Frédéric Daniel Jacinto Veiga

Licenciado em Engenharia Informática

Implementation of the RIF-PRD

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Carlos Viegas Damásio, Prof. Associado, Universidade Nova de Lisboa

Júri:

Presidente:	Nuno Correia
Arguente:	Salvador Pinto Abreu
Vogal:	Carlos Viegas Damásio

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2011

Implementation of the RIF-PRD

Agradecimentos

I thank Prof. Dr. Carlos Damásio for the opportunity of doing this work, always having the utmost predisposition for any doubt along the way and providing an extremely amicable and motivating work environment. For the endless support through these months I thank my family, my friends and Ana.

Resumo

O Rule Interchange Format (RIF) é uma recomendação da W3C que define um conjunto de dialectos para promover a interoperabilidade entre sistemas de regras. A necessidade destes dialectos proveio de um crescente número de linguagens de regras (lógicas ou de produção), com algumas tão específicas que o intercâmbio de regras entre sistemas diferentes é uma tarefa quase impossível. A criação de uma linguagem de regras convencional não seria bem aceite, como tal, a W3C optou por desenvolver o RIF, com o objectivo de fornecer uma solução para o intercâmbio de regras entre diferentes sistemas.

Um dos dialectos do RIF é o Rule Interchange Format Production Rule Dialect (RIF-PRD), que define uma linguagem de regras de produção orientada para a Semantic Web.

Até à data actual, não é conhecida nenhuma implementação completa de RIF-PRD, mas uma primeira especificação declarativa completa do RIF-PRD baseada em programação por conjuntos de resposta foi proposta na nona conferência internacional de Semantic Web (ISWC2010).

Nesta dissertação implementamos um motor de RIF-PRD baseando-nos nessa especificação, e desenvolveremos outras duas implementações, uma utilizando o sistema de regras de produção Jess, e outra recorrendo ao sistema de programação em lógica XSB. Após terminadas as três implementações, foi feita uma comparação entre elas, com o intuito de verificar se há benefícios em utilizar programação por conjuntos de resposta para implementar RIF-PRD, e concomitantemente detectar eventuais limitações na utilização de programação por conjuntos de resposta.

Palavras-chave: RIF-PRD, regras de produção, Semantic Web, programação por conjuntos de resposta, sistema de regras, Jess.

Abstract

The Rule Interchange Format (RIF) is a W3C Recommendation that defines a set of dialects that aim to provide interoperability between rule systems. The need for this set of dialects arose from the increasing number of rule languages available, many of them being so specific that exchanging rules between different rule systems is a hard if not even impossible task. Since defining a standard for a rule language would not be well accepted, W3C opted by creating RIF, with the sole purpose of providing a way to exchange rules between different rule systems.

One of these dialects is the Rule Interchange Format Production Rule Dialect (RIF-PRD), that defines a production rules language oriented towards the Semantic Web.

To the date there are no complete implementations of RIF-PRD, but a first fully declarative specification of RIF-PRD based in Answer Set Programming has been proposed in the 9th International Semantic Web Conference (ISWC 2010).

In this thesis we implement RIF-PRD following that specification and develop two other implementations, one using the production rule system Jess, and a third one using the logic programming system XSB. A comparison between the three implementations follows1, allowing to check on the one hand if there are benefits in using Answer Set Programming for implementing RIF-PRD and on the other hand also perceive if there are limitations to using it.

Keywords: RIF-PRD, production rules, Semantic Web, Answer Set Programming, rule system, Jess.

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introduction

1.1 Motivation

With the recent growth of the *Semantic Web*[1] popularity, the number of rule systems and rule languages also rose, mainly due to each rule system defining its own specific rule language. Since a standard rule language would be virtually inconceivable, the World Wide Web Consortium[2] (W3C) decided to create the Rule Interchange Format (RIF) [BK09] , a way to interoperate different rule languages.

RIF is intended to become a family of languages that are formed by dialects, as far as needed to support many different paradigms, e.g., logic programming and production rules. These dialects have a common core, and are intended to share as much syntax and semantics as possible. RIF possesses a XML based syntax and aims to provide interoperability between rule systems.

Logic programming and production rules paradigms both use rules as the main construct of their languages. The main difference lies that in logic programming, the conclusion of rules is a logical statement, whereas in production rules the effects of a fired rule might be the assertion, retraction and modification of facts, or even have other side effects.

One of the RIF dialects is the Rule Interchange Format Production Rule Dialect (RIF-PRD, section 2.2) [HPdSM10], that is oriented towards production rule languages.

As RIF-PRD is still a recent language, the number of known implementations is still scarce. In fact, there are only four known RIF-PRD implementations. Two of them are

[1] http://www.w3.org/standards/semanticweb/
[2] http://www.w3.org

mentioned in the RIF-PRD W3C website[3], one based in WebSphere ILOG JRules[4] by Changhai Ke (ILOG, an IBM company) and the other based in Oracle Business Rules (OBR)[5] by Gary Hallmark (Oracle). As far as we know, to the date, both are yet to be available. The two other implementations were developed by Pierre de Leusse, parte of the Yield RIF project, one based in Jess and the other in Drools[6]. The two partial implementations of RIF-PRD are available here[7].

To implement RIF-PRD, a non-monotonic language is required, since RIF-PRD itself is non-monotonic. With that in mind, a first complete RIF-PRD declarative specification based on Answer Set Programming (ASP) (section 2.3) has been presented [DAL10]. As there are still no complete RIF-PRD implementations, the work in [DAL10] is a guide for a first declarative implementation of RIF-PRD.

1.2 Objectives

The main objective of this thesis will be the implementation of RIF-PRD in three different paradigms: Answer Set Programming (iclingo), Production Rules (Jess) and Logic Programming (XSB), and consequently, a performance comparison between the implementations will be performed.

Regarding the ASP implementation, there's the objective of following the complete declarative specification presented in [DAL10] and assess it. The Jess implementation is expected to have a performance viable for real life applications, and finally, the main goal of the XSB implementation is to perform a first comparison of the incremental tabling mechanism of XSB with the incremental system of iClingo.

1.3 Contributions

The major contributions of this thesis are:

- Three RIF-PRD implementations, using different paradigms (Production Rules, ASP and Logic Programming). Since there are few known implementations of RIF-PRD, this is the major contribution of this work.
- Translation of RIF-PRD documents into Answer Set Programming (ASP) and Jess Markup Language (JessML, see section 3.2.2) using Extensible Stylesheet Language Transformations (XSLT).
- Evaluation of the feasibility/limitations of ASP systems to implement RIF-PRD.
- Benchmarking of the three implementations.

[3] http://www.w3.org/2005/rules/wiki/Implementations
[4] http://www-01.ibm.com/software/integration/business-rule-management/jrules/
[5] http://www.oracle.com/technetwork/middleware/business-rules/overview/index.html
[6] http://www.jboss.org/drools
[7] http://yieldrif.appspot.com/author/Pierre

1.4 Organization

We start in Chapter 2 by presenting an overview of production systems, RIF-PRD and answer set programming. Then, in Chapter 3, we will explore several tools for both production systems and answer set programming, that we will need for the implementation of RIF-PRD.

In Chapter 4 we will detail of our implementations, also presenting enhancements performed to the declarative specification of RIF-PRD [DAL10]. Our implementations will be tested in Chapter 5, where we present both the benchmark results and detail the limitations we encountered. Finally, we conclude in Chapter 6.

2

State of The Art

In this chapter we present an overview of the state of the art relevant to this dissertation. We start by presenting production systems, which will be used in one of the planned implementations (Production Rules paradigm). Plus, a brief analysis of RETE (the basic inference algorithm used by several production rules systems) is provided.

As the main focus of this dissertation is RIF-PRD, in section 2.2 we present its features, syntax and semantics. The ASP paradigm is also covered in this chapter, where we analyse the ASP language.

2.1 Production Systems

Production Systems (or production rules systems) are software systems that basically consist in a set of rules specifying behaviour. Each production system has a working memory, that stores facts and rules, and a rule interpreter. The rules (or productions) are *If . . . Then* statements, where the *If* part is the precondition necessary for the rule to fire, and the *Then* part is the action that is the consequence of a triggered rule. The difference between a triggered rule and a fired (or picked) rule is that, a triggered rule is a rule whose preconditions are met but the action part has not yet been executed, whereas a fired rule is a rule whose action part was executed.

The typical preconditions of a rule are simply facts, or conditions between facts, i.e., if the precondition of a rule is a fact, then when that fact is present in the working memory the rule will be triggered. The effects of a fired production rule are usually the assertion, retraction or modification of existing and/or new facts.

Example 1 (A basic production rule in the Jess rule language)**.**

```
(deftemplate car
(slot color)
(slot id)
(slot year (type INTEGER)))

(deftemplate newCar
(slot id))

(defrule r1
 ?c <- (car {year == 2010})
=>(assert (newCar (id ?c.id)))
  (modify ?c (color "black")))
```

In Jess, to declare a fact, first we need to declare a template that defines the arguments (slots) of that fact. In this example, *r1* is activated if there is a car in the fact base that is from the year 2010. The effects of *r1* are the assertion of a new fact called `newCar` that has the id of the car, and the modification of the car color to black.

The Object Management Group (OMG) is an international organization whose mission is to develop enterprise integration standards that provide real-world value[1]. The OMG Production Rule Representation Specification[2] defines production rule execution as a 3 phase process: Match, Conflict resolution and Act.

In the Match phase, rules are matched against the the current state of the knowledge base (hereafter we will refer to knowledge base as KB). For matching the production rules against the working memory, there are several strategies one can use. Some are naive, e.g. trying all rules in sequence stopping at the first match, whereas others are more complex, as the commonly used RETE [For90] algorithm. The RETE algorithm sacrifices memory for performance increase, and though that might be an issue in larger KB's, the RETE algorithm performance is several times better than naive implementations.

Regarding the Conflict Resolution phase, it only occurs if after the match phase there is more than one triggered rule (note that some production systems use a system of rule priority to reduce the number of these cases). The reason why it is important to decide which rule shall fire is that there might be mutually exclusive rules in conflict.

Generally each production system has more than one strategy that can be chosen to determine which rule will fire. These strategies are usually quite simple, being related to basic search algorithms applied to the order in which the rules were activated. Some of these strategies are: depth strategy, which uses the depth first search algorithm, and the breadth strategy, which is related to the breadth first search algorithm.

Regarding the Act phase, after the rules have been chosen and fired, the changes made by those rules will be applied to the working memory, where the changes can be

[1] http://www.omg.org/gettingstarted/gettingstartedindex.htm
[2] http://www.omg.org/spec/PRR/1.0/

the addition, deletion or modification of some elements. Since it is assumed that there will be deletion of elements, it is implied that the production system used in this work is non-monotonic, still, there are some production systems which are strictly monotonic.

2.1.1 RETE Algorithm

The RETE algorithm [For90] is an efficient pattern matching algorithm designed by Charles L. Forgy of Carnegie Mellon University for implementing production systems. It was first published in 1974, and nowadays it is used by many popular production systems, including CLIPS [Gia] and Jess [FH]. The RETE algorithm aims to reduce complexity of the matching process by avoiding to match every rule condition against every assertion in the working memory.

RETE is a network of nodes (a directed acyclic graph) that is divided into two parts, the alpha network and beta network. In the alpha network, the working memory elements (WMEs) are matched against each rule condition separately, and the results are stored in the alpha memory nodes. The beta network essentially makes the union of the matches stored in the alpha memory nodes through beta memory nodes and beta join nodes. The beta join nodes will perform the union between a alpha memory node and a beta memory node. The results will be stored in a child beta memory node. When a root-to-leaf path has facts that support all the conditions then the rule that corresponds to that path is triggered.

Example 2 (RETE computation example)**.**

Rule 1	Rule 2	Facts	
(R1 (car ?X) (year ?X 2010) (id ?X ?Y) => (assert (newCar(?Y))))	(R2 (car ?X) (year ?X 2007) (id ?X ?Y) => (assert (oldCar(?Y))))	f1: (car auto1) f2: (year auto1 2010) f3: (id auto1 123)	f4: (car auto2) f5: (year auto2 2007) f6: (id auto2 456)

In this example we have two rules (*r1* and *r2)*). The conditions of *r1* and *r2* are that a car $?X$ is from the year 2010 (resp. 2007) and that $?Y$ is the identifier of X. The effects of *r1* and *r2* is the assertion of a fact *newCar(?Y)* (resp. *oldCar(?Y)*) to the fact base. RETE will start by creating the network using the conditions of the rules. In the alpha network the facts will be matched with the conditions and the matches for $C1 = \{f1, f4\}$, $C2 = \{f2\}$, $C3 = \{f5\}$ and $C4 = \{f3, f6\}$. $B2$ is the set of facts from $C1$ and $C2$ that support the conditions of rule $R1$, therefore $B2 = \{f1, f2\}$. Analogously to $B2$, $B3$ is the set of facts from $C1$ and $C3$ that support the conditions of $R2$, hence $B3 = \{f4, f5\}$. Finally since $C4 = \{f3, f6\}$ both rules will have all their conditions met and both will be activated.

This is a very simplified explanation of RETE. A more complete, clear and easy to understand explanation is given in [Doo95].

The major downside to RETE is the considerable memory it uses. In larger problems

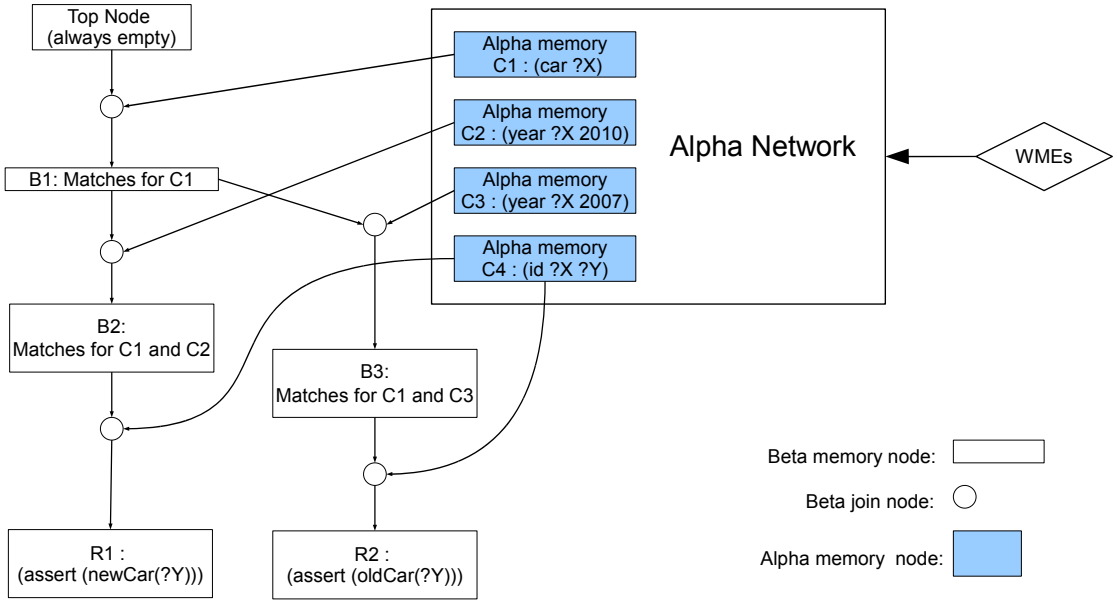


Figura 2.1: RETE Network (based in a [Doo95] example).

this might cause memory consumption issues. Still, the performance gain comparatively to naive strategies make RETE a better solution than those strategies.

The RETE algorithm only supports forward chaining, though a new version of RETE, RETE II has an implementation of backward chaining[3]. A comparative analysis[4] made to RETE and RETE II, have shown that RETE II can be several times faster than RETE (from a few seconds to a difference of hours).

2.2 RIF-PRD

2.2.1 RIF-PRD introduction

The Rule Interchange Format Production Rule Dialect (RIF-PRD)[HPdSM10] is an interchange format designed to become a standard for rule interchange between production systems. Until now, commonly the production systems would use an internal rule language to define rules. This means that sharing rules between production systems was virtually impossible due to syntax incompatibility, and of course possible semantic differences.

RIF-PRD inherits every feature of RIF-Core[5], as RIF-CORE is the base of all RIF dialects. RIF-Core corresponds to the language of definite Horn rules without function symbols with a standard first-order semantics [PRH+10].

[3] http://www.pst.com/clips_r2.htm
[4] http://www.pst.com/benchcr2.htm
[5] http://www.w3.org/TR/rif-core/

The main concept of Semantic Web is the ontology. An ontology represents knowledge of a domain in the form of concepts (classes), with relationships connecting those objects. This set of concepts, and the describable relationships among them, are reflected in the representational vocabulary with which a knowledge-based program represents knowledge. Thus, we can describe the ontology of a program by defining a set of representational terms. In such an ontology, definitions associate the names of entities in the universe of discourse (e.g., classes, relations, functions, or other concepts) with human-readable text describing what the names are meant to denote, and formal axioms that constrain the interpretation and well-formed use of these terms [Gru93].

