W3C Proposed Edited Recommendation

RIF Production Rule Dialect

W3C Proposed Edited Recommendation 11 December 2012

This version:
http://www.w3.org/TR/2012/PER-rif-prd-20121211/
Latest version:
http://www.w3.org/TR/rif-prd/
Previous version:
http://www.w3.org/TR/2010/REC-rif-prd-20100622/
Editors:
Christian de Sainte Marie, IBM/ILOG
Gary Hallmark, Oracle
Adrian Paschke, Freie Universitaet Berlin

A color-coded version of this document showing changes made since the previous version is also available.

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

Abstract

This document, developed by the Rule Interchange Format (RIF) Working Group, specifies the production rule dialect of the W3C rule interchange format (RIF-PRD), a standard XML serialization format for production rule languages.

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May Be Superseded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 12 documents:

1. RIF Overview
2. RIF Core Dialect
3. RIF Basic Logic Dialect
4. RIF Production Rule Dialect (this document)
5. RIF Framework for Logic Dialects
6. RIF Datatypes and Built-Ins 1.0
7. RIF RDF and OWL Compatibility
8. OWL 2 RL in RIF
9. RIF Combination with XML data
10. RIF In RDF
11. RIF Test Cases
12. RIF Primer

Summary of Changes

There have been no substantive changes since the previous version. For details on the minor changes see the change log and color-coded diff.

W3C Members Please Review By 8 January 2013

The W3C Director seeks review and feedback from W3C Advisory Committee representatives, via their review form by 8 January 2013. This will allow the Director to assess consensus and determine whether to issue this document as a W3C Edited Recommendation.

Others are encouraged by the Rule Interchange Format (RIF) Working Group to continue to send reports of implementation experience, and other feedback, to public-rif-comments@w3.org (public archive). Reports of any success or difficulty with the test cases are encouraged. Open discussion among developers is welcome at public-rif-dev@w3.org (public archive).

No Endorsement

Publication as a Proposed Edited Recommendation 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.

Table of Contents

1 Overview

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

The production rule dialect is one of a set of rule interchange dialects that also includes the RIF Core dialect ([RIF-Core]) and the RIF basic logic dialect ([RIF-BLD]).

RIF-Core, the core dialect of the W3C rule interchange format, is designed to support the interchange of definite Horn rules without function symbols ("Datalog"). RIF-Core has both a standard first-order semantics and an operational semantics. Syntactically, RIF-Core has a number of extensions of Datalog:

- frames as in F-logic [KLW95],
- internationalized resource identifiers (or IRIs, defined by [RFC-3987]) as identifiers for concepts, and
- XML Schema datatypes [XML-SCHEMA2].

RIF-Core is based on a rich set of datatypes and built-ins that are aligned with Web-aware rule system implementations [RIF-DTB]. In addition, the RIF RDF and OWL Compatibility document [RIF-RDF-OWL] specifies the syntax and semantics of combinations of RIF-Core, RDF, and OWL documents.

RIF-Core is intended to be the common core of all RIF dialects, and it has been designed, in particular, to be a useful common subset of RIF-BLD and RIF-PRD. RIF-PRD includes and extends RIF-Core, and, therefore, RIF-PRD inherits all RIF-Core features. These features make RIF-PRD a Web-aware (even a semantic Web-aware) language. However, it should be kept in mind that RIF is designed to enable interoperability among rule languages in general, and its uses are not limited to the Web.

This document targets designers and developers of RIF-PRD implementations. A RIF-PRD implementation is a software application that serializes production rules as RIF-PRD XML (producer application) and/or that deserializes RIF-PRD XML documents into production rules (consumer application).

1.1 Production rule interchange

Production rules 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 RIF-Core and its basic logic dialect extension, RIF-BLD). The *then* part contains actions. An action can assert facts, modify facts, retract facts, and have other side-effects. In general, an action is different from the conclusion of a logic rule, which contains only a logical statement. However, the conclusion of rules interchanged using RIF-Core can be interpreted, according to RIF-PRD operational semantics, as actions that assert facts in the knowledge base.

Example 1.1. The following are examples of production rules:

- *A customer becomes a "Gold" customer when his cumulative purchases during the current year reach $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.* ☐

Because RIF-PRD is a production rule interchange format, it specifies an abstract syntax that shares features with concrete production rule languages, and it associates the abstract constructs with normative semantics and a normative XML concrete syntax. Annotations (*e.g.* rule author) are the only constructs in RIF-PRD without a formal semantics.

The abstract syntax is specified in mathematical English, and the abstract syntactic constructs that are defined in the sections Abstract Syntax of Conditions, Abstract Syntax of Actions and Abstract Syntax of Rules and Rulesets, are mapped into the concrete XML constructs in the section XML syntax. A lightweight notation is used, instead of the XML syntax, to tie the abstract syntax to the specification of the semantics. A more complete presentation syntax is specified using an EBNF in Presentation Syntax. However, only the XML syntax and the associated semantics are normative. The normative XML schema is included in Appendix: XML Schema.

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

```
Prefix(ex <http://example.com/2008/prd1#>)
(* ex:rule_1 *)
Forall ?customer ?purchasesYTD (
 If   And( ?customer#ex:Customer
           ?customer[ex:purchasesYTD->?purchasesYTD]
           External(pred:numeric-greater-than(?purchasesYTD 5000)) )
 Then Do( Modify(?customer[ex:status->"Gold"]) ) )
```

☐

Production rules are statements of programming logic that specify the execution of one or more actions when their conditions are satisfied. Production rules have an operational semantics, that the OMG Production Rule Representation specification [OMG-PRR] summarizes 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 which rule instance will be executed;
3. *Act:* the state of the data source is changed, by executing the selected rule instance's 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 transitions result from executing the action part of instantiated rules. When several rules are found to be executable at the same time, during the rule execution process, a *conflict resolution strategy* is used to select the rule to execute. The section Conflict resolution specifies how a conflict resolution strategy can be attached to a rule set. RIF-PRD defines a default conflict resolution strategy.

In the section Semantics of condition formulas, the semantics of the condition part of rules in RIF-PRD is specified operationally, in terms of matching substitutions. To emphasize the overlap between the rule conditions of RIF-BLD and RIF-PRD, and to share the same RIF definitions for datatypes and built-ins [RIF-DTB], an alternative, and equivalent, specification of the semantics of rule conditions in RIF-PRD, using a model theory, is provided in the appendix Model-theoretic semantics of RIF-PRD condition formulas.

The semantics of condition formulas and the semantics of rules and rule sets make no assumption regarding how condition formulas are evaluated. In particular, they do not require that condition formula be evaluated using pattern matching. However, RIF-PRD conformance, as defined in the section Conformance and interoperability, requires only support for safe rules, that is, forward-chaining rules where the conditions can be evaluated based on pattern matching only.

In the section Operational semantics of actions, the semantics of the action part of rules in RIF-PRD is specified using a transition relation between successive states of the data source, represented by ground condition formulas, thus making the link between the model-theoretic semantics of conditions and the operational semantics of rules and rule sets.

The abstract syntax of RIF-PRD documents, and the semantics of the combination of multiple RIF-PRD documents, is specified in the section Document and imports.

In addition to externally specified functions and predicates, and in particular, in addition to the functions and predicates built-ins defined in [RIF-DTB], RIF-PRD supports externally specified actions, and defines one action built-in, as specified in the section Built-in functions, predicates and actions.

1.2 Running example

The same example rules will be used throughout the document to illustrate the syntax and the semantics of RIF-PRD.

The rules are about the status of customers at a shop, and the discount awarded to them. The rule set contains four rules, to be applied when a customer checks out:

1. Gold rule: *A "Silver" customer with a shopping cart worth at least $2,000 is awarded the "Gold" status*.
2. Discount rule: *"Silver" and "Gold" customers are awarded a 5% discount on the total worth of their shopping cart*.
3. New customer and widget rule: *A "New" customer who buys a widget is awarded a 10% discount on the total worth of her shopping cart, but she looses any voucher she may have been awarded*.
4. Unknown status rule: *A message must be printed, identifying any customer whose status is unknown (that is, neither "New", "Bronze", "Silver" or "Gold"), and the customer must be assigned the status: "New"*.

The *Gold rule* must be applied first; that is, e.g., a customer with "Silver" status and a shopping cart worth exactly $2,000 should be promoted to "Gold" status, before being given the 5% discount that would disallow the application of the *Gold rule* (since the total worth of his shopping cart would then be only $1,900).

In the remainder of this document, the prefix `ex1` stands for the fictitious namespace of this example: `http://example.com/2009/prd2#`.

2 Conditions

This section specifies the syntax and semantics of the condition language of RIF-PRD.

The RIF-PRD condition language specification depends on Section Constants, Symbol Spaces, and Datatypes, in the RIF data types and builtins specification [RIF-DTB].

2.1 Abstract syntax

The alphabet of the RIF-PRD condition language consists of:

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

For the sake of readability and simplicity, this specification introduces a notation for these constructs. The 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.

RIF-PRD supports externally defined functions only (including the built-in functions specified in [RIF-DTB]). RIF-PRD, unlike RIF-BLD, does not support uninterpreted function symbols (sometimes called logically defined functions).

RIF-PRD supports a form of negation. Neither RIF-Core nor RIF-BLD support negation, because logic rule languages use many different and incompatible kinds of negation. See also the RIF framework for logic dialects [RIF-FLD].

2.1.1 Terms

The most basic construct in the RIF-PRD condition language is the *term*. RIF-PRD defines several kinds of term: *constants*, *variables*, *lists* and *positional* terms.

Definition (Term).

1. *Constants and variables*. If `t` ∈ `Const` or `t` ∈ `Var` then `t` is a ***simple term***;
2. *List terms*. A ***list*** has the form `List(`t_1 ... t_m`)`, where m≥0 and t_1, ..., t_m are ground terms, i.e. without variables. A list of the form `List()` (i.e., a list in which `m=0`) is called the ***empty list***;
3. *Positional terms*. If `t` ∈ `Const` and t_1, ..., t_n, n≥0, are terms then `t(`t_1 ... t_n`)` is a ***positional term***.
   Here, the constant `t` represents a function and t_1, ..., t_n represent argument values. ☐

To emphasize interoperability with RIF-BLD, positional terms may also be written: `External(t(`t_1...t_n`))`.

Example 2.1.

- `List("New" "Bronze" "Silver" "Gold")` is a term that denotes the list of the values for a customer's status that are known to the system. The elements of the list, `"New"`, `"Bronze"`, `"Silver"` and `"Gold"` are terms denoting string constants;
- `func:numeric-multiply(?value, 0.90)` is a positional term that denotes the product of the value assigned to the variable `?value` and the constant `0.90`. That positional term can be used, for instance, to represent the new value, taking the discount into account, to be assigned a customer's shopping cart, in the rule *New customer and widget rule*. An alternative notation is to mark explicitly the positional term as externally defined, by wrapping it with the `External` indication: `External(func:numeric-multiply(?value, 0.90))` ☐

2.1.2 Atomic formulas

Atomic formulas are the basic tests of the RIF-PRD condition language.

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

1. *Positional atomic formulas*. If `t` ∈ `Const` and t_1, ..., t_n, n≥0, are terms then `t(`t_1 ... t_n`)` is a ***positional atomic formula*** (or simply an ***atom***)
2. *Equality atomic formulas*. `t = s` is an ***equality atomic formula*** (or simply an ***equality***), if `t` and `s` are terms
3. *Class membership atomic formulas*. `t#s` is a ***membership atomic formula*** (or simply ***membership***) if `t` and `s` are terms. The term `t` is the *object* and the term `s` is the *class*
4. *Subclass atomic formulas*. `t##s` is a ***subclass atomic formula*** (or simply a ***subclass***) if `t` and `s` are terms
5. *Frame atomic formulas*. `t[`p_1`->`v_1 ... p_n`->`v_n`]` is a ***frame atomic formula*** (or simply a ***frame***) if `t`, p_1, ..., p_n, v_1, ..., v_n, n ≥ 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*
6. *Externally defined atomic formulas.* If `t` is a positional atomic formula then `External(t)` is an ***externally defined atomic formula***. ☐

Class membership, subclass, and frame atomic formulas are used to represent classifications, class hierarchies and object-attribute-value relations.

Externally defined atomic formulas are used, in particular, for representing built-in predicates.

In the RIF-BLD specification, as is common practice in logic languages, atomic formulas are also called *terms*.

Example 2.2.

- The membership formula `?customer # ex1:Customer` tests whether the individual bound to the variable `?customer` is a member of the class denoted by `ex1:Customer`.
- The atom `ex1:Gold(?customer)` tests whether the customer represented by the variable `?customer` has the "*Gold*" status.
- Alternatively, gold status can be tested in a way that is closer to an object-oriented representation using the frame formula `?customer[ex1:status->"Gold"]`.
- The following atom uses the built-in predicate `pred:list-contains` to validate the status of a customer against a list of allowed customer statuses: `External(pred:list-contains(List("New", "Bronze", "Silver", "Gold"), ?status))`. ☐

2.1.3 Formulas

Composite truth-valued constructs are called *formulas*, in RIF-PRD.

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

More general formulas are constructed out of 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 ≥ 0, are condition formulas then so is And($φ_1$ ... $φ_n$), 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 ≥ 0, are condition formulas then so is Or($φ_1$ ... $φ_n$), 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 Not(φ), called a *negative* formula.
5. *Existentials*: If φ is a condition formula and $?V_1$, ..., $?V_n$, n>0, are variables then Exists $?V_1$ ... $?V_n$(φ) 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.

Example 2.3.

- The condition of the *New customer and widget rule*: *A "New" customer who buys a widget*, can be represented by the following RIF-PRD condition formula:

```
And( ?customer # ex1:Customer
     ?customer[ex1:status->"New"]
     Exists ?shoppingCart ?item ( And ( ?customer[ex1:shoppingCart->?shoppingCart]
                                        ?shoppingCart[ex1:containsItem->?item]
                                        ?item # ex1:Widget) )
                                )
    )
```

☐

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

- if *e* ∈ *Const*, then *Var(e)* = ∅;
- if *e* ∈ *Var*, then *Var(e)* = {*e*};
- if *e* is a *list term*, then *Var(e)* = ∅;
- if $f(arg_1...arg_n)$, $n ≥ 0$, is a positional term, then, $Var(f(arg_1...arg_n)) = \cup_{i=1...n} Var(arg_i)$;
- if $p(arg_1...arg_n)$, n ≥ 0, is an atom, then, $Var(p(arg_1...arg_n)) = Var(External(p(arg_1...arg_n))) = \cup_{i=1...n} Var(arg_i)$;
- 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, n ≥ 1, 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, n ≥ 0, are condition formulas, then $Var([And|Or](f_1...f_n)) = \cup_{i=0...n} Var(f_i)$;
- if *f* is a condition formula, then *Var(Not(f))* = *Var(f)*;
- if *f* is a condition formula and x_i ∈ *Var* for *i* = 1...n, *n* ≥ *1*, then, $Var(Exists\ x_1 ... x_n\ (f)) = Var(f) - \{x_1...x_n\}$.

Definition (Ground formula). A condition formula φ is a ***ground formula*** if and only if *Var*φ = ∅ 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

Not all formulas are well-formed in 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 the following subsets:

- A subset of individuals. The symbols in `Const` that belong to the primitive datatypes are required to be individuals;
- A subset for external function symbols;
- A subset of plain predicate symbols;
- A subset for external predicate symbols.

As seen from the following definitions, these subsets are not specified explicitly but, rather, are inferred from the occurrences of the symbols.

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 predicate in an atomic subformula of the form `s(...)` then `s` occurs in the *context of a (plain) predicate symbol*;
- If `s` occurs as a predicate in an atomic subformula `External(s(...))` then `s` occurs in the *context of an external predicate symbol*;
- If `s` occurs as a function in a term (which is not a subformula) `s(...)` (or `External(s(...))`) then `s` occurs in the *context of an (external) function symbol*;
- If `s` occurs in any other context (e.g. in a frame: `s[...]`, `...[s->...]`, or `...[...->s]`; or in a positional atom: `p(...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 positional term, `t` (or `External(t)`), or an external atomic formula, `External(t)`, `t` must be an instance of a schema in the coherent set of external schemas (Section Schemas for Externally Defined Terms in [RIF-DTB]) associated with the language of RIF-PRD;
- if `t` is an instance of a schema in the coherent set of external schemas associated with the language then `t` can occur only 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 Operational semantics of condition formulas

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

Informally, a condition formula is evaluated with respect to a state of facts and it is *satisfied*, or *true*, if and only if:

- it is an atomic condition formula and its variables are bound to individuals such that, when these constants are substituted for the variables, either
  - it matches a fact, or
  - it is implied by some background knowledge, or
  - it is an externally defined predicate, and its evaluation yelds *true*, or
- it is a compound condition formula: conjunction, disjunction, negation or existential; and it is evaluated as expected, based on the truth value of its atomic components.

The semantics is specified in terms of matching substitutions in the sections below. The specification makes no assumption regarding how matching substitutions are determined. In particular, it does not require from well-formed condition formulas that they can be evaluated using pattern matching only. However, RIF-PRD requires safeness from well-formed rules, which implies that all the variables in the left-hand side can be bound by pattern matching.

For compatibility with other RIF specifications (in particular, RIF data types and built-ins [RIF-DTB] and RIF RDF and OWL compatibility [RIF-RDF-OWL]), and to make explicit the interoperability with RIF logic dialects (in particular RIF Core [RIF-Core] and RIF-BLD [RIF-BLD]), the semantics of RIF-PRD condition formulas is also specified using model theory, in appendix Model theoretic semantics of RIF-PRD condition formulas.

