Declarative Semantics for the Rule Interchange Format Production Rule Dialect

Carlos Viegas Damásio, José Júlio Alferes, and João Leite

CENTRIA, Dep. Informática, FCT/Universidade Nova de Lisboa, Portugal
cd|jja|jleite@di.fct.unl.pt

Abstract. The Rule Interchange Format Production Rule Dialect (RIF-PRD) is a W3C Recommendation to define production rules for the Semantic Web, whose semantics is defined operationally via labeled terminal transition systems.
In this paper, we introduce a declarative logical characterization of the full default semantics of RIF-PRD based on Answer Set Programming (ASP), including matching, conflict resolution and acting.
Our proposal to the semantics of RIF-PRD enjoys several features. Being based on ASP, it enables a straightforward integration with Logic Programming rule based technology, namely for reasoning and acting with ontologies. Then, its full declarative logical character facilitates the investigation of formal properties of RIF-PRD itself. Furthermore, it turns out that our characterization based on ASP is flexible enough so that new conflict resolution semantics for RIF-PRD can easily be defined and encoded. Finally, it immediately serves as the declarative specification of an implementation, whose prototype we developed.

1 Introduction

In this paper we present a sound and complete declarative semantical characterization of the Production Rule Dialect of the Rule Interchange Format (RIF-PRD) [6] – including matching, conflict resolution and acting – based on Answer-Set Programming [11], accompanied by a prototypical implementation. While contributing to a better understanding of RIF-PRD, our proposal brings greater flexibility to RIF-PRD as it facilitates integration with other rule based technologies and is easily extensible e.g. with other conflict resolution strategies.

The W3C Rule Interchange Format (RIF) exists to enable interoperability among rule languages in general, allowing rules written for one application to be published, shared, and re-used in other applications and other rule engines. Whereas the core dialect of RIF [3] is designed to support the interchange of definite Horn rules without function symbols ("Datalog"), the Production Rule Dialect of RIF (RIF-PRD) [6] extends it to deal with production rules, and is currently a W3C Recommendation. Production rules can be seen as condition-action rules, and are particularly useful to specify behaviors and support the separation of business logic from business objects. According to RIF-PRD, the condition part of production rules is like the condition part of logic rules (as

covered by RIF-Core and its basic logic dialect extension, RIF-BLD [4]). Actions can assert facts, modify facts, retract facts, and have other side-effects, unlike conclusions of logic rules, which contain only a logical statement.

The following are examples of production rules taken from [6], about the status of customers, and corresponding discounts at checkout:
–**Gold rule**: "*Silver*" customers with shopping carts worth at least $2,000 are awarded the "*Gold*" status.
–**Discount rule**: "*Silver*" and "*Gold*" customers are awarded a 5% discount on the total worth of their shopping cart.
–**New customer and widget rule**: "*New*" customers who buy a widget are awarded a 10% discount on the total worth of their shopping carts, but loose any voucher they may have been awarded.
–**Unknown status rule**: a message must be printed, identifying any customer whose status is unknown (that is, neither "*New*", "*Bronze*", "*Silver*" nor "*Gold*"), and the customer must be assigned the status "*New*".

RIF-PRD specifies an abstract syntax and associates the abstract constructs with normative semantics and a normative XML concrete syntax. It also specifies a presentation syntax that provides a more succinct representation of production rules. For example, the third rule above can be represented as follows [6]:

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

The RIF-PRD operational semantics for production rules and rule sets is based on labeled terminal transition systems [14] where state transitions result from executing the action part of instantiated rules, according to the loop: **(Match)**: the rules are instantiated based on the definition of the rule conditions and the current state of the data source; **(Conflict resolution)**: a decision algorithm, often called the conflict resolution strategy, is applied to select which rule instance will be executed; **(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).

An important part of the control loop that governs the semantics concerns the conflict resolution strategy used to select one of the several available rules for execution. Strategies are denoted by keywords (of type `rif:IRI`), that are attached to rule sets permitting that production rule producers and consumers agree on a different semantics. RIF-PRD also prescribes a normative strategy, *forward chaining* denoted by `rif:forwardChaining`, which eliminates rules from a conflict set (a set of applicable rules) based on the following ordered criteria:

1.Refraction: eliminate rules that were already applied and whose conditions for application haven't changed since;
2.Priority: eliminate rules with lower priority;
3.Recency: eliminate rules that have been applicable for longer.

At the end of the application of these criteria, RIF-PRD prescribes that one of the remaining rules be chosen "in some way" (e.g. randomly).

The RIF-PRD W3C Recommendation is a crucial and significant step in standardizing the syntax and semantics of production rules, enabling their interoperability among rule languages in general, and not limited to the Web. However, there are some issues that require further attention, and some steps that need to be taken, in order to provide a better understanding and greater flexibility of RIF-PRD. One important component missing in [6] is a purely logical declarative semantics for RIF-PRD, which would serve as a counterpart to the operational semantics provided. Such a semantics would provide a better understanding and further insights into RIF-PRD, while facilitating the integration of production rules with declarative rules and Logic Programming rule based technology in general, useful e.g. for reasoning and acting with ontologies.

Another issue that needs further attention is that of providing alternatives to the default conflict resolution strategy. Though RIF-PRD foresees the specification of different conflict resolution strategies, there is no indication in [6] as to how such alternative strategies could be specified in a way that facilitates their shared understanding by document producers and consumers. We believe that any such strategy, including the one normatively specified by RIF-PRD, should be defined by a set of rules which precisely defines its meaning. In this case, the keyword for the strategy could be a URI for the set of rules which precisely defines the strategy.

In this paper, we present a sound and complete declarative semantical characterization of RIF-PRD – including matching, conflict resolution and acting – based on Answer-Set Programming (ASP) [11], that addresses these outstanding issues. As suggested by RIF-PRD designers, we assume RIF-Core strong safeness [3] in order to guarantee finite grounding in forward chaining mode.

ASP is a form of declarative programming, similar in syntax to traditional logic programming and close in semantics to non-monotonic logic, that is now widely recognized as a valuable tool for knowledge representation and reasoning. On the one hand, ASP is fully declarative in the sense that the program specifications resemble the problem specifications, the semantics is very intuitive, and there is extensive theoretical work that facilitates proving several properties of answer-set programs. On the other hand, ASP is very expressive, allowing for compact representations of all NP and coNP problems, or even more complex ones if disjunctive programs are used [7]. Other important characteristics of ASP include the use of default negation to allow for reasoning with assumptions and incomplete knowledge, as well as the existence of a number of well studied extensions such as preferences, revision, abduction, etc. More relevant for this work, are the recent results on $MKNF^+$ hybrid knowledge bases where a faithful, tight and flexible integration of description logics and rules has been

achieved [13]. The integration of rules with ontologies is also possible with dl-programs [8]. Finally, there are very efficient ASP solvers available (e.g. Clingo, DLV, Smodels, etc.).

Our proposal enjoys the following features that address the mentioned issues:

- Being based on ASP, it paves the way to a direct integration with Logic Programming based technology, viz. for reasoning and acting with ontologies;
- Being fully declarative, it facilitates the investigation of further formal properties of RIF-PRD, e.g. using the approach followed in [5];
- Enjoying the expressivity of ASP, it is flexible enough so that conflict resolution strategies for RIF-PRD are easily defined and encodable;
- Benefiting from the existence of efficient ASP solvers, it can be directly and efficiently implemented – which we have done using iClingo [9], and is, to the best of our knowledge, the first implementation of RIF-PRD.