Two of the currently popular ontology languages are RDF/S and OWL. Both are W3C recommendations for representing information in the web. According to [PRH+10], RIF-Core is compatible with RDF and OWL [Bru10], defining a syntax and semantics for integrated RIF-Core/RDF and RIF-Core/OWL languages. This feature is shared by RIF-PRD, thus making RIF-PRD a Web-aware language. In particular, it allows to equip RDF with production rules.

RIF-PRD specifies an abstract syntax that shares features with concrete production rule languages, and associates the abstract constructs with normative semantics and a normative XML concrete syntax. RIF-PRD also features a presentation syntax easier to read and understand than the equivalent abstract syntax, however it is not normative, but will be used here to illustrate the main features of the language.

Example 3 (Presentation syntax of a RIF-PRD rule)**.**

A car is painted black and listed as a new car if it is from the year 2010.

```
(* Annotation : rule r1 *)
Forall ?automobile ?Y (
If And (?automobile#Car
                ?automobile[Year->2010]
                ?automobile[Id->?Y]
               )
Then Do( Assert(newCar(?Y)) Modify( ?automobile[Color->"black"])))
```

Rules in RIF-PRD are composed by two parts: a *left hand side* which contains the conditions that are required for the rule to fire, and a *right hand side*, containing the actions the rule will perform when fired. These actions may result in changes on the KB (state of facts), instead of simple logical conclusion typical of logic rules.

The *left hand side* of a rule is composed by atomic formulas and formulas. If all the conditions of the rule are met, then the rule is activated (eligible to be fired).

The *right hand side* of a rule is an action block, which is a non-empty sequence of actions which might update the state of facts. It might also include action variable declarations, constructs that declare variables local to the action block.

In **Example 3**, the *left hand side* is composed by an *And* formula which contains three atomic formulas. The *right hand side* is formed by an atomic action (*Assert*) and a

composite action (*Modify*).

We proceed by presenting formally the RIF-PRD language. According to [HPdSM10], the alphabet of RIF-PRD consists in:

- a countably infinite set of constant symbols *Const*,
- a countably infinite set of variables symbols *Vars* (disjoint from *Const*),
- syntactic constructs, such as lists, function calls, relations (including equality, class membership and subclass relations), logical operators (conjunctions, disjunctions and negation) and existential conditions.

Due to its purpose, RIF-PRD language is composed by a set of constructs that try to cover most of the usual operations in rule production rules languages, such as terms, atomic formulas, formulas, atomic actions and composite actions.

Definition 1 (RIF-PRD Terms [HPdSM10]). *A RIF-PRD term is either a:*

- *simple term, where t is a simple term if* $t \in Const$ *or* $t \in Vars$*;*
- *list term,* $List(x_1 \dots x_n)$ *where* $n \geq 0$ *and* $x_1 \dots x_n$ *are ground terms;*
- *positional term,* $t(x_1 \dots x_n)$ *where* $t \in Const$ *and* $x_1 \dots x_n$*,* $n \geq 0$ *are terms, t is a function symbol and* $x_1 \dots x_n$ *are the arguments.*

In **Example 3**, the RIF-PRD term `?automobile` is a variable term.

The syntax of the RIF-PRD atomic formulas extends the corresponding notion of first order logic.

Definition 2 (RIF-PRD Atomic Formulas [HPdSM10]). *A RIF-PRD atomic formula is either:*

- *an atom* $t(x_1 \dots x_n)$*, where* $t \in Const$ *and* $x_1 \dots x_n$*,* $n \geq 0$ *are terms;*
- *an equality atomic formula* $t = s$*, where t and s are terms;*
- *a class membership atomic formula* $t\#s$*, where t and s are terms (representing that object t belongs to class s);*
- *a subclass atomic formula* $t\#\#s$*, where t and s are terms (representing that class t is a subclass of class s);*
- *a frame atomic formula* $t[p_1 \rightarrow v_1, \dots, p_n \rightarrow v_n]$*, where t is a term,* p_i *and* v_i *are terms. t is the object of the frame,* p_i *are the property or attribute names and* v_i *are the property or attribute values (an attribute/value pair may be called a slot);*
- *an externally defined atomic formula External(t), where t is an atom.*

For instance, in **Example 3**, `newCar(?Y)` is an atom and `?automobile#Car` is a class membership atomic formula, where `?automobile` is a variable term standing for an object of class `Car`. The formula `?automobile[Year->2010]` is a frame atomic formula.

2.2.2 RIF-PRD conditions

A RIF-PRD formula is a composite truth-valued construct. Constants, variables, lists and functions are *not* formulas. RIF-PRD formulas are either condition formulas or ground formulas. RIF-PRD formulas are exclusively used in the *left hand side* of the rules (with the exception of *And* formulas).

Definition 3 (RIF-PRD Formulas [HPdSM10])**.** *A RIF-PRD Condition formula is either a:*

- *atomic formula.*
- *conjunction. If $f_1, \ldots, f_n$, $n \geq 0$ are condition formulas, then so is And($f_1, \ldots, f_n$), called a conjunctive formula (And() is allowed and treated as a tautology, i.e., a formula that is always true). Conjunction in RIF-PRD might also be used to compose an unconditional action block formed by atoms and frames;*
- *disjunction. If $f_1, \ldots, f_n$, $n \geq 0$ are condition formulas, then so is Or($f_1, \ldots, f_n$), called a disjunctive formula (Or() is allowed and treated as a contradiction, i.e., a formula that is always false);*
- *negation. If f is a condition formula, then so is Not(f), called a negative formula;*
- *existential. If f is a condition formula and $?V_1, \ldots, ?V_n$, $n \geq 0$ are variables, then Exists $?V_1, \ldots, ?V_n$ (f) is an existential formula.*

In **Example 3**, there is a conjunctive formula `And(...)`, that holds whenever an `?automobile` of class `Car` from the year 2010 (`?automobile[Year->2010]`) has an id
(`?automobile[Id->?Y]`).

A condition formula *f* is a *ground formula* if it does not possess any variable term.

RIF-PRD provides the notion of well-formedness of formulas, that is a set of syntactic rules that all formulas must follow. A definition of well-formedness is present in [HPdSM10], which limits the occurrence of constants in formulas. We do not present this definition for the sake of simplicity. The set of all well-formed condition formulas is called **RIF-PRD condition language**.

The semantics of RIF-PRD is defined over a state of facts, *i.e.* a set of ground atomic formulas. The ground atomic formulas are obtained by matching substitution. Informally, a matching substitution of a given condition formula γ, assigns ground terms to every variable in γ, such that the resulting transformation exists in the current state of facts.

2.2.3 RIF-PRD state of the fact base

According to [HPdSM10], a state of the fact base, KB_ψ, is associated to every set of ground atomic formulas, ψ, that satisfies, at least, the following conditions, for all triple of constants c_1, c_2, c_3:

1. if $c_1\#\#c_2 \in \psi$ and $c_2\#\#c_3 \in \psi$, then $c_1\#\#c_3 \in \psi$;
2. and if $c_1\#c_2 \in \psi$ and $c_2\#\#c_3 \in \psi$, then $c_1\#c_3 \in \psi$.

Condition 1 captures transitivity of the subclass relation, while condition 2 formalizes class inheritance.

KB_ψ is represented by ψ; or, equivalently, by the conjunction of all the ground atomic formulas in ψ.

Each ground atomic formula in ψ represents a single fact, and, often, the ground atomic formulas, themselves, are called facts, as well.

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

- 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, such that its evaluation yields true;
- it is a compound condition formula: conjunction, disjunction, negation or existential; it is evaluated based on the truth value of its atomic components, with respect to the set of facts KB_ψ.

2.2.4 RIF-PRD actions

The RIF-PRD *right hand side* of a rule is composed by a set of actions. There are two RIF-PRD actions: atomic actions and compound actions.

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 after each atomic action is performed.

Definition 4 (RIF-PRD actions [HPdSM10])**.** *RIF-PRD atomic actions are:*

- $Assert(x)$*, where* x *is either an atom, a membership atomic formula or a frame. This action asserts* x *in the KB.*
- $Retract$*. In RIF-PRD there are three types of retract actions:*
  - $Retract(x)$*, where* x *is an atom or a frame. It removes* x *from the KB.*
  - $Retract(t)$*, where* t *is a term. It removes all frames whose object is* t *and all membership atomic formulas containing* t*.*

– *$Retract(t\ t_1)$, where t and t_1 are terms. This action removes all frames with object t and property t_1.*

There is but one RIF-PRD compound action, $Modify(x)$. It can be represented as the sequence of two atomic actions. Assume $x = o[s \leftarrow v]$, then $Modify(x)$ will do $Retract(o\ s)$ followed by $Assert(x)$.

In **Example 4**, the effects of the fired rule are an atomic action `Assert(newCar(123))`, that asserts a new fact into the fact base and a compound action `Modify ?car1[Color->"black"]`, that changes the property color of the frame object `?car1` to black.

It is important to keep in mind that the notion of well-formedness also extends to actions, action blocks and action variable declarations. The proper definition for these concepts is provided in [HPdSM10].

Definition 5 (RIF-PRD operational semantics of actions [HPdSM10])**.** *The semantics of the ground atomic actions in the RIF-PRD action language determines a relation, called the RIF-PRD transition relation: $\rightarrow_{RIF-PRD} \subseteq W \times L \times W$, 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 RIF-PRD atomic actions is specified by the transition relation $\rightarrow_{RIF-PRD} \subseteq W \times L \times W$. $(w, \alpha, w') \in \rightarrow_{RIF-PRD}$ if and only if $w \in W$, $w' \in W$, α is a ground atomic action, and one of the following is true:

1. *α is $Assert(\psi)$, where ψ is a ground atomic formula, and $w' = w \cup \{\psi\}$;*
2. *α is $Retract(\psi)$, where ψ is a ground atomic formula, and $w' = w \setminus \{\psi\}$;*
3. *α is $Retract(o\ s)$, where o and s are terms, and $w' = w \setminus \{o[s \rightarrow v]$ for all the values of $v\}$;*
4. *α is $Retract(o)$, where o is a constant, and $w' = w \setminus \{o[s \rightarrow v]$, $o\#c$ for all the values of terms s, v and $c\}$;*
5. *α is $Execute(\psi)$, where ψ is a ground atomic builtin action, and $w' = w$.*

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

Intuitively the semantics of the above actions is as follows:

- $Assert(\psi)$ asserts the fact ψ in the KB.
- $Retract(\psi)$ removes from the KB the fact ψ.
- $Retract(o\ s)$ removes from the KB every frame with object o and property s.

- $Retract(o)$ removes from the KB every frame with object o and every class membership formula with instance o.
- $Execute(\psi)$ executes a builtin action ψ, not causing any change to the KB.

Definition 6 (Action variable declaration and action block [DAL10]). *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 $o[s \rightarrow ?V]$ where o and s are ground terms. If $(?V_1\ b_1), \ldots, (?V_n\ b_n)$, $n \geq 0$, are action variable declarations, and if $a_1, \ldots, a_m$, $m \geq 1$, are simple actions, then $Do((?V_1\ b_1) \ldots (?V_n\ b_n)a_1 \ldots a_m)$ denotes an action block.*

The notion of well-formedness defined for condition formulas also extends to action blocks. Furthermore, for an action block to be well-formed:

- 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.

2.2.5 RIF-PRD rules

A RIF-PRD program is formed by a set of rules that are able to modify the KB. A RIF-PRD rule is formed by two parts: a *left hand side*, with the conditions of the rule, and a *right hand side*, that has the actions of that rule.

Definition 7 (RIF-PRD rules [HPdSM10]). *A rule r is a RIF-PRD rule if it is either:*

- *an unconditional action block, that is a rule with an empty left hand side;*
- *a conditional action block, that is a rule of the form $If\ \alpha\ Then\ \beta$, where α is a RIF-PRD condition language formula and β is a well-formed action block.*
- *a rule with variable declaration, that is a rule of the form*

$$Forall ?v_1 \ldots ?v_n \text{ such that } (p_1 \ldots p_m)(rule)$$

$?v_1 \ldots ?v_n$, $n \geq 1$, are variables, $?p_1 \ldots ?p_m$, $m \geq 1$, are condition formulas (also called patterns) and rule is a RIF-PRD rule.

The rule presented in the **Example 3** is an example of a rule with variable declaration, as it declare two variables, `?automobile` and `?Y`.

Mark that the notion of well-formedness also applies to RIF-PRD rules. Another important notion referred in [HPdSM10] is safeness. 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. The notion of safeness also extends to variables and groups. We assume all rules are safe, unless stated otherwise.

2.2.6 RIF-PRD system and transitional states

As RIF-PRD is a production rule oriented dialect, one of the basic concepts we need to grasp is the concept of state.

Definition 8 (RIF-PRD state [HPdSM10]). *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, $s_1, \ldots, s_n$, $n \geq 0$, 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.*

As said before, RIF-PRD is a stateful language. Then, the operational semantics of RIF-PRD can be defined as follows.

Definition 9 (RIF-PRD System [HPdSM10]).

- *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:*
  - *$(facts(s), a, facts(s')) \in \rightarrow_{RIF-PRD} *$, where $\rightarrow_{RIF-PRD} *$ denotes the transitive closure of the transition relation $\rightarrow_{RIF-PRD}$ that is determined by the specification of the semantics of the atomic actions supported by RIF-PRD;*
  - *$a = actions(picked(s))$;*
- *$T \subseteq S$, a set of final system states.*

Example 4 (Production Rule System behaviour)**.**

Consider $facts(s_0)$ an initial fact base and the presented rule in example 3:

$facts(s_0) = \{car1[Id \rightarrow 123 \quad Year \rightarrow 2010 \quad Color \rightarrow white]$,
$car2[Id \rightarrow 456 \quad Year \rightarrow 2007 \quad Color \rightarrow red]\}$

Rule r_1 fires and the changes the fact base into:

$facts(s_1) = \{car1[Id \rightarrow 123 \quad Year \rightarrow 2010 \quad Color \rightarrow black]$,
$car2[Id \rightarrow 456 \quad Year \rightarrow 2007 \quad Color \rightarrow red]$, $newCar(123)\}$

Definition 10 (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, $facts(s) \in W$, where R denotes the set of all the rules and W denotes the set of all the states of the fact base.*

The rules in the conflict set may be also called fireable rules.

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. The group construct is used to group sets of rules and to associate them with a conflict resolution strategy. Many production rule systems use priorities associated with rules as part of their conflict resolution strategy. In RIF-PRD, the group is also used to carry the priority information that may be associated with the interchanged rules.

Definition 11 (RIF-PRD groups [HPdSM10])**.** *A group consists of a, possibly empty, set of rules and groups, associated with a resolution conflict strategy and, a priority. If strategy is an IRI that identifies a conflict resolution strategy, if priority is an integer, and if each r_{gj}, $0 \leq j \leq n$, is either a rule or a group, then any of the following represents a group:*

- *Group $(r_{g0} \ldots r_{gn})$, $n \geq 0$;*
- *Group strategy $(r_{g0} \ldots r_{gn})$, $n \geq 0$;*
- *Group priority $(r_{g0} \ldots r_{gn})$, $n \geq 0$;*
- *Group strategy priority $(r_{g0} \ldots r_{gn})$, $n \geq 0$.*

If a conflict resolution strategy is not explicitly attached to a group, the strategy defaults to rif:forwardChaining (see section 2.2.7 for an explanation how rif:forwardChaining operates).

2.2.7 RIF-PRD conflict resolution

In RIF-PRD, conflict resolution strategies are denoted by a keyword of the type (`rif:IRI`) that is attached to the rule set. The current version of RIF-PRD provides a default strategy 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 [DAL10].

The *rif:forwardChaining* strategy is formed by three strategy elements: refraction, priority and recency. Note that there is a specific order to apply these three elements to the conflict set, and if a rule of the conflict set has been discarded in a given element, the rule does not belong to the conflict set in the following strategy elements.

The first element to be applied to the conflict set is refraction, that discards a given rule ϕ in state κ, if ϕ has already been fired and the conditions that made ϕ eligible to fire in κ still hold.

The second element is priority, that returns the set of pickable rules with the highest priority. The last element is recency, that returns the set of rules that have been activated more recently. If at this point there is still more than one rule in the conflict set, then one rule is chosen in some way internally determined by the engine.

The semantics of RIF-PRD is quite complex but a declarative logical characterization of the full semantics of RIF-PRD based on Answer Set Programming (ASP, see section 2.3) was proposed in [DAL10], presented at the 9th International Semantic Web Conference (ISWC 2010)[6]. This article will be the core to one of the RIF-PRD implementations proposed in this dissertation, for this we need to overview the basic ASP concepts.

2.3 Answer Set Programming

Answer Set Programming (ASP) is a declarative programming paradigm oriented towards solving combinatorial declarative search problems.

The goal of ASP is to obtain solutions of a given logic program in the form of answer sets, by reducing search problems into computing stable models (The notion of a stable model was first introduced by Gelfond & Lifschitz [GL88]). In order to generate stables models, answer set solvers are used.