The two specifications are equivalent and normative.

2.2.1 Matching substitution

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 $\sigma = \{t_i/x_i\}_{i=1..n}$ where $Dom(\sigma) = \{x_i\}_{i=1..n}$ and $\sigma(x_i) = t_i$, $i = 1..n$. ☐

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

Because RIF-PRD covers only externally defined interpreted functions, a ground positional term can always be replaced by the (non-positional) ground term to which it evaluates. As a consequence, a ground RIF-PRD formula can always be restricted, without loss of generality, to contain no positional term; that is, to be such that any ground positional terms have been replaced with the non-positional ground terms to which they evaluate. In the remainder of this document, it will always be assumed that a ground condition formula never contains any positional term. As a consequence, a ground substitution never assigns a ground positional term to the variables in its domain.

If t is a term or a condition formula, and if σ is a ground substitution such that $Var(t) \in Dom(\sigma)$, $\sigma(t)$ denotes the ground term or the ground condition formula obtained by substituting, in t:

- $\sigma(x)$ for all $x \in Var(t)$, and
- the externally defined results of interpreting a function with ground arguments, for all externally defined terms.

Definition (Matching substitution). Let ψ be a RIF-PRD condition formula; let σ be a ground substitution such that $Var(\psi) \subseteq Dom(\sigma)$; and let Φ be a set of ground RIF-PRD atomic formulas.

We say that the ground substitution σ ***matches*** ψ to Φ if and only if one of the following is true:

- ψ is an atomic formula and either
  - $\sigma(\psi) \in \Phi$, or
  - ψ is a frame with multiple slots, $\mathtt{o[s_1\text{->}v_1...s_n\text{->}v_n]}$, $n > 1$, and there is one i, $1 \le i \le n$, such that σ matches the conjunction $\mathtt{And(o[s_i\text{->}v_i]\ o[s_1\text{->}v_1...s_{i-1}\text{->}v_{i-1}\ s_{i+1}\text{->}v_{i+1}...s_n\text{->}v_n]}$ to Φ; or
  - ψ is an equality formula, $\mathtt{t_1 = t_2}$, and either
    - $\sigma(t_1)$ and $\sigma(t_2)$ are the same ground term;
    - or the ground terms $\sigma(t_1)$ and $\sigma(t_2)$ are list terms with the same length $n \ge 0$ and, for all i, $0 \le i \le n\text{-}1$, such that l_{1_i} and l_{2_i} are the ground terms of rank i in $\sigma(t_1)$ and $\sigma(t_2)$, respectively, either l_{1_i} and l_{2_i} are both constants in symbol spaces that are data types and they have the same value, or $\mathtt{l_{1_i} = l_{2_i}} \in \Phi$,
    - or the ground terms $\sigma(t_1)$ and $\sigma(t_2)$ are constants in symbol spaces that are data types and they have the same value; or
  - ψ is a membership formula `o # c`, and there is a ground term c' such that σ matches the conjunction `And(o#c' c'##c)` to Φ, or
  - ψ is an external atomic formula and the external definition maps $\sigma(\psi)$ to **t** (or *true*),
- ψ is `Not(f)` and σ does not match the condition formula `f` to Φ,
- ψ is $\mathtt{And(f_1\ ...\ f_n)}$ and either $\mathtt{n = 0}$ or $\forall\, i$, $1 \le i \le n$, σ matches $\mathtt{f_i}$ to Φ,
- ψ is $\mathtt{Or(f_1\ ...\ f_n)}$ and $\mathtt{n} > 0$ and $\exists\, i$, $1 \le i \le n$, such that σ matches $\mathtt{f_i}$ to Φ, or
- ψ is $\mathtt{Exists\ ?v_1\ ...\ ?v_n\ (f)}$, and there is a substitution σ' that extends σ in such a way that σ' agrees with σ where σ is defined, and $Var(f) \subseteq Dom(\sigma')$; and σ' matches `f` to Φ. ☐

2.2.2 Condition satisfaction

We define, now, what it means for a *state of the fact base* to satisfy a condition formula. The satisfaction of condition formulas in a state of the fact base provides formal underpinning to the operational semantics of rule sets interchanged using RIF-PRD.

Definition (State of the fact base). A ***state of the fact base***, w_Φ, is associated to every set of ground atomic formulas, Φ, that contains no frame with multiple slots and that satisfies all the following conditions:

- for every equality formula $\mathtt{t_1 = t_2} \in \Phi$, if t_1 and t_2 are, both, constants in symbol spaces that are data types, then they have the same value;
- for every equality formula $\mathtt{t_1 = t_2} \in \Phi$, either t_1 is not a constant in a symbol space that is a data type, or t_2 is not a list term;
- for every pair of constants c_1 and c_2, if $\mathtt{c_1 = c_2} \in \Phi$, then $\mathtt{c_2 = c_1} \in \Phi$;
- for every triple of constants c_1, c_2 and c_3, if $\mathtt{c_1 = c_2} \in \Phi$ and $\mathtt{c_2 = c_3} \in \Phi$, then $\mathtt{c_1 = c_3} \in \Phi$;
- for all triple of constants c_1, c_2, c_3, if $\mathtt{c_1\#\#c_2} \in \Phi$ and $\mathtt{c_2\#\#c_3} \in \Phi$, then $\mathtt{c_1\#\#c_3} \in \Phi$.

We say that w_Φ is *represented* by Φ; or, equivalently, by the conjunction of all the ground atomic formulas in Φ. ☐

Each ground atomic formula in Φ represents a single **fact**, and, often, the ground atomic formulas, themselves, are called *facts*, as well. Notice that the restriction that Φ can contain only single slot frames, in the definition of a state of the fact base is not a limitation: given the definition of a matching substitution, a frame with multiple slots is only syntactic shorthand for the semantically equivalent conjunction of single slot frames.

Definition (Condition satisfaction). A RIF-PRD condition formula ψ is ***satisfied*** in a state of the fact base, w, if and only if w is represented by a set of ground atomic formulas Φ, and there is a ground substitution σ that matches ψ to Φ. ☐

Alternative, but equivalent, definitions of a state of the fact base and of the satisfaction of a condition are given in the appendix Model theoretic semantics of RIF-PRD condition formulas: they provide the formal link between the model theory of RIF-PRD condition formulas and the operational semantics of RIF-PRD documents.

3 Actions

This section specifies the syntax and semantics of the RIF-PRD *action* language. The conclusion of a production rule is often called the *action* part, the *then* part, or the *right-hand side*, or *RHS*.

The RIF-PRD action language is used to add, delete and modify facts in the fact base. As a rule interchange format, RIF-PRD does not make any assumption regarding the nature of the data sources 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 represent the facts that the actions affect. In the same way, the semantics of the actions is specified in terms of how their execution affects the evaluation of rule conditions.

3.1 Abstract syntax

The alphabet of the RIF-PRD action language includes symbols to denote:

- the assertion of a fact represented by a positional atom, a frame, or a membership atomic formula,
- the retraction of a fact represented by a positional atom or a frame,
- the retraction of all the facts about the values of a given slot of a given frame object,
- the addition of a new frame object,
- the removal of a frame object and the retraction of all the facts about it, represented by the corresponding frame and class membership atomic formulas,
- the replacement of all the values of an object's attribute by a single, new value,
- the execution of an externally defined action, and
- a sequence of these actions, including the declaration of local variables and a mechanism to bind a local variable to a frame slot value or a new frame object.

3.1.1 Actions

The RIF-PRD action language includes constructs for actions that are atomic, from a transactional point of view, and constructs that represent compounds of atomic actions. Action constructs take constructs from the RIF-PRD condition language as their arguments.

Definition (Atomic action). An ***atomic action*** is a construct that represents an atomic transaction. An atomic action can have several different forms and is defined as follows:

1. *Assert simple fact*: If φ is a positional atom, a single slot frame or a membership 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 simple fact*: If φ is a positional atom or a single slot frame in the RIF-PRD condition language, then `Retract(φ)` is an atomic action. φ is called the *target* of the action.
3. *Retract all slot values*: If `o` and `s` are terms in the RIF-PRD condition language, then `Retract(o s)` is an atomic action. The pair `(o, s)` is called the *target* of the action.
4. *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.
5. *Execute*: if φ is a positional atom in the RIF-PRD condition language, then `Execute(φ)` is an atomic action. φ is called the *target* of the action. ☐

Definition (Compound action). A ***compound action*** is a construct that can be replaced equivalently by a pre-defined, and fixed, sequence of atomic actions. In RIF-PRD, a compound action can have three different forms, defined as follows:

1. *Assert compound fact*: If φ is a frame with multiple slots: $φ = o[s_1\text{->}v_1...s_n\text{->}v_n]$, *n > 1*; then `Assert(φ)` is a compound action, defined by the sequence $Assert(o[s_1\text{->}v_1])$ ... $Assert(o[s_n\text{->}v_n])$. φ is called the *target* of the action.
2. *Retract compound fact*: If φ is a frame with multiple slots: $φ = o[s_1\text{->}v_1...s_n\text{->}v_n]$, *n > 1*; then `Retract(φ)` is a compound action, defined by the sequence $Retract(o[s_1\text{->}v_1])$ ... $Retract(o[s_n\text{->}v_n])$. φ is called the *target* of the action.
3. *Modify fact*: if φ is a frame in the RIF-PRD condition language: $φ = o[s_1\text{->}v_1...s_n\text{->}v_n]$, *n > 0*; then `Modify(φ)` is a compound action, defined by the sequence: $Retract(o\ s_1)$ ... $Retract(o\ s_n)$, followed by `Assert(φ)`. φ is called the *target* of the action. ☐

Definition (Action). A ***action*** is either an atomic action or a compound action. ☐

Definition (Ground action). An action with target `t` is a ***ground action*** if and only if

- `t` is an atomic formula and *Var*(t) = ∅;
- or `t = (o, s)` is a pair of terms and *Var*(o) = *Var*(s) = ∅.

☐

Example 3.1.

- `Assert( ?customer[ex1:voucher->?voucher] )` and `Retract( ?customer[ex1:voucher->?voucher] )` denote two atomic actions with the frame `?customer[ex1:voucher->?voucher]` as their target,
- `Retract( ?customer ex1:voucher )` denotes an atomic action with the pair of terms `(?customer, ex1:voucher)` as its target,
- `Modify(?customer[ex1:voucher->?voucher])` denotes a compound action with the frame `?customer[ex1:voucher->?voucher]` as its target. `Modify(?customer[ex1:voucher->?voucher])` can always be equivalently replaced by the sequence: `Retract( ?customer ex1:voucher )` then `Assert( ?customer[ex1:voucher->?voucher] );`
- `Retract( ?voucher )` denotes an atomic action whose target is the individual bound to the variable `?voucher`,
- `Execute( act:print("Hello, world!") )` denotes an atomic action whose target is the externally defined action `act:print`. ☐

3.1.2 Action blocks

The *action block* is the top level construct to represent the conclusions of RIF-PRD rules. An action block contains a non-empty sequence of 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.

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()`. In that case, the notation for the action variable declaration is: `(?o New());`
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 associated notation is: `(?value o[s->?value]).` ☐

Definition (Action block). If `(v1 p1), ..., (vn pn)`, $n \geq 0$, are action variable declarations, and if `a1, ..., am`, $m \geq 1$, are actions, then `Do((v1 p1) ... (vn pn) a1 ... am)` denotes an **action block**. ☐

Example 3.2. In the following action block, a local variable `?oldValue` is bound to a value of the attribute `value` of the object bound to the variable `?shoppingCart`. The `?oldValue` is then used to compute a new value, and the `Modify` action is used to overwrite the old value with the new value in the fact base:

```
Do( (?oldValue ?shoppingCart[ex1:value->?oldValue])
    Modify( ?shoppingCart[ex1:value->func:numeric-multiply(?oldValue 0.90)] ) )
```

☐

3.1.3 Well-formed action blocks

Not all action blocks are well-formed in RIF-PRD:

- one and only one action variable binding can assign a value to each action variable, and
- the assertion of a membership atomic formula is meaningful only if it is about a frame object that is created in the same action block.

The notion of well-formedness, already defined for condition formulas, is extended to actions, action variable declarations and action blocks.

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

- α is an `Assert` and its target is a well-formed atom, a well-formed frame or a well-formed membership atomic formula,
- α is a `Retract` with one single argument and its target is a well-formed term or a well-formed atom or a well-formed frame atomic formula,
- α is a `Retract` with two arguments: `o` and `s`, and both are well-formed terms,
- α is a `Modify` and its target is a well-formed frame, or
- α is an `Execute` and its content is an instance of the coherent set of external schemas (Section Schemas for Externally Defined Terms in RIF data types and builtins [RIF-DTB]) associated with the RIF-PRD language (section Built-in functions, predicates and actions). ☐

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()`, or
- the action variable binding, *p*, is a well formed frame atomic formula, *p* = `o[a1->t1...an->tn]`, $n \geq 1$, and the action variable, v occurs in the position of a slot value, and nowhere else, that is: v ∈ `{t1 ... tn}` and v ∉ *Var(o)* ∪ *Var(a1)* ∪ ... ∪ *Var(an)* and ∀ `ti`, either *v* = `ti` or *v* ∉ *Var(*`ti`*)*. ☐

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

- if *f* is an action with target *t* and *t* is an atomic formula, then *Var(f) = Var(t)*;
- if *f* is an action with target *t* and *t* is a pair, *(o, s)* of terms, then *Var(f) = Var(o)* ∪ *Var(s)*;
- if *f* is a frame object declaration, `New()`, then *Var(f)* = ∅.

Definition (Well-formed action block). An action block is ***well-formed*** if and only if all of the following are 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 variable binding, that is: if b_1 = `(v1 p1)` and b_2 = `(v2 p2)` are two action variable declarations in the action block with different bindings: $p_1 \neq p_2$, then $v_1 \neq v_2$,
- in addition, the action variable declarations, if any, are partially ordered by the ordering defined as follows: if b_1 = `(v1 p1)` and b_2 = `(v2 p2)` are two action variable declarations in the action block, then $b_1 < b_2$ if and only if $v_1 \in Var(p_2)$,
- all the actions in the action block are well-formed actions, and
- if an action in the action block asserts a membership atomic formula, `Assert(t1 # t2)`, then the object term in the membership atomic formula, `t1`, is an action variable that is declared in the action block and the action variable binding is a frame object declaration. ☐

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 semantics of the atomic actions in a RIF-PRD document.

The effect 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 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.

The semantics of a compound action follows directly from the semantics of the atomic actions that compose it.

Individual states of the fact base are represented by sets of ground atomic formulas (Section Satisfaction of a condition). In the following, the operational semantics of RIF-PRD actions, rules, and rule sets is specified by describing the changes they induce in the fact base.

Definition (RIF-PRD transition relation). The semantics of RIF-PRD atomic actions is specified by the **transition relation** $\rightarrow_{\textbf{RIF-PRD}} \subseteq W \times L \times W$. $(w, \alpha, w') \in \rightarrow_{\text{RIF-PRD}}$ if and only if $w \in W$, $w' \in W$, α is a ground atomic action, and one of the following is true, where *Φ* is a set of ground atomic formulas that represents *w* and *Φ*' is a set of ground atomic formulas that represent *w*':

1. α is `Assert(φ)`, where φ is a ground atomic formula, and $\Phi' = \Phi \cup \{\varphi\}$;
2. α is `Retract(φ)`, where φ is a ground atomic formula, and $\Phi' = \Phi \setminus \{\varphi\}$;
3. α is `Retract(o s)`, where `o` and `s` are constants, and *Φ' = (Φ \ {o[s->v] | for all the values of v});*
4. α is `Retract(o)`, where `o` is a constant, and *Φ' = Φ \ {o[s->v] | for all the values of terms s and v} - {o#c | for all the values of term c}*;
5. α is `Execute(φ)`, where φ is a ground atomic builtin action, and $\Phi' = \Phi$. ☐

Rule 1 says that all the atomic condition formulas that were satisfied before an assertion will be satisfied after, and that, in addition, the atomic condition formulas that are satisfied by the asserted ground formula will be satisfied after the assertion. No other atomic condition formula will be satisfied after the execution of the action.

Rule 2 says that all the atomic condition formulas that were satisfied before a retraction will be satisfied after, except if they are satisfied only by the retracted fact. No other atomic condition formula will be satisfied after the execution of the action.

Rule 3 says that all the condition formulas that were satisfied before the retraction of all the values of a given slot of a given object will be satisfied after, except if they are satisfied only by one of the frame formulas about the object and the slot that are the target of the action, or a conjunction of such formulas. No other condition formula will be satisfied after the execution of the action.

Rule 4 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. No other condition formula will be satisfied after the execution of the action.

Rule 5 says that all the condition formulas that were satisfied before the execution of an action builtin will be satisfied after. No other condition formula will be satisfied after the execution of the action.