The remainder of this paper is structured as follows: in Sect. 2 we review ASP; in Sect. 3 we present a sound and complete translation of RIF-PRD rule sets into ASP; in Sect. 4 we address the specification of conflict resolution strategies in ASP, illustrating with a sound and complete encoding of *forward chaining*, the RIF-PRD normative strategy; we conclude in Sect. 5.

2 Answer Set Programming

In this Section we start by describing the syntax and semantics of Answer-set Programming, before we introduce *iClingo*[9], an incremental answer-set system. We follow the presentation in [9], with some modifications.

The language is built from a set $\mathcal{F}$ of constants and function symbols (including the natural numbers and usual arithmetic operators), a set $\mathcal{V}$ of variable symbols, and a set $\mathcal{P}$ of predicate symbols (including the binary equality and inequality predicates, and ordinary arithmetic comparison operators). We assume that $\mathcal{V}$ contains a distinguished parameter symbol $\boldsymbol{\kappa}$ (varying over natural numbers). The set $\mathcal{T}$ of terms is the smallest set containing $\mathcal{V}$ and all expressions of the form $f(t_1, ..., t_n)$, where $f \in \mathcal{F}$ and $t_i \in \mathcal{T}$ for $0 \leq i \leq n$. The set $\mathcal{A}$ of atoms contains all expressions of the form $p(t_1, ..., t_n)$, where $p \in \mathcal{P}$ and $t_i \in \mathcal{T}$ for $1 \leq i \leq n$. A literal is an atom a or its (default) negation **not** a. Given a set L of literals, let $L^+ = \{a \in \mathcal{A} \mid a \in L\}$ and $L^- = \{a \in \mathcal{A} \mid \textbf{not } a \in L\}$. A logic program over $\mathcal{A}$ is a set of rules of the form $a_0 \leftarrow a_1, ..., a_m, \textbf{not } a_{m+1}, ..., \textbf{not } a_n$, where $a_i \in \mathcal{A}$ for $0 \leq i \leq n$. For a rule r of the form above, let $head(r) = a$ be the head of r, $body(r) = \{a_1, ..., a_m, \textbf{not } a_{m+1}, ..., \textbf{not } a_n\}$ be the body of r, and $atom(r) = \{head(r)\} \cup body(r)^+ \cup body(r)^-$. For a program P, let $head(P) = \{head(r) \mid r \in P\}$ and $atom(P) = \bigcup_{r \in P} atom(r)$. Given an expression $e \in \mathcal{T} \cup \mathcal{A}$, let $var(e)$ denote the set of all variables occurring in e, and given a rule r, let $var(r)$ denote the set of all variables occurring in r. Expression $e \in \mathcal{T} \cup \mathcal{A}$ is ground if $var(e) = \emptyset$. The ground instantiation of a program P is defined as $grd(P) = \{r\theta \mid r \in P, \theta : var(r) \to \mathcal{U}\}$ where $\mathcal{U} = \{t \in \mathcal{T} \mid var(t) = \emptyset\}$. Similarly, $grd(\mathcal{A}) = \{a \in \mathcal{A} \mid var(a) = \emptyset\}$.

A set $M \subseteq grd(\mathcal{A})$ is an answer set [11,1] of a program P over A if M is the $\subseteq$-smallest model of $\{head(r) \leftarrow body(r)^+ \mid r \in grd(P), body(r)^- \cap M = \emptyset\}$. The set of answer-sets of P is denoted by $AS(P)$. The semantics of integrity constraints is given through a program transformation where an integrity constraint of the form $\leftarrow a_1, ..., a_m, \mathbf{not}\ a_{m+1}, ..., \mathbf{not}\ a_n$ is a shorthand for the rule $a' \leftarrow a_1, ..., a_m, \mathbf{not}\ a_{m+1}, ..., \mathbf{not}\ a_n, \mathbf{not}\ a'$ where a' is a new atom.

2.1 *iClingo*

Real-world applications such as planning or model checking include a parameter encoding the size of a solution. In Answer Set Programming (ASP), essentially a propositional formalism, this is dealt with by considering one problem instance after another by gradually increasing the bound on the solution size. In most cases, Answer-Set Programming systems simply produce a ground set of rules for each problem instance, incurring in a high efficiency cost.

iClingo[1][9] is an incremental ASP (iASP) system where both the grounder as well as the solver are implemented in a stateful way, interleaving grounding and solving within incremental computations. Both the grounder and the solver maintain their previous states while increasing an incremental parameter. At each incremental step, the grounder just produces ground rules generated from the current program slice, i.e. generated by instantiating the incremental parameter with the current value. Such ground program slices are gradually passed to the solver that accumulates ground rules and computes answer sets for them.

In the context of *iClingo*, the concept of a *(parametrized) domain description* is introduced, as being a triple $\langle B, S[\boldsymbol{\kappa}], Q[\boldsymbol{\kappa}]\rangle$ of logic programs where $S[\boldsymbol{\kappa}]$ and $Q[\boldsymbol{\kappa}]$ contain a (single) parameter $\boldsymbol{\kappa}$ ranging over the natural numbers. The base program B describes static knowledge, independent of parameter $\boldsymbol{\kappa}$. Program $S[\boldsymbol{\kappa}]$ contains knowledge that accumulates with increasing values of $\boldsymbol{\kappa}$. Program $Q[\boldsymbol{\kappa}]$ contains knowledge that is specific for each value of $\boldsymbol{\kappa}$. Given a *domain description* $\Pi = \langle B, S[\boldsymbol{\kappa}], Q[\boldsymbol{\kappa}]\rangle$ and an integer $i \geq 1$, let $P[i] = B \cup \left(\bigcup_{1 \leq j \leq i} S[j]\right) \cup Q[i]$, and $AS(\Pi_i)$ denote $AS(P[i])$, $min(\Pi)$ denote the minimum integer such that $AS(\Pi_i) \neq \emptyset$, and $AS(\Pi)$ denote $AS\left(\Pi_{min(\Pi)}\right)$. The goal is then to determine $AS(\Pi)$. *iClingo* accepts *domain descriptions* Π[2] and computes $AS(\Pi)$ by incrementally constructing and solving for $P[i]$. Detailed information regarding the implementation of *iClingo* can be found in [9].

---

[1] *iClingo* is part of *Potassco*, a set of tools for Answer Set Programming developed at the University of Potsdam, and available at `http://potassco.sourceforge.net`

[2] Function symbols with non-zero arity may lead to logic programs over an infinite Herbrand base. To maintain decidability at each iteration, it is important to restrict the language to fragments for which finite equivalent ground programs are guaranteed to exist. Level-restricted (or λ-restricted) logic programs [10] constitute such a fragment, where finiteness is guaranteed by the requirement that any variable in a rule be bound to a finite set of ground terms via a predicate not subject to positive recursion through that rule.

3 Fact bases, states, conditions and rules

In this Section we synthetically overview some of the main concepts of the Production Rule dialect of RIF [6] and provide a mapping of RIF-PRD initial states (*fact base*) and rule sets into iASP which is sound and complete wrt. the possible traces of execution of the rules on the initial state. For now, we do not consider the inclusion of a conflict resolution strategy – it will be dealt with in Sect. 4.