ASP programs consist of rules that look like Prolog rules, but the computational mechanisms used in ASP are different: they are based on the ideas that have led to the creation of fast satisfiability solvers for propositional logic [Lif08].

Definition 12 (ASP Literals)**.** *As stated previously, the ASP syntax is similar to Prolog. Its main constructs are:*

- *a countably infinite set of constant symbols Const, generally assumed to begin with a lower case symbol and/or are double-quoted;*

[6] http://iswc2010.semanticweb.org/

- *a countably infinite set of variable symbols Vars (disjoint from Const), generally assumed to begin with upper case symbol;*
- *Terms: constants, variables and $p(t_1 \dots t_n)$, where $p \in Const$ and $t_1 \dots t_n$ are terms;*
- *Ground Terms: constants, and $p(t_1 \dots t_n)$, where $p \in Const$ and $t_1 \dots t_n$ are terms; without variables;*
- *Atoms: $p(t_1 \dots t_n)$ where p ∈ Const and $t_1 \dots t_n$ are terms;*
- *Literals: atom a or its complementary ¬a (where ¬ represents explicit negation);*
- *NAF-Literals: a literal l or not l (negation as failure).*

An ASP program is a set of rules. These rules have the form $l_0 \leftarrow l_1, \dots, l_n, not\ c_1, \dots, not\ c_n$ where $l_0 \in$ *Literals* is called left hand side (or head) of the rule, and $l_1, \dots, l_n, not\ c_1, \dots, not\ c_n \in$ *NAF-Literals* is the right hand side (or body) of the rule.

Definition 13 (ASP rules)**.** *ASP rules can be either:*

- *Facts, that are rules that no body and are assumed to be true (e.g. man(john));*
- *Constraints, that are rules that have no head. Their function is to discard irrelevant models (e.g. ← man(john), not(human(john)). In this example, all models where john is a man and is not a human are discarded;*
- *Normal Rules, that are rules that have the form $l_0 \leftarrow l_1 \dots l_n$, (e.g. man(X) ← human(X), male(X).), where $l_1, \dots, l_n \in NAF - Literals$.*

ASP solvers usually support more kinds of rules, namely choice rules. These rules are an extension that allow to make the programs more declarative and optimize the computation of answer sets.

Informally [You], a stable model M of a ground program P is a set of ground atoms such that:

- Every rule is satisfied, i.e., for any rule in P

  $l_0 \leftarrow l_1, \dots, l_m, not\ c_1, \dots, not\ c_n$, if l_i are satisfied in M and not c_i are also satisfied (*not* c_j is satisfied if c_j is not in M), then l_0 is in M.

- Every *literal* $\in M$ can be derived from a rule by non-circular reasoning.

Definition 14 (Model and least model)**.** *If a ground program does not have negation of failure in its body, then it has always an least model. A model L is the least model of program P if, for every model M of P, $L \subseteq M$.*

Definition 15 (Stable model [Syr]). *Let M be a set of atoms from program P. The reduct P^M w.r.t. M is obtained by deleting from P each rule that has a negative literal not l in its body when $l \in M$; and then by removing all negative literals in the bodies of remaining rules. If the least model of P^M coincides with M, then M is a stable model of P [Syr].*

Example 5 (Models of an ASP program).

Logic Program:	Grounding:	Stable models:
p(a). r(X) :- p(X), not(q(X)). q(X) :- p(X), not(r(X)).	p(a). r(a) :- p(a), not(q(a)). q(a) :- p(a), not(r(a)).	{p(a) , r(a)} and {p(a) , q(a)}

In this example, there are two stable models *{p(a) , r(a)}* and *{p(a) , q(a)}*. By the definition we get:

$M_1 = \{p(a)\ , r(a)\}$	$M_2 = \{p(a)\ , q(a)\}$
$P^{M_1} =$ { p(a). r(a) :- p(a). }	$P^{M_2} =$ { p(a). q(a) :- p(a). }
$least(P^{M_1}) = M_1$	$least(P^{M_2}) = M_2$

Typically, generation of candidate solutions is deductively implemented, and constraints, that are a special case of rules which have an empty head, are used to discard unintended models.

Two tools that are probably most known in the ASP community are Smodels[7] and Lparse[7]. Smodels is an answer set solver that has as a main front-end Lparse, that performs the grounding of the logic program. Other well known tools are clasp[8] and dlv[9].

As mentioned in 2.2, a declarative logical characterization of the full default semantics of RIF-PRD based on Answer Set Programming (ASP, see section 2.3) was proposed [DAL10], including the 3 stage process of RIF-PRD, match, conflict resolution and act.

The choice of ASP by the authors is based in that fact that 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. Plus, ASP is very expressive, allowing for compact representations of all NP and coNP decision problems, or even more complex ones if disjunctive programs are used [DAL10].

The article [DAL10] provides a RIF-PRD translation for ASP of: atomic formulas, fact bases, states, condition formulas, actions and rules. The 3 stages, matching, conflict resolution and acting are also translated into ASP. The authors of [DAL10] support that any

[7] http://www.tcs.hut.fi/Software/smodels/
[8] http://www.cs.uni-potsdam.de/clasp/
[9] http://www.dbai.tuwien.ac.at/proj/dlv/

conflict resolution strategy (including those normatively specified by RIF-PRD) should be defined by a set of rules, in which the keyword to denote the strategy would by an URI for the set of rules that define the strategy. An explanation of this process is provided in the article. This will be the core to the ASP based RIF-PRD implementation proposed in this dissertation, which will be detailed in the following chapter.

2.3.1 Incremental Answer Set Programming

One of the limitations of ASP is that problems such as model checking or planning can only be dealt considering one solution size at each problem instance, meaning that at each new problem instance there is redundancy, as the problem is entirely re-processed. According to [GKK+08b], with Incremental Answer Set Programming (iASP) this issue is addressed, by gradually processing the extensions to a problem rather than repeatedly re-processing the entire (extended) problem .

In [GKK+08b], an iASP domain specification is provided, represented as a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter κ ranging over the natural numbers. The base program B is meant to describe static knowledge, independent of parameter κ. The role of P is to capture knowledge accumulating with increasing κ, whereas Q is specific for each value of κ. The goal of iASP is to decide if the program

$$R[\kappa/i] = B \cup P \bigcup_{1 \leq j \leq i} P[\kappa/j] \cup Q[\kappa/i]$$

has an answer set for some (minimum) integer $i \geq 1$.

RIF-PRD is a stateful language, therefore, as iASP features the addition of a states to the computation of a ASP program, it is possible to implement RIF-PRD using iASP. Presently, the only ASP system that supports incremental mode is iClingo, that we shall detail in Chapter 3.

2.4 Conclusions

In this section we introduced several concepts that will be used and addressed in this work, in particular we presented an overview of both RIF-PRD and the logical paradigms that will be used to implement it.

In the first section, we introduced production systems, that uses production rules paradigm, one of the paradigms used in this dissertation to implement RIF-PRD. Furthermore, we also provided some specific details about the production rules implementation, by presenting RETE, an algorithm used by several production systems (including the one that will be used in this work) for pattern matching of rules.

We then presented RIF-PRD, the main topic of this work. A brief overview of the syntax and semantics was provided, along with some technical details of the language.

In section 2.3 we introduced Answer Set Programming, a declarative programming paradigm that will be one the paradigms used to implement RIF-PRD in this dissertation.

3

Technology

In this chapter, an overview of several relevant tools and libraries is presented. Not all of the tools will be used in this thesis, but a brief explanation about the choice of tools shall be given as well. The tools presented include XSLT, production systems (CLIPS and Jess), ASP systems (iclingo, oclingo), XSB Prolog and Benchmark (LUBM and OpenRuleBench).

3.1 XSLT

The eXtensible Stylesheet Language Transformation (XSLT) [Kay07] is a W3C Recommendation for transforming XML based documents. It is a declarative XML based language, that creates a new file with the results of the transformations applied to a XML document, leaving the original document unaltered.

A transformation expressed in XSLT is achieved by a set of template rules. A template rule associates a pattern, which matches nodes in the source document, with a sequence constructor. In many cases, evaluating the sequence constructor will cause new nodes to be constructed, which can be used to produce part of a result tree. The structure of the result trees can be completely different from the structure of the source trees. In constructing a result tree, nodes from the source trees can be filtered and reordered, and arbitrary structure can be added. This mechanism allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures [Kay07].

Commonly, XSLT is used to format and present XML documents, though it may be used to a wide range of other transformation tasks. In this work, it will be used to translate RIF-PRD/XML syntax into both ASP, JessML (a XML-based language supported by Jess production rules system - see below) and XSB.

3.2 Production rules systems

In this section we will present a few production rules systems considered for one of our RIF-PRD implementations. A production rules system executes rules in order to achieve a goal. The execution of rules follows a three step phase: match, conflict resolution and act (see section 2.1).

3.2.1 CLIPS

C Language Integrated Production System (CLIPS)[1] [Gia] is an expert system tool oriented to the development of rule and/or object based systems. It was developed by the NASA Software Technology Branch in 1986, and nowadays it is one of the most popular expert systems.

One of CLIPS key features is flexibility, being a multi-paradigm tool [Gia]. CLIPS provides to the user three paradigms: production rules, object-oriented paradigm, and procedural paradigm, that can be used separately or combined. The rules, facts and objects in a given CLIPS program are treated as data that stimulates the execution via the inference engine.

The rule based paradigm allows the user to implement its own rules in the CLIPS rule language. Regarding the Object oriented programming paradigm, CLIPS provides a language called CLIPS Object Oriented Language, that has some common features with SmallTalk[2] and Common LISP Object System (CLOS)[3] . As for the procedural paradigm, it is present in the form of Deffunctions (user built functions) and generic functions.

The basic elements of CLIPS are: a fact list and instance list, that work as a global memory for data, a knowledge base, which contains all the rules, and an inference engine that controls the execution of rules.

The working process of CLIPS follows the 3 steps explained in 2.1, also using RETE (section 2.1.1) algorithm for pattern matching of rules. Since CLIPS uses RETE (that uses forward chaining), programs written in CLIPS are data-driven.

There are a few languages similar to CLIPS, as Jess (section 3.2.2) and CLIPS/R2[4] , that was developed by Production Systems Technologies (founded by Charles Forgy, the inventor of RETE). One of the main features of CLIPS/R2 is the implementation of a newer version of RETE, RETE II [5], that has better performance than the original RETE.

3.2.2 Jess

Jess[6] is a rule engine and scripting environment written in JAVA. It was developed by Ernest Friedman-Hill at Sandia National Laboratories as part of an internal research project,

[1] http://clipsrules.sourceforge.net/
[2] http://www.smalltalk.org/main/
[3] http://www.dreamsongs.com/CLOS.html
[4] http://www.pst.com/clips_r2.htm
[5] http://www.pst.com/rete2.htm
[6] http://www.jessrules.com/

with the first release coming in late 1995. It began as a CLIPS [Gia] clone, but nowadays has many features that differentiate it from its parent, though it still maintains a very similar syntax with CLIPS. Here's a little example of a Jess rule:

Example 6 (Jess rule language example)**.**

```
(defrule r1
 ?c <- (car {year == 2010})
=>(assert (newCar (id ?c.id))))
```

In this example, the condition of the rule is to match `?c` with a car from the year 2010. The effect of the rule is the assertion of a fact `newCar((id ?c.id))`.

Some features that Jess provides are: a rule language that permits to express powerful concepts with ease; rule loop prevention; improved error reporting and Eclipse-based rule development environment. Still regarding the rule language, note that Jess also provides XML support for rule development.

The working process of Jess follows the three steps mentioned in section 2.1, match, conflict resolution, act. By default, Jess will stop its execution when there are no more rules activated, though this can be changed by the programmer.

For pattern matching purposes, Jess uses the well known RETE [For90] algorithm, which makes Jess much faster than a simple set of cascading if-then statements in a loop.

As with any other rule language, the Jess working memory stores facts. To create a fact in Jess first you must declare a template for that fact. In the **Example 6**, the fact car has to belong to a template called car, and that has at least a slot called year.

Example 7 (Jess template example)**.**

```
(deftemplate car (slot year))
```

3.2.2.1 Jess conditional formulas

The basic formula applied to a left hand side (LHS) of a Jess rule are called patterns. A pattern is used to match facts in the Jess working memory. In the **Example 6** $\{ (caryear == 2010) \}$is a pattern that matches facts of the template car and that have a slot year with value 2010.

The Jess language offers several conditional formulas to be used in the LHS of rules, such as:

- conjunctive formula, $and(\phi_1 \dots \phi_n)$, where $\phi_1 \dots \phi_n$ are conditional formulas or patterns;
- disjunctive formula, $or(\phi_1 \dots \phi_n)$, where $\phi_1 \dots \phi_n$ are conditional formulas or patterns;

- existential formula, $exists(\psi)$, where ψ is a Jess pattern. This existential formula will not be used in our implementation, as it is different from the RIF-PRD existential formula;
- universal quantification formula, $forall(\psi_1 \dots \psi_n)$, where $\psi_1 \dots \psi_n$, are Jess patterns. This formula will not be used in our implementation, as it has a different meaning from the RIF-PRD quantification formula *Forall*;
- negation $not(\phi)$, where ϕ is a conditional formula or pattern;
- test conditional element, $(test(\sigma))$, where σ is a boolean function applied to a fact, such as "greater than"and "equal to"functions.

Example 8 (Jess conditions example)**.**

```
(deftemplate car (slot year) (slot id))

(defrule r2
(or (car (year 2000) (id xyz)) (car (year 2011) (id xyz)))
(not (car {year < 1999}))
(car (id abc) (year ?x))
(test (eq (?x 2005))
=> ...
)
```

In this example, the left hand side of rule $r2$ will be satisfied if the following conditions are met:

- the car xyz is either from year 2000 or 2011;
- there is not a car in the Jess working memory older than 1999;
- the car abc is from year 2005.

3.2.2.2 Jess actions

The right hand side of a Jess rule is formed by actions. Jess language offers three different type of actions:

- assertion, $Assert(\alpha)$, where α is a fact;
- retraction, $Retract(\beta)$, where β is either an integer representing the identifier of a fact, or a *Jess.Fact* JAVA object (we will describe the process to obtain this identifier in the next chapter);
- modification, $Modify(\alpha\ (\beta\ \gamma))$, where α is a fact, β is a slot of α and γ is the new value of β.

Example 9 (Jess actions example)**.**

```
(deftemplate car (slot year) (slot id))

(defrule r2
...
=>
(assert (car (id aaa) (year 2010))
)
```

3.2.2.3 Jess conflict resolution

Regarding the conflict resolution, Jess provides two strategies, depth strategy and breadth strategy. By default, Jess will use the depth strategy.

In the depth strategy, when in case of conflict of rules that share the same salience[7], the rules that have been activated more recently will be fired. This strategy is identical to the default RIF-PRD conflict resolution strategy $rif : forwardChaining$.

The breadth strategy is the simplest one, rules are fired in the order which they are activated.

Example 10 (Demonstration of Jess's conflict resolution strategies)**.**

```
(deftemplate car                    (defrule r1
(slot brand)                        (declare (salience 5))
(slot year (type INTEGER))          (car  year == 2010)
(slot color (default white)))      => (printout t "R1 activated"crlf))

(defrule r2                         (defrule r3
(declare (salience 10))             (declare (salience 10))
(car  year == 2007)                 (car  year == 2010)
=>                                  =>
(printout t "R2 activated"crlf))    (printout t "R3 activated"crlf))
```

- Using the depth strategy, the order which the rules will be fired is : *r2*, *r3*, *r1*. This is due to the activation of *r2* being more recent than *r3*, that is caused by the order of the assertion of facts.
- Using the breadth strategy, the order will be : *r3*,*r2*,*r1*, since the activation of *r3* happens before the activation of *r2*, due to the order of the assertion of facts.

Although these are the two given strategies, Jess allows the user to create his own conflict resolution strategies using JAVA.

Since Jess uses RETE [For90], reasoning with rules will be made through forward chaining. This means that Jess will match the preconditions of the rules with the working memory until all preconditions are met and the rule is activated.

[7]Salience is a property that all rules possess, that indicates the priority of the rule, being higher salience equivalent to higher priority.

Despite the usage of RETE algorithm, Jess also has built-in support for backward chaining. According to [FH] is that with a backward chaining system, the rule engine seeks steps to activate rules whose preconditions are not met, a behaviour called "goal seeking". It is also stated that this explanation is very simplistic since full explanation requires a good understanding of the underlying algorithms used by Jess.

3.2.3 Drools

A third considered option to implement RIF-PRD using a production rules system was Drools. According to [PNM+07], Drools is a business rule management system(BRMS) and an enhanced Rules Engine implementation, ReteOO, based on Charles Forgy's Rete algorithm tailored for the Java language. Drools also provides for Declarative Programming and is flexible enough to match the semantics of a given problem domain with Domain Specific Languages, graphical editing tools, web based tools and developer productivity tools. Similarly to Jess, Drools is also JAVA written and allows the user to develop a conflict resolution strategy.