Example 3.3. Assume an initial state of the fact base that is represented by the following set, w_0, of ground atomic formulas, where `_c1`, `_v1` and `_s1` denote individuals and where `ex1:Customer`, `ex1:Voucher` and `ex1:ShoppingCart` represent classes:

Initial state:

- w_0 = *{_c1#ex1:Customer _v1#ex1:Voucher _s1#ex1:ShoppingCart _c1[ex1:voucher->_v1] _c1[ex1:shoppingCart->_s1] _v1[ex1:value->5] _s1[ex1:value->500]}*

1. `Assert( _c1[ex1:status->"New"] )` denotes an atomic action that adds to the fact base, a fact that is represented by the ground atomic formula: `_c1[ex1:status->"New"]`. After the action is executed, the new state of the fact base is represented by
   - w_1 = *{_c1#ex1:Customer _v1#ex1:Voucher _s1#ex1:ShoppingCart _c1[ex1:voucher->_v1] _c1[ex1:shoppingCart->_s1] _v1[ex1:value->5] _s1[ex1:value->500] _c1[ex1:status->"New"]}*
2. `Retract( _c1[ex1:voucher->_v1] )` denotes an atomic action that removes from the fact base, the fact that is represented by the ground atomic formula `_c1[ex1:voucher->_v1]`. After the action, the new state of the fact base is represenetd by:
   - w_2 = *{_c1#ex1:Customer _v1#ex1:Voucher _s1#ex1:ShoppingCart _c1[ex1:shoppingCart->_s1] _v1[ex1:value->5] _s1[ex1:value->500] _c1[ex1:status->"New"]}*
3. `Retract( _v1 )` denotes an atomic action that removes the individual denoted by the constant `_v1` from the fact base. All the class membership and the object-attribute-value facts where `_v1` is the object are removed. After the action, the new state of the fact base is represenetd by:
   - w_3 = *{_c1#ex1:Customer _s1#ex1:ShoppingCart _c1[ex1:shoppingCart->_s1] _s1[ex1:value->500] _c1[ex1:status->"New"]}*
4. `Retract( _s1 ex1:value )` denotes an atomic action that removes all the object-attribute-value facts that assign a `ex1:value` to the `ex1:ShoppingCart _s1`. After the action, the new state of the fact base is represented by
   - w_4 = *{_c1#ex1:Customer _s1#ex1:ShoppingCart _c1[ex1:shoppingCart->_s1] _c1[ex1:status->"New"]}*
5. `Assert( _s1[ex1:value->450] ) adds in the fact base_the single fact that is represented by the ground frame: <tt>_s1[ex1:value->450]`. After the action, the new state of the fact base is represented by:
   - w_5 = *{_c1#ex1:Customer _s1#ex1:ShoppingCart _c1[ex1:shoppingCart->_s1] _s1[ex1:value->450] _c1[ex1:status->"New"]}*
6. `Execute(` act:print`(func:concat("New customer: " _c1)) )` denotes an action that does not impact the state of the fact base, but that prints a string to an output stream. After the action, the new state of the fact base is represented by:
   - w_6 = w_5 = *{_c1#ex1:Customer _s1#ex1:ShoppingCart _c1[ex1:shoppingCart->_s1] _s1[ex1:value->450] _c1[ex1:status->"New"]}*

Notice that steps 4 and 5 can be equivalently replaced by the single compound action:

- `Modify( _s1[ex1:value->450] )`, which denotes an action that replaces all the object-attribute-value facts that assign a `ex1:value` to the `ex1:ShoppingCart _s1` by the single fact that is represented by the ground frame: `_s1[ex1:value->450]`.

☐

4 Production rules and rule sets

This section specifies the syntax and semantics of RIF-PRD rules and rule sets.

4.1 Abstract syntax

The alphabet of the RIF-PRD rule language includes the alphabets of the RIF-PRD condition language and the RIF-PRD action language and adds symbols for:

- combining a condition and an action block into a rule,
- declaring (some) variables that are free in a rule *R*, specifying their bindings, and combining them with *R* into a new rule *R'* (with fewer free variables),
- grouping rules and associating specific operational semantics to groups of rules.

4.1.1 Rules

Definition (Rule). A ***rule*** can be one of:

- 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 rule,
- a *rule with variable declaration*: if $?v_1 \ldots ?v_n$, $n \geq 1$, are variables; $p_1 \ldots p_m$, $m \geq 1$, are condition formulas (called *patterns*), and `rule` is a rule, then `Forall` $?v_1 \ldots ?v_n$ `such that` $(p_1 \ldots p_m)$ `(rule)` is a rule. ☐

Example 4.1. The *Gold rule*, from the running example: *A "Silver" customer with a shopping cart worth at least $2,000 is awarded the "Gold" status*, can be represented using the following rule with variable declaration:

```
Forall ?customer such that And( ?customer # ex1:Customer
                                ?customer[ex1:status->"Silver"] )
  (Forall ?shoppingCart such that And( ?shoppingCart # ex1:ShoppingCart
                                      ?customer[ex1:shoppingCart->?shoppingCart] )
     (If Exists ?value (And( ?shoppingCart[ex1:value->?value]
                             pred:numeric-greater-than-or-equal(?value 2000))
      Then Do( Modify( ?customer[ex1:status->"Gold"] ) ) )
```

☐

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

4.1.2 Groups

As was already mentioned in the 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*, needs to be interchanged along with the rules. In RIF-PRD, the *group* construct is used to group sets of rules and to associate them with a conflict resolution strategy. Many production rule systems

W3C Proposed Edited Recommendation

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 with the interchanged rules.

Definition (Group). A ***group*** consists of a, possibly empty, set of rules and groups, associated with a conflict resolution strategy and, a priority. If `strategy` is an IRI that identifies 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 represents 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), $n \ge 0$.

If a conflict resolution strategy is not explicitly attached to a group, the strategy defaults to `rif:forwardChaining` (specified below, in section Conflict resolution). ☐

4.1.3 Safeness

The definitions in this section are unchanged from the definitions in the section Safeness in [RIF-Core], except for the definition of *RIF-PRD rule safeness*, that is extended from the definition of *RIF-Core rule safeness*. The definitions are reproduced for the reader's convenience.

Intuitively, safeness of rules guarantees that all the variables in a rule can be bound, using pattern matching only, before they are used, in a test or in an action.

To define safeness, we need to define, first, the notion of *binding patterns* for externally defined functions and predicates, as well as under what conditions variables are considered *bound*.

Definition (Binding pattern). (from [RIF-Core]) ***Binding patterns*** for externally defined functions and predicates are lists of the form (p_1, ..., p_n), such that p_i=b or p_i=u, for $1 \le i \le n$: b stands for a "bound" and u stands for an "unbound" argument. ☐

Each external function or predicate has an associated list of ***valid binding patterns***. We define here the binding patterns valid for the functions and predicates defined in [RIF-DTB].

Every function or predicate `f` defined in [RIF-DTB] has a valid binding pattern for each of its schemas with only the symbol `b` such that its length is the number of arguments in the schema. In addition,

- the external predicate `pred:iri-string` has the valid binding patterns (b, u) and (u, b) and
- the external predicate `pred:list-contains` has the valid binding pattern (b, u).

The functions and predicates defined in [RIF-DTB] have no other valid binding patterns.

To keep the definitions concise and intuitive, boundedness and safeness are defined, in [RIF-Core], for condition formulas in disjunctive normal form, that can be existentially quantified themselves, but that contain, otherwise, no existential sub-formula. The definitions apply to any valid RIF-Core condition formula, because they can always, in principle, be put in that form, by applying the following syntactic transforms, in sequence:

1. if *f* contains existential sub-formulas, all the quantified variables are renamed, if necessary, and given a name that is unique in *f*, and the scope of the quantifiers is extended to *f*. Assume, for instance, that *f* has an existential sub-formula, *sf* = *Exists* $v_1...v_n$ *(sf')*, $n \ge 1$, such that the names $v_1...v_n$ do not occur in *f* outside of *sf*. After the transform, *f* becomes *Exists* $v_1...v_n$ *(f')*, where *f'* is *f* with *sf* replaced by *sf'* . The transform is applied iteratively to all the existential sub-formulas in *f*;
2. the (possibly existentially quantified) resulting formula is rewritten in disjunctive normal form ([Mendelson97], p. 30).

In RIF-PRD, the definitions apply to conditions formulas in the same form as in [RIF-Core], with the exception that, in the disjunctive normal form, negated sub-formulas can be atomic formulas or existential formulas: in the latter case, the existentially quantified formula must be, itself, in disjunctive normal form, and contain no further existential sub-formulas. The definitions apply to any valid RIF-PRD condition formula, because they can always, in principle, be put in that form, by applying the above syntactic transform, modified as follows to take negation into account:

- if the condition formula under consideration, *f*, contains negative sub-formulas, existential formulas that occur inside a negated formula are handled as if they were atomic formulas, with respect to the two processing steps. Extending the scope of an existential quantifier beyond a negation would require its transformation into an universal quantifier, and universal formulas are not part of RIF-PRD condition language;
- in addition, the two pre-processing steps are applied, separately, to these existentially quantified formulas, to be able to determine the status of the existentially quantified variables with respect to boundedness.

Definition (Boundedness). (from [RIF-Core]) An external term External(f(t_1,...,t_n)) is ***bound*** in a condition formula, if and only if `f` has a valid binding pattern (p_1, ..., p_n) and, for all *j*, $1 \le j \le n$, such that p_j=b, t_j is bound in the formula.

A variable, *v*, is ***bound*** in an atomic formula, *a*, if and only if

- *a* is neither an equality nor an external predicate, and *v* occurs as an argument in *a*;
- or *v* is bound in the conjunctive formula *f* = *And(a)*.

A variable, *v*, is ***bound*** in a conjunction formula, *f* = *And*($c_1...c_n$), $n \ge 1$, if and only if, either

- *v* is bound in at least one of the conjuncts;
- or *v* occurs as the *j-th* argument in a conjunct, c_i, that is an externally defined predicate, and the *j-th* position in a binding pattern that is associated with c_i is u, and all the arguments that occur, in c_i, in positions with value *b* in the same binding pattern are bound in *f'* = *And*($c_1...c_{i-1}$ $c_{i+1}...c_n$);
- or *v* occurs in a conjunct, c_i, that is an equality formula, and *v* occurs as the term on one side of the equality, and the term on the other side of the equality is bound in *f'* = *And*($c_1...c_{i-1}$ $c_{i+1}...c_n$).

A variable, *v*, is ***bound*** in a disjunction formula, if and only if *v* is bound in every disjunct where it occurs;

A variable, *v*, is ***bound*** in an existential formula, Exists v_1,...,v_n (f'), $n \ge 1$, if and only if *v* is bound in *f'*. ☐

Notice that the variables, v_1,...,v_n, that are existentially quantified in an existential formula *f* = *Exists* $v_1,...,v_n$ *(f')*, are bound in any formula, *F*, that contains *f* as a sub-formula, if and only if they are bound in *f*, since they do not exist outside of *f*.

Definition (Variable safeness). (from [RIF-Core]) A variable, *v*, is ***safe*** in a condition formula, *f*, if and only if

- *f* is an atomic formula and *f* is not an equality formula in which both terms are variables and *v* occurs in *f*;
- or *f* is a conjunction, , *f* = *And*($c_1...c_n$), $n \ge 1$, and *v* is safe in at least one conjunct in *f*, or *v* occurs in a conjunct, c_i, that is an equality formula in which both terms are variables, and *v* occurs as the term on one side of the equality, and the variable on the other side of the equality is safe in *f'* = *And*($c_1...c_{i-1}$ $c_{i+1}...c_n$);
- or *f* is a disjunction, and *v* is safe in every disjunct;
- or *f* is an existential formula, *f* = *Exists* $v_1,...,v_n$ *(f')*, $n \ge 1$, and *v* is safe in *f'* . ☐

Notice that the two definitions, above, are not extended for negation and, followingly, that an universally quantified (rule) variable is never bound or safe in a condition formula as a consequence of occurring in a negative formula.

The definition of *rule safeness* is replaced by the following one, that extends the one for RIF-Core rules.

Definition (RIF-PRD rule safeness). A RIF-PRD rule, *r*, is ***safe*** if and only if

- *r* is an unconditional action block, and $Var(r) = \emptyset$;
- or *r* is a conditional action block, `If C Then A`, and all the variables in *Var(A)* are safe in *C*, and all the variables in *Var(r)* are bound in *C*;
- or *r* is a rule with variable declaration, ∀ $v_1...v_n$ such that $p_1...p_m$ (r'), $n \geq 1$, $m \geq 0$, and either
  - *r'* is an unconditional action block, `A`, and the conditional action block `If And(`$p_1...p_m$`) Then A` is safe;
  - or *r'* is a conditional action block, `If C Then A`, and the conditional action block `If And(C `$p_1...p_m$`) Then A` is safe;
  - or *r'* is a rule with variable declaration, ∀ $v'_1...v'_{n'}$ such that $p'_1...p'_{m'}$ (r"), $n' \geq 1$, $m' \geq 0$, and the rule with variable declaration ∀ $v_1...v_n$ $v'_1...v'_{n'}$ such that $p_1...p_m$ $p'_1...p'_{m'}$ (r"), is safe. ☐

Definition (Group safeness). (from [RIF-Core]) A group, `Group (`$s_1...s_n$`)`, $n \geq 0$, is ***safe*** if and only if

- it is empty, that is, $n = 0$;
- or s_1 and ... and s_n are safe. ☐

4.1.4 Well-formed rules and groups

If *f* is a rule, *Var(f)* is the set of the free variables in *f*.

Definition (Well-formed rule). A rule, `r`, is a ***well-formed rule*** if and only if either

- `r` is an unconditional well-formed action block, `a`,
- or `r` is a conditional action block where the condition formula, `c`, is a well-formed condition formula, and the action block, `a`, is a well-formed action block,
- or `r` is a quantified rule (with or without patterns), `Forall V [such that P](r')`, and
  - each of the patterns, $p_i \in P = \{p_1, ..., p_n\}$, $n \geq 0$, is a well-formed condition formula,
  - and the quantified rule, `r'`, is a well-formed rule. ☐

Definition (Well-formed group). A group is ***well-formed group*** if and only if it is safe and it contains only well-formed groups, $g_1...g_n$, $n \geq 0$, and well-formed rules, $r_1...r_m$, $m \geq 0$, such that $Var(r_i) = \emptyset$ for all i, $0 \leq i \leq m$. ☐

The variables that are universally quantified in a rule are sometimes called *rule variables* in the remainder of this document, to distinguish them from the action variables and from the existentially quantified variables. The function *CVar*, that maps a rule to the set of its rule variables is defined as follows:

- if *r* is a conditional or unconditional action block, $CVar(r) = \emptyset$
- if *r* is a rule with variable declaration, `Forall ?`v_1`...?`v_n` (r')`, $CVar(r) = CVar(r') \cup \{?v_1...?v_n\}$.

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

4.2 Operational semantics of rules and rule sets

4.2.1 Motivation and example

As mentioned in the Overview, the description of a production rule system as a transition system is used to specify the semantics of production rules and rule sets 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 4.2. The Rif Shop, Inc. is a rif-raf retail chain, with brick and mortar shops all over the world and virtual storefronts in many on-line shops. The Rif Shop, Inc. maintains its customer fidelity management policies in the form of production rule sets. The customer management department uses RIF-PRD to publish rule sets to all the shops and licensees so that everyone uses the latest version of the rules, even though several different rule engines are in use (in fact, some of the smallest shops actually run the rules by hand).

Here is a small rule set that governs discounts and customer status updates at checkout time (to keep the example short, this is a subset of the rules described in the running example):

```
(* ex1:CheckoutRuleset *)
Group rif:forwardChaining (

  (* ex1:GoldRule *)
  Group 10 (
    Forall ?customer such that (And( ?customer # ex1:Customer
                                     ?customer[ex1:status->"Silver"] ) )
      (Forall ?shoppingCart such that (?customer[ex1:shoppingCart->?shoppingCart])
         (If Exists ?value (And( ?shoppingCart[ex1:value->?value]
                                 pred:numeric-greater-than-or-equal(?value 2000))
          Then Do( Modify( ?customer[ex1:status->"Gold"] ) ) ) ) )

  (* ex1:DiscountRule *)
  Group (
    Forall ?customer such that (And( ?customer # ex1:Customer ) )
      (If Or (?customer[ex1:status->"Silver"]
              ?customer[ex1:status->"Gold"] )
       Then Do( (?s ?customer[ex1:shoppingCart->?s])
                (?v ?s[ex1:value->?v])
                Modify( ?s[ex1:value->func:numeric-multiply(?v 0.95)] ) ) ) )
```

To see how the rule set works, consider the case of a shop where the checkout processing of customer John is about to start. The initial state of the fact base can be represented as follows:

w_0 = *{_john#ex1:Customer _john[ex1:status->"Silver"] _s1#ex1:ShoppingCart _john[ex1:shoppingCart->_s1] _s1[ex1:value->2000]}*

When instantiated against w_0, the first pattern in the "Gold rule", `And( ?customer#ex1:Customer ?customer[ex1:status->"Silver"] )`, yields the single matching substitution: *{(_john/?customer)}*. The second pattern in the same rule also yields a single matching substitution: *{(_john/?customer)(_s1/?shoppingCart)}*, for which the existential condition is satisfied.

Likewise, the instantiation of the "Discount rule" yields a single matching substitution that satisfies the condition: *{(_john/?customer)}*. The conflict set is: {ex1:GoldRule/{(_john/?customer)(_s1/?shoppingCart)}, ex1:DiscountRule/{(_john/?customer)}}

The instance *ex1:GoldRule/{(_john/?customer)(_s1/?shoppingCart)}* is selected because of its higher priority. The ground compound action: `Modify(_john[ex1:status->"Gold"])`, is executed, resulting in a new state of the fact base, represented as follows:

w_1 = *{_john#ex1:Customer _john[ex1:status->"Gold"] _s1#ex1:ShoppingCart _john[ex1:shoppingCart->_s1] _s1[ex1:value->2000]}*

In the next cycle, there is no substitution for the rule variable `?customer` that matches the pattern to the state of the fact base, and the only matching rule instance is: ex1:DiscountRule/{(_john/?customer)}, which is selected for execution. The action variables `?s` and `?v` are bound, based on the state of the fact base, to `_s1` and `200`, respectively, and the ground compound action, `Modify(_s1[ex1:value->1900])`, is executed, resulting in a new state of the fact base:

w_2 = *{_john#ex1:Customer _john[ex1:status->"Gold"] _s1#ex1:ShoppingCart _john[ex1:shoppingCart->_s1] _s1[ex1:value->1900]}*

In w_2, the only matching rule instance is, again: ex1:DiscountRule/{(_john/?customer)}. However, that same instance has already been selected and the corresponding action has been executed. Nothing has changed in the state of the fact base that would justify that the rule instance be selected gain. The principle of refraction applies, and the rule instance is removed from consideration.

