
            </Member>
         </target>
      </Assert>
      <Assert>
         <target>
            <Frame>
               <object><Var>p</Var></object>
               <slot rif:ordered="yes">
                  <Const type="rif:iri">http://rif.example.com/2008/joe#wei
                  <Const type="xsd:decimal">100</Const>
               </slot>
            </Frame>
         </target>
      </Assert>
   </actions>
</Do>
```

5.3.3.2 And

An action block that contains only assert atomic actions can be serialized using the `And` element, for compatibility with RIF-Core and RIF-BLD.

However, the atomic formulas allowed as conjuncts are restricted to atoms (positional or with named arguments), frames, and membership or subclass atomic formulas.

In that position, an `And` element must contain at least one sub-element.

```
<And>
   <formula> [ Atom | Frame | Member | Subclass ] </formula>+
</And>
```

5.3.3.3 ATOMIC

For compatibility with RIF-Core and RIF-BLD, an action block that contains only a single assert atomic action can be serialzed as the `ATOMIC` that serializes the target of the assert action.

However, the only atomic formulas allowed in tat position are the ones that are allowed as targets to an atomic assert action: atoms (positional or with named arguments), frames, and membership or subclass atomic formulas.

```
[ Atom | Frame | Member | Subclass ]
```

5.4 Rules and Groups

This section specifies the XML constructs that are used, in RIR-PRD, to serialize rules and groups.

5.4.1 RULE

In RIF-PRD, the `RULE` class of constructs is used to serialize rules, that is, unconditional as well as conditional actions, or rules with bound variables.

As an abstract class, `RULE` is not associated with specific XML markup in RIF-PRD instance documents.

```
[ Implies | Forall | ACTION_BLOCK ]
```

5.4.1.1 ACTION_BLOCK

An unconditional action block is serialized, in RIF-PRD XML, using the `ACTION_BLOCK` class of construct.

5.4.1.2 Implies

Conditional actions are serialized, in RIF-PRD, using the XML element `Implies`.

The `Implies` element contains an optional `if` sub-element and a `then` sub-element:

- the optional `if` element contains an element from the `FORMULA` class of constructs, that serializes the condition of the rule;
- the required `then` element contains one element from the `ACTION_BLOCK` class of constructs, that serializes its conlusion.

```
<Implies>
   <if> FORMULA </if>?
   <then> ACTION_BLOCK </then>
</Implies>
```

5.4.1.3 Forall

The `Forall` construct is used, in RIF-PRD, to represent rules with bound variables.

The `Forall` element contains:

- one or more `declare` sub-elements, each containing a `Var` element that represents one of the universally quantified variable;
- zero or more `pattern` sub-elements, each containing an element from the `FORMULA` group of constructs, serializing one binding pattern;
- exactly one `formula` sub-element that serializes the formula in the scope of the variables binding, and that contains an element of the `RULE` group.

```
<Forall>
   <declare> Var </declare>+
   <pattern> FORMULA </pattern>*
   <formula> RULE </formula>
</Forall>
```

Editor's Note: Nested `Forall`s make explicit the scope of the declared variables, and, thus, impose an order on the evaluation of the `pattern` and `if` `FORMULA`s in a rule. That ordering and the use of patterns to constrain the binding of variables may be of practical significance for some production rule systems, but they are irrelevant with respect to the intended semantics of the rules being interchanged (although they would be relevant if RIF-PRD was to be extended to support some kind of "else" or "case" construct). In addition, RIF-BLD does not allow nested `Forall` and does not support the association of contraining `pattern`s to declared variables. The working group seeks feedback regarding whether nested `Forall` and constraininting `pattern` should be supported in RIF-PRD, to the cost of reducing the interoperability with RIF-BLD.

Example 2.10. The example below shows how the CMP rule extract: *"if a chicken owns a potato and ..."* could be serialized using a binding `pattern` `FORMULA`:

```
<Forall>
   <declare><Var>chicken</Var></declare>
   <formula>
      <Forall>
         <declare><Var>potato</Var></declare>
         <pattern>
            <External>
               <content>
                  <Atom>
                     <Const type="rif:iri">
                        http://rif.example.com/2008/joe#owns
                     </Const>
                     <Args rif:ordered="yes">
                        <Var>chicken</Var>
                        <Var>potato</Var>
                     </Args>
                  </Atom>
               </content>
            </External>
         </pattern>
         <formula>
            ...
         </formula>
      </Forall>
   </formula>
</Forall>
```

5.4.2 Group

The `Group` construct is used to serialize a group.

The `Group` element has zero or one `behavior` sub-element and zero or more `sentence` sub-elements:

- the `behavior` element contains
  - zero or one `ConflictResolution` sub-element that contains exactly one IRI. The IRI identifies the conflict resolution strategy that is associated with the `Group`; and
  - zero or one `Priority` sub-element that contains exactly one signed integer between -10,000 and 10,000. The integer associates a priority with the `Group`'s sentences;
- a `sentence` element contains either a `Group` element or an element of the `RULE` abstract class of constructs.

```
<Group>
   <behavior>
      <ConflictResolution> xsd:anyURI </ConflictResolution>?
      <Priority> -10,000 ≤ xsd:int ≤ 10,000 </Priority>?
   </behavior>?
   <sentence> [ RULE | Group ] </sentence>*
</Group>
```

5.5 Constructs carrying no semantics

5.5.1 Document

The `Document` is the root element in a RIF-PRD instance document.

The `Document` contains zero or one `payload` sub-element, that must contain a `Group` element.

```
<Document>
   <payload> Group </payload>?
</Document>
```

5.5.2 Metadata

Metadata can be associated with any concrete class element in RIF-PRD: those are the elements with a CamelCase tagname starting with an upper-case character:

```
CLASSELT = [ TERM | ATOMIC | FORMULA | ACTION | RULE | Group | Document
```

An identifier can be associated to any instance element of the abstract `CLASSELT` class of constructs, as an optional `id` sub-element that MUST contain a `Const` of type `rif:local` or `rif:iri`.

Metadata can be included in any instance of a concrete class element using the `meta` sub-element.

The RIF-PRD `Frame` construct is used to serialize metadata: the content of the `Frame`'s `object` sub-element identifies the object to which the metadata is associated:, and the `Frame`'s `slot`s represent the metadata properly said as property-value pairs.

If the all the metadata is related to the same object, the `meta` element can contain a single `Frame` sub-element. If metadata related to several different objects need be serialized, the `meta` role element can contain an `And` element with zero or more `formula` sub-elements, each containing one `Frame` element.