3.3 ASP Systems

In this section we will present a few ASP systems. The main component of an ASP system is the answer set solver, that computes the answer sets of a given logic program. One of the RIF-PRD implementations proposed in this dissertation will be developed using one of the tools presented in this section. Despite not being presented in here, there are other ASP systems that were considered to be used in this work, for example, DLV[8].

3.3.1 iclingo

iclingo[9] is and ASP *incremental system* [GKK+08c] that is an extension of the system clingo [GKK+08a] written in C++. the system iclingo links internally the grounder gringo [GKK+08a] and the solver clasp [GKK+08a].

The major feature of iClingo is the support of incremental answer set computation (see section 2.3.1). With incremental computation, the basic notion is the notion of state. A state is defined by a set of facts, and, when the state is incremented, the facts that are not defined as incremental are not passed on to the next state. A state is incremented when there are no more rules to fire in the current state.

In addition to the constructs that are available in clingo, iclingo supports statements with the form: *#base*, *#cumulative constant* and *#volatile constant*.

Regarding the statement *#base*, it is used to declare that a certain part of the program is static, therefore it will only be processed once in the beginning of the incremental computation and kept in all iterations. In contrast, *#cumulative constant* and *#volatile constant*

[8] http://www.dbai.tuwien.ac.at/proj/dlv/
[9] http://potassco.sourceforge.net/

are used to declare a (symbolic) *constant* as a placeholder for incremental step numbers [GKK+08a]. This constant will be always updated with current step number. In the program parts below a *#cumulative* the facts, rules and integrity constraints are accumulated over a whole incremental computation. Regarding the *#volatile* program parts, the facts, rules and integrity constraints computed in one step are dismissed before the next incrementation. No other ASP system has this feature.

As stated in [GKK+08a], the incremental computational process of iclingo starts with grounding. This is executed by the *gringo* part of iclingo, and it ensures that no rule is grounded twice with the same substitution of variables. First of all, the static parts of the program are grounded. The cumulative and volatile parts of the program are grounded stepwise.

The second step of the incremental computational process is executed by *clasp*, ending when an answer set is obtained.

Here's an example of an iclingo program, obtaining the least number of colors necessary to color an arbitrary graph, each coloring being a differente stable model.

Example 11 (iclingo program)**.**

```
#base.

% The graph
node(a;b;c).

edge(a,b).
edge(b,c).
edge(c,a).

#cumulative k.

% a new color k is added to the program.
color(k).

% a node might have this color or not.
{ color(X,k) } :- node(X).

% cannot color nodes with color k which have been previously colored
:- color(X,k), color(X,V), V != k.

#volatile k.

% Every node must have exactly one color
1 {color(X,V): color(V)} 1 :- node(X).

% A graph to be colored correctly must not have vertices of an edge
colored with the same color.
:- edge(X1,X2), color(V), color(X1,V), color(X2,V).
```

One extra feature that gringo offers is the possibility of using the scripting language Lua[10]. With Lua, gringo's input language can be enriched by arithmetical functions and implicit domains, and even provide a link to database systems.

3.3.2 oclingo

oclingo [GGKS11] is an ASP system developed in 2010 by Torsten Grote, at the university of Potsdam, as a master thesis. Similarly to iclingo, oclingo links internally the grounder gringo and the solver clasp.

Until now, ASP applications were mostly static. This means that if a dynamic program (a program that may receive real time inputs) were to be solved through these applications, the computational process of the ASP application would reprocess and rebuild everything it had performed whenever a new input was made. These ASP applications are also called offline ASP systems.

The main goal of oclingo is to be an online ASP system. This means that whereas an offline ASP system would reprocess a program whenever a input was submitted, the online ASP system only processes the whole program once, and updates whenever an input is submitted. These changes translate in redundancy decrease and performance boost against the offline ASP systems.

As stated in [GGKS11], this way of solving is called online because the system communicates with the real world and gets input in real time. Therefore, the ASP systems that manage to dynamically find solutions to solve these online programs, are online ASP systems.

3.4 XSB

XSB is a research-oriented, commercial-grade Logic Programming system for Unix and Windows-based platforms. It includes nearly all functionality of ISO-Prolog, plus several other features mentioned in [SS], as tabling, fast loading of large files by the $load\_dync/1$ predicate, and a variety of indexing techniques for asserted code including variable-depth indexing on several alternate arguments, fixed-depth indexing on combined arguments, trie-indexing.

These three features will be relevant to this work, as we will be using large files for testing, and use trie-indexing plus tabling for storing the facts, and incrementally update them.

3.4.1 XSB tabling

One of the limitations of Prolog is the risk of infinite loops (even in some perfectly reasonable programs). This happens because Prolog is based on a depth-first search through trees that are built using program clause resolution (SLD) [SS]. In order to surpass this

[10] http://www.lua.org/

limitation, XSB features SLG [SW94] evaluation. SLG evaluation in XSB appears in the form of *tabling*. According to [SS], during computation of a goal to a logic program, each subgoal S is registered in a table the first time it is called, and unique answers to S are added to the table as they are derived. When subsequent calls are made to S, the evaluation ensures that answers to S are read from the table rather than being re-derived using program clauses.

Therefore, tabling is useful to ensure that XSB terminates in many cases where Prolog won't. Furthermore, XSB provides a heap garbage collector for tabled predicates, guaranteeing that when a table is abolished, its space is properly reclaimed. We assume the user is acquainted with Prolog. We illustrate briefly the main significant features of XSB-Prolog that will be explored in this work.

To perform *tabling*, *table* declarations are used as follows:

Example 12 (Tabling example [SS])**.**

```
:- table ancestor/2.

ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

In Prolog systems, this program will enter in an infinite loop. In XSB, due to *tabling*, this program will terminate since $ancestor/2$ is a tabled predicate. XSB also supports negated calls to tabled predicates under the Well-Founded Semantics [VGRS91], but this feature will not be used in our work.

Furthermore, XSB features *incremental tabling*. Contrary to *non-incremental tabling*, *incremental tabling* allows update and removal of tabled predicates, as long as the table predicate is defined as a dynamic predicate. There are several operations that can be performed in incremental tabled predicates, such as $incr\_assert/1$, that asserts a predicate into an incremental table and automatically updating that table, and $incr\_assert\_inval/1$, that asserts a predicate into an incremental table without updating that table. The update of a table can be forced by calling the operation $incr\_table\_update/0$.

Here's an example showing the difference of *non-incremental* and *incremental tabling* [SS]:

Example 13 (Non-incremental and incremental tabling in XSB)**.**

```
:- table p/2.

p(X,Y) :- q(X,Y), Y <= 5.

:- dynamic q/2.

q(a,1).
q(b,3).
q(c,5).
```

```
|?- p(X,Y), writeln([X,Y]), fail.

[c,5]
[b,3]
[a,1]


|?-assert(q(d,4)).

|?-p(X,Y),writeln([X,Y]),fail.
[c,5]
[b,3]
[a,1]
```

As we can see, with the use of non-incremental tabling, at the end of the program there were inconsistencies, as predicate $q(d, 4)$ does not show up in the second query. This happens because the second query does not re-evaluate the answers, it simply retrieves the values of the table created in the first query, which completed the tables.

Consider the *incremental tabling* version:

```
:- table p/2 as incremental.

p(X,Y) :- q(X,Y), Y <= 5.

:- dynamic q/2.

q(a,1).
q(b,3).
q(c,5).

|?- import incr_assert/1 from increval.
```

```
|?- p(X,Y), writeln([X,Y]), fail.

[c,5]
[b,3]
[a,1]


|?-assert(q(d,4)).

|?-p(X,Y),writeln([X,Y]),fail.

[d,4]
[c,5]
[b,3]
[a,1]
```

As we can see, after asserting the predicate $q(d, 4)$ into an incremental table (using the operation $incr\_assert/1$), the second query returns all the previous answers including $q(d, 4)$, as the table created upon the first query was updated with the $incr\_assert/1$ call.

The major downside to incremental tabling is that performance-wise, it comes at an expensive cost. There is an alternative to the above predicates, that performance-wise is less expensive. These predicates are $incr\_assert\_inval/1$ and $incr\_retractall\_inval/1$, which are similar to the above predicates except they do not update automatically the tables, instead they mark the tables as invalid. To update the tables a call to the predicate $incr\_table\_update/0$ must be done.

3.4.2 XSB tries

Tries, a variant of discrimination nets, provide a complete discrimination for terms, and permit a lookup and possible insertion to be performed in a single pass through a term [RRS+99].

Comparing with standard dynamic code, the insertion and deletion of tries is 4-5 times faster. Furthermore, since there is no distinction in trie-dynamic code between index and the code itself, in some cases trie storage might occupy far less space than standard dynamic code.

Still, trie storage is not without tradeoffs: only facts can be stored in a trie, no ordering is preserved among the facts (unlike standard dynamic code), and it does not support duplicate facts.

In XSB a trie-indexing term can be defined as follows:

$$:- index(fact/1, trie).$$

Assuming the set of facts ϕ

$$\{rt(a, f(a, b), a), rt(a, f(a, X), Y), rt(b, V, d)\}$$

the trie storage of ϕ is

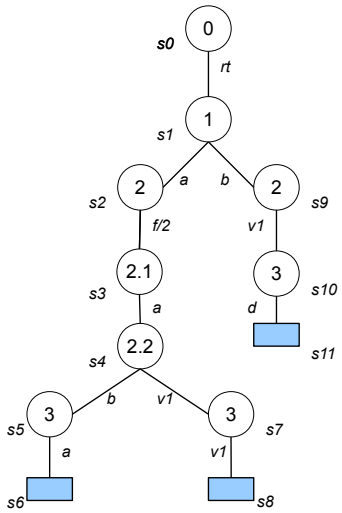


Figura 3.1: Trie storage.

where each node relates to an instruction in XSB's virtual machine. We will use tries to store the dynamic fact base.

3.5 Benchmarking and Evaluation

To test our translations, two kinds of tests were made: W3C test cases and benchmarking. With the W3C test cases we wanted to see if our translations performed correctly the

translation, whereas the benchmarking assesses if the performances of our translations are competitive.

3.5.1 Evaluation

The W3C has a series of tests cases meant to evaluate if an implementation of RIF-PRD is working as intended. These tests are available in the W3C website [11]. We did not test every case in the W3C test case repository, as our translations have a few limitations, which will be detailed in the next chapter.

3.5.2 Benchmarking

3.5.2.1 LUBM

Lehigh University Benchmarking (LUBM)[12] is a benchmark oriented towards Semantic Web knowledge base systems with respect to use in large OWL applications [GPH05].

One of the features of LUBM is that it supports very different types of systems : systems supporting RDFS reasoning, systems providing partial OWL Lite reasoning, and systems that are complete or almost complete for OWL Lite [GPH05].

The main goals of LUBM are [GPH05]:

- Support extensional queries : Extensional queries are queries about the instance data over ontologies.
- Arbitrary scaling of data : In order to evaluate the ability of systems to handle large ABoxes we need to be able to vary the size of data, and see how the system scales.
- Ontology of moderate size and complexity

3.5.2.2 OpenRuleBench

OpenRuleBench[13] is a benchmark oriented towards rule systems, supporting several types of rule systems (e.g. Prolog based, deductive databases, production rules).

Along with OpenRuleBench, the developers have made freely available various rule sets, real-world data and data generators, and scripts freely available [LFWK09].

3.6 Conclusions

After exploring each tool mentioned before, the list of tools and libraries chosen is:

- Jess, to implement a RIF-PRD translation using production rules paradigm. Jess provides a XML-based language (JessML), making it easier to translate RIF-PRD documents using XSLT. Also, being JAVA written, it was easier to work with Jess,

[11] http://www.w3.org/2005/rules/test/repository/tc/
[12] http://swat.cse.lehigh.edu/projects/lubm/index.htm
[13] http://rulebench.projects.semwebcentral.org/

whereas CLIPS, is written in C. Regarding Drools, it would also be a viable choice, but since Jess has a more intuitive documentation, we chose Jess over Drools;

- iClingo, to implement a RIF-PRD translation using answer set programming. iClingo's main feature is that it is an incremental system, which was the base to the RIF-PRD translation;
- XSB, to implement a RIF-PRD translation using logic programming. XSB was chosen mainly due to its incremental tabling support;
- XSLT, to translate RIF-PRD documents into Jess, iClingo and XSB.
- LUBM, as a benchmark to evaluate the three implementations above. LUBM was the choice as it is one of a kind. Furthermore, OpenRuleBench provides data files (CLASP, DLV and Jess) to test with LUBM, which we used in this work.

4

Implementation

We present three implementations of RIF-PRD using three different approaches. The basis of the three implementations is the declarative logical characterization [DAL10], that is fully represented in the iASP implementation of RIF-PRD. The XSB and Jess versions were implemented to perform tests and see if the iASP version could compete performance-wise with a production rules system and a logic programming system.

We provide a comparison of the three implementations in section 5. The tests were performed using the Lehigh University Benchmark (LUBM) [1].

[1] http://swat.cse.lehigh.edu/projects/lubm/

4.1 RIF-PRD to ASP implementation

The ASP implementation of RIF-PRD is based in [DAL10], that presents a declarative logical characterization of the RIF-PRD language using incremental Answer Set Programming. This approach allows a better understanding of the RIF-PRD language and is flexible enough to allow the development of new conflict resolution strategies other than the *rif:forwardChaining* strategy, that is the RIF-PRD default (and so far, the only) strategy available.

The simplest constructs of RIF-PRD are terms and atomic formulas. These are the base for every other construct in the RIF-PRD language.

Definition 16 (Atomic formulas translation [DAL10])**.** *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 $s[p \rightarrow o]$ is mapped into term $frame(s, p, o)$.*

This representation assumes that a ground frame $t[p_1 \rightarrow o_1 \ldots p_n \rightarrow o_n]$ is represented by the facts $frame(t, p_1, o_1), \ldots, frame(t, p_n, o_n)$.

RIF-PRD is a stateful language, that feature alone made us choose *iClingo* 3.3.1 since it is an incremental system, which means it provides support to stateful programs.

In each state there is a KB associated to it, containing the facts that are true in that state.

Definition 17 (Facts translation[DAL10])**.** *Consider an initial fact base Φ:*

> *Program $\pi_{INIT(\Phi)}$ is formed by $fact(\varphi, INIT)$, foreach $\varphi \in \Phi$, where $INIT$ is an integer constant identifying the initial state. This is the initial fact base of the program.*
>
> *Program $\pi_{CHANGE[\kappa]}$ is formed by the rules:*
>
> $$fact(F, \kappa) \leftarrow fact(F, \kappa - 1), not\ retract(F, \kappa - 1).$$
> $$fact(F, \kappa) \leftarrow assert(F, \kappa - 1).$$
>
> *The constant κ is the placeholder for incremental step numbers.*

These two rules deal with the maintenance of facts through consecutive states. The first rule says that a fact exists in a determined state κ if it was present in the previous state $(\kappa - 1)$ and it was not retracted in $\kappa - 1$.

The second rule says that if a fact was asserted in the previous state $(\kappa - 1)$ then it is present in the current state κ.

Definition 18 (States translation [DAL10]). *To ensure the maintenance of facts there are few more rules that are needed, that deal with the class hierarchy expressed in facts. These rules are:*

$$state(F, \kappa) \leftarrow fact(F, \kappa).$$

$$state(sub(C1, C2), \kappa) \leftarrow fact(sub(C1, C2), INIT).$$

$$state(isa(O1, C2), \kappa) \leftarrow state(isa(O1, C1), \kappa), state(sub(C1, C2), \kappa).$$

$$state(sub(C1, C3), \kappa) \leftarrow state(sub(C1, C2), \kappa), state(sub(C2, C3), \kappa).$$

The first rule states that the fact base in κ, belongs to state κ.

The second rule says that any subclass membership fact in the program is present since the initial state.

The third and fourth rule ensure respectively class inheritance and subclass transitivity, and correspond to the conditions expressed in section 2.2.2.

4.1.1 RIF-PRD conditions

The translation of RIF-PRD condition formulas is the result of applying the Lloyd-Topor's transformation [LT84] to obtain the normal rules equivalent to the RIF-PRD condition formulas.

Definition 19 (Conditions translation [DAL10]). *Let Φ be an arbitrary condition formula and κ an execution step. Define condition iASP formula Φ' and program $\pi^{\Phi}_{COND}[\kappa]$ inductively as follows:*

- *If Φ is an atomic formula φ then $\Phi'[\kappa] = state(\varphi', \kappa)$ and $\pi^{\Phi}_{COND}[\kappa] = \{\}$;*
- *If $\Phi = And(\phi_1 \dots \phi_n)$ then $\Phi'[\kappa] = (\phi_1 \dots \phi_n)$ and $\pi^{\Phi}_{COND}[\kappa] = \bigcup_{1 \leq i \leq n} \pi^{\phi_i}_{COND}[\kappa]$;*
- *If $\Phi = Or(\phi_1 \dots \phi_n)$ then $\Phi'[\kappa] = or_{\Phi}(X_1 \dots X_m, \kappa)$ where $X_1 \dots X_m$ are the free variables of Φ and or_{Φ} is a new predicate symbol, and $\pi^{\Phi}_{COND}[\kappa] = \bigcup_{1 \leq i \leq n} \left(\pi^{\phi_i}_{COND}[\kappa] \cup \{or_{\Phi}(X_1 \dots X_m, \kappa) \leftarrow \phi_i'[\kappa]\}\right)$;*
- *If $\Phi = Exists\ ?V_1 \dots ?V_n\ (\phi)$ then $\Phi'[\kappa] = exists_{\Phi}(X_1 \dots X_m, \kappa)$ where $X_1 \dots X_m$, are the free variables of Φ and $exists_{\Phi}$ is a new predicate symbol, and $\pi^{\Phi}_{COND}[\kappa] = \pi^{\phi}_{COND}[\kappa] \cup \{exists_{\Phi}(X_1 \dots X_m, \kappa) \leftarrow \phi'[\kappa]\}$;*
- *If $\Phi = Not(\phi)$ then $\Phi'[\kappa] = not\ arg_{\Phi}(X_1 \dots X_m, \kappa)$ where $?X_1 \dots ?X_m$, are the free variables of Φ and arg_{Φ} is a new predicate symbol, and $\pi^{\Phi}_{COND}[\kappa] = \pi^{\phi}_{COND}[\kappa] \cup \{arg_{\Phi}(X_1 \dots X_m, \kappa) \leftarrow \phi'[\kappa]\}$;*

The first rule states that any atomic formula Φ will be translated to iASP as $state(\Phi, \kappa)$.

The $And(\phi_1 \dots \phi_n)$ condition formula refers to the sequence of $\phi_1 \dots \phi_n$ formulas translated into ASP.

The $Or(\phi_1 \dots \phi_n)$ conditional formula will be translated as the predicate $or_\Phi(X_1 \dots X_m)$. For each ϕ_i, a new rule $or_\Phi(X_1 \dots X_m)$ will be added to the program, presenting the translation of ϕ_i.

The $Exists\ ?V_1 \dots ?V_n\ (\phi)$ formula is translated to a predicate $exists_\Phi(X_1 \dots X_m, \kappa)$. Also, a new rule $exists_\Phi(X_1 \dots X_m, \kappa)$ will be added to the program, where $X_1 \dots X_m$ are the free variables of that rule. This rule represents the translation of ϕ into iASP.

The $Not(\phi)$ formula will be translated as $not\ arg_\Phi(X_1 \dots X_m)$. Also, a new rule $arg_\Phi(X_1 \dots X_m, \kappa)$ will be added to the program, where $X_1 \dots X_m$ are the free variables of that rule. This rule represents the translation of ϕ into iASP.

Example 14 (Conditions of a RIF-PRD rule translated to iASP)**.**