RIF-PRD defines rules with action heads for performing changes over a set of facts (i.e. an extensional logic database) dependent on logical conditions over a logical state derived from this set of facts. The underlying logical language is constructed from a first-order alphabet.

3.1 Atomic formulas and conditions

RIF-PRD defines the notion of term as in ASP, except for the introduction of the special list term which, for all purposes in the rest of this paper, can be seen as an ordinary complex term. Terms are used to construct atomic formulas.

Definition 1 (RIF-PRD term and atomic formulas). *A term is either an arbitrary constant* `c`*, an arbitrary variable* `?V`*, a lists of ground terms* `List(`$\mathtt{g}_1$ `...`$\mathtt{g}_n$`)`*, or a (complex) positional terms* `f(`$\mathtt{t}_1$ `...`$\mathtt{t}_n$`)` *formed from a constant* `f` *and a sequence of arbitrary term arguments* $\mathtt{t}_1 \ldots \mathtt{t}_n$ *with* $n \geq 1$*.*

Given arbitrary terms `t`*,* `s`*, and* $\mathtt{p}_i$*,* $\mathtt{t}_i$ *where* $1 \leq i \leq n$*, atomic formulas are ordinary atoms (i.e. positional terms), equality of terms (*`t=s`*), membership of object* `t` *in class* `s` *(*`t#s`*), subclass relation (*`t##s`*), frames (*`t[`$\mathtt{p}_1$`->`$\mathtt{t}_1$ `...` $\mathtt{p}_n$`->`$\mathtt{t}_n$`]`*), or externally defined terms (* `External(t)`*).*

In RIF-PRD, there is no syntactical distinction between positional terms and ordinary atoms. Equality is used to check if two terms are identical, while membership atomic formulas `t#s` are used to represent that the object denoted by term `t` belongs to the class denoted by `s`. A subclass atomic formula `t##s` expresses that `t` is a subclass of `s`. A frame term `t[`$\mathtt{p}_1$`->`$\mathtt{t}_1$ `...` $\mathtt{p}_n$`->`$\mathtt{t}_n$`]` roughly states that the object denoted by term `t` has for each property $\mathtt{p}_i$ the value $\mathtt{t}_i$. Externally defined terms are used for representing built-in functions, e.g. to perform numerical operations. Condition formulas are to be used in the antecedents of production rules to define conditions for their applicability, corresponding syntactically to a fragment of first-order logic without universal quantifiers.

Definition 2 (RIF-PRD condition formulas). *Condition formulas are inductively defined from atomic formulas, conjunction* `And(`ϕ_1 `...`ϕ_n`)` *and disjunction* `Or(`ϕ_1 `...`ϕ_n`)` *of conditional formulas, negation* `Not(`ϕ`)` *or existential quantification* `Exists ?`$\mathtt{v}_1$ `...?`$\mathtt{v}_m$ `(`ϕ`)`*, where* ϕ*,* $\phi_1 \ldots \phi_n$ *are condition formulas and* `?`$\mathtt{v}_1$ `...?`$\mathtt{v}_m$ *are variables.*

3.2 Fact bases and states

The knowledge dynamics is captured by a set of ground atomic formulas – the *fact base* – which changes through the addition and removal of atomic formulas. The execution of a RIF-PRD production rule system starts with an initial fact

base, and proceeds by updating it step by step. At a given step of the execution $\boldsymbol{\kappa}$ a fact base will be encoded in iASP by a set of facts of the form `fact(`φ'`, `$\boldsymbol{\kappa}$`)` where φ' is the translation of the RIF-PRD ground atomic formula φ.

Definition 3 (Translation of atomic formulas). *An atomic RIF-PRD formula φ is translated into the iASP term φ' as follows:*

- *A positional atom, an equality or an externally defined term φ is mapped into itself;*
- *A membership atomic formula* `t#s` *is mapped into term $isa(t, s)$;*
- *A subclass atomic formula* `t##s` *is mapped into term $sub(t, s)$;*
- *A frame atomic formula* `s[p->o]` *is mapped into term $frame(s, p, o)$.*

This representation assumes that a ground frame `t[`$\mathtt{p}_1$`->`$\mathtt{t}_1$ `...` $\mathtt{p}_n$`->`$\mathtt{t}_n$`]` is represented by the set of facts $frame(t, p_1, t_1), \ldots, frame(t, p_n, t_n)$. For simplicity of presentation, externally defined formulas are mapped into themselves. However, a concrete implementation should implement these resorting to their own built-ins; this is ignored in the translation.

Definition 4 (Fact bases translation). *Consider an initial fact base Φ.*

- *Program $\pi_{\texttt{INIT}}(\Phi)$ is formed by $fact(\varphi, 0)$, for each $\varphi \in \Phi$.*
- *Program $\pi_{\texttt{FLUENT}}(\Phi)$ is formed by $fluent(\varphi)$, for each formula φ that may occur in a fact base.*
- *Program $\pi_{\texttt{CHANGE}}[\boldsymbol{\kappa}]$ is formed by the rules:*

$$fact(F, \boldsymbol{\kappa}) \leftarrow fluent(F), fact(F, \boldsymbol{\kappa} - 1), \textbf{\textit{not}}\ retract(F, \boldsymbol{\kappa} - 1).$$
$$fact(F, \boldsymbol{\kappa}) \leftarrow fluent(F), assert(F, \boldsymbol{\kappa} - 1).$$

$\pi_{\texttt{INIT}}$ collects the initial fact base which will be updated using the rules in $\pi_{\texttt{CHANGE}}[\boldsymbol{\kappa}]$. The first rule states that fluents which are not retracted in the previous step remain in the fact base (inertia), while the second states that fluents asserted in the previous step will be added. Notice that the things which can be added or deleted are collected in program $\pi_{\texttt{FLUENT}}$. For simplicity, the definition of predicate $fluent/1$ is extensional but could also be defined intensionally by rules. Also note that by RIF-Core strong safeness at each step there may exist only a finite number of alternatives which can be dealt with in practice. Another essential use of predicate $fluent/1$ is to ground variables in the final iASP domain description.

Definition 5 (States translation). *Program $\pi_{\texttt{STATES}}[\boldsymbol{\kappa}]$ is formed by the rules:*

$$state(F, \boldsymbol{\kappa}) \leftarrow fact(F, \boldsymbol{\kappa}).$$
$$state(F, \boldsymbol{\kappa}) \leftarrow fact(F, 0), \textbf{\textit{not}}\ fluent(F).$$
$$state(isa(O1, C2), \boldsymbol{\kappa}) \leftarrow fluent(isa(O1, C1)), fluent(sub(C1, C2)), state(isa(O1, C1), \boldsymbol{\kappa}), state(sub(C1, C2), \boldsymbol{\kappa}).$$
$$state(sub(C1, C3), \boldsymbol{\kappa}) \leftarrow fluent(sub(C1, C2)), fluent(sub(C2, C3)), state(sub(C1, C2), \boldsymbol{\kappa}), state(sub(C2, C3), \boldsymbol{\kappa}).$$

The first rule includes in the state of step $\boldsymbol{\kappa}$ the fact base of $\boldsymbol{\kappa}$. The second states that any non-fluent (static) fact holding at the initial fact base also holds at step $\boldsymbol{\kappa}$. According to RIF-PRD semantics the set of initial facts can be arbitrarily ground atomic formula but actions are syntactically limited to specific types of

formula (e.g. it is impossible to change subclass atomic formulas). The third rule captures class inheritance while the last one expresses transitivity of the subclass relationship, imposed to any state by the semantics of RIF-PRD.