```
<CLASSELT>
    <id> Const </id>?
    <meta>
       [ Frame
         |
         <And>
            <formula> Frame </formula>*
         </And>
       ]
    </meta>?
    other CLASSELT content
 </CLASSELT>
```

Notice that the content of the `meta` sub-element of an instance of a RIF-PRD class element is not necessarily associated to that same instance element: only the content of the `object` sub-element of the `Frame` that represents the metadata specifies what the metadata is about, not where it is included in the instance RIF document.

It is suggested to use Dublin Core, RDFS, and OWL properties for metadata, along the lines of http://www.w3.org/TR/owl-ref/#Annotations -- specifically owl:versionInfo, rdfs:label, rdfs:comment, rdfs:seeAlso, rdfs:isDefinedBy, dc:creator, dc:description, dc:date, and foaf:maker.

Example 2.11. TBC

6 Presentation syntax

To make it easier to read, a non-normative, lightweight notation was introduced to complement the mathematical english specification of the abstract syntax and the semantics of RIF-PRD. This section specifies a presentation syntax for RIF-PRD, that extends that notation. The presentation syntax is not normative. However, it may help implementers by providing a more succinct overview of RIF-PRD syntax.

Editor's Note: An uptodate version of the RIF-PRD presentation synatx will be included in a future version of this document.

7 References

[CIR04]

Production Systems and Rete Algorithm Formalisation, Cirstea H., Kirchner C., Moossen M., Moreau P.-E. Rapport de recherche n° inria-00280938 - version 1 (2004).

[CURIE]

CURIE Syntax 1.0 - A compact syntax for expressing URIs, W3C note 27 October 2005, M. Birbeck (ed.).

[HAK07]
Data Models as Constraint Systems: A Key to the Semantic Web, Hassan Ait-Kaci, Constraint Programming Letters, 1:33--88, 2007.

[PLO04]
A Structural Approach to Operational Semantics, Gordon D. Plotkin, Journal of Logic and Algebraic Programming, Volumes 60-61, Pages 17-139 (July - December 2004).

[PRR07]
Production Rule Representation (PRR), OMG specification, version 1.0, 2007.

[RDF-CONCEPTS]
Resource Description Framework (RDF): Concepts and Abstract Syntax, Klyne G., Carroll J. (Editors), W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. Latest version available at http://www.w3.org/TR/rdf-concepts/.

[RDF-SCHEMA]
RDF Vocabulary Description Language 1.0: RDF Schema, Brian McBride, Editor, W3C Recommendation 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/. Latest version available at http://www.w3.org/TR/rdf-schema/.

[RFC-3066]
RFC 3066 - Tags for the Identification of Languages, H. Alvestrand, IETF, January 2001, http://www.isi.edu/in-notes/rfc3066.txt.

[RFC-3987]
RFC 3987 - Internationalized Resource Identifiers (IRIs), M. Duerst and M. Suignard, IETF, January 2005, http://www.ietf.org/rfc/rfc3987.txt.

[RIF-BLD]
RIF Basic Logic Dialect Harold Boley, Michael Kifer, eds. W3C Working Draft, 30 July 2008, http://www.w3.org/TR/2008/WD-rif-bld-20080730/. Latest version available at http://www.w3.org/TR/rif-bld/.

[RIF-Core]
RIF Core Harold Boley, Gary Hallmark, Michael Kifer, Adrian Paschke, Axel Polleres, Dave Reynolds, eds. W3C Editor's Draft, 18 December 2008, http://www.w3.org/2005/rules/wg/draft/ED-rif-core-20081218/. Latest version available at http://www.w3.org/2005/rules/wg/draft/rif-core/.

[RIF-DTB]
RIF Datatypes and Built-Ins 1.0 Axel Polleres, Harold Boley, Michael Kifer, eds. W3C Editor's Draft, 18 December 2008, http://www.w3.org/2005/rules/wg/draft/ED-rif-dtb-20081218/. Latest version available at http://www.w3.org/2005/rules/wg/draft/rif-dtb/.

[XDM]
XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C Recommendation, World Wide Web Consortium, 23 January 2007. This version is http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/. Latest version available at http://www.w3.org/TR/xpath-datamodel/.

[XML-SCHEMA2]
XML Schema Part 2: Datatypes Second Edition, W3C Recommendation, World Wide Web Consortium, 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/. Latest version available at http://www.w3.org/TR/xmlschema-2/.

[XPath-Functions]
XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Recommendation, World Wide Web Consortium, 23 January 2007, http://www.w3.org/TR/2007/REC-xpath-functions-20070123/. Latest version available at http://www.w3.org/TR/xpath-functions/.

8 Appendix: XML schema

TBD

9 Appendix: Compatibility with RIF-BLD

9.1 Syntactic compatibility between RIF-PRD and RIF-BLD

Editor's Note: RIF-PRD and RIF-BLD [RIF-BLD]] share essentially the same presentation syntax and XML syntax. Future versions of this, or another, RIF document will include a complete, construct by construct, comparison table of RIF-PRD and RIF-BLD presentation and XML syntaxes.

9.2 Semantic compatibility between RIF-PRD and RIF-BLD

The intended semantics of any RIF XML document which is both a syntactically valid RIF-PRD document and a syntactically valid RIF-BLD document is the same whether it is considered a RIF-PRD or a RIF-BLD document. For any input set of facts, the set of rules contained in the document must produce the same output set of facts whether it is consumed as a RIF-PRD or a RIF-BLD document.

Proof. TBC