```
(* Annotation : rule r1 *)
If Exists ?color  ?car (
        And (
                ?car#vehicle
                INeg( ?car[Id->123] )
                Or(
                  ?car[color->?color Id->456]
                  ?car[color->?color Id->789]
   ))        )
Then Do( ... ))
```

The iASP translation of the conditional formula in the body of the rule is as follows:

```
exists1(k) :- state(isa(VAR_car,vehicle),k),
  not arg1(VAR_car,k), or1(VAR_car,VAR_color,k).

arg1(VAR_car,k) :- state(frame(VAR_car,id,123),k).

or1(VAR_car,VAR_color,k) :- state(frame(VAR_car,color,VAR_color),k),
  state(frame(VAR_car,id,456),k).

or1(VAR_car,VAR_color,k) :- state(frame(VAR_car,color,VAR_color),k),
  state(frame(VAR_car,id,789),k).
```

In ASP, the name of a variable has to start with a capital letter. For that matter, in our translation, every variable starts with the prefix `VAR_`. Instead of using the formula Φ to identify the new generated predicates, we use a new numeric identifier for each predicate generated. Duplicates may be generated in this process.

4.1.2 RIF-PRD actions

RIF-PRD defines several actions for modifying the fact base. Consider a rule r_i, for each action α of r_i, a new rule will be added to the program:

$$action(\alpha_j, \kappa + j) \leftarrow instance(r_i, subs(V_1 \dots V_n), \kappa).$$

where α_j is a RIF-PRD action, $instance(r_i, subs(V_1 \dots V_n), \kappa)$ corresponds to the bindings determined by a rule r_i that was fired in state κ, and $subs(v_1, \dots, v_n)$ are the free variables in the rule.

Definition 20 (Effects of RIF-PRD actions [DAL10])**.** *For each action predicate $action(\alpha, \kappa)$, there will be changes in the KB. Program $\pi_{ACTIONS}[\kappa]$ is:*

- *Assert fact:*

  $$assert(F, \kappa) \leftarrow action(assert(F), \kappa).$$

- *Retract fact:*

  $$retract(F, \kappa) \leftarrow action(retract(F), \kappa).$$

- *Retract all slot values:*

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

- *Retract object:*

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

The compound action Modify is translated as the sequence of two atomic actions, retract all slot values and assert fact.

Note that the execute actions do not have an effect in the KB and should be interpreted externally [DAL10].

Example 15 (RIF-PRD actions translated to iASP)**.**

```
If(...)
Then Do(
        Assert(newCar#vehicle)
        Assert(newCar[Color->black Id->?carId])
        Modify(newCar[Color->white])
        Retract(newCar)
)
```

The iASP translation is as follows:

```
action(assert(isa(newCar,vehicle),k+1) :- instance(r1,subs,k).
action(assert(frame(newCar,color,black),k+2) :- instance(r1,subs,k).
action(assert(frame(newCar,id,VAR_carId),k+3) :- instance(r1,subs,k).
action(retract_slots(newCar,color),k+4):- instance(r1,subs,k).
action(assert(frame(newCar,color,white),k+5) :- instance(r1,subs,k).
action(retract_object(newCar),k+6) :- instance(r1,subs,k).
```

Notice that there are no free variables in the Action Block, so we use an empty `subs` term. Also mark the translations of the compound action in the $\kappa + 4$ and $\kappa + 5$ actions.

4.1.3 RIF-PRD rules

There are three types of RIF-PRD rules: unconditional action block, conditional action block and a quantified rule. In this subsection we will approach the three types.

Definition 21 (Translation of RIF-PRD rules [DAL10])**.** *Let r_i be a RIF production rule and let id be a unique identifier assigned to that rule (i.e. its "name"). Program $\pi^{r_i}_{RULE}[\kappa]$ is constructed as follows [DAL10]:*

- *If r_i is $Do((?V_1\ b_1)\ldots(?V_n\ b_n)\ a_1\ldots a_m)$ then include in $\pi^{r_i}_{RULE}[\kappa]$ the fact $fireable(rule(id, subs), \kappa)$.*
- *If r_i is $If\ \Phi\ Then\ Do((?V_1\ b_1)\ldots(?V_n\ b_n)\ a_1\ldots a_m)$ then include $\pi^{\Phi}_{COND}[\kappa]$ in $\pi^{r_i}_{RULE}[\kappa]$, and the following rule where $X_1\ldots X_l$ are the free variables of r_i: $fireable(rule(id, subs(X_1\ldots X_l), \kappa) \leftarrow \Phi'[\kappa]$.*
- *If r_i is $Forall\ ?V_1\ldots?V_n$ such that $(p_1\ldots p_m)\ If\ \Phi\ Then\ Do(B)$ then treat this as the conditional action block $If\ And(p_1\ldots p_m\ \Phi)\ Then\ Do(B)$.*

Additionally, from the action block $Do((?V_1\ b_1)\ldots(?V_n\ b_n)\ a_1\ldots a_m)$ in the conclusion of r_i, add to the program $\pi^{r_i}_{RULE}[\kappa]$, for each $1 \leq j \leq m$, the rule:

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

RIF-PRD also allows to declare variables in the right hand side of a rule. These variables are called action variables and can be one of two types: $New()$ for generating a new identifier, or a frame $o[p \rightarrow ?V]$ where $?V$ is the action variable.

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

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

The predicate $fireable(rule(id, subs(\ldots)), \kappa)$, represents the conditions of a RIF-PRD rule. Given a rule r_i, when $fireable(r_i, \kappa)$ holds in state κ, r_i might be fired. If fired, the predicate $picked(r_i, \kappa)$ will hold, and consequently $instance(r_i, \kappa)$ holds. At this point,

the actions a_j from rule r_i will be executed in increasing order of $\kappa + j$. The implementation of $picked/2$ will be presented subsequently, and depends on the conflict resolution strategy.

Example 16 (Translation of a RIF-PRD rule to iASP)**.**

```
(* Annotation : r2 *)
Forall ?car such that (
If Exists ?buyer (
        And(
                ?car[forSale->true]
                ?car[owner->blank]
                ?car[hasBuyer->?buyer]))
Then Do(
        (?buyer ?car[hasBuyer->?buyer])
        (?newVar New())
        Modify(?car[owner->?buyer])
        Modify(?car[forSale->false])
        Assert(?newVar#Receipt)
))
```

And the respective iASP translation:

```
fireable(rule(r2,subs(VAR_car),k) :- exists1(VAR_car,k).

exists1(VAR_car,k) :- state(frame(VAR_car,forSale,true)),
  state(frame(VAR_car,owner,blank)),
  state(frame(VAR_car,hasBuyer,VAR_buyer)).

action(retract_slots(VAR_car, owner),k+1) :-
  instance(rule(r2,subs(VAR_buyer,VAR_car,Var_newVar),k).

action(assert(frame(VAR_car,owner,VAR_buyer),k+2) :-
  instance(rule(r2,subs(VAR_buyer,VAR_car,Var_newVar),k).

action(retract_slots(VAR_car forSale),k+3) :-
  instance(rule(r2,subs(VAR_buyer,VAR_car,Var_newVar),k).

action(assert(frame(VAR_car,forSale,false),k+4) :-
  instance(rule(r2,subs(VAR_buyer,VAR_car,Var_newVar),k).

action(assert(member(VAR_newVar,Receipt),k+5) :-
  instance(rule(r2,subs(VAR_buyer,VAR_car,Var_newVar),k).

instance(rule(r2,subs(VAR_buyer,VAR_car),k) :-
  picked(rule(r2,subs(VAR_car)),k),
  state(frame(VAR_car,hasBuyer,VAR_buyer),k), VAR_newVar = obj(r2,2,k).
```

The next step is to explain the semantics of picking rules in iASP. When a rule r_i is chosen in a state κ, the following $\kappa + \gamma$ states, where γ is the number of actions of r_i, are

called transitional states. Transitional states are meant to execute actions and consequently changes to the KB, therefore rules are only picked in non transitional states.

Definition 22 (Pick rule [DAL10]). *Program $\pi_{PICK}[\kappa]$ is formed by:*

$$picked(Rule, \kappa) \leftarrow pickable(Rule, \kappa), not\ picked\_other(Rule, \kappa), not\ transitional(\kappa).$$

$$picked\_other(Rule, \kappa) \leftarrow pickable(Other, \kappa), Rule\ != Other, picked(Other, \kappa).$$

$$picked(\kappa) \leftarrow picked(Rule, \kappa).$$

$$transitional(\kappa) \leftarrow action(A, \kappa).$$

To pick a rule, that rule must be pickable in the current state κ and there cannot be another rule picked in κ. For a rule r_i to be pickable, the predicate $pickable(r_i, \kappa)$ must hold. This predicate is captured the program $\pi_{ONE}[\kappa]$ with a single rule:

$$pickable(Rule, \kappa) \leftarrow fireable(Rule, \kappa).$$

This means that, for the time being, any fireable rule is pickable. We will make this more complex later on to handle $rif : forwardChaining$.

The execution of a translated RIF-PRD program will terminate when a state is reached that is non transitional and there are no rules to pick.

Definition 23 (Termination).

$$\leftarrow not\ final(\kappa).$$

$$final(\kappa) \leftarrow not\ transitional(\kappa), not\ picked(\kappa).$$

Definition 24 (Rule set translation [DAL10]). *The translation of a RIF-PRD set RS with an initial fact base w is the iASP domain specification $\Pi_{RULESET}(RS, w) = \langle B_{RS}(w), S_{RS}(RS), [\kappa], Q_{RS}[\kappa] \rangle$ where:*

$$B_{RS} = \pi_{INIT}[\kappa]$$

$$S_{RS}[\kappa] = \pi_{CHANGE}[\kappa] \cup \pi_{STATES}[\kappa] \cup \pi_{ACTION}[\kappa] \cup \pi_{PICK}[\kappa] \cup \pi_{ONE}[\kappa] \cup \bigcup_{r_i \in RS} \pi^{r_i}_{RULE}[\kappa]$$

$$Q_{RS} = \pi_{HALT}[\kappa]$$

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:

Theorem 1 (Correctness of translation [DAL10]). *Let RS be a rule set and w an initial fact base. Then* [2]*:*

Soundness: If $M \in AS(\Pi_{RULESET}(RS, w)_n)$ and $(c_1 \dots c_n)$ is the increasing sequence of integers such that the $transitional(c_j) \notin M$, $1 \leq j \leq m$, then, for every $i : 1 \leq i \leq m-1$ $(State^i(M), Picked^i(M), State^{i+1}(M)) \rightarrow PRD$, 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 $(s_1 \dots s_m)$ 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 $M \in AS$ $(\Pi_{RULESET}(RS, w)_n)$ for some $n \geq m$ such that the sequence of integers $(c_1 \dots c_m)$, constructed from M as above, is such that $State^i(M) = s_i$, for all $1 \leq i \leq m$.

Thus the above theorem justifies that the translation does not limit any possible con?ict resolution strategy, since all possible traces can be generated.

4.1.4 RIF-PRD conflict resolution

In RIF-PRD, only one rule can be chosen per state, though, it is common to have more than one rule activated in the same state κ. To select which rule shall be fired, we resort to conflict resolution strategies.

Definition 25 (Strategy [DAL10]). *Program $\pi_{STRATEGY}[\kappa]$ is formed by:*

$$pickable(Rule, \kappa) \leftarrow fireable(Rule, \kappa), not\ rejected(Rule, \kappa).$$

$$rejected(Rule, \kappa) \leftarrow rejected(Rule, \kappa, S), st\_element(S).$$

This predicate holds when the conditions of the rule are satisfied in κ (predicate $fireable(r_i, \kappa)$), and the rule is not rejected. A rule being rejected means it did not satisfy all the requirements the chosen conflict resolution strategy applies.

These semantics make this implementation flexible enough to allow user made strategies, as the user will only need to implement the predicates $rejected(Rule, \kappa, S)$ and $st\_element(S, N)$, according to the intended strategy. If no strategy is defined, $pickable(Rule, \kappa)$ will hold for every fireable rule instance in κ, being equivalent to the previous program $\pi_{ONE}[\kappa]$.

Not every rule in a given state κ will applied to the conflict resolution. The rule needs to be active in κ.

Definition 26 (Active Rules [DAL10]).

$$inactive(Rule, \kappa, N) \leftarrow st\_element(\_, N), st\_element(S, N1), N1 < N, rejected(Rule, \kappa, S).$$

$$active(Rule, \kappa, N) \leftarrow not\ inactive(Rule, \kappa, N), st\_element(\_, N).$$

[2] $\rightarrow PRD$ 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 s_i.

As seen in section 2.2 RIF-PRD has one default conflict resolution strategy: $rif : forwardChaining$. It is formed by three elements:

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

We also need the rule $st\_element(S) \leftarrow st\_element(S, \_)$. to obtain the defined elements regardless its order.

The `rif:forwardChanining` strategy will apply the strategy elements in the given order, not applying a element x to a rule instance if it did not pass the previous element $x - 1$. Therefore, if a rule instance does fail in the refraction element, it should be discarded, thus not be tested for priority. Special care is necessary to propagate information from non-transitional states to transitional ones.

Definition 27 (Refraction strategy element)**.** *The first strategy element of $rif : forwardChaining$ is Refraction. Once a rule is picked in a given state, it is rejected on following states while it remains fireable.*

$$rejected(Rule, \kappa, refraction) \leftarrow fireable(Rule, \kappa), picked(Rule, \kappa{-}1), nottransitional(\kappa).$$

$$rejected(Rule, \kappa, refraction) \leftarrow rejected(Rule, \kappa - 1, refraction), transitional(\kappa).$$

$$rejected(Rule, \kappa, refraction) \leftarrow fireable(Rule, \kappa), rejected(Rule, \kappa{-}1, refraction),$$
$$not\ transitional(\kappa).$$

Definition 28 (Priority strategy element)**.** *The second strategy element is Priority. All rule instances for which there has a higher priority instance active, are discarded. As said before, rules that were discarded in the Refraction phase, are not tested in Priority.*

$$rejected(rule(Id, Var), \kappa, priority) \leftarrow fireable(rule(Id, Var), \kappa),$$
$$fireable(rule(Id2, Var2), \kappa), Id\,!= Id2, priority(Id, P), priority(Id2, P2),$$
$$P < P2, active(rule(Id2, Var2), \kappa, N), st\_element(priority, N).$$

Definition 29 (Recency strategy element)**.** *The third and final strategy element of $rif : forwardChaining$ is Recency. A rule is rejected if there is another active rule that has been activated after and is fireable.*

$$rejected(rule(Id, Var), \kappa, recency) \leftarrow fireable(Rule, \kappa),\ fireable(Other, \kappa),\ Rule\ != Other,$$
$$recency(Rule, TR, \kappa), recency(Other, TO, \kappa), TO < TR, state(TR), state(TO),$$
$$active(Other, \kappa, B), st\_element(recency, N).$$

$$recency(Rule, \kappa, \kappa) \leftarrow fireable(Rule, \kappa), not\ fireable(Rule, \kappa - 1).$$

$$recency(Rule, K, \kappa) \leftarrow recency(Rule, K, \kappa - 1), transitional(k), state(K).$$

$$recency(Rule, K, \kappa) \leftarrow fireable(Rule, \kappa), recency(Rule, K, \kappa - 1),$$
$$not\ transitional(\kappa), state(K).$$

New conflict resolution strategies can be added by adding new $st\_element(N, X)$ and implementing the respective $rejected(Rule, \kappa, X)$ predicates. This concludes the specification of RIF-PRD in iASP.

4.1.5 Transformation Analysis

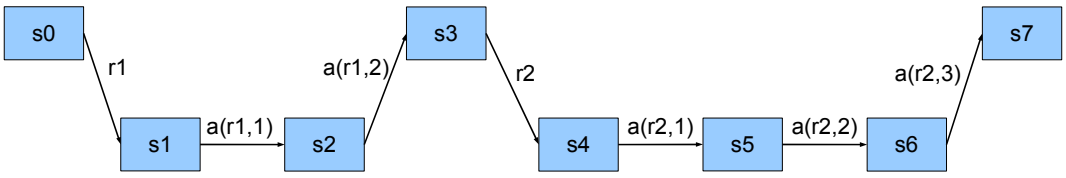
In this section we will demonstrate how our iASP implementation of RIF-PRD works, as well detail the limitations of this implementation.

Consider the states $s0 \ldots s7$, where $s0$ and $s7$ are respectively the initial and final state of the iASP program. Furthermore, let the two rules $r1$, $r2$ be:

```
(defrule r1
=>
(assert (act (r 1) (n 1)))
(assert (act (r 1) (n 2))))
```

```
(defrule r2
(act (r 1) (n 1))
=>
(assert (act (r 2) (n 1)))
(assert (act (r 2) (n 2)))
(assert (act (r 2) (n 3))))
```

have actions $a(r1, 1)$, $a(r1, 2)$, $a(r2, 1)$, $a(r2, 2)$, $a(r2, 3)$. Then:



state	s0	s1	s2	s3	s4	s5	s6	s7
state type	cycle	trans.	trans.	cycle	trans.	trans.	trans.	cycle
KB status	KB	KB	KB + a(r1,1)	KB + a(r1,1) + a(r1,2)	KB + a(r1,1) + a(r1,2)	KB + a(r1) + a(r2,1)	KB + a(r1) + a(r2,1) + a(r2,2)	KB + a(r1) + a(r2)

Figura 4.1: Execution of a RIF-PRD (iASP implementation) program.

The execution of our iASP program starts with an initial state, with an initial KB. The initial state is a system cycle state, therefore one of the activated rules, in this case $r1$, will be fired.

The next state is $s1$, that is a transitional state. The KB from $s0$ will be copied to $s1$ without changes. In $s1$ the first action of $r1$, $a(r1, 1)$ will be executed.

The following state is $s2$, that still is a transitional state, since we are still dealing with actions of $r1$. The KB of $s2$ is the KB of $s1$ plus the changes that $a(r1, 1)$ made into the KB. In $s2$, the second action of $r1$, $a(r1, 2)$ will be executed.

We now reach state $s3$, a cycle state. The KB is now the KB from $s2$ plus the changes that the action $a(r1, 2)$ made to the KB of $s2$. In this state $r2$ is fired.

The next states will be similar to $s1$ and $s2$, until we reach $s7$, a cycle state. Since there are no more rules to fire, the program terminates.

4.1.6 iASP implementation enhancements

4.1.6.1 Number of states reduced

In the original implementation (see **Figure 4.1.6.1**), the KB is copied at each state. As for each action of a fired rule a new state is created, the KB is being copied far too many times. With the reduction of all transitional states of a fired rule to a unique transitional state, the number of times the KB is copied is drastically reduced in rules with several actions. This achieved by simultaneously executing all the actions of a fired rule, yet maintaining their order to avoid any possible conflicts.

By reducing the number of transitional states per rule to one, we were able to introduce a new rule to our program that checks if a state is transitional, by calculating if that state is even or odd.

$$transitional(\kappa) \leftarrow (\kappa \; mod \; 2) == 0.$$

Definition 30 (Fact base improvement)**.**

$$fact(F, \kappa) \leftarrow not \; transitional(\kappa), \; fact(F, \kappa - 2), \; not \; retracted(F, \kappa - 1).$$

$$fact(F, \kappa) \leftarrow not \; transitional(\kappa), \; assert(F, \kappa - 1, A), \; not \; retractedAfter(F, \kappa - 1, A).$$

$$retracted(F, \kappa) \leftarrow retract(F, \kappa, A), \; not \; assertedAfter(F, \kappa, A).$$

$$assertedAfter(F, \kappa, A) \leftarrow retract(F, \kappa, A), \; assert(F, \kappa, B), \; B > A.$$

$$retractedAfter(F, \kappa, A) \leftarrow retract(F, \kappa, B), \; assert(F, \kappa, A), \; B > A.$$

A fact F holds in state κ if it was present in $\kappa - 2$ (the previous non-transitional state) and $retracted(F, \kappa - 1)$ holds. $retracted(F, \kappa)$ holds if F was retracted in κ and there was not an assert of F in the sequence of the fired action block.

A fact F holds in κ if it did not exist in the previous non-transitional state and was asserted in $\kappa - 1$ (previous transitional state) without being retracted afterwards.

Definition 31 (Actions improvement)**.** *The improved translation of RIF-PRD actions is then:*

$action(\gamma, k, \psi) : -instance(r_i, \phi, k).$

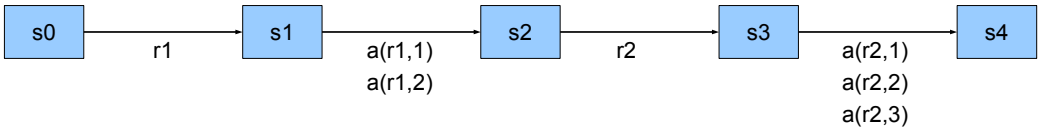
where γ represents a RIF-PRD action, ψ represents the position of γ in the execution order, and ϕ are the free variables of r_i.

Example 17 (Actions improvement)**.**
Consider the original translation:

```
action(assert(isa(newCar,vehicle),k+1) :- instance(r1,subs,k).
action(assert(frame(newCar,color,black),k+2) :- instance(r1,subs,k).
action(assert(frame(newCar,id,333),k+3) :- instance(r1,subs,k).
```

Executing all actions in the same state raises the need of maintain the order which the actions will be executed. Therefore, the new translation is as follows:

```
action(assert(isa(newCar,vehicle),k,1) :- instance(r1,subs,k).
action(assert(frame(newCar,color,black),k,2) :- instance(r1,subs,k).
action(assert(frame(newCar,id,333),k,3) :- instance(r1,subs,k).
```



state	s0	s1	s2	s3	s4
state type	cycle	trans.	cycle	trans.	cycle.
KB status	KB	KB	KB + a(r1)	KB + a(r1)	KB + a(r1) + a(r2)

Figura 4.2: New execution of a RIF-PRD (iASP implementation) program .

In the worst case, the new execution requires the same number of the states when compared to the the original transformation (in programs with rules that have several actions this proportion will be significantly lower when compared to the original execution). As said before, for every rule that is fired, all of its actions are executed in the same transitional state, therefore reducing the number of states and consequently, the number of times the KB is copied.

4.1.6.2 Reducing the number of facts in the KB

This improvement filters the number of facts in the KB. To achieve this we introduce the notion of plausible fact. A fact is plausible if it matches the conditions of any rule in the program.

Definition 32 (Plausible facts enhancement)**.** *for every* $state(\psi, k)$ *predicate in a fireable/2,* $exists_\phi/2$*,* $not_\phi/2$*,* $or_\phi/2$ *rule, add to the program a rule of the form:*

$$plausible(\psi, k) \leftarrow fact(\psi, k-2).$$

To guarantee that facts that were asserted in $\kappa - 1$ *are not ignored, a further rule is added to the program:*

$$plausible(\psi, \kappa) \leftarrow not\ transitional(\kappa), assert(\psi, \kappa - 1, P).$$

The KB now only stores facts that match conditions of the rules, plus the asserted facts of the previous transitional state.

Example 18 (Plausible facts enhancement)**.**
Consider the following rule:

```
fireable(rule(r1,subs),k) :- state(isa(VAR_X,student),k).
```

A rule $plausible(F, \kappa)$ will be added:

```
plausible(isa(VAR_X,student),k) :- fact(isa(VAR_X,student,k-2).
```

The fact base and state maintenance will now be:

$fact(F, \kappa) \leftarrow not\ transitional(\kappa), plausible(F, \kappa), not\ retracted(F, \kappa - 1).$

$fact(F, \kappa) \leftarrow not\ transitional(\kappa), assert(F, \kappa - 1, A), not\ retractedAfter(F, \kappa - 1, A).$

$state(F, \kappa) \leftarrow fact(F, \kappa).$

$state(sub(C1, C2), \kappa) \leftarrow fact(sub(C1, C2), INIT).$

$state(isa(O1, C2), \kappa) \leftarrow state(isa(O1, C1), \kappa), state(sub(C1, C2), \kappa).$

$state(sub(C1, C3), \kappa) \leftarrow state(sub(C1, C2), \kappa), state(sub(C2, C3), \kappa).$

This improvement will filter the facts that are passed on through the successive states, being the KB free of facts that would never be used by any rule in the program, reducing the size of the grounded program.

4.1.7 Conflict resolution focusing

The next enhancement we performed was reducing the number of generated rules in the grounding. With the original translation, a large number of rules were generated due to the conflict resolution. The issue resided in the priority and recency elements, that generated a large number rules for comparing the priority and recency of fireable rules. By performing some changes to those two strategy elements the number of generated rules could be reduced, affecting positively the performance of our implementation. The following improvement also limits the strategy to *rif:forwardChaining* instead of considering general conflict resolution strategies.

Definition 33 (Conflict resolution improvement)**.**

$$picked(Rule, \kappa) \leftarrow pickable(Rule, \kappa), not\ pick\_other(Rule, \kappa).$$

$$pick\_other(Rule, \kappa) \leftarrow pickable(Rule, \kappa), pickable(Rule1, \kappa), Rule1 > Rule.$$

$$pickable(rule(Id, Var), \kappa) \leftarrow fireable(rule(Id, Var), \kappa), not\ refracted(rule(Id, Var), \kappa),$$
$$priority(Id, P), max\_priority(P, \kappa), recency(rule(Id, Var), R, \kappa), max\_recency(R, \kappa).$$

$$\%Priority\ Rules\%$$

$$max\_priority(P, \kappa) \leftarrow priorities\_of\_fireable(P, \kappa), not\ no\_max\_prio(P, \kappa).$$

$$no\_max\_prio(P, \kappa) \leftarrow priorities\_of\_fireable(P, \kappa), priority(\_, P1),$$
$$P1 > P, priorities\_of\_fireable(P1, \kappa).$$

$$priorities\_of\_fireable(P, \kappa) \leftarrow fireable(rule(Id, Var), \kappa),$$
$$not\ refracted(rule(Id, Var), \kappa), priority(Id, P).$$

$$\%Recency\ Rules\%$$

$$max\_recency(R, \kappa) \leftarrow recencies\_of\_prioritary(R, \kappa), not\ no\_max\_rec(R, \kappa).$$

$$no\_max\_rec(R, \kappa) \leftarrow recencies\_of\_prioritary(R, \kappa), state(R1),$$
$$R < R1, recencies\_of\_prioritary(R1, \kappa).$$

$$recencies\_of\_prioritary(R, \kappa) \leftarrow max\_priority(P, \kappa), priority(Id, P),$$
$$fireable(rule(Id, Var), \kappa), not\ refracted(rule(Id, Var), \kappa), recency(Rule, R, \kappa).$$

$$recency(Rule, \kappa, \kappa) \leftarrow fireable(Rule, \kappa), not\ fireable(Rule, \kappa - 2).$$

$$recency(Rule, K, \kappa) \leftarrow fireable(Rule, \kappa), recency(Rule, K, \kappa - 2), state(K).$$

$$\%Refraction\ Rules\%$$

$$refracted(Rule, \kappa) \leftarrow fireable(Rule, \kappa), picked(Rule, \kappa - 2).$$

$$refracted(Rule, \kappa) \leftarrow fireable(Rule, \kappa - 2), refracted(Rule, \kappa - 2).$$

With this new implementation of the *rif:forwardChaining* strategy, we reduced the number of generated rules in the grounding by removing the rules $rejected/3$ from the original translation.

In the original implementation, the order of the three strategy elements was ensured by the rules $inactive/3$ and $active/3$. In the new implementation, since this order is ensured in the rule $pickable/2$, these two rules were eliminated, thus reducing the number of rules generated in the grounding.

The priority element rules and recency element rules computation was drastically modified. In the original implementation, to get the set of rules with higher priority, given a set of rules χ, representing the rules that were not discarded by refraction, we would check for every $r_i \in \chi$ if there were not any rules in χ with higher priority than r_i.

In the new implementation, given the set of rules χ, that were not discarded by refraction, we first check the highest priority π achievable in χ with the rule $max\_priority/2$. Then, we compare every rule $r_i \in \chi$ with π. Finally, the set of rules with priority π is returned.

Originally, to get the most recent rules, given a set of rules χ , representing the rules that were not discarded by priority, we would check for every $r_i \in \chi$ if there were not any rules in χ that were more recent than r_i.

Analogously to the new priority element rules, given the set of rules χ that were not discarded by priority, we first check the most recent value φ achievable in χ with the rule $max\_recency/2$. Then, we compare every rule $r_i \in \chi$ with φ. The set of rules with recency φ is returned.

The refraction element rules did not suffer changes with the new implementation.

4.2 RIF-PRD to Jess implementation

The second translation maps RIF-PRD to Jess. Jess is a production rule system, hence, the RIF-PRD translation was less complex than the one to iASP, as Jess supports by default some of the constructs present in RIF-PRD.

The goal of this implementation was to provide a comparison to the iASP translation, while providing a production rules system implementation of RIF-PRD, that is closer to what might be used in business applications.

The working memory in Jess stores facts. In Jess, facts are the most basic construct, and shall be used to implement the atomic formulas of RIF-PRD. Facts must have a template declared before they can be used, therefore, based upon the iASP declarative semantics, we created four templates:

- frame - `(deftemplate frame (slot _obj) (slot _pred) (slot _val))`
- membership atomic formula - `(deftemplate member (slot _elem) (slot _isa))`
- subclass atomic formula - `(deftemplate sub (slot _sub) (slot _super))`
- atom - `(deftemplate atom (slot _op) (multislot _args))`

Definition 34 (Atomic formulas translation)**.**
The Jess translation of atomic formulas is defined as follows:

- *frame atomic formula* $o[p \rightarrow v]$ *is mapped to* $frame(\_obj\ o, \_pred\ p, \_val\ v)$*, where* o *is the object,* p *is the property and* v *the value;*
- *membership atomic formula* $o\#c$ *is mapped to* $member(\_elem\ o, \_isa\ c)$*, where* o *is of class* c*;*
- *subclass atomic formula* $s\#\#c$ *is mapped to* $sub(\_sub\ s, \_super\ c)$*, where* s *is subclass of* c*;*
- *atom is mapped* $t(a_1, \ldots, a_n)$ *to* $atom(\_op\ t, \_args\ a_1, \ldots, a_n)$*, where* t *is the name of the atom, and* $a_1, \ldots, a_n$ *are the arguments.*

Regarding states translation, Jess internally maintains and updates the fact base (called working memory in Jess). As a result, there was only a need to implement the class hierarchy rules:

Definition 35 (Class hierarchy rules translation)**.**

```
(defrule MAIN::ruleCH
(member (_elem O1) (_isa C1))
(sub (_sub C1) (_super C2))
=>
(assert (member (_elem O1) (_isa C2))))