This leaves the conflict set empty, and the system, having detected a final state, stops.

The result of the execution of the system is w_2. ☐

4.2.2 Rules normalization

A rule, *R*, whose condition, rewritten in disjunctive normal form as described in section Safeness, consists of more than one disjunct, is equivalent, logically as well as operationally, to a set (or conjunction) of rules that have, all, the same conclusion as *R*, and each rule has one of the disjuncts as its condition: the rule *R: If* C_1 *Or ... Or* C_n *Then A* is equivalent to the set of rules $\{r_{i=0..n} | r_i$*: If* C_i *Then A*$\}$.

Without loss of generality, and to keep the specification as simple and intuitive as possible, the operational semantics of production rules and rule sets is specified, in the following sections, for rules and rule sets that have been normalized as follows:

1. All the rules are rewritten in disjunctive normal form as described in section Safeness;
2. Each rule is replaced by a group of rules
   - with the same priority as the rule it replaces,
   - that contains as many rules as the condition of the original rule in disjunctive normal form contains disjuncts,
   - where the condition, in each rule in the group is one of the disjunct in the condition of the original rule,
   - and where all the rules in the group have a different condition and the same action part as the original rule.

In the same way, without loss of generality, and to keep the specification as simple and intuitive as possible, the operational semantics of production rules and rule sets is specified, in the following sections, for rules and rule sets where all the compound actions have been replaced by the equivalent sequences of atomic actions.

4.2.3 Definitions and notational conventions

Formally, a production rule system is defined as a labeled terminal transition system (e.g. PLO04), for the purpose of specifying the 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;
- → ⊆ C × L × C is the transition relation, that is: *(c, a, c')* ∈ → iff there is a transition labeled *a* from the state *c* to the state *c'* . In the case of a production rule system: in the state *c* of the fact base, the execution of action *a* causes a transition to state the *c'* of the fact base;
- **T** ⊆ C is the set of final states, that is, the set of all the states *c* from which there are no transitions: T = {*c* ∈ C | ∀ *a* ∈ L, ∀ *c'* ∈ C, (*c, a, c'*) ∉ →}. ☐

For many purposes, a representation of the states of the fact base is an appropriate representation of the states of 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 of the rule instances that where selected and that triggered the transition in those states.

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* ∈ *R*, and a ground substitution, σ, such that *CVar(r)* ⊆ *Dom(σ)*, where *CVar(r)* denotes the set of the rule variables in *r*, the result, *ri* = *σ(r)*, of the substitution of the constant *σ(?x)* for each variable ?x ∈ *CVar(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 *CVar(r)*.

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 set of ground atomic formulas 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 instance, *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 set of ground atomic condition formulas that represents *w*;
- *rule(ri)* is a rule with variable declaration: `Forall` $?v_1...?v_n$ $(p_1...p_m)$ `(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 set of ground atomic condition formulas that represents *w*, and the rule instance *ri*' matches *w*, where *rule(ri')* = `r'` and *substitution(ri')* = *substitution(ri)*. ☐

Definition (Action instance). Given a state of the fact base, *w* ∈ *W*, 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 $(v_i\ p_i)$, $0 \leq i \leq n$, 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* matches *w*, the substitution *σ = substitution(ri)* is extended to the action variables $v_1...v_n$, $n \geq 0$, in the following way:

- if the binding, p_i, associated to v_i, in the action variable declaration, is the declaration of a new frame object: (v_i `New()`), then $\sigma(v_i) = c_{new}$, where c_{new} is a constant of type `rif:IRI` that does not occur in any of the ground atomic formulas in *w*;
- if v_i is assigned the value of a frame's slot by the action variable declaration: (v_i `o[s->`v_i`]`), then $\sigma(v_i)$ is a ground term such that the substitution σ matches the frame formula `o[s->`v_i`]` to *w*.

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*.

Notice that RIF-PRD does not specify semantics for the case where there is no matching substitution for the binding frame formula `o[s->vi]` in an action variable declaration `(vi o[s->vi])`. Indeed, although the rule might be valid from an interchange viewpoint, applying it in a context where object `o` has no value for attribute `s` is applying it outside the domain where it is meaningful, and the specification of the context where an otherwise valid RIF-PRD rule is validly applicable is out of the scope of RIF-PRD.

The components of the states of a production rule system seen as a transition system can now be defined more precisely. To avoid 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***) is either a ***system cycle state*** or a ***system transitional state***. Every production rule system state, *s*, cycle or transitional, is characterized by

- a state of the fact base, *facts(s)*;
- if *s* is not the current state, an ordered set of rule instances, *picked(s)*, defined as follows:
  - if *s* is a system cycle state, *picked(s)* is the ordered set of rule instances picked by the conflict resolution strategy, among the set of all the rule instances that matched *facts(s)*;
  - if *s* is a system transitional state, *picked(s)* is the empty set;
- if *s* is not the initial state, a previous system state, *previous(s)*, defined as follows: given a system cycle state, s_c, and given the sequence of system transitional states, $s_1,...,s_n$, $n \geq 0$, such that the execution of the first ground atomic action in *action(picked(s_c))* transitioned the system from s_c to s_1 and ... and the *n-th* ground atomic action in *action(picked(s_c))* transitioned the system from s_{n-1} to s_n, then *previous(s)* = s_n if and only if the *(n+1)-th* ground atomic action in *action(picked(s_c))* transitioned the system from s_n to *s*. ☐

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

Definition (Conflict set). Given a rule set, $RS \subseteq R$, and a system state, *s*, the ***conflict set*** determined by *RS* in *s* is 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*. ☐

The rule instances that are in the conflict set are, sometimes, said to be *fireable*.

In each non-final cycle 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 the resulting sequence of ground atomic actions is executed. This is sometimes called: *firing* the selected instances.

4.2.4 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, called the *system cycle 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 $\rightarrow_{PRS}$ states that a production rule system can transition from one system cycle state to another only if the state of facts in the latter system cycle 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 cycle state are determined only by how rule instances are *picked* for execution, from the conflict set, by the conflict resolution strategy.

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

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

Using $\rightarrow^{*}_{PRS}$ to denote the transitive closure of the transition relation $\rightarrow_{PRS}$, there are zero, one or more final states of the system, $s' \in T$, such that:

$$Eval(RS, LS, H, w) \rightarrow^{*}_{PRS} s'.$$

The execution of a rule set, *RS*, in a state, *w*, of a fact base, may result in zero, one or more final state of the fact base, *w' = facts(s')*, depending on the conflict resolution strategy and the set of final system states.

Therefore, the behavior of a RIF-PRD production rule system also 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, and
2. how the set *T* of final system states is precisely defined.

4.2.5 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*), because it accounts for a common conflict resolution strategy used in most forward-chaining production rule systems. That conflict resolution strategy selects a single rule instance for execution.

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 responsibility of the producers and consumers of such document to agree on the intended conflict resolution strategies that are denoted by such non-standard keywords. Future or non-standard conflict resolution strategies may select an ordered set of rule instances for execution, instead of a single one: the functions *picked* and *actions*, in the previous section, have been defined to take this case into account.

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 (cycle and transitional);
- *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 the number of consecutive system states, cycle and transitional, in which they have been in the conflict set, and only the most recently fireable ones are eligible for firing. Note that the recency rule, used alone, results in depth-first processing.

Many existing production rule systems implement also some kind of *fire the most specific rule first* strategy, in combination with the above. However, whereas they agree on the definition of refraction and the priority or recency ordering, existing production rule systems vary widely on the precise definition of the specificity ordering. As a consequence, rule instance specificity was not included in the basic conflict resolution strategy that RIF-PRD specifies normatively.

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 further consideration;
2. The remaining rule instances are ordered by decreasing priority, and only the rule instances with the highest priority are kept for further consideration;
3. The remaining rule instances are ordered by decreasing recency, and only the most recent rule instances are kept for further consideration;
4. Any remaining tie is broken is some way, 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`, for any given system cycle state, *s*, the list denoted by *picked(s)* contains a single rule instance. By definition, if *s* is a system transitional state, *picked(s)* is the empty set.

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 a 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))*.

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 explicitly associated, or zero.

Example 4.3. Consider the following RIF-PRD document:

```
Document (
  Prefix( ex2 <http://example.com/2009/prd3#> )
  (* ex2:ExampleRuleSet *)
  Group (
    (* ex2:Rule_1 *) Forall ...
    (* ex2:HighPriorityRules *)
    Group 10 (
      (* ex2:Rule_2 *) Forall ...
      (* ex2:Rule_3 *)
      Group 9 (Forall ... ) )
    (* ex2:NoPriorityRules *)
    Group (
      (* ex2:Rule_4 *) Forall ...
      (* ex2:Rule_5 *) Forall ... )
 )
```

No conflict resolution strategy is identified explicitly, so the default strategy `rif:forwardChaining` is used.

Because the *ex2:ExampleRuleSet* group does not specify a priority, the default priority *0* is used. Rule 1, not being in any other group, inherits its priority, *0*, from the top-level group.

Rule 2 inherits its priority, *10*, from the enclosing group, identified as *ex2:HighPriorityRules*. Rule 3 specifies its own, lower, priority: *9*.

Since neither Rule 4 nor Rule 5 specify a priority, they inherit their priority from the enclosing group *ex2:NoPriorityRules*, which does not specify one either, and, thus, they inherit *0* from the top-level group, *ex2:ExampleRuleSet*. ☐

Given a set of rule instances, *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}*. RIF-PRD does not specify the tie-break rule more precisely: how a single instance is selected from the remaining set is implementation specific.

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; 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.

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 of the following sequence of steps:

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

Example 4.4. Consider, from example 4.2, the conflict set that the rule set `ex1:CheckoutRuleset` determines in the system state, s_2, that corresponds to the state w_2 = *facts*(s_2) of the fact base, and use it to initialize the set of rule instance considered for firing, *picked*(s_2):

conflictSet(`ex1:CheckoutRuleset`*,* s_2*) = { ex1:DiscountRule/{(_john/?customer)} } = picked(*s_2*)*

W3C Proposed Edited Recommendation

The single rule instance in the conflict set, *ri = ex1:DiscountRule/{(_john/?customer)}*, did already belong to the conflict sets in the two previous states, *conflictSet(ex1:CheckoutRuleset, s_1)* and *conflictSet(ex1:CheckoutRuleset, s_0)*; so that its recency in s_2 is: *recency(ri, s_2) = 3*.

On the other hand, that rule instance was fired in system state s_1: *picked(s_1) = (ex1:DiscountRule/{(_john/?customer)})*; so that, in s_2, it has been last fired one cycle before: *lastPicked(ri, s_2) = 1*.

Therefore, *lastPicked(ri, s_2) < recency(ri, s_2)*, and *ri* is removed from *picked(s_2)* by refraction, leaving *picked(s_2)* empty. ☐

4.2.6 Halting test

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 Document and imports

This section specifies the structure of a RIF-PRD document and its semantics when it includes import directives.

5.1 Abstract syntax

In addition to the language of conditions, actions, and rules, RIF-PRD provides a construct to denote the import of a RIF or non-RIF document. Import enables the modular interchange of RIF documents, and the interchange of combinations of multiple RIF and non-RIF documents.

5.1.1 Import directive

Definition (Import directive). An ***import directive*** consists of:

- an IRI, the *locator*, that identifies and locates the document to be imported, and
- an optional second IRI that identifies the *profile* of the import. ☐

RIF-PRD gives meaning to one-argument import directives only. Such directives can be used to import other RIF-PRD and RIF-Core documents. Two-argument import directives are provided to enable import of other types of documents, and their semantics is covered by other specifications. For example, the syntax and semantics of the import of RDF and OWL documents, and their combination with a RIF document, is specified in [RIF-RDF-OWL].

5.1.2 RIF-PRD document

Definition (RIF-PRD document). A RIF-PRD ***document*** consists of zero or more import directives, and zero or one group. ☐

Definition (Imported document). A document is said to be ***directly imported*** by a RIF document, *D*, if and only if it is identified by the locator IRI in an import directive in *D*. A document is said to be ***imported*** by a RIF document, *D*, if it is directly imported by *D*, or if it is imported, directly or not, by a RIF document that is directly imported by *D*. ☐

Definition (Document safeness). (from [RIF-Core]) A document is ***safe*** if and only if it

- it contains a safe group, or no group at all,
- and all the documents that it imports are safe. ☐

5.1.3 Well-formed documents

Definition (Conflict resolution strategy associated with a document). A ***conflict resolution strategy is associated with a RIF-PRD document***, *D*, if and only if

- it is explicitly or implicitly attached to the top-level group in *D*, or
- it is explicitly or implicitly attached to the top-level group in a RIF-PRD document that is imported by *D*. ☐

Definition (Well-formed RIF-PRD document). A RIF-PRD document, *D*, is ***well-formed*** if and only if it satisfies all the following conditions:

- the locator IRI provided by all the import directives in *D*, if any, identify well-formed RIF-PRD documents,
- *D* contains a well-formed group or no group at all,
- *D* has only one associated conflict resolution strategy (that is, all the conflict resolution strategies that can be associated with it are the same), and
- every non-`rif:local` constant that occurs in *D* or in one of the documents imported by *D*, occurs in the same context in *D* and in all the documents imported by *D*. ☐

The last condition in the above definition makes the intent behind the `rif:local` constants clear: occurrences of such constants in different documents can be interpreted differently even if they have the same name. Therefore, each document can choose the names for the `rif:local` constants freely and without regard to the names of such constants used in the imported documents.

5.2 Operational semantics of RIF-PRD documents

The semantics of a well-formed RIF-PRD document that contains no import directive is the semantics of the rule set that is represented by the top-level group in the document, evaluated with the conflict resolution strategy that is associated to the document, and the default halting test, as specified above, in section Halting test.

The semantics of a well-formed RIF-PRD document, *D*, that imports the well-formed RIF-PRD documents D_1, ..., D_n, $n \geq 1$, is the semantics of the rule set that is the union of the rule sets represented by the top-level groups in *D* and the imported documents, with the `rif:local` constants renamed to ensure that the same symbol does not occur in two different component rule sets, and evaluated with the conflict resolution strategy that is associated to the document, and the default halting test.

6 Built-in functions, predicates and actions

In addition to externally specified functions and predicates, and in particular, in addition to the functions and predicates built-ins defined in [RIF-DTB], RIF-PRD supports externally specified actions, and defines action built-ins.

The syntax and semantics of action built-ins are specified like for the other buit-ins, as described in the section Syntax and Semantics of Built-ins in [RIF-DTB]. However, their formal semantics is trivial: action built-ins behave like predicates that are always true, since action built-ins, in RIF-PRD, MUST NOT affect the semantics of the rules.

Although they must not affect the semantics of the rules, action built-ins may have other side effects.

RIF action built-ins are defined in the namespace: http://www.w3.org/2007/rif-builtin-action#. In this document, we will use the prefix: `act:` to denote the RIF action built-ins namespace.

6.1 Built-in actions

6.1.1 act:print

- *Schema*:

  `(?arg; act:print(?arg))`

- *Domains*:

  The value space of the single argument is `xs:string`.

- *Mapping:*

  When s belongs to its domain, $\boldsymbol{I}_{\text{truth}} \circ \boldsymbol{I}_{\text{External}}$(?arg; act:print(?arg))(s) = **t**.

  If an argument value is outside of its domain, the truth value of the function is left unspecified.

- *Side effects:*

  The value of the argument MUST be printed to an output stream, to be determined by the user implementation.

7 Conformance and interoperability

7.1 Semantics-preserving transformations

RIF-PRD conformance is described partially in terms of semantics-preserving transformations.

The intuitive idea is that, for any initial state of facts, the conformant consumer of a conformant RIF-PRD document must reach at least one of the final state of facts intended by the conformant producer of the document, and that it must never reach any final state of facts that was not intended by the producer. That is:

- a conformant RIF-PRD producer, *P*, must translate any rule set from its own rule language, L_P, into RIF-PRD, in such a way that, for any possible initial state of the fact base, the RIF-PRD translation of the rule set must never produce, according to the semantics specified in this document, a final state of the fact base that would not be a possible result of the execution of the rule set according to the semantics of L_P (where the state of the facts base are meant to be represented in L_P or in RIF-PRD as appropriate), and
- a conformant RIF-PRD consumer, *C*, must translate any rule set from a RIF-PRD document into a rule set in its own language, L_C, in such a way that, for any possible initial state of the fact base, the translation in L_C of the rule set, must never produce, according to the semantics of L_C, a final state of the fact base that would not be a possible result of the execution of the rule set according to the semantics specified in this document (where the state of the facts base are meant to be represented in L_C or in RIF-PRD as appropriate).

Let T be a set of datatypes and symbol spaces that includes the datatypes specified in [RIF-DTB] and the symbol spaces `rif:iri` and `rif:local`. Suppose also that E is a set of external predicates and functions that includes the built-ins listed in [RIF-DTB] and in the section Built-in actions. We say that a rule *r* is a $\textit{RIF-PRD}_{\text{T,E}}$ rule if and only if

- *r* is a well-formed RIF-PRD rule,
- all the datatypes and symbol spaces used in *r* are in T, and
- all the externally defined functions and predicates used in *r* are in E.