Conditions are matched to a given state. However, the case of non-atomic formulas introduces extra complexity:

Definition 6 (Conditions translation). *Let Φ be an arbitrary condition formula and $\boldsymbol{\kappa}$ an execution step. Define condition iASP formula Φ' and program $\pi^{\Phi}_{\texttt{COND}}[\boldsymbol{\kappa}]$ inductively as follows:*

- *If Φ is an atomic formula φ then $\Phi'[\boldsymbol{\kappa}] = state(\varphi', \boldsymbol{\kappa})$ and $\pi^{\Phi}_{\texttt{COND}}[\boldsymbol{\kappa}] = \{\}$;*
- *If $\Phi = \texttt{And(}\phi_1 \ \ldots \phi_n\texttt{)}$ then $\Phi'[\boldsymbol{\kappa}] = (\phi'_1, \ldots, \phi'_n)$ and $\pi^{\Phi}_{\texttt{COND}}[\boldsymbol{\kappa}] = \bigcup_{1 \leq i \leq n} \pi^{\phi_i}_{\texttt{COND}}[\boldsymbol{\kappa}]$;*
- *If $\Phi = \texttt{Or(}\phi_1 \ \ldots \phi_n\texttt{)}$ then $\Phi'[\boldsymbol{\kappa}] = or_{\Phi}(X_1, \ldots, X_m, \boldsymbol{\kappa})$ where $?X_1, \ldots ?X_m$, are the free variables of Φ and or_{Φ} is a new predicate symbol, and $\pi^{\Phi}_{\texttt{COND}}[\boldsymbol{\kappa}] = \bigcup_{1 \leq i \leq n} \left(\pi^{\phi_i}_{\texttt{COND}}[\boldsymbol{\kappa}] \cup \{or_{\Phi}(X_1, \ldots, X_m, \boldsymbol{\kappa}) \leftarrow \phi'_i[\boldsymbol{\kappa}]\} \right)$;*
- *If $\Phi = \texttt{Exists ?V}_1 \ \ldots\texttt{?V}_n \ (\phi)$ then $\Phi'[\boldsymbol{\kappa}] = exists_{\Phi}(X_1, \ldots, X_m, \boldsymbol{\kappa})$ where $?X_1, \ldots ?X_m$, are the free variables of Φ and $exists_{\Phi}$ is a new predicate symbol, and $\pi^{\Phi}_{\texttt{COND}}[\boldsymbol{\kappa}] = \pi^{\phi}_{\texttt{COND}}[\boldsymbol{\kappa}] \cup \{exists_{\Phi}(X_1, \ldots, X_m, \boldsymbol{\kappa}) \leftarrow \phi'[\boldsymbol{\kappa}]\}$;*
- *If $\Phi = \texttt{Not(}\phi\texttt{)}$ then $\Phi'[\boldsymbol{\kappa}] = \textbf{\textit{not}}\ arg_{\Phi}(X_1, \ldots, X_m, \boldsymbol{\kappa})$ where $?X_1, \ldots ?X_m$, are the free variables of Φ and arg_{Φ} is a new predicate symbol, and $\pi^{\Phi}_{\texttt{COND}}[\boldsymbol{\kappa}] = \pi^{\phi}_{\texttt{COND}}[\boldsymbol{\kappa}] \cup \{arg_{\Phi}(X_1, \ldots, X_m, \boldsymbol{\kappa}) \leftarrow \phi'[\boldsymbol{\kappa}]\}$;*

Basically, this transformation applies Lloyd-Topor's transformation [12] to obtain the corresponding normal rules capturing the conditional formula, taking into account what is true in the current step. Mark that both a (conjunctive) goal $\Phi'[\boldsymbol{\kappa}]$ and a program $\pi^{\Phi}_{\texttt{COND}}[\boldsymbol{\kappa}]$ is returned for each condition formula Φ. Additional details and justification of this process can be found in [1].

3.3 Actions and rules

The RIF-PRD language defines several atomic actions for updating the fact base, and these will be used to define the effects of RIF-PRD production rules.

Definition 7 (RIF-PRD atomic actions). *An atomic action is a simple construct that represents an atomic transaction.*

1. *Assert fact: If Φ is a positional atom, a frame or a membership atomic formula in the RIF-PRD condition language, then* `Assert(`Φ`)` *is an atomic action.*
2. *Retract fact: If Φ is a positional atom or a frame in the RIF-PRD condition language, then* `Retract(`Φ`)` *is an atomic 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.*
4. *Retract object: If* `t` *is a term in the RIF-PRD condition language, then* `Retract(t)` *is an atomic action.*
5. *Execute: if Φ is a positional atom in the RIF-PRD condition language, then Execute(Φ) is an atomic action.*

The arguments of the action are dubbed the target of the action.

The effects of RIF-PRD atomic actions are captured by our translation using the following iASP rules.

Definition 8 (Effects of actions). *Program* $\pi_{\texttt{ACTIONS}}[\boldsymbol{\kappa}]$ *is:*
$assert(F, \boldsymbol{\kappa}) \leftarrow action(assert(F), \boldsymbol{\kappa}).$
$retract(F, \boldsymbol{\kappa}) \leftarrow action(retract(F), \boldsymbol{\kappa}).$

$retract(isa(O, C), \boldsymbol{\kappa}) \leftarrow action(retract\_object(O), \boldsymbol{\kappa}), fact(isa(O, C), \boldsymbol{\kappa}).$
$retract(frame(O, S, V), \boldsymbol{\kappa}) \leftarrow action(retract\_object(O), \boldsymbol{\kappa}), fact(frame(O, S, V), \boldsymbol{\kappa}).$

$retract(frame(O, S, V), \boldsymbol{\kappa}) \leftarrow action(retract\_slots(O, S), \boldsymbol{\kappa}), fact(frame(O, S, V), \boldsymbol{\kappa}).$

Note that the *execute* actions do not have an effect in the fact base and should be interpreted externally. The first two rules of program $\pi_{\texttt{ACTIONS}}[\boldsymbol{\kappa}]$ apply when an assert (resp. retract) action occurs at step $\boldsymbol{\kappa}$, whose effects in the fact base have been defined previously in program $\pi_{\texttt{CHANGE}}$. The next two rules translate a retract object action into a set of simultaneous retracts, while the last one takes care of the retract all slots action. The interaction of rules with the fact base is performed via the *action*/2 predicate to be defined subsequently.

Actions are combined sequentially into action blocks, allowing binding patterns for binding variables occurring in the actions. Additionally, RIF-PRD defines a compound `Modify` frame action which can be substituted by a sequence of a retract all slot values followed by an assert; it is assumed that such a replacement has been performed.

Definition 9 (Action variable declaration and action blocks). *An action variable declaration is a pair* `(?V b)` *where* `?V` *is a variable and* `b` *is binding having one of the forms:* `New()` *for generating a new identifier, or a frame* `o[s->?V]` *where* `o` *and* `s` *are ground terms. If* $\texttt{(?V}_1\ \texttt{b}_1\texttt{)}, \ldots, \texttt{(?V}_n\ b_n\texttt{)}$, $n \geq 0$, *are action variable declarations, and if* $\texttt{a}_1, \ldots, \texttt{a}_m$, $m \geq 1$, *are simple actions, then* $\texttt{Do((?V}_1\ \texttt{b}_1\texttt{)} \ldots\texttt{(?V}_n\ \texttt{b}_n\texttt{)}\ \texttt{a}_1 \ldots\texttt{a}_m\texttt{)}$ *denotes an action block.*

Finally, the RIF Production Rules are captured by the following definition. Mark that well-formedness conditions are imposed to rules and conditions, which we are ignoring in this summary presentation.