(defrule MAIN::ruleSH
(sub (_sub C1) (_super C2))
(sub (_sub C2) (_super C3))
=>
(assert (sub (_sub C1) (_super C3))))
```

The first rule is class inheritance. If $O1$ is of class $C1$ and class $C1$ is subclass of $C2$, then $O1$ is of class $C2$. The second rule represents transitivity of subclass relationship. If $C1$ is subclass of $C2$ and $C2$ is subclass of $C3$, then $C1$ is subclass of $C3$ (see sections 2.2.3).

4.2.1 RIF-PRD conditions

The translation of RIF-PRD conditions to Jess uses Jess internal constructs with the exception of the Exists formula, that closely follows the iASP translation.

Definition 36 (Conditions translation)**.** *Let Φ be an arbitrary condition formula. Define condition Jess formula Φ' as follows:*

- *If Φ is an atomic formula φ then $\Phi' = \varphi'$ where φ' is the corresponding Jess template;*
- *If $\Phi = And(\phi_1 \dots \phi_n)$ then $\Phi' = (\phi_1 \dots \phi_n)$;*
- *If $\Phi = Or(\phi_1 \dots \phi_n)$ then $\Phi' = or(\phi_1 \dots \phi_n)$ where $or(\dots)$ is a Jess conditional element;*
- *If $\Phi = Not(\phi)$ then $\Phi' = (not\ (\phi))$, where $not(\dots)$ is a Jess conditional element;*
- *If $\Phi = Exists\ ?V_1 \dots ?V_n\ (\phi)$ then $\Phi' = (ex\ (\_num\ \alpha)\ (\_var\ ?X_1 \dots ?X_n)$. The fact $(ex\ (\_num\ \alpha)\ (\_var\ ?X_1 \dots ?X_n)$ has two slots, $(\_num\ \alpha)$ which is the number of exists formulas seen in the program so far, and a multislot $(\_var\ ?X_1 \dots ?X_n)$, that are the free variables of the formula. Additionally, a new rule $exists_\alpha$ is added to the program:*

  $(defrule\ MAIN :: exists\alpha$

  $(declare\ (salience\ 1000))$

  (ϕ)

  $\Rightarrow$

  $(ex\ (\_num\ \alpha)\ (\_var\ ?X_1 \dots ?X_n)))$

The declared salience in the above rule avoids inconsistencies has it fires before any rule of the RIF-PRD program.

Example 19 (Conditions translation)**.**

```
(* Annotation : rule r1 *)
If And(
        ?car#vehicle
        Exists ?color (
                And (
                    INeg( ?car[Id->123] )
                    Or(
                      ?car[color->?color Id->456]
                      ?car[color->?color Id->789]
           )        )        )        )
Then Do( assert( ?car[status->forSale] ))
```

The Jess translation is as follows:

```
(defrule MAIN::rule1
(member (_elem ?car) (_isa vehicle))
(ex (_num 1) (_var ?car))
=> (assert (?car (status forSale))))

(defrule MAIN::exists1
(declare (salience 10000))
(not (frame (_obj ?car) (_pred Id) (_val 123)))
(or (And(
          (frame (_obj ?car) (_pred color) (_val ?color))
          (frame (_obj ?car) (_pred Id) (_val 456))))
        (And(
          (frame (_obj ?car) (_pred color) (_val ?color))
          (frame (_obj ?car) (_pred Id) (_val 789))))
)
=> (assert (ex (_num 1) (_var ?car))))
```

The numbering mechanism $(\_num\ slots)$ is used in the translation to simplify generation of the auxiliary predicates required for existential formulas.

4.2.2 RIF-PRD actions

The Jess translation of RIF-PRD actions use internal Jess constructs. Though, some limitations of Jess made the translation of some actions non-trivial.

Definition 37 (Actions translation)**.** *Consider ϕ as a fact:*

- *Assert fact: It is translated to $(assert(\phi))$;*
- *Retract fact: It is translated to $(retract?p))$ where $?p$ is a Jess.Fact java object correspondent to ϕ. To obtain this, a series of queries (to match the 4 atomic formulas) were added to the program. Also, the following statements were added to the action part of the rule:*

$$(bind\ ?result\ (run-query*\ MAIN::query_\phi\ args_\phi))$$
$$(while\ (call\ ?result\ next)\ (bind\ ?p\ (call\ ?result\ getObject\ type_\phi))$$
$$(retract\ ?p)))$$

- *Retract all slots: Consider a terms t_1 and t_2. Retract all slots will be translated using the same method as Retract Fact, using one query to catch all frames with object t_1 and property t_2.*
- *Retract Object: Consider a term t. Retract object will be translated using the same method as Retract Fact, using one query to catch all membership atomic formulas where t is an object and a second one to catch all frames with object t.*
- *Modify: Consider a frame f. Modify will be translated in a similar way to the retract actions, by using a query to catch the frame f, and then executing the jess intern action $(modify\ f\ new\_v)$, where new_v is the new value of the property of f.*

The $query_\phi$ represents the query that obtains all facts (in the form of *Jess.Fact* JAVA objects) that correspond to ϕ, where $args_\phi$ are the arguments of ϕ. The variable $?result$ returns an iterator of $query_\phi$. Afterwards, each element of $?result$ is bound to $?p$ and consequently retracted.

Example 20 (Actions translation)**.**

```
(* Annotation : rule r1 *)
If Exists ?color  ?car (
        And (
            ?car#vehicle
            INeg( ?car[Id->123] )
            Or(
              ?car[color->?color Id->456]
              ?car[color->?color Id->789]
   ))        )
Then Do(
        assert( newCar[color->?color] )
        retract( ?car[color->?color] )))
```

The Jess translation is as follows:

```
(defrule MAIN::rule1
 (ex (_num 1) (_var ?color ?car))
=>
  assert( frame(newCar,color,?color))
  (bind ?result (run-query* MAIN::frameQuery ?car color))
  (while (call ?result next) (bind ?p (call ?result getObject frame))
  (retract ?p)))
)
```

As Jess does not allow pattern matching in the rhs of a rule, we opted by implementation queries that return a specific fact needed for retract actions. Several queries were implemented in order to cover the four different types of atomic formulas. In the above example a query for obtaining a frame atomic formula was used. The implementation of the query above is as follows:

```
(defquery MAIN::frameQuery
   (declare (variables ?obj ?pred))
   ?frame <- (frame (_obj ?obj) (_pred ?pred)))
```

4.2.3 RIF-PRD rules

The Jess translation of RIF-PRD rules follows closely the iASP translation, by transforming *Forall* quantified rules into conditional action blocks.

Definition 38 (Translation of RIF-PRD rules)**.** *Let r_i be a RIF production rule and let id be a unique identifier assigned to that rule (i.e. its "name"):*

If r_i is $Do((?V_1\ b_1) \ldots (?V_n\ b_n)\ a_1 \ldots a_m)$ then include in the program the following rule:

$$(defrule\ r_i\ \ \Rightarrow\ \ (bind_{V1}) \ldots (bind_{Vn})\ (a_1) \ldots (a_m))$$

If r_i is $If\ \Phi\ Then\ Do((?V_1\ b_1) \ldots (?V_n\ b_n)\ a_1 \ldots a_m)$ then include in the program the following rule where $X_1 \ldots X_l$ are the free variables of r_i:

$$(defrule\ r_i\ \ \Phi\ \ \Rightarrow\ \ (bind_{V1}) \ldots (bind_{Vn})\ (a_1) \ldots (a_m))$$

If r_i is $Forall\ ?V_1 \ldots ?V_n$ such that $(p_1 \ldots p_m)\ If\ \Phi\ Then\ Do(B)$ then include in the program the following rule:

$$(defrule\ r_i\ \ (p_1) \ldots (p_m)\ (\Phi)\ \ \rightarrow\ \ (B))$$

Furthermore, in the above rules substitute $(bind_{Vi})$ by:

$(bind\ \_r\alpha\_v\beta)$, if V_i is a $New()$ action variable. α is the number id associated to the rule, and β is the number action variables defined before (and inclusive) $bind_{Vi}$.

If $(bind_{Vi})$ is a frame $o[p \rightarrow V_i]$, then the substitution will not be made in the right hand side of the rule. In this case, a new pattern will be added to the left hand side of the rule, $(frame\ (\_obj\ \psi)\ (\_pred\ \phi)\ (\_val\ V_i))$. It is then extremely important to use different variable names for regular variables and action variables, otherwise conflicts may and most certainly will occur, causing the rule not to fire correctly.

At each state, a new rule is chosen by the Jess engine. This is done internally, using the RETE algorithm (see section 2.1.1) to that effect, which will return all the pickable

rules in the current state. If more than one rule is returned, then Jess will choose one by applying a conflict resolution strategy to the set of pickable rules.

Jess features the possibility to add user made conflict resolution strategies. Hence, it was possible to implement the $rif : forwardChaining$ strategy.

The execution of a RIF-PRD program in Jess will terminate when there are no more eligible rules to fire. This will be done internally by the Jess engine.

4.2.4 RIF-PRD conflict resolution

Jess offers by default two different conflict resolution strategies: depth strategy and breadth strategy. In depth strategy, if two rules have the same priority, the most recent one will fire. In breadth strategy, the rules are fired in the order they were activated.

One of the three elements of the *rif:forwardChaining* strategy is refraction. Since the Jess engine by default applies refraction to all the rules, no matter what strategy is used, then the depth strategy of Jess is equivalent to the *rif:forwardChaining* strategy of RIF-PRD.

To provide a better understanding on how to develop *rif:forwardChaining*, we present a JAVA implementation of *rif:forwardChaining* (using Jess libraries) in this subsection.

The $rif : forwardChaining$ strategy is formed by three elements: refraction, priority and recency. The Jess rule engine by default performs refraction regardless the chosen conflict resolution strategy.

The priority element is implemented in the $priority(Activation\ act1, Activation\ act2)$ method. It evaluates the rules priority and returns an integer representing which rule has higher priority. If both rules have the same priority, it returns 0.

```
public int priority(Activation act1,Activation act2)
{
        int sal1 = act1.getSalience();
        int sal2 = act2.getSalience();
        if( sal1  < sal2)
           return 1;
        else if ( sal1 > sal2)
           return -1;
        else return 0;
}
```

The recency element is implemented in the $recency(Activation\ act1, Activation\ act2)$ method. It evaluates the rules activation time and returns an integer representing which rule is more recent. If both rules have the same recency, it returns 0.

```
public int recency(Activation act1,Activation act2)
{
        int time1 = act1.getToken().getTime();
        int time2 = act2.getToken().getTime();
        if (time1 < time2)
           return 1;
        else if (time1 > time2)
          return -1;
        else return 0;
}
```

The $rif : forwardChaining$ Jess implementation main method is $compare(Activation\ act1, Activation\ act2)$, that compares two rules and returns an integer representing one of the rules.

```
public int compare(Activation act1, Activation act2)
{
        int result = priority(act1,act2);
        if(result == 0){
           result = recency(act1,act2);
           if(result == 0)
              return 1;
        }
        return result;
}
```

The compare method starts by applying the priority element to the rules $act1$ and $act2$. If none of the rules is discarded, the element recency will be applied. Finally if the two rules still are in conflict, then the rule `act1` is chosen.

4.3 RIF-PRD to XSB implementation

The third implementation we implemented was RIF-PRD to XSB. XSB is a logic programming system that offers nearly all functionality of ISO-PROLOG.

This implementation is meant mostly to compare to the iASP translation. Our main goal was to study the performance impact of the XSB incremental tabling mechanism in comparison with the incremental system of iClingo.

To maintain the fact base, we used a recent and hardly explored XSB feature: incremental tabling. With incremental tabling, facts can be stored and updated when changes occur in the program, such as modification or removal of a fact. The table used in our implementation to maintain the KB is an incremental table γ that stores predicates $fact/1$. Whenever an assert action is executed, the asserted fact is stored in γ. Consequently, if a retract action is executed, the retracted facts are removed from γ, and tables that depend on γ are updated.

Definition 39 (Facts and states translation)**.** *The states translation follows closely the ASP translation 18. The only difference is the absence of the argument κ, since XSB is not an incremental system.*

$$:- dynamic\ fact/1\ as\ incremental.$$

$$:- index(fact/1, trie).$$

$$:- table\ state/1\ as\ incremental.$$

$$state(F) \leftarrow fact(F).$$

$$state(isa(O1, C2)) \leftarrow state(isa(O1, C1)), state(sub(C1, C2)).$$

$$state(sub(C1, C3)) \leftarrow state(sub(C1, C2)), state(sub(C2, C3)).$$

The above rules correspond to the definition expressed in 18. Thus any change in the $fact/1$ table will incrementally update the table $state/1$.

Example 21 (Facts and states update)**.**

Consider a KB formed by the three following facts:

```
fact(isa(kit,car)).
fact(sub(car,motor_vehicle)).
fact(sub(motor_vehicle,vehicle)).

| ?- fact(X), writeln(X), fail.

sub(motor_vehicle,vehicle)
sub(car,motor_vehicle)
isa(kit,car)

no
```

```
| ?- state(X), writeln(X), fail.

sub(motor_vehicle,vehicle)
sub(car,motor_vehicle)
isa(kit,car)
isa(kit,motor_vehicle)
isa(kit,vehicle)
sub(car,vehicle)

no
```

Consider now the assertion of a new fact $isa(kat, car)$. $state/1$ table will be updated as follows:

```
| ?- incr_assert(fact(isa(kat,car))).

| ?- state(X), writeln(X), fail.
sub(motor_vehicle,vehicle)
sub(car,motor_vehicle)
isa(kat,car)
isa(kit,car)
isa(kat,motor_vehicle)
isa(kit,motor_vehicle)
isa(kat,vehicle)
isa(kit,vehicle)
sub(car,vehicle)
```

4.3.1 RIF-PRD conditions

Analogously to the states translation, the translation of RIF-PRD conditions to XSB follows closely the iASP translation (see **Definition 19**), with the absence of the κ argument.

Example 22 (XSB translation of Example 14)**.**

```
fireable(rule(rule1,subs)) :- exists1.

exists1 :- state(isa(VAR_car,vehicle)),
  not arg1(VAR_car), or1(VAR_car,VAR_color).

arg1(VAR_car) :- state(frame(VAR_car,id,123)).

or1(VAR_car,VAR_color) :- state(frame(VAR_car,color,VAR_color)),
  state(frame(VAR_car,id,456)).

or1(VAR_car,VAR_color) :- state(frame(VAR_car,color,VAR_color)),
  state(frame(VAR_car,id,789)).
```

4.3.2 RIF-PRD actions

As seen before, the facts are stored in tables. Therefore, the translation of RIF-PRD actions to XSB will be essentially operations to those tables. The two operations used are: $incr\_assert\_inval/1$ and $incr\_retractall\_inval/1$. $incr\_assert\_inval/1$ asserts a fact to a table without updating the table. The table update will come after all actions of the rule have been executed, with the call of the operation $incr\_table\_update/0$. $inc\_retractall\_inval/1$ will retract a fact from a table, without updating the table.

Definition 40 (Actions translation)**.** *Consider ϕ a fact. The XSB translation of RIF-PRD actions is as follows:*

- *Assert fact ϕ :*

  $incr\_assert\_inval(\phi)$*;*

- *Retract fact ϕ :*

  $incr\_retractall\_inval(\phi)$*;*

- *Retract all slots γ β :*

  $incr\_retractall\_inval(fact(frame(\gamma, \beta, \_)))$*, where γ and β are RIF-PRD terms;*

- *Retract object γ:*

  $incr\_retractall\_inval(fact(frame(\gamma, \_, \_)))$,
  $incr\_retractall\_inval(fact(member(\gamma, \_)))$ *where γ is a RIF-PRD term;*

- *Modify :*

  $incr\_retractall\_inval(fact(frame(\gamma, \beta, \_)))$,
  $incr\_assert\_inval(fact(frame(\gamma, \beta, \psi)))$ *where* γ, β *and* ψ *are RIF-PRD terms.*

For each RIF-PRD rule r_i, a new rule will be added to the program:

$$actions(Step, rule(Id_{r_i}, subs(S_1 \dots S_n)) :- picked(Step, rule(Id_{r_i}, subs(S_1 \dots S_n)), bind_{v1}, \dots, bind_{vb}, r_{i_{ACTION_1}}, \dots, r_{i_{ACTION_m}}.$$

where $S_1 \dots S_n$ are the free variables of r_i, and $bind_{v_i}$ is $state(frame(o, s, V_i), \kappa)$ if $b_i = o[s \rightarrow ?V_i]$, otherwise, $b_i = New()$, and let $bind_{v_i}$ be $V_i = obj(id, i, \kappa)$ with obj and arbitrary but fixed constant symbol.

Since XSB follows closely the iASP translation, the $Step$ variable is equivalent to the κ in the iASP translation, it indicates a state.

The body goals $r_{i_{ACTION_1}}, \dots, r_{i_{ACTION_m}}$ are the actions of r_i, as obtained by Definition 40. It is important to mention that the order of the actions is crucial, as they shall be executed from left to right when $actions/2$ predicate is called.

Example 23 (Translation of Example 15)**.**