Suppose, further, that C is a set of conflict resolution strategies that includes the one specified in section Conflict resolution, and that H is a set of halting tests that includes the one specified in section Halting test: we say that a rule set , *R*, is a $\textit{RIF-PRD}_{\text{T,E,C,H}}$ rule set if and only if

- *R* contains only $\textit{RIF-PRD}_{\text{T,E}}$ rules,
- the conflict resolution strategy that is associated to *R* is in C, and
- the halting test that is associated to *R* is in H.

Given a $\textit{RIF-PRD}_{\text{T,E,C,H}}$ rule set, *R*, an initial state of the fact base, *w*, a conflict resolution strategy $c \in$ C and a halting test $h \in$ H, let $F_{R,w,c,h}$ denote the set of all the sets, *f*, of RIF-PRD ground atomic formulas that represent final states of the fact base, *w'* , according to the operational semantics of a RIF-PRD production rule system, that is: $f \in F_{R,w,c,h}$ if and only if there is a state, *s'* , of the system, such that $Eval(R, c, h, w) \rightarrow^{*}_{PRS} s'$ and $w' = facts(s')$ and *f* is a representation of *w'* .

In addition, given a rule language, *L*, a rule set expressed in *L*, R_L, a conflict resolution strategy, *c*, a halting test, *h*, and an initial state of the fact base, *w*, let $F_{L,R_L, c, h, w}$ denote the set of all the formulas in *L* that represent a final state of the fact base that an *L* processor can possibly reach.

Definition (Semantics preserving mapping).

- A mapping from a $\textit{RIF-PRD}_{\text{T,E,C,H}}$, *R*, to a rule set, R_L, expressed in a language *L*, is **semantics-preserving** if and only if, for any initial state of the fact base, *w*, conflict resolution strategy, *c*, and halting test, *h*, it also maps each *L* formula in $F_{L,R_L, c, h, w}$ onto a set of RIF-PRD ground formulas in $F_{R,w,c,h}$;
- A mapping from a rule set, R_L, expressed in a language *L*, to a $\textit{RIF-PRD}_{\text{T,E,C,H}}$, *R*, is **semantics-preserving** if an only if, for any initial state of the fact base, *w*, conflict resolution strategy, *c*, and halting test, *h*, it also maps each set of ground RIF-PRD atomic formulas in $F_{R,w,c,h}$ onto an *L* formula in $F_{L,R_L, c, h, w}$. ☐

7.2 Conformance Clauses

Definition (RIF-PRD conformance).

- A RIF processor is a ***conformant*** $\textit{RIF-PRD}_{\text{T,E,C,H}}$ ***consumer*** iff it implements a *semantics-preserving mapping* from the set of all *safe* $\textit{RIF-PRD}_{\text{T,E,C,H}}$ rule sets to the language *L* of the processor;
- A RIF processor is a ***conformant*** $\textit{RIF-PRD}_{\text{T,E,C,H}}$ ***producer*** iff it implements a *semantics-preserving mapping* from a subset of the language *L* of the processor to a set of *safe* $\textit{RIF-PRD}_{\text{T,E,C,H}}$ rule sets;
- An ***admissible document*** is an XML document that conforms to all the syntactic constraints of RIF-PRD, including ones that cannot be checked by an XML Schema validator;
- A ***conformant RIF-PRD consumer*** is a conformant $\text{RIF-PRD}_{\text{T,E,C,H}}$ consumer in which T consists only of the symbol spaces and datatypes, E consists only of the externally defined functions and predicates, C consists only of the conflict resolution strategies, and H consists only of halting tests that are required by RIF-PRD. The required symbol spaces are `rif:iri` and `rif:local`, and the datatypes and externally defined terms (built-ins) are the ones specified in [RIF-DTB] and in the section Built-in actions. The required conflict resolution strategy is the one that is identified as `rif:forwardChaining`, as specified in section Conflict resolution; and the required halting test is the one specified in section Halting test. A conformant RIF-PRD consumer must reject any document containing features it does not support.
- A ***conformant RIF-PRD producer*** is a conformant $\text{RIF-PRD}_{\text{T,E,C,H}}$ producer which produces documents that include only the symbol spaces, datatypes, externals, conflict resolution strategies and halting tests that are required by RIF-PRD. ☐

In addition, conformant RIF-PRD producers and consumers SHOULD preserve annotations.

7.3 Interoperability

[RIF-Core] is specified as a specialization of RIF-PRD: all valid [RIF-Core] documents are valid RIF-PRD documents and must be accepted by any conformant RIF-PRD consumer.

Conversely, it is desirable that any valid RIF-PRD document that uses only abstract syntax that is defined in [RIF-Core] be a valid [RIF-Core] document as well. For some abstract constructs that are defined in both RIF-Core and RIF-PRD, RIF-PRD defines alternative XML syntax that is not valid RIF-Core XML syntax. For example, an action block that contains no action variable declaration and only assert atomic actions can be expressed in RIF-PRD using the XML elements `Do` or `And`. Only the latter option is valid RIF-Core XML syntax.

To maximize interoperability with RIF-Core and its non-RIF-PRD extensions, a conformant RIF-PRD consumer SHOULD produce valid [RIF-Core] documents whenever possible. Specifically, a conformant RIF-PRD producer SHOULD use only valid [RIF-Core] XML syntax to serialize a rule set that satisfies all of the following:

- the conflict resolution strategy is effectively equivalent to the stratagy that RIF-PRD identifies by the IRI `rif:forwardChaining`,
- no condition formula contains a negation, in any rule in the rule set,
- no rule in the rule set has an action block that contains an action variable declaration, and
- in all the rules in the rule set, the action block contains only assert atomic actions.

When processing a rule set that satisfies all the above conditions, a RIF-PRD producer is guaranteed to produce a valid [RIF-Core] XML document by applying the following rules recursively:

1. *Remove redundant information*. The `behavior` role element and all its sub-elements should be omitted in the RIF-PRD XML document;
2. *Remove nested rule variable declarations*. If the `rule` inside a rule with variable delcaration, r_1, is also a rule with variable declaration, r_2, all the rule variable delarations and all the patterns that occur in r_1 should be added to the rule variable declarations and the patterns that occur in r_2, and, after the transform, r_1 should be replaced by r_2, in the rule set. If the names of some variables declared in r_1 are the same as the names of some variables declared in r_2, the former names must be changed prior to the transform.;
3. *Remove patterns*. If a pattern occurs in a rule with variable declaration, r_1:
   - if the `rule` inside r_1 is a unconditional action block, r_2, r_2 should be transformed into a conditional action block, where the condition is the pattern, and the pattern should be removed from r_1,
   - if the `rule` inside r_1 is a conditional action block, r_2, the formula that represents the condition in r_2 should be replaced by the conjunction of that formula and the formula that represents the pattern, and the pattern should be removed from r_1;
4. *Convert action blocks*. The action block, in each rule, should be replaced by a conjunction, and, inside the conjunction, each assert action should be replaced by its target atomic formula.

Example 7.1. Consider the following rule, *R*, derived from the *Gold rule*, in the running example, to have only assertions in the action part:

```
R: Forall ?customer such that (And( ?customer # ex1:Customer
                                    ?customer[status->"Silver"] ) )
      (Forall ?shoppingCart such that (?customer[shoppingCart->?shoppingCart])
         (If Exists ?value (And( ?shoppingCart[value->?value]
                                 pred:numeric-greater-than-or-equal(?value 2000))
          Then Do( Assert(ex1:Foo(?customer))
                   Assert(ex1:Bar(?shoppingCart)) ) ) )
```

The serialization of *R* in the following RIF-Core conformant XML form does not impacts its semantics (see example 8.12 for another valid RIF-PRD XML serialization, that is not RIF-Core conformant):

```
<Forall>
   <declare><Var>?customer</Var></declare>
   <declare><Var>?shoppingCart</Var></declare>
   <formula>
      <Implies>
         <if>
            <And>
               <formula>   <!-- first pattern -->
                  <And>
                     <formula><Member> ... </Member></formula>
                     <formula><Frame> ... </Frame></formula>
                  </And>
               </formula>
               <formula>   <!-- second pattern -->
                  <Member> ... </Member>
               </formula>
               <formula>   <!-- original existential condition -->
                  ...
               </formula>
           </And>
        </if>
        <then>
           <And>
              <formula>   <!-- serialization of ex1:Foo(?customer) -->
                 ...
              </formula>
              <formula>   <!-- serialization of ex1:Bar(?shoppingCart) -->
                 ...
              </formula>
        </then>
     </Implies>
  </formula>
</Forall>
```

□

8 XML Syntax

This section specifies the concrete XML syntax of RIF-PRD. The concrete syntax is derived from the abstract syntax defined in sections 2.1, 3.1 and 4.1 using simple mappings. The semantics of the concrete syntax is the same as the semantics of the abstract syntax.

8.1 Notational conventions

8.1.1 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`.

8.1.2 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>
```

8.1.3 Syntactic components

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

- ***Abstract classes*** are defined only by their subclasses: they are 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.

8.2 Relative IRIs and XML base

Relative IRIs are allowed in RIF-PRD XML syntax, anywhere IRIs are allowed, including constant types, symbol spaces, location, and profile. The attribute `xml:base` [XML-Base] is used to make them absolute.

8.3 Conditions

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

8.3.1 TERM

The TERM class of constructs is used to serialize terms, be they simple terms, that is, constants and variables; lists; or positional terms, the latter 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 | List | External ]
```

8.3.1.1 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 be a `rif:iri`;
- 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 `rdf:plainLiteral`. A compliant implementation MUST ignore the `xml:lang` attribute if the type of the `Const` is not `rdf:plainLiteral`.

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

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

Example 8.1.

a. A constant with built-in type *xsd:integer* and value *2,000*:

```
<Const type="xsd:integer">2000</Const>
```

b. The *Customer* class, in the running example, is identified by a constant of type `rif:iri`, in the namespace *http://example.com/2009/prd2#*:

```
<Const type="rif:iri">
   http://example.com/2009/prd2#Customer
</Const>
```

☐

8.3.1.2 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 NCName.

```
<Var> xsd:NCName </Var>
```

8.3.1.3 List

In RIF, the List element is used to serialize a list.

The List element contains an optional items element, that contains one or more TERMs (without variables) that serialize the elements of the list. The order of the sub-elements is significant and MUST be preserved. This is emphasized by the fixed value "yes" of the mandatory attribute ordered in the items element.

```
<List>
  <items ordered="yes"> GROUNDTERM+ </items>?
</List>
```

Example 8.2.

- The list of customer status values from example 2.1:

```
<List>
  <items ordered="yes">
    <Const type="xsd:string> New </Const>
    <Const type="xsd:string> Bronze </Const>
    <Const type="xsd:string> Silver </Const>
    <Const type="xsd:string> Gold </Const>
  </items>
</List>
```

- The empty list:

```
<List>
</List>
```

□

8.3.1.4 External

As a TERM, the External element is used to serialize a positional term. In RIF-PRD, a positional term represents always a call to an externally defined function, e.g. a built-in, a user-defined function, a query to an external data source, etc.

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:

- The External and the content elements 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 alone;
- 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 externally defined function to be applied to the args TERMs;
- The optional args element contains one 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 attribute ordered.

```
<External>
   <content>
      <Expr>
         <op> Const </op>
         <args ordered="yes"> TERM+ </args>?
      </Expr>
   </content>
</External>
```

Example 8.3. The example shows one way to serialize, in RIF-PRD, the product of a variable named ?value and the xsd:decimal value *0.9*, where the operation conforms to the specification of the built-in func:numeric-multiply, as specified in [RIF-DTB].

RIF built-in functions are associated with the namespace http://www.w3.org/2007/rif-builtin-function#.

```
<External>
   <content>
      <Expr>
         <op> <Const type="rif:iri"> http://www.w3.org/2007/rif-builtin-function#numeric-multiply </Const> </op>
         <args ordered="yes">
            <Var> ?value </Var>
            <Const type="xsd:decimal"> 0.9 </Const>
         </args>
      </Expr>
   </content>
</External>
```

□

8.3.2 ATOMIC

The ATOMIC class is used to serialize atomic formulas: positional 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 ]
```

8.3.2.1 Atom

In RIF, the Atom element is used to serialize a positional atomic formula.

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

- 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 one 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 is emphasized by the required value `"yes"` of the attribute `ordered`.

```
<Atom>
   <op> Const </op>
   <args ordered="yes"> TERM+ </args>?
</Atom>
```

Example 8.4. The example shows the RIF XML serialization of the positional atom *ex1:gold(?customer)*, where the predicate symbol *gold* is defined in the example namespace *http://example.com/2009/prd2#*.

```
<Atom>
   <op>
      <Const type="rif:iri">
         http://example.com/2009/prd2#gold
      </Const>
   </op>
   <args ordered="yes">
      <Var> ?customer </Var>
   </args>
</Atom>
```

☐

8.3.2.2 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, that serialize the terms of the equality. The order of the sub-elements is not significant.

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

8.3.2.3 Member

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

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

- the `instance` elements must be a construct from the TERM abstract class that serializes the reference to the object;
- the `class` element must be a construct from the TERM abstract class that serializes the reference to the class.

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

Example 8.5. The example shows the RIF XML serialization of class membership atom that tests whether a variable named `?customer` belongs to a class identified by the name `Customer` in the namespace *http://example.com/2009/prd2#*

```
<Member>
   <instance> <Var> ?customer </Var> </instance>
   <class>
      <Const type="rif:iri">
         http://example.com/2009/prd2#Customer
      </Const>
   </class>
</Member>
```

☐

8.3.2.4 Subclass

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

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

- the `sub` element must be a construct from the TERM abstract class that serializes the reference to the sub-class;
- the `super` elements must be a construct from the TERM abstract class that serializes the reference to the super-class.

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

8.3.2.5 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 that serializes the reference to the frame's object;
- zero to many `slot` elements, each serializing 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 `ordered`.

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

W3C Proposed Edited Recommendation

Example 8.6. The example shows the RIF XML serialization of an expression that states that the object denoted by the variable *?customer* has the value denoted by the string "*Gold*" for the property identified by the symbol *status* that is defined in the example namespace *http://example.com/2009/prd2#*

```
<Frame>
   <object> <Var> ?customer </Var> </object>
   <slot ordered="yes">
      <Const type="rif:iri">
         http://example.com/2009/prd2#status
      </Const>
      <Const type="xsd:string> Gold </Const>
   </slot>
</Frame>
```

☐

8.3.2.6 External

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

When it is an `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.

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.

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

Example 8.7. The example below shows the RIF XML serialization of an externally defined atomic formula that tests whether the value denoted by the variable named *?value* is greater than or equal to the integer value *2000*, where the test is intended to behave like the built-in predicate pred:numeric-greater-than-or-equal as specified in [RIF-DTB]:

RIF built-in predicates are associated with the namespace *http://www.w3.org/2007/rif-builtin-predicate#*.

```
<External>
   <content>
      <Atom>
         <op> <Const type="rif:iri"> http://www.w3.org/2007/rif-builtin-predicate#numeric-greater-than-or-equal </Const> </op>
         <args ordered="yes">
            <Var> ?value </Var>
            <Const type="xsd:integer"> 2000 </Const>
         </args>
      </Atom>
   </content>
</External>
```

☐

8.3.3 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 | INeg | Exists ]
```

8.3.3.1 ATOMIC

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

8.3.3.2 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, that serializes one of the conjuncts.

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

8.3.3.3 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, that serializes one of the disjuncts.

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

8.3.3.4 INeg

The kind of negation that is used in RIF-PRD is serialized using the `INeg` element.

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

```
<INeg>
   <formula> FORMULA </formula>
</INeg>
```

8.3.3.5 Exists

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

The `Exists` element contains:

- one or more `declare` sub-elements, each containing one 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, that serializes the formula in the scope of the quantifier.

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

Example 8.8. The example shows the RIF XML serialization of a condition on the existence of a value greater than or equal to 2.000, in the *Gold rule* of the running example, as represented in example 4.2.

```
<Exists>
   <declare> <Var> ?value </Var> </declare>
   <formula>
      <And>
         <Frame>
            <object> <Var> ?shoppingCart </Var> </object>
            <slot ordered="yes">
               <Const type="rif:iri">
                  http://example.com/2009/prd2#value
               </Const>
               <Var> ?value </Var>
            </slot>
         </Frame>
         <External>
            <content>
               <Atom>
                  <op> <Const type="rif:iri"> http://www.w3.org/2007/rif-builtin-predicate#numeric-greater-than-or-equal </Const> </op>
                  <args ordered="yes">
                     <Var> ?value </Var>
                     <Const type="xsd:integer"> 2000 </Const>
                  </args>
               </Atom>
            </content>
         </External>
      </And>
   </formula>
</Exists>
```

☐

8.4 Actions

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

8.4.1 ACTION

The `ACTION` class of elements is used to serialize the actions: *assert*, *retract*, *modify* and *execute*.

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

```
[ Assert | Retract | Modify | Execute ]
```

8.4.1.1 Assert

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

The `Assert` element has one `target` sub-element that contains an `Atom`, a `Frame` or a `Member` element that represents the target of the action.

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

8.4.1.2 Retract

The `Retract` construct is used to serialize retract atomic actions.

The `Retract` element has one `target` sub-element that contains an `Atom`, a `Frame`, a `TERM`, or a pair of `TERM` constructs that represent the target of the action. The `target` element has an optional attribute, `ordered`, that MUST be present when the element contains two `TERM` sub-elements: the order of the sub-elements is significant and MUST be preserved. This is emphasized by the required value "`yes`" of the attribute.

```
<Retract>
   <target ordered="yes"?>
      [ Atom | Frame | TERM | TERM TERM ]
   </target>
</Retract>
```

8.4.1.3 Modify

A compound modification is serialized using the `Modify` element.