Definition 10 (RIF production rule). *A rule can be one of:*
- *An (unconditional) action block* $\texttt{Do((?V}_1\ \texttt{b}_1\texttt{)}\ldots\texttt{(?V}_n\ \texttt{b}_n\texttt{)}\ \texttt{a}_1\ldots\texttt{a}_m\ \texttt{)}$.
- *A conditional action block* $\texttt{If}\ \Phi\ \texttt{Then Do((?V}_1\ b_1\texttt{)}\ldots\texttt{(?V}_n\ b_n\texttt{)}\ \texttt{a}_1\ldots\texttt{a}_m\texttt{)}$, *where* Φ *is a condition formula and the conclusion is an action block.*
- *A quantified rule* $\texttt{Forall ?V}_1\ldots\texttt{?V}_n\ \texttt{such that (p}_1\ldots\texttt{p}_m\texttt{) (r)}$, *where each* $\texttt{p}_i$ *is a conditional formula (a pattern) and* r *is a RIF Production rule.*

Without loss of generality we assume that quantified rules have only one level of universal quantification, i.e. the rule r is limited to be a conditional action block since it is always possible to write quantified rules in this way, by variable renaming and appending patterns.

Definition 11 (Translation of a RIF production rule). *Let* ri *be a RIF production rule and let* id *be a unique identifier assigned to that rule (i.e. its "name"). Program* $\pi^{ri}_{\texttt{RULE}}[\boldsymbol{\kappa}]$ *is constructed as follows:*

– *If ri is* `Do((?V1 b1)...(?Vn bn) a1...am )` *then include in* $\pi^{ri}_{\texttt{RULE}}[\boldsymbol{\kappa}]$ *the fact* $fireable(rule(id, subs), \boldsymbol{\kappa})$.
– *If ri is* `If` Φ `Then Do( (?V1 b1) ...(?Vn bn) a1 ...am )` *then include* $\pi^{\Phi}_{\texttt{COND}}[\boldsymbol{\kappa}]$ *in* $\pi^{ri}_{\texttt{RULE}}[\boldsymbol{\kappa}]$, *and the following rule where* $?X_1, \ldots, ?X_l$ *are the free variables of ri:* $fireable(rule(id, subs(X_1, \ldots, X_l), \boldsymbol{\kappa})) \leftarrow \Phi'$.
– *If ri is* `Forall ?V1...?Vn such that (p1...pm) (If` Φ `Then Do(`B`))` *then treat this as the conditional action block* `If And(p1...pm` Φ`) Then Do(`B`)`.

Additionally, from the action block `Do((?V1 b1)...(?Vn bn) a1...am )` *in the conclusion of ri add to program* $\pi^{ri}_{\texttt{RULE}}[\boldsymbol{\kappa}]$, *for each* $1 \leq j \leq m$, *the rule:*

$$action(a'_j, \boldsymbol{\kappa} + j) \leftarrow instance(id, subs(V_1, \ldots, V_n, X_1, \ldots, X_l), \boldsymbol{\kappa}).$$

Finally, include in $\pi^{ri}_{\texttt{RULE}}[\boldsymbol{\kappa}]$ *the rule below, where* $bind_{v_i}$ *is* $state(frame(o, s, V_i), \boldsymbol{\kappa})$ *if* $b_i =$ `o[s->?Vi]`. *Otherwise* $b_i =$ `New()`, *and let* $bind_{v_i}$ *be* $V_i = obj(id, i, \boldsymbol{\kappa})$ *with* obj *an arbitrary but fixed constant symbol.*

$$instance(id, subs(V_1, \ldots, V_n, X_1, \ldots, X_l), \boldsymbol{\kappa}) \leftarrow picked(rule(id, subs(X_1, \ldots, X_l)), \boldsymbol{\kappa}), \\ bind_{V_1}, \ldots, bind_{V_n}.$$

Predicate $fireable(rule(id, subs(\ldots)), \boldsymbol{\kappa})$ holds in step $\boldsymbol{\kappa}$ whenever the rule identified by id has a condition true, and thus may be applied. The complex term $sub(\ldots)$ keeps the substitution of variables for which the condition matches state $\boldsymbol{\kappa}$, and is also used to distinguish between different matching instances of the same rule. If the rule is picked for execution then $picked(rule(id, subs(\ldots)), \boldsymbol{\kappa})$ will hold and consequently action a_j will be executed in step $k + j$ with the action instance (i.e. substitution of variables) collected in auxiliary predicate $instance/3$.

Example 1. Consider the rule presented in the introduction of this paper. Its encoding into iASP as constructed by $\pi_{\texttt{RULE}}$ transformation is shown below, following the usual answer-set convention of variables beginning with upper-case and, to simplify the presentation, the constants belonging to namespace $ex1$ are represented using CURIE notation:

$$fireable(rule(widget, subs(Cust)), \boldsymbol{\kappa}) \leftarrow state(isa(Cust, \text{ex1:Customer}), \boldsymbol{\kappa}), \\ state(frame(Cust, \text{status}, \text{“New”}), \boldsymbol{\kappa}), exists_1(Cust, \boldsymbol{\kappa}).$$
$$exists_1(Cust, \boldsymbol{\kappa}) \leftarrow state(frame(Cust, \text{shoppingCart}, Cart), \boldsymbol{\kappa}), \\ state(frame(Cart, \text{containsItem}, Item), \boldsymbol{\kappa}), state(isa(Item, \text{ex1:Widget}), \boldsymbol{\kappa}).$$

$$action(retract(frame(Cust, \text{voucher}, Voucher)), \boldsymbol{\kappa} + 1) \leftarrow \\ instance(widget, subs(Cust, S, Val, Voucher), \boldsymbol{\kappa}).$$
$$action(retract\_object(Voucher)), \boldsymbol{\kappa} + 2) \leftarrow \\ instance(widget, subs(Cust, S, Val, Voucher), \boldsymbol{\kappa}).$$
$$action(retract\_slots(S, \text{value}), \boldsymbol{\kappa} + 3) \leftarrow \\ instance(widget, subs(Cust, S, Val, Voucher), \boldsymbol{\kappa}).$$
$$action(assert(frame(S, \text{value}, Val * 90/100)), \boldsymbol{\kappa} + 4) \leftarrow \\ instance(widget, subs(Cust, S, Val, Voucher), \boldsymbol{\kappa}).$$

$$instance(widget, subs(Cust, S, Val, Voucher), \boldsymbol{\kappa}) \leftarrow \\ picked(rule(widget, subs(Cust), \boldsymbol{\kappa}), state(frame(Cust, \text{shoppingCart}, S), \boldsymbol{\kappa}), \\ state(frame(S, \text{value}, Val), \boldsymbol{\kappa}), state(frame(Cust, \text{voucher}, Voucher), \boldsymbol{\kappa}).$$

It is clear from the example that the fireable conditions are not yet connected to the rules performing the actions, which will be tackled next. First, it is necessary to pick one rule for execution from the pickable ones (i.e. the ones which fire and can be executed). This is straightforward to encode:

Definition 12 (Pick rule). *Program* $\pi_{\texttt{PICK}}[\boldsymbol{\kappa}]$ *is formed by:*

$picked(Rule, \boldsymbol{\kappa}) \leftarrow pickable(Rule, \boldsymbol{\kappa}), \textbf{\textit{not}}\ picked\_other(Rule, \boldsymbol{\kappa}), \textbf{\textit{not}}\ transitional(k).$
$picked\_other(Rule, \boldsymbol{\kappa}) \leftarrow pickable(Other, \boldsymbol{\kappa}), pickable(Rule, \boldsymbol{\kappa}), Rule! = Other, picked(Other, \boldsymbol{\kappa}).$
$picked(\boldsymbol{\kappa}) \leftarrow picked(Rule, \boldsymbol{\kappa}).$
$transitional(\boldsymbol{\kappa}) \leftarrow action(A, \boldsymbol{\kappa}).$

The execution of RIF-PRD proceeds by first picking one rule, then performing its actions sequentially, then picking another rule, performing its actions, etc.... The steps in which the fact base is being updated are dubbed "transitional" in the RIF-PRD recommendation. The first two rules in $\pi_{\texttt{PICK}}[\boldsymbol{\kappa}]$ choose exactly one alternative (i.e. a rule) from the pickable rules, when $\boldsymbol{\kappa}$ is not a transitional step. If no strategy is defined, the general operational semantics prescribes that all fireable rules are pickable, which can be captured by the program $\pi_{\texttt{ONE}}[\boldsymbol{\kappa}]$ with the single rule $pickable(Rule, \boldsymbol{\kappa}) \leftarrow fireable(Rule, \boldsymbol{\kappa})$. Computation terminates in a non-transitional step where no rule is picked. This is captured by $\pi_{\texttt{HALT}}[\boldsymbol{\kappa}]$, which ends our translation of a RIF-PRD rule set, summarized in Def. 14.

Definition 13 (Termination). *Program* $\pi_{\texttt{HALT}}[\boldsymbol{\kappa}]$ *is defined by:*

$\leftarrow \textbf{\textit{not}}\ final(\boldsymbol{\kappa}).$
$final(\boldsymbol{\kappa}) \leftarrow \textbf{\textit{not}}\ transitional(\boldsymbol{\kappa}), \textbf{\textit{not}}\ picked(\boldsymbol{\kappa}).$

Definition 14 (Rule set translation). *The translation of a RIF-PRD rule set RS with initial fact base w and set of fluents F is the iASP domain specification* $\mathit{\Pi}_{\texttt{RULESET}}(RS, w) = \langle B_{\texttt{RS}}(w), S_{\texttt{RS}}(RS)\,[\boldsymbol{\kappa}]\,, Q_{\texttt{RS}}\,[\boldsymbol{\kappa}]\rangle$ *where:*

$$
\begin{array}{ll}
B_{\texttt{RS}}(w) & = \pi_{\texttt{INIT}}(w) \cup \pi_{\texttt{FLUENT}}(F) \\
S_{\texttt{RS}}(RS)[\boldsymbol{\kappa}] & = \pi_{\texttt{CHANGE}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{STATES}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{ACTION}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{PICK}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{ONE}}[\boldsymbol{\kappa}] \cup \bigcup_{ri \in RS} \pi^{ri}_{\texttt{RULE}}[\boldsymbol{\kappa}] \\
Q_{\texttt{RS}}\,[\boldsymbol{\kappa}] & = \pi_{\texttt{HALT}}[\boldsymbol{\kappa}]
\end{array}
$$

An advantage of this encoding is that all possible "traces" of execution can be generated by the iASP system, where each different trace corresponds to an answer set. Formally[3]:

Theorem 1 (Correctness of translation). *Let RS be a rule set and w an initial fact base. Then*[4]*:*
Soundness*: If* $M \in AS\,(\mathit{\Pi}_{\texttt{RULESET}}(RS, w)_n)$ *and* $(c_1, \ldots, c_m)$ *is the increasing sequence of integers such that* $transitional(c_j) \notin M, 1 \leq j \leq m$*, then, for every* $i : 1 \leq i \leq m - 1$ $(State^i(M), Picked^i(M), State^{i+1}(M)) \in \rightarrow_{PRD}$*,*

[3] Lack of space prevents us from presenting the proofs of theorems.

[4] $\rightarrow_{PRD}$ stands for the transition system which serves as the basis for defining the semantics of RIF-PRD, $ConflictSet(RS, s_i)$ the set of all applicable rules in state s_i. Lack of space prevents us from presenting the semantics of RIF-PRD, which is available in [6].

where $State^i(M)$ *denotes the set of formulae* Φ *such that* $state(\Phi', c_i) \in M$ *and* $Picked^i(M)$ *the name of the (only) rule* R *such that* $picked(R, c_i) \in M$.
Completeness*: If* $(s_1, \ldots, s_m)$ *is a sequence of non-transitional states such that* $w = s_1$*, and for each pair* (s_i, s_{i+1}) *there exists a rule* $r \in ConflictSet(RS, s_i)$ *such that* $(s_i, r, s_{i+1}) \in \rightarrow_{PRD}$*, then, there exists* $M \in AS\,(\Pi_{\texttt{RULESET}}(RS, w)_n)$ *for some* $n \geq m$ *such that the sequence of integers* $(c_1, \ldots, c_m)$*, constructed from* M *as above, is such that* $State^i(M) = s_i$*, for all* $1 \leq i \leq m$.

4 Conflict resolution strategies

For selecting (ideally one) among these possible executions (or traces), as mentioned in the Introduction RIF-PRD foresees the existence of *conflict resolution strategies*. Each of the strategies is denoted by a keyword (of type `rif:IRI`), that is attached to the rule set. The current version of RIF-PRD prescribes a normative strategy, *forward chaining*, denoted by `rif:forwardChaining`, and anticipates the specification of additional keywords, each corresponding to an additional strategy for selecting rules in conflict. Furthermore, it also allows for the inclusion of other keywords, not specified in the RIF-PRD specification, in which case it is the responsibility of the producers and consumers of those documents to agree on the strategy denoted by the keywords.

Our stance is that any conflict resolution strategy should be defined by a set of rules, including those normatively specified by RIF-PRD, which precisely defines its meaning. In this case, the keyword for the strategy could be a URI for the set of rules which precisely defines the strategy. In this section we show that iASP, along with the translation defined in the previous section, is expressive enough to specify conflict resolution strategies. In particular, we show how to specify conflict resolution strategies, and illustrate by precisely characterizing the `rif:forwardChaining` strategy.

4.1 General definition of strategies

A conflict resolution strategy is defined in [6] by an algorithm that, in a series of steps, selects from the set of all fireable rules in some state, a subset of (pickable) rules from which one is finally picked for execution. For example, the `rif:forwardChaining` strategy can be summarized as the following algorithm:

Definition 15 (Forward chaining algorithm). *Given a conflict set (i.e. a set of fireable rules):*

1. *Remove all rules which where previously applied and, since their last application, the conditions that made them applicable haven't changed* – refraction.
2. *The remaining rules are ordered by decreasing* priority*, and only the rule instances with the highest priority are kept. Recall that in RIF-PRD every rule is assigned a priority which is a natural number.*
3. *The remaining rules are ordered by decreasing* recency*, and only the most recent rule instances are kept. Here, a rule is more recent than another if it is (consecutively) applicable for less prior states than the other.*

Each of these steps applies one strategy element (refraction, priority and recency). In [6], a fourth (tie-break) element is considered, to be applied after these 3, stating that one of the remaining rules should be picked in some "implementation specific way" [6]. Here we do not need to consider this last step. On the one hand, the translation is such that each answer set is guaranteed to reflect the application of a single rule at each state. On the other hand, the existence of more than one answer set reflects the fact that there may be more than one pickable rule at some state after the application of these 3 strategy elements. As a result of the translation, each answer set encodes one possible sequence of application of rules, and one can either consider all resulting answer-sets, or arbitrarily pick one of them.

For encoding such a strategy in a set of iASP rules, to be added to the domain description obtained from the translation of the previous section, we first need to replace the rule of $\pi_{\texttt{ONE}}[\boldsymbol{\kappa}]$ which specified that all fireable rules are pickable, by a set of general rules allowing for restrictions on pickable rules. Accordingly, a rule is pickable if it is fireable and it is not rejected by one of the strategy elements:

Definition 16 (Strategy). *Program* $\pi_{\texttt{STRATEGY}}[\boldsymbol{\kappa}]$ *is formed by the rules*

$$pickable(Rule, \boldsymbol{\kappa}) \leftarrow fireable(Rule, \boldsymbol{\kappa}), \textbf{\textit{not}}\ rejected(Rule, \boldsymbol{\kappa}).$$
$$rejected(Rule, \boldsymbol{\kappa}) \leftarrow rejected(Rule, \boldsymbol{\kappa}, S), st\_element(S).$$

Note that, without any defined strategy, $\pi_{\texttt{STRATEGY}}[\boldsymbol{\kappa}]$ has exactly the same effect as $\pi_{\texttt{ONE}}[\boldsymbol{\kappa}]$. In fact, if there are no rules for neither $rejected/3$ nor $st\_element/1$, $rejected(Rule, \boldsymbol{\kappa})$ is false in all answer-sets for all rules and $\boldsymbol{\kappa}$, and so pickable is true for all fireable rules, as is the case in $\pi_{\texttt{ONE}}[\boldsymbol{\kappa}]$.

Strategy elements are identified by a name. Then, for each strategy, facts to specify the order of application of the elements must be added. For example, for `rif:forwardChaining` the specification of the order of elements is as follows:

$$st\_element(refraction, 1). \quad st\_element(priority, 2). \quad st\_element(recency, 3).$$

For referring to the element without its order of application, the following rule is also needed $st\_element(S) \leftarrow st\_element(S, \_)$.

In general, for the definition of conflict resolution strategies, a predicate is needed to indicate whether a rule is active when a given strategy element is being applied. For example, in `rif:forwardChaining`, if a rule is removed by refraction, then that rule should no longer be available for consideration (i.e. active) when considering the priority-element. The specification of this predicate is quite straightforward: a rule is inactive if there is a strategy element prior in the application order which rejected it, and active otherwise.

Definition 17 (Active Rules). *Program* $\pi_{\texttt{ACTIVE}}[\boldsymbol{\kappa}]$ *is defined by*

$$inactive(Rule, \boldsymbol{\kappa}, N) \leftarrow st\_element(\_, N), st\_element(S, N1), N1 < N, rejected(Rule, \boldsymbol{\kappa}, S).$$
$$active(Rule, \boldsymbol{\kappa}, N) \leftarrow \textbf{\textit{not}}\ inactive(Rule, \boldsymbol{\kappa}, N), st\_element(\_, N).$$

The iASP domain description associated with a RIF-PRD rule set becomes:

Definition 18 (RIF-PRD domain description). *The RIF-PRD iASP domain description of a rule set RS with initial fact base w and fluents F is $\Pi_{\texttt{RS}}(RS, w) = \langle B_{\texttt{RS}}(w), S_{\texttt{RS}}(RS)\,[\boldsymbol{\kappa}]\,, Q_{\texttt{RS}}\,[\boldsymbol{\kappa}]\rangle$ with $B_{\texttt{RS}}(w)$ and $Q_{\texttt{RS}}$ as in Def. 14, and*

$$S_{\texttt{RS}}(RS)[\boldsymbol{\kappa}] = \pi_{\texttt{CHANGE}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{STATES}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{ACTION}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{PICK}}[\boldsymbol{\kappa}] \cup \\ \cup \bigcup_{ri \in RS} \pi^{ri}_{\texttt{RULE}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{STRATEGY}}[\boldsymbol{\kappa}] \cup \pi_{\texttt{ACTIVE}}[\boldsymbol{\kappa}]$$

Theorem 2. *Theorem 1 holds if we replace $\Pi_{\texttt{RULESET}}(RS, w)$ with $\Pi_{\texttt{RS}}(RS, w)$.*

4.2 Defining one specific strategy

To completely specify one conflict resolution strategy, we add facts defining the strategy elements and their application order (as above for `rif:forwardChaining`) and define, for each element, which rules are rejected. Below we show how this can be done for each of the elements in the `rif:forwardChaining` algorithm.

Refraction Once a rule is picked at some state, then it is rejected by refraction from that state onwards, for as long as the rule remains fireable. The test for the rule being fireable is only done in states when the system is not being updated.

$$\begin{aligned} &rejected(Rule, \boldsymbol{\kappa}, refraction) \leftarrow fireable(Rule, \boldsymbol{\kappa}), picked(Rule, \boldsymbol{\kappa} - 1).\\ &rejected(Rule, \boldsymbol{\kappa}, refraction) \leftarrow rejected(Rule, \boldsymbol{\kappa} - 1, refraction), transitional(\boldsymbol{\kappa}).\\ &rejected(Rule, \boldsymbol{\kappa}, refraction) \leftarrow fireable(Rule, \boldsymbol{\kappa}), rejected(Rule, \boldsymbol{\kappa} - 1, refraction)\\ &\qquad\qquad\qquad\qquad \textbf{not } transitional(\boldsymbol{\kappa}). \end{aligned}$$

Priority All rules for which there is another (different) active fireable rule with a strictly higher priority should be rejected. We do not need to test that rejected rules are active (i.e. not rejected by a previous strategy element), since according to $\pi_{\texttt{STRATEGY}}[\boldsymbol{\kappa}]$ a rejected rule is never pickable.

$$\begin{aligned} rejected(rule(Id, Var), \boldsymbol{\kappa}, priority) \leftarrow\ & fireable(rule(Id, Var), \boldsymbol{\kappa}),\\ & fireable(rule(Id2, Var2), \boldsymbol{\kappa}), Id! = Id2, priority(Id, P), priority(Id2, P2),\\ & P < P2, active(rule(Id2, Var2), \boldsymbol{\kappa}, N), strategy(priority, N). \end{aligned}$$

Recency A rule is rejected if there is a more recent one also active and fireable. We use an auxiliary predicate ($recency/3$) that, for each rule instance and state $\boldsymbol{\kappa}$, determines the number of consecutive states before $\boldsymbol{\kappa}$ that the instance has been fireable. Then, a rule is rejected if there is another one which is more recent. Predicate $state(K)$ is just used for grounding, and is true for any state K.