The `Modify` element has one `target` sub-element that contains one `Frame` that represents the target of the action.

```
<Modify>
     <target> Frame </target>
</Modify>
```

Example 8.9. The example shows the RIF XML representation of the action that updates the status of a customer, in the *Gold rule*, in the running example, as represented in example 4.2: `Modify(?customer[status->"Gold"])`

```
<Modify>
   <target>
      <Frame>
         <object>
            <Var> ?customer </Var>
         </object>
         <slot ordered="yes">
            <Const type="rif:iri"> http://example.com/2009/prd2#status </Const>
            <Const type="xsd:string"> Gold </Const>
         </slot>
      </Frame>
   </target>
</Modify>
```

The action could be equivalently serialized as the sequence of a `Retract` and an `Assert` atomic actions:

```
<Retract>
   <target ordered="yes">
      <Var> ?customer </Var>
      <Const type="rif:iri"> http://example.com/2009/prd2#status </Const>
   </target>
</Retract>
<Assert>
   <target>
      <Frame>
         <object>
            <Var> ?customer </Var>
         </object>
         <slot ordered="yes">
            <Const type="rif:iri"> http://example.com/2009/prd2#status </Const>
            <Const type="xsd:string"> Gold </Const>
         </slot>
      </Frame>
   </target>
</Assert>
```

□

8.4.1.4 Execute

The execution of an externally defined action is serialized using the `Execute` element.

The `Execute` element has one `target` sub-element that contains an `Atom`, that represents the externally defined action to be executed.

The op `Const` in the `Atom element` must be a symbol of type `rif:iri` that must uniquely identify the externally defined action to be applied to the `args` TERMs.

```
<Execute>
     <target> Atom </target>
</Execute>
```

Example 8.10. The example shows the RIF XML serialization of the message printing action, in the *Unknonw status rule*, in the running example, using the `act:print` action built-in.

The namespace for RIF-PRD action built-ins is *http://www.w3.org/2007/rif-builtin-action#*.

```
<Execute>
  <target>
    <Atom>
      <op>
        <Constant type="rif:iri"> http://www.w3.org/2007/rif-builtin-action#print </Const>
      </op>
      <args ordered="yes">
        <External>
          <content>
            <Expr>
              <op>
                <Constant type="rif:iri"> http://www.w3.org/2007/rif-builtin-function#concat </Const>
              </op>
              <args ordered="yes">
                <Const type="xsd:string> New customer: </Const>
                 ?customer
              </args>
            </Expr>
          </content>
        </External>
      </args>
    </Atom>
```

W3C Proposed Edited Recommendation

```
    </target>
  </Execute>
```

☐

8.4.2 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 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 SHOULD be serialized like the conclusions of logic rules using RIF-Core or RIF-BLD, that is, as a single asserted `Atom` or `Frame`, or as a conjunction of the asserted facts, using the `And` element.

In the latter case, to conform with the definition of an action block well-formedness, the formulas that serialize the individual conjuncts MUST be atomic `Atom`s and/or `Frame`s.

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

```
[ Do | And | Atom | Frame ]
```

8.4.2.1 New

The `New` element is used to serialize the construct used to create a new frame identifer, in an action variable declaration.

The `New` element is always empty.

```
<New />
```

8.4.2.2 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 the serialization of an action variable binding, that assigns an initial value to the declared variable, that is: either a `frame` or the empty element `New`;
- one `actions` sub-element that serializes the sequence of actions in the action block, and that contains, accordingly, a sequence of one or more sub-elements of the `ACTION` class. The order of the actions is significant, and the order MUST be preserved, as emphasized by the required `ordered="yes"` attribute.

```
<Do>
   <actionVar ordered="yes">
      Var
      [ New | Frame ]
   </actionVar>*
   <actions ordered="yes">
      ACTION+
   </actions>
</Do>
```

Example 8.11. The example shows the RIF XML serialization of an action block that asserts that a customer gets a new $5 voucher.

```
<Do>
   <actionVar ordered="yes">
      <Var>?voucher</Var>
      <New />
   </actionVar>
   <actions  ordered="yes">
      <Assert>
         <target>
            <Member>
               <instance><Var>?voucher</Var></instance>
               <class>
                  <Const type="rif:iri">http://example.com/2009/prd2#Voucher</Const>
               </class>
            </Member>
         </target>
      </Assert>
      <Assert>
         <target>
            <Frame>
               <object><Var>?voucher</Var></object>
               <slot ordered="yes">
                  <Const type="rif:iri">http://example.com/2009/prd2#value</Const>
                  <Const type="xsd:integer">5</Const>
               </slot>
            </Frame>
         </target>
      </Assert>
      <Assert>
         <target>
            <Frame>
               <object><Var>?customer</Var></object>
               <slot ordered="yes">
                  <Const type="rif:iri">http://example.com/2009/prd2#voucher</Const>
                  <Var>?voucher</Var>
               </slot>
            </Frame>
         </target>
      </Assert>
   </actions>
</Do>
```

W3C Proposed Edited Recommendation

□

8.5 Rules and Groups

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

8.5.1 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 ]
```

8.5.1.1 ACTION_BLOCK

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

8.5.1.2 Implies

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

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

- the required `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 conclusion.

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

8.5.1.3 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 one `Var` element that represents one of the declared rule variables;
- zero or more `pattern` sub-elements, each containing one element from the `FORMULA` group of constructs, that serializes one 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>
```

Example 8.12. The example shows the rule variables declaration part of the *Gold rule*, from the running example, as represented in example 4.2.

```
<Forall>
   <declare><Var>?customer</Var></declare>
   <pattern>
      <And>
         <formula><Member> ... </Member></formula>
         <formula><Frame> ... </Frame></formula>
      </And>
   </pattern>
   <formula>
      <Forall>
         <declare><Var>?shoppingCart</Var></declare>
         <pattern><Member> ... </Member></pattern>
         <formula>
            <Implies> ... </Implies>
         </formula>
      </Forall>
   </formula>
</Forall>
```

□

8.5.2 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`;
  - 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>
```

8.6 Document and directives

8.6.1 Import

The `Import` directive is used to serialize the reference to an RDF graph, an OWL ontology or another RIF document to be combined with a RIF document. The `Import` directive is inherited from [RIF-Core]. The abstract syntax and semantics of RDF graph and OWL ontology imports are specified in [RIF-RDF-OWL].

The `Import` directive contains:

- exactly one `location` sub-element, that contains an IRI, that serializes the location of the RDF or OWL document to be combined with the RIF document;
- zero or one `profile` sub-element, that contains an IRI. The admitted values for that constant and their semantics are listed in the section Profiles of Imports, in [RIF-RDF-OWL].

```
<Import>
   <location> xsd:anyURI </location>
   <profile> xsd:anyURI </profile>?
</Import>
```

8.6.2 Document

The `Document` is the root element of any RIF-PRD instance document.

The `Document` contains zero or more `directive` sub-elements, each containing an `Import` directive, and zero or one `payload` sub-element, that must contain a `Group` element.

```
<Document>
   <directive> Import </directive>*
   <payload> Group </payload>?
</Document>
```

The semantics of a document that imports RDF and/or OWL documents is specified in [RIF-RDF-OWL] and [RIF-BLD]. The semantics of a document that does not import other documents is the semantics of the rule set that is serialised by the `Group` in the `document`'s `payload` sub-element, if any.

8.7 Constructs carrying no semantics

8.7.1 Annotation

Annotations 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 | ACTION_BLOCK | New | RULE | Group | Document | Import ]
```

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:iri`.

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

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

If all the annotations are related to the same object, the `meta` element can contain a single `Frame` sub-element. If annotations 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, that serializes the annotations relative to one identified object.

```
<any concrete element in CLASSELT>
    <id> Const </id>?
    <meta>
       [ Frame
         |
         <And>
            <formula> Frame </formula>*
         </And>
       ]
    </meta>?
    other CLASSELT content
 </any concrete element in 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 annotations specifies what the annotations are about, not where it is included in the instance RIF document.

It is suggested to use Dublin Core, RDFS, and OWL properties for annotations, 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 8.13. The example shows the structure of the document that contains the runnig example rule set, as represented in example 4.2, including annotations such as rule set and rule names.

```
<Document>
  <payload>
    <Group>
      <id><Const type="rif:iri">http://example.com/2009/prd2#CheckoutRuleSet</Const></id>
      <meta>
        <Frame>
          <object><Const type="rif:iri">http://example.com/2009/prd2#CheckoutRuleSet</Const></object>
          <slot ordered="yes">
            <Const type="rif:iri">http://dublincore.org/documents/dcmi-namespace/#creator</Const>
            <Const type="xsd:string>W3C RIF WG</Const>
          </slot>
          <slot>
            <Const type="rif:iri">http://dublincore.org/documents/dcmi-namespace/#description</Const>
            <Const type="xsd:string">Running example rule set from the RIF-PRD specification</Const>
```

```
          </slot>
        </Frame>
      </meta>
      <behavior> ... </behavior>
      <sentence>
        <Group>
          <id><Const type="rif:iri">http://example.com/2009/prd2#GoldRule</Const></id>
          <behavior> ... </behavior>
          <sentence><Forall> ... </Forall></sentence>
        </Group>
      </sentence>
      <sentence>
        <Group>
          <id><Const type="rif:iri">http://example.com/2009/prd2#DiscountRule</Const></id>
          <sentence><Forall> ... </Forall></sentence>
        </Group>
      </sentence>
    </Group>
  </payload>
</Document>
```

□

9 Presentation syntax (Informative)

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.

The EBNF for the RIF-PRD presentation syntax is given as follows. For convenience of reading we show the entire EBNF in its four parts (rules, conditions, actions, and annotations).

Rule Language:

```
Document      ::= IRIMETA? 'Document' '(' Base? Prefix* Import* Group? ')'
Base          ::= 'Base' '(' ANGLEBRACKIRI ')'
Prefix        ::= 'Prefix' '(' Name ANGLEBRACKIRI ')'
Import        ::= IRIMETA? 'Import' '(' LOCATOR PROFILE? ')'
Group         ::= IRIMETA? 'Group' Strategy? Priority? '(' (RULE | Group)* ')'
Strategy      ::= Const
Priority      ::= Const
RULE          ::= (IRIMETA? 'Forall' Var+ (' such that ' FORMULA+)?  '(' RULE ')') | CLAUSE
CLAUSE        ::= Implies | ACTION_BLOCK
Implies       ::= IRIMETA? 'If' FORMULA 'Then' ACTION_BLOCK
LOCATOR       ::= ANGLEBRACKIRI
PROFILE       ::= ANGLEBRACKIRI
```

Action Language:

```
ACTION  ::= IRIMETA? (Assert | Retract | Modify | Execute )
Assert        ::= 'Assert' '(' IRIMETA? (Atom | Frame | Member) ')'
Retract       ::= 'Retract' '(' ( IRIMETA? (Atom | Frame) | TERM | TERM TERM ) ')'
Modify        ::= 'Modify'  '(' IRIMETA? Frame ')'
Execute       ::= 'Execute' '(' IRIMETA? Atom ')'
ACTION_BLOCK  ::= IRIMETA? ('Do (' ('(' IRIMETA? Var IRIMETA? (Frame | 'New()') ')')* ACTION+  ')' |
                  'And (' ( IRIMETA? (Atom | Frame) )* ')' | Atom | Frame)
```

Condition Language:

```
FORMULA       ::= IRIMETA? 'And' '(' FORMULA* ')' |
                  IRIMETA? 'Or' '(' FORMULA* ')' |
                  IRIMETA? 'Exists' (IRIMETA? Var)+ '(' FORMULA ')' |
                  ATOMIC |
                  IRIMETA? NEGATEDFORMULA  |
                  IRIMETA? Equal |
                  IRIMETA? Member |
                  IRIMETA? Subclass |
                  IRIMETA? 'External' '(' IRIMETA? Atom ')'
ATOMIC        ::= IRIMETA? (Atom | Frame)
Atom          ::= UNITERM
UNITERM       ::= (IRIMETA? Const) '(' (TERM* ')'
GROUNDUNITERM ::= (IRIMETA? Const) '(' (GROUNDTERM* ')'
NEGATEDFORMULA ::= 'Not' '(' FORMULA ')' | 'INeg' '(' FORMULA ')'
Equal         ::= TERM '=' TERM
Member        ::= TERM '#' TERM
Subclass      ::= TERM '##' TERM
Frame         ::= TERM '[' (TERM '->' TERM)* ']'
TERM          ::= IRIMETA? (Const | Var | List | 'External' '(' Expr ')')
GROUNDTERM    ::= IRIMETA? (Const | List | 'External' '(' GROUNDUNITERM ')')
Expr          ::= UNITERM
List          ::= 'List' '(' GROUNDTERM* ')'
Const         ::= '"' UNICODESTRING '"^^' SYMSPACE | CONSTSHORT
Var           ::= '?' Name
Name          ::= NCName
SYMSPACE      ::= ANGLEBRACKIRI | CURIE
```

Annotations:

```
IRIMETA       ::= '(*' IRICONST? (Frame | 'And' '(' Frame* ')')? '*)'
```

A NEGATEDFORMULA can be written using either *Not* or *INeg*. *INeg* is short for *inflationary negation* and is preferred over 'Not' to avoid ambiguity about the semantics of the negation.

The RIF-PRD presentation syntax does not commit to any particular vocabulary and permits arbitrary Unicode strings in constant symbols, argument names, and variables. Constant symbols can have this form: `"UNICODESTRING"^^SYMSPACE`, where `SYMSPACE` is an `ANGLEBRACKIRI` or `CURIE` that represents the identifier of the symbol space of the constant, and `UNICODESTRING` is a Unicode string from the lexical space of that symbol space. `ANGLEBRACKIRI` and `CURIE` are defined in the section Shortcuts for Constants in RIF's Presentation Syntax in [RIF-DTB]. Constant symbols can also have several shortcut forms, which are represented by the non-terminal `CONSTSHORT`. These shortcuts are also defined in the same section of [RIF-DTB]. One of them is the `CURIE` shortcut, which is extensively used in the examples in this document. Names are XML `NCNames`. Variables are composed of `NCName` symbols prefixed with a ?-sign.

Example 9.1. Here is the transcription, in the RIF-PRD presentation syntax, of the complete RIF-PRD document corresponding to the running example:

```
Document(

Prefix( ex1 <http://example.com/2009/prd2> )

(* ex1:CheckoutRuleset *)
Group  (
  (* ex1:GoldRule *)
  Group  (
    Forall ?customer such that And(?customer # ex1:Customer
                                   ?customer[ex1:status -> "Silver"])
      (Forall ?shoppingCart such that ?customer[ex1:shoppingCart -> ?shoppingCart]
         (If Exists ?value (And(?shoppingCart[ex1:value -> ?value]
                                External(pred:numeric-greater-than-or-equal(?value 2000))))
          Then Do(Modify(?customer[ex1:status -> "Gold"])))))

  (* ex1:DiscountRule *)
  Group (
    Forall ?customer such that ?customer # ex1:Customer
      (If Or( ?customer[ex1:status -> "Silver"]
              ?customer[ex1:status -> "Gold"])
       Then Do ((?s ?customer[ex1:shoppingCart ->  ?s])
                (?v ?s[ex1:value -> ?v])
                Modify(?s [ex1:value -> External(func:numeric-multiply (?v 0.95))]))))

  (* ex1:NewCustomerAndWidgetRule *)
  Group (
    Forall ?customer such that And(?customer # ex1:Customer
                                    ?customer[ex1:status -> "New"] )
      (If Exists ?shoppingCart ?item
                 (And(?customer[ex1:shoppingCart -> ?shoppingCart]
                      ?shoppingCart[ex1:containsItem -> ?item]
                      ?item # ex1:Widget ) )
       Then Do( (?s ?customer[ex1:shoppingCart -> ?s])
                (?val ?s[ex1:value -> ?val])
                Retract(?customer ex1:voucher)
                Modify(?s[ex1:value -> External(func:numeric-multiply(?val 0.90))]))))