$$\begin{aligned} rejected(rule(Id, Var), \boldsymbol{\kappa}, recency) \leftarrow\ & fireable(Rule, \boldsymbol{\kappa}),\ fireable(Other, \boldsymbol{\kappa}),\\ & Rule\ != \ Other, recency(Rule, TR, \boldsymbol{\kappa}), recency(Other, TO, \boldsymbol{\kappa}), TO < TR,\\ & state(TR), state(TO), active(Other, \boldsymbol{\kappa}, N),\ st\_element(recency, N). \end{aligned}$$

$$\begin{aligned} &recency(Rule, \boldsymbol{\kappa}, \boldsymbol{\kappa}) \leftarrow fireable(Rule, \boldsymbol{\kappa}), not\ fireable(Rule, \boldsymbol{\kappa} - 1).\\ &recency(Rule, K, \boldsymbol{\kappa}) \leftarrow recency(Rule, K, \boldsymbol{\kappa} - 1), transitional(\boldsymbol{\kappa}), state(K).\\ &recency(Rule, K, \boldsymbol{\kappa}) \leftarrow fireable(Rule, \boldsymbol{\kappa}), recency(Rule, K, \boldsymbol{\kappa} - 1),\\ &\qquad\qquad\qquad\qquad \textbf{not } transitional(\boldsymbol{\kappa}), state(K). \end{aligned}$$

The set composed by all rules described in this subsection is meant to encode the `rif:forwardChaining`, and we denote it by $\pi_{\texttt{rif:fc}}[\boldsymbol{\kappa}]$.

The next theorem shows in which terms the encoding is correct with respect to the RIF-PRD `rif:forwardChaining` as described in [6]:

Theorem 3 (Correctness for `rif:forwardChaining`). *Let RS be a rule set, w an initial fact base, and $\langle B_{\texttt{RS}}(w), S_{\texttt{RS}}(RS)\,[\boldsymbol{\kappa}]\,, Q_{\texttt{RS}}\,[\boldsymbol{\kappa}]\rangle$ the corresponding iASP domain description as in Def. 18. Let LS be the* `rif:forwardChaining` *strategy of definition 15, and H the halting test that halts whenever no rule is picked. Let $\Pi_{\texttt{rif:fc}}(RS, w) = \langle B_{\texttt{RS}}(w), S_{\texttt{RS}}(RS)\,[\boldsymbol{\kappa}] \cup \pi_{\texttt{rif:fc}}[\boldsymbol{\kappa}], Q_{\texttt{RS}}\,[\boldsymbol{\kappa}]\rangle$. Then*[5]*:*
Soundness*: if $M \in AS\,(\Pi_{\texttt{rif:fc}}(RS, w))$, then there exists a state s_f such that $Eval(RS, LS, H, w) \rightarrow^*_{PRD} s_f$ and where s_f is the set of all formulae Φ such that $state(\Phi', min\,(\Pi_{\texttt{rif:fc}}(RS, w))) \in M$.*
Completeness*: if $Eval(RS, LS, H, w) \rightarrow^*_{PRD} s_f$, then there exists an M such that $M \in AS\,(\Pi_{\texttt{rif:fc}}(RS, w))$ and $\forall \Phi \in s_f, state(\Phi', min\,(\Pi_{\texttt{rif:fc}}(RS, w))) \in M$.*

One can impose other conflict resolution strategies, by specifying different rejection rules. For example, `rif:forwardChaining` behaves in a depth-first manner, in that it always selects the rule that has been more recently applied. Imposing a breadth-first strategy can be accomplished by simply changing "$TO < TR$" into "$TO > TR$" in the rule defining the rejection by recency, thus obtaining $\pi_{\texttt{st:breadth}}[\boldsymbol{\kappa}]$. Also note that `rif:forwardChaining` does not behave in a purely depth-first manner since it only applies recency after removing rules with less priority. For a strategy where a depth-first behavior is more important than complying with the declared priority of rules, one can simply change the facts that impose the order in the application of strategy elements, e.g. by including the facts $st\_element(priority, 3)$ and $st\_element(recency, 2)$ instead.

5 Conclusions

In this paper, we presented a declarative logical characterization of RIF-PRD through a sound and complete transformation into ASP, which can be seen as an equivalent alternative to the transitional semantics proposed in [6], giving further insights into RIF-PRD and providing for an immediate implementation using *iASP*, which we have developed using iClingo[9]. This transformation considers not only the RIF-PRD rule sets and their transitions, but also the conflict resolution strategies which are essential to select among applicable rules. We have illustrated how the default normative strategy – *forward chaining* – is encodable in ASP, and have shown that ASP provides an appropriate language in which to precisely define alternative non-standard conflict resolution strategies, which are also foreseen in [6], facilitating their development and unambiguous sharing, due to the simple, expressive and well known semantics of ASP. The

[5] $Eval(RS, LS, H, w)$ is the input function of the RIF-PRD production rule system that is responsible for choosing one among the rules in the conflict set and for the halting conditions. $\rightarrow^*_{PRD}$ is the transitive closure of $\rightarrow_{PRD}$.

work in [2] uses the Situation Calculus, although without handling the idiosyncrasies of RIF-PRD. A Situation Calculus based approach like the one in [2] could have been followed, although with extra complexity introduced by the situation terms which would not be easily handled by answer set solvers. A critique of the Situation Calculus is made in [5], where it is shown how to capture the semantics of rule production systems in μ-calculus and FPL. This work captures a result equivalent to our Theorem 1, thus not handling other conflict resolution strategies. We expect to use the work of [5] to study the formal properties of our translation. An implementation using an external DL reasoner is underway to assess the practicality of our approach, namely by comparing with more traditional approaches like CLIPS or JESS.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. C. Baral and J. Lobo. Characterizing production systems using logic programming and situation calculus. Available from `http://www.public.asu.edu/?cbaral/papers/char-prod-systems.ps`.
3. H. Boley, G. Hallmark, M. Kifer, A. Paschke, A. Polleres, and D. Reynolds, editors. *RIF Core Dialect*. W3C Recommendation, 22 June 2010. `http://www.w3.org/TR/2010/REC-rif-core-20100622/`.
4. H. Boley and M. Kifer, editors. *RIF Basic Logic Dialect*. W3C Recommendation, 22 June 2010. `http://www.w3.org/TR/2010/REC-rif-bld-20100622/`.
5. Jos de Bruijn and Martín Rezk. A logic based approach to the static analysis of production systems. In *Proc. of Web Reasoning and Rule Systems RR 2009*, volume 5837 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2009.
6. C. de Sainte Marie, G. Hallmark, and A. Paschke, editors. *RIF Production Rule Dialect*. W3C Recommendation, 22 June 2010. `http://www.w3.org/TR/2010/REC-rif-prd-20100622/`.
7. T. Eiter and G. Gottlob. Expressiveness of stable model semantics for disjunctive logic programs with functions. *Journal of Logic Programming*, 33(2):167–178, 1997.
8. Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495 – 1539, 2008.
9. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental asp solver. In *Procs. of ICLP 2008*, volume 5366 of *LNCS*, pages 190–205. Springer, 2008.
10. M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In *Procs. of LPNMR 2007*, volume 4483 of *LNCS*, pages 266–271. Springer, 2007.
11. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Procs. of ICLP 1990*, pages 579–597. MIT Press, 1990.
12. J. W. Lloyd and R. W. Topor. Making prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.
13. Boris Motik and Riccardo Rosati. Reconciling description logics and rules. *J. ACM*, 57(5), 2010.
14. Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and ALgebraic Programming*, 60-61:17–139, 2004.