  (* ex1:UnknownStatusRule *)
  Group (
    Forall ?customer such that ?customer # ex1:Customer
      (If Not(Exists ?status
                     (And(?customer[ex1:status -> ?status]
                          External(pred:list-contains(List("New" "Bronze" "Silver" "Gold") ?status)) )))
       Then Do( Execute(act:print(External(func:concat("New customer: " ?customer))))
                Assert(?customer[ex1:status -> "New"]))))
 )
)
```

□

10 Acknowledgements

This document is the product of the Rules Interchange Format (RIF) Working Group (see below) whose members deserve recognition for their time and commitment. The editors extend special thanks to Harold Boley and Changhai Ke for their thorough reviews and insightful discussions; the working group chairs, Chris Welty and Christian de Sainte Marie, for their invaluable technical help and inspirational leadership; and W3C staff contact Sandro Hawke, a constant source of ideas, help, and feedback.

The regular attendees at meetings of the Rule Interchange Format (RIF) Working Group at the time of the publication were: Adrian Paschke (Freie Universitaet Berlin), Axel Polleres (DERI), Chris Welty (IBM), Christian de Sainte Marie (IBM), Dave Reynolds (HP), Gary Hallmark (ORACLE), Harold Boley (NRC), Jos de Bruijn (FUB), Leora Morgenstern (IBM), Michael Kifer (Stony Brook), Mike Dean (BBN), Sandro Hawke (W3C/MIT), and Stella Mitchell (IBM).

11 References

11.1 Normative references

[OMG-PRR]
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, at http://www.ietf.org/rfc/rfc3066.

[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 Proposed Edited Recommendation, 11 December 2012, http://www.w3.org/TR/2012/PER-rif-bld-20121211/. Latest version available at http://www.w3.org/TR/rif-bld/.

[RIF-Core]
RIF Core Dialect Harold Boley, Gary Hallmark, Michael Kifer, Adrian Paschke, Axel Polleres, Dave Reynolds, eds. W3C Proposed Edited Recommendation, 11 December 2012, http://www.w3.org/TR/2012/PER-rif-core-20121211/. Latest version available at http://www.w3.org/TR/rif-core/.

[RIF-DTB]
RIF Datatypes and Built-Ins 1.0 Axel Polleres, Harold Boley, Michael Kifer, eds. W3C Proposed Edited Recommendation, 11 December 2012, http://www.w3.org/TR/2012/PER-rif-dtb-20121211/. Latest version available at http://www.w3.org/TR/rif-dtb/.

[RIF-FLD]
RIF Framework for Logic Dialects Harold Boley, Michael Kifer, eds. W3C Proposed Edited Recommendation, 11 December 2012, http://www.w3.org/TR/2012/PER-rif-fld-20121211/. Latest version available at http://www.w3.org/TR/rif-fld/.

[RIF-RDF-OWL]
RIF RDF and OWL Compatibility Jos de Bruijn, Chris Welty, eds. W3C Proposed Edited Recommendation, 11 December 2012, http://www.w3.org/TR/2012/PER-rif-rdf-owl-20121211/. Latest version available at http://www.w3.org/TR/rif-rdf-owl/.

[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-Base]
XML Base (second edition), W3C Recommendation, World Wide Web Consortium, 28 January 2009, http://www.w3.org/TR/2009/REC-xmlbase-20090128/. Latest version available at http://www.w3.org/TR/xmlbase/.

[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/.

11.2 Informational 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, S. McCarron, M. Birbeck, Editors, W3C Working Group Note, 16 December 2010, http://www.w3.org/TR/2010/NOTE-curie-20101216 . Latest version available at http://www.w3.org/TR/curie.

[Enderton01]
A Mathematical Introduction to Logic, Second Edition, H. B. Enderton. Academic Press, 2001.

[FIT02]
Fixpoint Semantics for Logic Programming: A Survey, Melvin Fitting, Theoretical Computer Science. Vol. 278, no. 1-2, pp. 25-51. 6 May 2002.

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

[KLW95]
Logical foundations of object-oriented and frame-based languages, M. Kifer, G. Lausen, J. Wu. Journal of ACM, July 1995, pp. 741--843.

[Mendelson97]
Introduction to Mathematical Logic, Fourth Edition, E. Mendelson. Chapman & Hall, 1997.

[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).

12 Appendix: Model-theoretic semantics of RIF-PRD condition formulas

This appendix provides an alternative specification of the Semantics of condition formulas, and it is also normative.

This alternative specification is provided for the convenience of the reader, for compatibility with other RIF specifications, such as [RIF-DTB] and [RIF-RDF-OWL], and to make explicit the interoperability with RIF logic dialects, in particular [RIF-Core] and [RIF-BLD].

12.1 Semantic structures

The key concept in a model-theoretic semantics of a logic language is the notion of a semantic structure [Enderton01, Mendelson97].

Definition (Semantic structure). A ***semantic structure***, $\boldsymbol{I}$, is a tuple of the form <$\boldsymbol{TV}$, $\boldsymbol{DTS}$, $\boldsymbol{D}$, $\boldsymbol{D}_{\text{ind}}$, $\boldsymbol{D}_{\text{func}}$, $\boldsymbol{I}_{\text{C}}$, $\boldsymbol{I}_{\text{V}}$, $\boldsymbol{I}_{\text{list}}$, $\boldsymbol{I}_{\text{P}}$, $\boldsymbol{I}_{\text{frame}}$, $\boldsymbol{I}_{\text{sub}}$, $\boldsymbol{I}_{\text{isa}}$, $\boldsymbol{I}_{=}$, $\boldsymbol{I}_{\text{external}}$, $\boldsymbol{I}_{\text{truth}}$>. Here $\boldsymbol{D}$ is a non-empty set of elements called the ***Herbrand domain*** of $\boldsymbol{I}$, that is, the set of all ground terms which can be formed by using the elements of `Const`. $\boldsymbol{D}_{\text{ind}}$, $\boldsymbol{D}_{\text{func}}$ are nonempty subsets of $\boldsymbol{D}$. $\boldsymbol{D}_{\text{ind}}$ is used to interpret the elements of `Const` that are individuals and $\boldsymbol{D}_{\text{func}}$ 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. $\boldsymbol{TV}$ denotes the set of truth values that the semantic structure uses and $\boldsymbol{DTS}$ is a set of identifiers for primitive datatypes (please refer to Section Datatypes in the RIF data types and builtins specification [RIF-DTB] for the semantics of datatypes).

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

The other components of $\boldsymbol{I}$ are *total* mappings defined as follows:

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

   This mapping interprets constant symbols. In addition:

- If a constant, c ∈ `Const`, is an *individual* then it is required that $\boldsymbol{I}_C$(c) ∈ $\boldsymbol{D}_{ind}$.
- If c ∈ `Const`, is a *function symbol* then it is required that $\boldsymbol{I}_C$(c) ∈ $\boldsymbol{D}_{func}$.

2. $\boldsymbol{I}_V$ maps `Var` to $\boldsymbol{D}_{ind}$.

   This mapping interprets variable symbols.

3. $\boldsymbol{I}_{list}$: $\boldsymbol{D}_{ind}^{*}$ → $\boldsymbol{D}_{ind}$ is used to interpret lists.

   In addition, this mapping is required to satisfy the following conditions:

   - $\boldsymbol{I}_{list}$ is an injective one-to-one function.
   - $\boldsymbol{I}_{list}$($\boldsymbol{D}_{ind}$) is disjoint from the value spaces of all data types in ***DTS***.

4. $\boldsymbol{I}_P$ maps ***D*** to functions $\boldsymbol{D*}_{ind}$ → ***D*** (here $\boldsymbol{D*}_{ind}$ is a set of all sequences of any finite length over the domain $\boldsymbol{D}_{ind}$).

   This mapping interprets positional terms atoms.

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

   This mapping interprets frame terms. An argument, d ? $\boldsymbol{D}_{ind}$, to $\boldsymbol{I}_{frame}$ represents an object and the finite bag `{<a1,v1>`, ..., `<ak,vk>}` represents a bag of attribute-value pairs for d. We will see shortly how $\boldsymbol{I}_{frame}$ 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. 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 while `?B` and `?D` are instantiated with b. (We shall see later that `o[a->b a->b]` is equivalent to `o[a->b]`.)

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

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

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

   It gives meaning to the equality operator.

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

   It is used to define truth valuation for formulas.

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

    For every external schema, σ, associated with the language, $\boldsymbol{I}_{external}$(σ) 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, function or action, $\boldsymbol{I}_{external}$(σ) is specified so that:

    - If σ is a schema of a built-in function then $\boldsymbol{I}_{external}$(σ) must be the function defined in [RIF-DTB];
    - If σ is a schema of a built-in predicate then $\boldsymbol{I}_{truth}$ `o` ($\boldsymbol{I}_{external}$(σ)) (the composition of $\boldsymbol{I}_{truth}$ and $\boldsymbol{I}_{external}$(σ), a truth-valued function) must be as specified in [RIF-DTB];
    - If σ is a schema of a built-in action then $\boldsymbol{I}_{truth}$ `o` ($\boldsymbol{I}_{external}$(σ)) (the composition of $\boldsymbol{I}_{truth}$ and $\boldsymbol{I}_{external}$(σ), a truth-valued function) must be as specified in the section Built-in actions in this document.

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

- ***I***(k) = $\boldsymbol{I}_C$(k), if k is a symbol in `Const`;
- ***I***(?v) = $\boldsymbol{I}_V$(?v), if ?v is a variable in `Var`;
- For list terms, the mapping is defined as follows:
  - ***I***(`List( )`) = $\boldsymbol{I}_{list}$(<>). Here <> denotes an empty list of elements of $\boldsymbol{D}_{ind}$. (Note that the domain of $\boldsymbol{I}_{list}$ is $\boldsymbol{D}_{ind}^{*}$, so $\boldsymbol{D}_{ind}^{0}$ is an empty list of elements of $\boldsymbol{D}_{ind}$.)
  - ***I***(`List(t`$_1$ ... `t`$_n$`)`) = $\boldsymbol{I}_{list}$(***I***(t$_1$), ..., ***I***(t$_n$)), if n>0.
- ***I***(`p(t`$_1$ `... t`$_n$`)`) = $\boldsymbol{I}_P$(***I***(p))(***I***(t$_1$),...,***I***(t$_n$));
- ***I***(`o[a`$_1$`->v`$_1$ `... a`$_k$`->v`$_k$`]`) = $\boldsymbol{I}_{frame}$(***I***(o))({<***I***(a$_1$),***I***(v$_1$)>, ..., <***I***(a$_n$),***I***(v$_n$)>})
  Here {...} denotes a bag of attribute/value pairs.
- ***I***(c1##c2) = $\boldsymbol{I}_{sub}$(***I***(c1), ***I***(c2));
- ***I***(o#c) = $\boldsymbol{I}_{isa}$(***I***(o), ***I***(c));
- ***I***(x=y) = $\boldsymbol{I}_{=}$(***I***(x), ***I***(y));
- ***I***(`External(t)`) = $\boldsymbol{I}_{external}$(σ)(***I***(s$_1$), ..., ***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 ***I***(`External(t)`) is well-defined.

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

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

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

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

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

12.2 Interpretation of condition formulas

This section defines how a semantic structure, ***I***, determines the truth value *TVal*$_I$(φ) of a condition formula, φ.

We define a mapping, *TVal*$_I$, from the set of all condition formulas to ***TV***. Note that the definition implies that *TVal*$_I$(φ) is defined *only if* the set ***DTS*** of the datatypes of ***I*** includes all the datatypes mentioned in φ and $\boldsymbol{I}_{external}$ 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 $TVal_I$:

- *Positional atomic formulas*: $TVal_I$(r(t1 ... tn)) = $\boldsymbol{I}_{truth}(\boldsymbol{I}$(r(t1 ... tn)));
- *Equality*: $TVal_I$(x = y) = $\boldsymbol{I}_{truth}(\boldsymbol{I}$(x = y)).
  To ensure that equality has precisely the expected properties, it is required that:
  - $\boldsymbol{I}_{truth}(\boldsymbol{I}$(x = y)) = **t** if $\boldsymbol{I}$(x) = $\boldsymbol{I}$(y) and that $\boldsymbol{I}_{truth}(\boldsymbol{I}$(x = y)) = **f** otherwise. This is tantamount to saying that $TVal_I$(x = y) = **t** iff $\boldsymbol{I}$(x) = $\boldsymbol{I}$(y);
- *Subclass*: $TVal_I$(sc ## cl) = $\boldsymbol{I}_{truth}(\boldsymbol{I}$(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 ∈ ***D***, if $TVal_I$(c1 ## c2) = $TVal_I$(c2 ## c3) = **t** then $TVal_I$(c1 ## c3) = **t**;
- *Membership*: $TVal_I$(o # cl) = $\boldsymbol{I}_{truth}(\boldsymbol{I}$(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 o, cl, scl ∈ ***D***, if $TVal_I$(o # cl) = $TVal_I$(cl ## scl) = **t** then $TVal_I$(o # scl) = **t**;
- *Frame*: $TVal_I$(o[a_1->v_1 ... a_k->v_k]) = $\boldsymbol{I}_{truth}(\boldsymbol{I}$(o[$a_1$->$v_1$ ... a_k->v_k])).
  Since the bag of attribute/value pairs represents the conjunctions of all the pairs, the following is required, if k > 0:
  - $TVal_I$(o[a_1->v_1 ... a_k->v_k]) = **t** if and only if $TVal_I$(o[a_1->v_1]) = ... = $TVal_I$(o[a_k->v_k]) = **t**;
- *Externally defined atomic formula*: $TVal_I$(External(t)) = $\boldsymbol{I}_{truth}(\boldsymbol{I}_{external}(\sigma)(\boldsymbol{I}(s_1), ..., \boldsymbol{I}(s_n)))$, if t is an atomic formula that 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;
- *Conjunction*: $TVal_I$(And(c_1 ... c_n)) = **t** if and only if $TVal_I$(c_1) = ... = $TVal_I$(c_n) = **t**. Otherwise, $TVal_I$(And(c_1 ... c_n)) = **f**.
  The empty conjunction is treated as a tautology, so $TVal_I$(And()) = **t**;
- *Disjunction*: $TVal_I$(Or(c_1 ... c_n)) = **f** if and only if $TVal_I$(c_1) = ... = $TVal_I$(c_n) = **f**. Otherwise, $TVal_I$(Or(c_1 ... c_n)) = **t**.
  The empty disjunction is treated as a contradiction, so $TVal_I$(Or()) = **f**;
- *Negation*: $TVal_I$(Not(c)) = **f** if and only if $TVal_I$(c) = **t**. Otherwise, $TVal_I$(Not(c)) = **t**;
- *Existence*: $TVal_I$(Exists ?v_1 ... ?v_n (φ)) = **t** if and only if for some $\boldsymbol{I}$*, described below, $TVal_{I*}$(φ) = **t**.
  Here $\boldsymbol{I}$* is a semantic structure of the form <***TV***, ***DTS***, ***D***, $\boldsymbol{D}_{ind}$, $\boldsymbol{D}_{func}$, $\boldsymbol{I}_C$, $\boldsymbol{I}^*_V$, $\boldsymbol{I}_{list}$, $\boldsymbol{I}_P$, $\boldsymbol{I}_{frame}$, $\boldsymbol{I}_{sub}$, $\boldsymbol{I}_{isa}$, $\boldsymbol{I}_=$, $\boldsymbol{I}_{external}$, $\boldsymbol{I}_{truth}$>, 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 ?v_1,...,?v_n. ☐

12.3 Condition satisfaction

We define, now, what it means for a *state of the fact base* to satisfy a condition formula. The satisfaction of condition formulas in a state of the fact base provides formal underpinning to the operational semantics of rule sets interchanged using RIF-PRD.

Definition (Models). A semantic structure ***I*** is a ***model*** of a condition formula, φ, written as ***I*** |= φ, iff $TVal_I$(φ) = **t**. ☐

Definition (Herbrand interpretation). Given a non-empty set of constants, Const, the *Herbrand domain* is the set of all the ground terms that can be formed using the elements of Const, and the *Herbrand base* is the set of all the well-formed ground atomic formulas that can be formed with the elements in the Herbrand domain.

A semantic structure, ***I***, is a ***Herbrand interpretation***, if the set of all the ground formulas which are true with respect to ***I*** (that is, of which ***I*** is a model), is a subset of the corresponding Herbrand base, B_I. ☐

In RIF-PRD, the semantics of condition formulas is defined with respect to semantic structures where the domain, ***D*** is the Herbrand domain that is determined by the set of all the constants, Const; that is, with respect to Herbrand interpretations.

Definition (State of the fact base). To every Herbrand interpretation ***I***, we associate a ***state of the fact base***, w_I, that is represented by the subset of the Herbrand base that contains exactly the ground atomic formulas of which ***I*** is a model; or, equivalently, by the conjunction of all these ground atomic formulas. ☐

Definition (Condition satisfaction). A RIF-PRD condition formula φ is ***satisfied*** in a state of the fact base, w_I, if and only if ***I*** is a model of φ. ☐

At the syntactic level, the interpretation of the variables by a valuation function $\boldsymbol{I}_V$ is realized by a substitution. As a consequence, a ground substitution *σ* matches a condition formula *ψ* to a set of ground atomic formulas *Φ* if and only if *σ* realizes the valuation function $\boldsymbol{I}_V$ of a semantics structure ***I*** that is a model of *ψ* and *Φ* is a representation of a state of the fact base, w_I (as defined above), that is associated to ***I***; that is, if and only if *ψ* is satisfied in w_I (as defined above).

This provides the formal link between the satisfaction of a condition formula, as defined above, and a matching substitution, and, followingly, between the alternative definitions of a state of facts and the satisfaction of a condition, here and in section Semantics of condition formulas.

13 Appendix: XML schema

The RIF-PRD XML Schema is specified below as a redefinition and an extension of the RIF-Core XML Schema [RIF-Core] and is also available at http://www.w3.org/2010/rif-schema/prd/.

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema
    targetNamespace="http://www.w3.org/2007/rif#"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xml="http://www.w3.org/XML/1998/namespace"
    xmlns="http://www.w3.org/2007/rif#"
    elementFormDefault="qualified">

<xs:import namespace='http://www.w3.org/XML/1998/namespace'
           schemaLocation='http://www.w3.org/2001/xml.xsd'/>


<!-- ================================================== -->
<!-- Redefine some elements in the Core Conditions      -->
<!-- Extension of the choice                            -->
<!-- ================================================== -->

   <xs:group name="ATOMIC">
      <xs:choice>
         <xs:element ref="Atom"/>
         <xs:element ref="Frame"/>
         <xs:element ref="Member"/>
         <xs:element ref="Equal"/>
         <xs:element ref="Subclass"/> <!-- Subclass is not in RIF-Core -->
```

```
         <xs:element name="External" type="External-FORMULA.type"/>
      </xs:choice>
   </xs:group>

   <xs:group name="FORMULA">
      <xs:choice>
         <xs:group   ref="ATOMIC"/>
         <xs:element ref="And"/>
         <xs:element ref="Or"/>
         <xs:element ref="Exists"/>
         <xs:element ref="INeg"/> <!-- INeg is nt in RIF-Core -->
      </xs:choice>
   </xs:group>

<!-- ================================================= -->
<!-- Additional elements to the Core Condition schema   -->
<!-- ================================================= -->

   <xs:element name="Subclass">
<!--                                                   -->
<!--     <Subclass>                                    -->
<!--        <sub> TERM </sub>                          -->
<!--        <super> TERM </super>                      -->
<!--     </Subclass>                                   -->
<!--                                                   -->
      <xs:complexType>
         <xs:sequence>
            <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
            <xs:element name="sub">
               <xs:complexType>
                  <xs:group ref="TERM" minOccurs="1" maxOccurs="1"/>
               </xs:complexType>
            </xs:element>
            <xs:element name="super">
               <xs:complexType>
                  <xs:group ref="TERM" minOccurs="1" maxOccurs="1"/>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="INeg">
<!--                                                   -->
<!--     <INeg>                                        -->
<!--        <formula> FORMULA </formula>               -->
<!--     </INeg>                                       -->
<!--                                                   -->
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
          <xs:element ref="formula" minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
      </xs:complexType>
   </xs:element>

<!-- =========================================== -->
<!-- CoreCond.xsd starts here                    -->
<!-- =========================================== -->

  <xs:complexType name="External-FORMULA.type">
    <!-- sensitive to FORMULA (Atom) context-->
    <xs:sequence>
      <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
      <xs:element name="content" type="content-FORMULA.type"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="content-FORMULA.type">
    <!-- sensitive to FORMULA (Atom) context-->
    <xs:sequence>
      <xs:element ref="Atom"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="And">
    <xs:complexType>
      <xs:sequence>
        <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="formula" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Or">
    <xs:complexType>
      <xs:sequence>
        <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="formula" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Exists">
```

```
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="declare" minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="formula"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="formula">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="FORMULA"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="declare">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Var"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Atom">
  <!--
Atom           ::= UNITERM
  -->
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="UNITERM"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:group name="UNITERM">
  <!--
UNITERM        ::= Const '(' (TERM* ')'
  -->
  <xs:sequence>
    <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
    <xs:element ref="op"/>
    <xs:element name="args" type="args-UNITERM.type" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:group>

<xs:group name="GROUNDUNITERM">
  <!-- sensitive to ground terms
GROUNDUNITERM        ::= Const '(' (GROUNDTERM* ')'
  -->
  <xs:sequence>
    <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
    <xs:element ref="op"/>
    <xs:element name="args" type="args-GROUNDUNITERM.type" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:group>


<xs:element name="op">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Const"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="args-UNITERM.type">
  <!-- sensitive to UNITERM (TERM) context-->
  <xs:sequence>
    <xs:group ref="TERM" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="ordered" type="xs:string" fixed="yes"/>
</xs:complexType>


<xs:complexType name="args-GROUNDUNITERM.type">
  <!-- sensitive to GROUNDUNITERM (TERM) context-->
  <xs:sequence>
    <xs:group ref="GROUNDTERM" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="ordered" type="xs:string" fixed="yes"/>
</xs:complexType>

<xs:element name="Equal">
  <!--
Equal          ::= TERM '=' ( TERM | IRIMETA? 'External' '(' Expr ')' )
  -->
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="left"/>
      <xs:element ref="right"/>
    </xs:sequence>
```

```
  </xs:complexType>
</xs:element>

<xs:element name="left">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="TERM"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="right">
  <xs:complexType>
      <xs:group ref="TERM"/>
  </xs:complexType>
</xs:element>

<xs:element name="Member">
  <!--
Member         ::= TERM '#' TERM
  -->
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="instance"/>
      <xs:element ref="class"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="instance">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="TERM"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="class">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="TERM"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Frame">
  <!--
Frame          ::= TERM '[' (TERM '->' TERM)* ']'
  -->
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="object"/>
      <xs:element name="slot" type="slot-Frame.type" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="object">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="TERM"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="slot-Frame.type">
  <!-- sensitive to Frame (TERM) context-->
  <xs:sequence>
    <xs:group ref="TERM"/>
    <xs:group ref="TERM"/>
  </xs:sequence>
  <xs:attribute name="ordered" type="xs:string" fixed="yes"/>
</xs:complexType>

<xs:group name="TERM">
  <!--
TERM           ::= IRIMETA? (Const | Var | External | List )
  -->
    <xs:choice>
       <xs:element ref="Const"/>
       <xs:element ref="Var"/>
       <xs:element name="External" type="External-TERM.type"/>
       <xs:element ref="List"/>
    </xs:choice>
</xs:group>

<xs:group name="GROUNDTERM">
  <!--
GROUNDTERM           ::= IRIMETA? (Const | List | 'External' '(' 'Expr' '(' GROUNDUNITERM ')' ')')
  -->
    <xs:choice>
       <xs:element ref="Const"/>
```

```
      <xs:element ref="List"/>
      <xs:element name="External" type="External-GROUNDUNITERM.type"/>
    </xs:choice>
</xs:group>

<xs:element name="List">
  <!--
    List          ::= 'List' '(' GROUNDTERM* ')'
  -->
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
      <xs:element name="items">
        <xs:complexType>
          <xs:sequence>
            <xs:group ref="GROUNDTERM" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="ordered" type="xs:string" fixed="yes"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="External-TERM.type">
  <!-- sensitive to TERM (Expr) context-->
  <xs:sequence>
    <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
    <xs:element name="content" type="content-TERM.type"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="External-GROUNDUNITERM.type">
  <!-- sensitive to GROUNDTERM (Expr) context-->
  <xs:sequence>
    <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
    <xs:element name="content" type="content-GROUNDUNITERM.type"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="content-TERM.type">
  <!-- sensitive to TERM (Expr) context-->
  <xs:sequence>
    <xs:element ref="Expr"/>
  </xs:sequence>
</xs:complexType>


 <xs:complexType name="content-GROUNDUNITERM.type">
  <!-- sensitive to GROUNDTERM (Expr) context-->
  <xs:sequence>
    <xs:element name="Expr" type="content-GROUNDEXPR.type"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="content-GROUNDEXPR.type">
  <!-- sensitive to GROUNDEXPR context-->
  <xs:sequence>
    <xs:group ref="GROUNDUNITERM"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="Expr">
  <!--
Expr          ::= Const '(' (TERM | IRIMETA? 'External' '(' Expr ')')* ')'
  -->
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="op"/>
      <xs:element name="args" type="args-Expr.type" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="args-Expr.type">
  <!-- sensitive to Expr (TERM) context-->
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:group ref="TERM"/>
  </xs:choice>
  <xs:attribute name="ordered" type="xs:string" fixed="yes"/>
</xs:complexType>

<xs:element name="Const">
  <!--
Const         ::= '"' UNICODESTRING '"^^' SYMSPACE | CONSTSHORT
  -->
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="type" type="xs:anyURI" use="required"/>
    <xs:attribute ref="xml:lang"/>
  </xs:complexType>
```

```
  </xs:element>

  <xs:element name="Var">
    <!--
  Var           ::= '?' NCName
    -->
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:group name="IRIMETA">
    <!--
  IRIMETA   ::= '(*' IRICONST? (Frame | 'And' '(' Frame* ')')? '*)'
    -->
    <xs:sequence>
      <xs:element ref="id" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="meta" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:group>

  <xs:element name="id">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Const" type="IRICONST.type"/>   <!-- type="&rif;iri" -->
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="meta">
    <xs:complexType>
     <xs:choice>
       <xs:element ref="Frame"/>
       <xs:element name="And" type="And-meta.type"/>
     </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="And-meta.type">
  <!-- sensitive to meta (Frame) context-->
    <xs:sequence>
      <xs:element name="formula" type="formula-meta.type" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="formula-meta.type">
    <!-- sensitive to meta (Frame) context-->
    <xs:sequence>
      <xs:element ref="Frame"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="IRICONST.type" mixed="true">
    <!-- sensitive to location/id context-->
    <xs:sequence/>
    <xs:attribute name="type" type="xs:anyURI" use="required" fixed="http://www.w3.org/2007/rif#iri"/>
  </xs:complexType>

<!-- ============================================ -->
<!-- Definition of the actions (not in RIF-Core)  -->
<!-- ============================================ -->

   <xs:element name="New">
    <xs:complexType>
      <xs:sequence>
        <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>


   <xs:group name="INITIALIZATION">
      <xs:choice>
         <xs:element ref="New"/>
         <xs:element ref="Frame"/>
      </xs:choice>
   </xs:group>

   <xs:element name="Do">
<!--                                                 -->
<!--     <Do>                                        -->
<!--        <actionVar ordered="yes">                -->
<!--          Var                                    -->
<!--          INITIALIZATION                         -->
<!--        </actionVar>*                            -->
<!--        <actions ordered="yes">                  -->
<!--          ACTION+                         -->
<!--        </actions>                               -->
<!--     </Do>                                       -->
<!--                                                 -->
      <xs:complexType>
         <xs:sequence>
```

```
            <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
            <xs:element name="actionVar" minOccurs="0" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="Var" minOccurs="1" maxOccurs="1"/>
                     <xs:group ref="INITIALIZATION" minOccurs="1" maxOccurs="1"/>
                  </xs:sequence>
                  <xs:attribute name="ordered" type="xs:string" fixed="yes"/>
               </xs:complexType>
            </xs:element>
            <xs:element name="actions" minOccurs="1" maxOccurs="1">
               <xs:complexType>
                  <xs:sequence>
                     <xs:group ref="ACTION" minOccurs="1" maxOccurs="unbounded"/>
                  </xs:sequence>
                  <xs:attribute name="ordered" type="xs:string" fixed="yes"/>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:group name="ACTION">
      <xs:choice>
         <xs:element ref="Assert"/>
         <xs:element ref="Retract"/>
         <xs:element ref="Modify"/>
         <xs:element ref="Execute"/>
      </xs:choice>
   </xs:group>

   <xs:element name="Assert">
<!--                                                   -->
<!--     <Assert>                                      -->
<!--        <target> [ Atom                            -->
<!--                 | Frame                           -->
<!--                 | Member ]                        -->
<!--        </target>                                  -->
<!--     </Assert>                                     -->
<!--                                                   -->
      <xs:complexType>
         <xs:sequence>
            <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
            <xs:element name="target" minOccurs="1" maxOccurs="1">
               <xs:complexType>
                  <xs:choice>
                     <xs:element ref="Atom"/>
                     <xs:element ref="Frame"/>
                     <xs:element ref="Member"/>
                  </xs:choice>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
     </xs:complexType>
   </xs:element>

   <xs:element name="Retract">
<!--                                                   -->
<!--     <Retract>                                     -->
<!--        <target ordered="yes"?>                    -->
<!--           [ Atom                                  -->
<!--             | Frame                               -->
<!--             | TERM                                -->
<!--             | TERM TERM ]                         -->
<!--        </target>                                  -->
<!--     </Assert>                                     -->
<!--                                                   -->
      <xs:complexType>
         <xs:sequence>
            <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
            <xs:element name="target" minOccurs="1" maxOccurs="1">
               <xs:complexType>
                  <xs:choice>
                     <xs:element ref="Atom"/>
                     <xs:element ref="Frame"/>
                     <xs:group ref="TERM"/>
                     <xs:sequence>
                        <xs:group ref="TERM"/>
                        <xs:group ref="TERM"/>
                     </xs:sequence>
                  </xs:choice>
                  <xs:attribute name="ordered" type="xs:string" fixed="yes"/>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="Modify">
<!--                                                   -->
<!--     <Modify>                                      -->
<!--        <target> Frame </target>                   -->
<!--     </Modify>                                     -->
<!--                                                   -->
```

W3C Proposed Edited Recommendation

```
      <xs:complexType>
       <xs:sequence>
        <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
        <xs:element name="target" minOccurs="1" maxOccurs="1">
         <xs:complexType>
          <xs:sequence>
           <xs:element ref="Frame"/>
          </xs:sequence>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="Execute">
 <!--                                                      -->
 <!--     <Execute>                                        -->
 <!--        <target> Atom </target>                       -->
 <!--     </Execute>                                       -->
 <!--                                                      -->
      <xs:complexType>
       <xs:sequence>
        <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
        <xs:element name="target" minOccurs="1" maxOccurs="1">
         <xs:complexType>
          <xs:sequence>
           <xs:element ref="Atom"/>
          </xs:sequence>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
    </xs:element>

 <!-- ================================================== -->
 <!-- Redefine Group related Core construct              -->
 <!-- ================================================== -->

    <xs:complexType name="Group-contents">
       <xs:sequence>
          <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
          <xs:element ref="behavior" minOccurs="0" maxOccurs="1"/>
                      <!-- behavior in not in RIF-Core -->
          <xs:element ref="sentence" minOccurs="0" maxOccurs="unbounded"/>
       </xs:sequence>
    </xs:complexType>

 <!-- ================================================== -->
 <!-- Group related addition                             -->
 <!-- ================================================== -->

    <xs:element name="behavior">
 <!--                                                      -->
 <!--     <behavior>                                       -->
 <!--        <ConflictResolution>                          -->
 <!--           xsd:anyURI                                 -->
 <!--        <ConflictResolution>?                         -->
 <!--        <Priority>                                    -->
 <!--           -10,000 ≤ xsd:int ≤ 10,000         -->
 <!--        </Priority>?                                  -->
 <!--     </behavior>                                      -->
 <!--                                                      -->
       <xs:complexType>
          <xs:sequence>
             <xs:element name="ConflictResolution" minOccurs="0" maxOccurs="1" type="xs:anyURI"/>
             <xs:element name="Priority" minOccurs="0" maxOccurs="1">
                <xs:simpleType>
                   <xs:restriction base="xs:int">
                      <xs:minInclusive value="-10000"/>
                      <xs:maxInclusive value="10000"/>
                   </xs:restriction>
                </xs:simpleType>
             </xs:element>
          </xs:sequence>
       </xs:complexType>
    </xs:element>

 <!-- ================================================== -->
 <!-- Redefine rule related Core constructs              -->
 <!-- ================================================== -->

   <xs:group name="RULE">
     <!--
    RULE      ::= (IRIMETA? 'Forall' Var+ (' such that ' FORMULA+)?  '(' RULE ')') | CLAUSE
     -->
     <xs:choice>
        <xs:element name="Forall" type="Forall-premises"/>
        <xs:group   ref="CLAUSE"/>
     </xs:choice>
   </xs:group>

   <xs:complexType name="Forall-premises">
     <xs:sequence>
       <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
```

```
        <xs:element ref="declare" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="pattern" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:group ref="FORMULA"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <!-- different from formula in And, Or and Exists -->
        <xs:element name="formula">
          <xs:complexType>
            <xs:group ref="RULE"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>

    <xs:group name="CLAUSE">
      <!--
    CLAUSE   ::= Implies | ACTION_BLOCK
      -->
      <xs:choice>
        <xs:element ref="Implies"/>
        <xs:group ref="ACTION_BLOCK"/>
      </xs:choice>
    </xs:group>

     <xs:complexType name="then-part">
             <xs:group ref="ACTION_BLOCK" minOccurs="1" maxOccurs="1"/>
     </xs:complexType>

  <!-- ================================================== -->
  <!-- Rule related additions                             -->
  <!-- ================================================== -->

    <xs:group name="ACTION_BLOCK">
      <!--
    ACTION_BLOCK   ::= 'Do (' (Var (Frame | 'New'))* ACTION+  ')' |
                       'And (' (Atom | Frame)* ')' | Atom | Frame
      -->
      <xs:choice>
       <xs:element  ref="Do"/>
       <xs:element  name="And" type="And-then.type"/>
       <xs:element  ref="Atom"/>
       <xs:element  ref="Frame"/>
      </xs:choice>
    </xs:group>

  <!-- ================================================== -->
  <!-- CoreRule.xsd starts here                           -->
  <!-- ================================================== -->

    <xs:element name="Document">
      <!--
    Document  ::= IRIMETA? 'Document' '(' Base? Prefix* Import* Group? ')'
      -->
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
          <xs:element ref="directive" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element ref="payload" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="directive">
      <!--
    Base and Prefix represented directly in XML
      -->
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Import"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="payload">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Group" type="Group-contents"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="Import">
      <!--
    Import    ::= IRIMETA? 'Import' '(' IRICONST PROFILE? ')'
      -->
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
          <xs:element ref="location"/>
          <xs:element ref="profile" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
```

```
      </xs:complexType>
    </xs:element>

    <xs:element name="location" type="xs:anyURI"/>

    <xs:element name="profile" type="xs:anyURI"/>

    <xs:element name="sentence">
     <xs:complexType>
       <xs:choice>
         <xs:element name="Group" type="Group-contents"/>
         <xs:group ref="RULE"/>
         </xs:choice>
     </xs:complexType>
    </xs:element>

    <xs:element name="Implies">
      <!--
    Implies   ::= IRIMETA? ATOMIC  ':-' FORMULA
      -->
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="IRIMETA" minOccurs="0" maxOccurs="1"/>
          <xs:element ref="if"/>
          <xs:element name="then" type="then-part"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="if">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="FORMULA"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:complexType name="And-then.type">
      <!-- sensitive to then (And) context-->
      <xs:sequence>
        <xs:element name="formula" type="formula-then.type" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="formula-then.type">
      <!-- sensitive to then (And) context-->
      <xs:choice>
       <xs:element  ref="Atom"/>
       <xs:element  ref="Frame"/>
      </xs:choice>
    </xs:complexType>

  </xs:schema>
```

14 Appendix: Change Log (Informative)

This appendix summarizes the changes to this document since its publication as a Recommendation on 22 June 2010:

- Errata as described in the section Proposed Errata for RIF-PRD on the RIF errata (2010-2012) page have been implemented.