```
actions(Step,rule(r1,subs)) :- picked(Step,rule(r1,subs)),
 incr_assert_inval(isa(newCar,vehicle)),
 incr_assert_inval(frame(newCar,color,black)),
 incr_assert_inval(frame(newCar,id,333)),
 incr_retractall_inval(frame(newCar,color,_)),
 incr_assert_inval(frame(newCar,color,white)),
 incr_retractall_inval(frame(newCar,_,_)),
 incr_retractall_inval(isa(newCar,_)).
```

Contrary to the $fact/1$ and $state/1$ predicates we need to maintain the step in the action rules because of the conflict resolution strategy (see below).

4.3.3 RIF-PRD rules

Analogously to the states translation, the translation of RIF-PRD conditions to XSB follows closely the ASP translation (see **Definition 19**), with the absence of the κ argument.

As happens with the other implementation, at each state, a rule will be chosen to fire by XSB.

A new table was added to the program, that stores the predicate $fireable/2$, allowing to know which rules might be eligible to fire in a given state.

If it is the first state, XSB will choose one of the fireable rules. In the successive states, the picked rule will be one that is fireable at that state, and that successfully passes the conflict resolution strategy.

Definition 41 (Pick rule)**.** *The act of picking a rule can then be implemented as follows:*

$$pick(Step, Rule) :- pickable(Step, Rule), assert(picked(Step, Rule)).$$

$$pickable(1, Rule) :- fireable(Rule).$$

$$pickable(Step, Rule) :- Step > 1, max\_priority(Step, MaxP), max\_recency(Step, MaxR),$$
$$fireable(Rule), Rule = rule(Name, \_), priority(Name, MaxP), tnot\, refracted(Step, Rule),$$
$$recency(Step, Rule, MaxR).$$

The $pickable/2$ predicate returns a rule that is fireable in the current state and that passes the $rif : forwardChaining$ strategy. If it is the first state, only the priority element will be applied, as the refraction and recency can not be applied in the first state.

4.3.4 RIF-PRD conflict resolution

Let us recall the above $pickable/2$ rule:

$$pickable(Step, Rule) :- Step > 1, max\_priority(Step, MaxP), max\_recency(Step, MaxR),$$
$$fireable(Rule), Rule = rule(Name, \_), priority(Name, MaxP),$$
$$tnot\, refracted(Step, Rule), recency(Step, Rule, MaxR).$$

The $max\_priority/2$ will return the highest priority $MaxP$ possessed by a fireable rule in state $Step$. Analogously, the $max\_recency/2$ will return the state $MaxR$ of the most recent fireable rule in $Step$. Afterwards, $Rule$ will unify with a RIF-PRD rule that has priority $MaxP$, recency $MaxR$ and that was not in the $refracted/2$ table.

It is important to mention that the tables that store the $max\_priority/2$ and $max\_recency/2$ predicates have a different definition than the above tables mentioned, resorting to partial order answer subsumption.

$$:- table\ max\_priority(\_, po('>'/2)).$$

$$:- table\ max\_recency(\_, po('>'/2)).$$

These two tables store at each step α the answers to predicates $priority/2$ and $recency/2$ of every pickable rule in α. These answers will be stored in an ordered way, simplifying the current implementation of $rif : forwardChaining$ in XSB, where only the solution in the second argument will be kept and returned, that is, only highest priority rules and most recent rules will be kept in the table.

The $max\_priority/2$ returns the highest priority P of a non refracted rule $Rule$ in state $Step$.

$$max\_priority(Step, P) :- fireable(Rule), tnot\ refracted(Step, Rule), \\ Rule{=}rule(R, \_), priority(R, P).$$

The $max\_recency/2$ returns the state Rec of the most recent fireable rule in $Step$.

$$max\_recency(Step, Rec) :- recency(Step, Rule, Rec), tnot\ refracted(Step, Rule).$$

If a rule is fireable in the initial state, then its recency is 1. In the successive states S, if a rule had recency $R1$ in $S-1$ then the recency in S will also be $R1$, otherwise, the recency in S will be equal to S.

$$recency(1, Rule, 1) :- fireable(Rule).$$

$$recency(S, Rule, R) :- S > 1, S1\ is\ S-1, fireable(Rule), \\ (recency(S1, Rule, R1) -> R = R1; R = S).$$

Considering the current state $Step$, a rule instance is refracted if it was picked in the previous state $Step-1$ and still is fireable in $Step$.

Also, a rule $Rule$ is refracted in $Step$ if it is fireable in $Step$, but was refracted in $S1$, which corresponds to the activation state of $Rule$.

$$refracted(Step, Rule) :- Step > 1, S1\ is\ Step-1, \\ fireable(Rule), picked(S1, Rule).$$

$$refracted(Step, Rule) :- Step > 1, S1\ is\ Step-1, \\ fireable(Rule), refracted(S1, Rule), \quad recency(S1, Rule, \_).$$

The execution of a translated RIF-PRD program to XSB will be initiated by calling the predicate $rif\_prd(0)$. The execution then will loop through the rule $rif\_prd(Step)$ until termination. In other words, the program terminates when there are no more eligible rules to fire.

Definition 42 (RIF-PRD main loop)**.**

$$rif\_prd(Step) :- pickAll(Step), pick(Step, Rule), !, \\ actions(Step, Rule), updateKB(Step), NewStep\ is\ Step+1, \\ rif\_prd(NewStep).$$

$$rif\_prd(\_).$$

The program will start by getting all the eligible rules with $pickall(Step)$, and will choose one of them in $pick(Step, Rule)$. Then, the actions of $Rule$ will be executed, the tables will be updated (with $updateKB(Step)$) and finally, a new state $NewStep$ will initiate. The cut guarantees that only one rule will be picked for execution at each step.

The $pickall(Step)$ at each Step ensures that the tables for the predicates $state/1$, $fireable/1$, $recency/3$, $refracted/3$ and $pickable/2$ are not incomplete.

$$pcickAll(Step) :- state(\_), fail.$$

$$pickAll(Step) :- fireable(\_), fail.$$

$$pickAll(Step) :- recency(Step, \_, \_), fail.$$

$$pickAll(Step) :- refracted(Step, \_), fail.$$

$$pickAll(Step) :- pickable(Step, \_), fail.$$

$$pickAll(\_).$$

This is an usual programming technique in XSB used to complete the tables.

The $updateKB(Step)$ updates and cleans the tables in the program. The definition is as follows:

$$updateKB(Step) :- incr\_table\_update, clean\_tables(Step).$$

$$clean\_tables(1) :- !.$$

$$clean\_tables(2) :- !.$$

$$clean\_tables(Step) :- Step2 \; is \; Step - 2,$$
$$abolish\_table\_call(recency(Step2, \_, \_)),$$
$$abolish\_table\_call(pickable(Step2, \_)),$$
$$abolish\_table\_call(refracted(Step2, \_)).$$

The $incr\_table\_update$ call will update the incremental tables. The $clean\_tables(Step)$ will clean unnecessary facts that still were stored in tables. Since only the previous state of predicates $recency/3$, $pickable/2$ and $refracted/2$ are needed for the $rif : forwardChaining$ strategy, the tables that harbour these predicates can erase all the information regarding the antepenultimate state.

As we seen in **Definition 42**, the execution of a RIF-PRD program will loop through the $rif\_prd(Step)$ rule. When a state is reached where there are no eligible rules to fire, XSB will call $rif\_prd(\_)$ and terminate, as it is a fact.

4.4 Conclusions

In this chapter we detailed the three implementations that were basis of the this work. The first implementation to be developed was the iASP, and by far the most complex implementation in this work, mainly due to the management of the free variables and action variables in the program. Other issues were the renaming of variables and IRI's so there would be no conflicts with the iClingo engine, and the generation of the identifiers for the exists, or and not formulas.

The implementation of RIF-PRD to Jess was more intuitive than the iASP implementation, as some of the constructs of RIF-PRD are present in the Jess system. We found some intricate issues to be solved, mostly related to the retract and modify actions, the exists conditional formula and the declaration of action variables. Due to the argument restrictions of the Jess retract formula, several queries had to be developed to return a Jess.Fact JAVA object used in the Jess retract construct. The action variables raised one limitation in the Jess implementation, as they had to be implemented in the left hand side of the rules, since Jess does not allow pattern matching in the right hand side of the rules. To implement the exists conditional formula we had to create a new Jess template, and, analogously to the iASP translation, create rules for each exists formula in a program.

The third and last implementation was the XSB implementation. The core of this implementation is the same as the iASP implementation, therefore this was the implementation that raised less issues. We evalute in pratice the three implementations in the next chapter.

5

Evaluation and Benchmarking

In this chapter we present evaluation and benchmark of our three RIF-PRD implementations. To evaluate our implementations, we used W3C RIF-PRD test cases [1]. To test the performance of our implementations, LUBM[2] was the chosen tool.

5.1 Implementation, compilation and execution

In this section we detail the implementation, compilation and execution of our three RIF-PRD transformations.

5.1.1 Implementation

Our three transformations of RIF-PRD were implemented using a XSLT stylesheet. Each transformation requires its own stylesheet, with circa of 1700 lines of code.

The largest transformation is the Jess stylesheet (that transforms RIF-PRD language to JessML), which was mainly caused by the XSL templates required to manage the templates and queries needed by the Jess implementation. Some of the additional problems with the Jess transformation were the action variables declaration, as Jess does not feature action variables, and the implementation of the retract and modify operations. Furthermore, in JessML, when using a Jess term, there is an attribute that defines that term as a constant or a variable. This was also one of the issues that the Jess implementation imposed, as it requires special care to define if a term is a constant or a variable.

The most complex implementation is the iClingo implementation, which requires

[1] http://www.w3.org/2005/rules/test/repository/PRDTests.xml
[2] http://swat.cse.lehigh.edu/projects/lubm/

passage of several parameters through diverse XSL templates to maintain the free variables and action variables, among other information. Furthermore, the iClingo XSL transformation processes the RIF-PRD document three times in order to complete the exists, or and negation conditional formulas and the plausible facts enhancement (see section 4.1.6.2).

The XSB XSL transformation follows closely the iClingo tranformation, therefore it was less complex to implement than the two other transformations.

To generate the data files needed to run the LUBM tests in our implementations, we created a JAVA class that translates original CLASP tests into our three implementations specific language.

5.1.2 Compilation and execution

To create a translated RIF-PRD document, we apply a XSL transformation correspondent to the desired implementation (iClingo, Jess, XSB) to the RIF-PRD document. To execute the translated RIF-PRD documents, the following commands must be used:

- iClingo : `.\iclingo.exe .\translatedDocument.txt`
- Jess : `.\jess .\translatedDocument.xml`
- XSB : `.\xsb.exe -e "['translatedDocument.P'], rif_prd(0), halt."`

5.2 Evaluation

The W3C provides RIF-PRD test cases to evaluate a RIF-PRD processor, to see if it works as intended. According to [MMP10], these tests are divided in five types:

- Import Rejection: The test is passed if the processor indicates that the input document is rejected, and failed otherwise.
- Syntax
  - Positive Syntax: the test passes if the processor indicates that the document is a syntactically correct RIF-PRD document;
  - Negative Syntax: the test passes if the processor indicates that the document is a syntactically incorrect RIF-PRD document;
- Semantic
  - Positive Entailment : a conformant RIF consumer should report that the conclusion is entailed by the premises, should not report that the answer is undecided, and must not report that the conclusion is not entailed by the premises.
  - Negative Entailment: a conformant RIF consumer should report that the conclusion is not entailed by the premises, should not report that the answer is

undecided, and must not report that the conclusion is entailed by the premises.

5.2.1 Limitations

In this section we will present the limitations of our three RIF-PRD implementations.

5.2.1.1 General limitations

- No support for datatypes;
- No support for Equal atomic formula;
- No support for List terms;
- Limited support for RIF built-in actions. The supported built-in actions are:
  - greater than or equal ($\geq$) boolean operation;
  - greater than ($>$) boolean operation;
  - lower than or equal ($\leq$) boolean operation;
  - lower than ($<$) boolean operation;
  - subtraction arithmetic operation;
  - addition arithmetic operation;
  - multiply arithmetic operation;

5.2.1.2 Jess limitations

- All rules must have salience lower than 1000. Higher values are reserved for internal operations of the Jess implementation of RIF-PRD;
- Action variables and variables declared in the LHS, must have different names;

5.2.2 Tests performed

In this section we list the W3C test cases that we performed. The tests performed and successfully passed are:

- Positive Syntax
  - CoreSafeness
- Positive Entailment
  - Assert
  - AssertRetract
  - AssertRetract2

- ChainNumericAdd1
- ChainNumericSubtract2
- Frames
- FrameSlotsAreIndependent
- Modify
- ModifyLoop
- PositionalArguments

- Negative Entailment
  - Retract

Furthermore, we created some other tests, which we partially provide in the Appendix A, which check other RIF-PRD features not covered in the official RIF-PRD test cases.

5.3 Benchmarking

In this section we will present the results obtained in the LUBM benchmark, presenting first the performance of the CLASP, DLV and Jess engines, and then the performance of our three implementations of RIF-PRD.

5.3.1 Benchmarking Tools

To evaluate the performance of our implementations, we used the Lehigh University Benchmark (LUBM) [3]. To generate the data needed to run these tests, we used OpenRuleBench[4], as it provides data in various formats, such as DLV (ASP) and Jess. The tests were performed in a Intel® Core™2 Quad Processor Q6600 (8M Cache, 2.40 GHz, 1066 MHz FSB) with 4GB Kingston® DDR3 1333MHZ.

[3] http://swat.cse.lehigh.edu/projects/lubm/
[4] http://rulebench.projects.semwebcentral.org/

5.3.2 Results

5.3.2.1 Default LUBM

In this section we present the results of LUBM tests using CLASP, DLV and Jess engines.

Tabela 5.1: CLASP, DLV and Jess LUBM performance

Univs	Queries	CLASP		DLV		Jess	
		Mean (s)	Var (s)	Mean (s)	Var (s)	Mean (s)	Var (s)
1	1	1.485	0.000	1.392	0.000	5.210	0.000
	2	1.491	0.000	1.510	0.000	5.765	0.001
	9	1.604	0.000	6.450	0.000	6.579	0.000
	14	1.510	0.008	1.492	0.001	5.305	0.001
5	1	9.114	0.001	8.820	0.000	26.173	0.016
	2	9.830	0.000	9.690	0.000	27.785	0.019
	9	9.932	0.003	221.147	0.038	32.109	1.255
	14	9.207	0.000	9.410	0.002	26.170	0.013
10	1	19.625	0.003	18.592	0.000	52.724	0.072
	2	20.643	0.000	21.904	0.000	55.370	0.135
	9	20.973	0.003	956.087	0.130	125.606	0.193
	14	19.313	0.003	19.806	0.006	52.635	0.071
20	1	42.043	0.003	40.226	0.001	110.087	0.249
	2	44.032	0.001	47.654	0.002	187.645	2.559
	9	45.397	0.012	> 1000	-	468.741	9.147
	14	41.176	0.011	42.852	0.030	109.376	0.571
50	1	105.525	0.213	104.360	1.325	> 1000	-
	2	108.120	0.039	148.223	0.032	> 1000	-
	9	108.376	0.088	> 1000	-	> 1000	-
	14	103.107	0.015	112.349	0.095	> 1000	-

> 1000: Execution timeout

5.3.2.2 Translated LUBM

In this section we present the results of LUBM tests using our RIF-PRD implementations. Note that the data files used in this section were created by translating the original LUBM tests of CLASP and Jess to iClingo and Jess (the original data files of Jess in OpenRuleBench have many specific templates, e.g. for universities and teachers). In our Jess implementation of RIF-PRD, the data is divided under four templates: atoms, frames, membership and subclass atomic formulas.

Tabela 5.2: RIF-PRD Translated LUBM performance

Univs	Queries	iClingo		XSB		Jess	
		Mean (s)	Var (s)	Mean (s)	Var (s)	Mean (s)	Var (s)
1	1	3.306	0.000	2.287	0.000	7.571	0.003
	2	N/F	-	> 1000	-	10.247	0.001
	9	N/F	-	> 1000	-	14.335	0.043
	14	N/F	-	> 1000	-	7.364	0.002
5	1	9.114	0.001	16.019	0.023	50.681	0.092
	2	9.830	0.000	> 1000	-	58.863	0.037
	9	9.932	0.003	> 1000	-	180.370	1.955
	14	9.207	0.000	> 1000	-	50.041	0.041
10	1	18.309	0.000	33.070	0.073	213.281	2.195
	2	N/F	-	> 1000	-	213.188	0.166
	9	N/F	-	> 1000	-	826.171	52.454
	14	N/F	-	> 1000	-	213.808	4.679
20	1	84.575	0.024	N/F	-	890.384	41.139
	2	N/F	-	> 1000	-	508.825	81.254
	9	N/F	-	> 1000	-	> 1000	-
	14	N/F	-	> 1000	-	246.775	0.996
50	1	347.929	5.302	N/F	-	> 1000	-
	2	N/F	-	> 1000	-	> 1000	-
	9	N/F	-	> 1000	-	> 1000	-
	14	N/F	-	> 1000	-	> 1000	-

N/F: Aborted execution

> 1000: Execution timeout

For query 1, we observed that it is several times slower than the CLASP LUBM performance but still manages to terminate under 1000 seconds except in the fifty universities test.

In the one university tests, the queries 2, 9 and 14 aborted execution after 100 seconds, returning the exception `std::bad_alloc`. The reason for that exception is that the machine ran out of memory while executing the queries. This happens due to the large number of facts that are being generated in the grounding of each incremental state.

For the five, ten, twenty and fifty university tests, the behaviour, as expected, was similar.

In the Jess implementation of RIF-PRD tests we observed that for all universities the execution terminates, though in the tests of twenty and fifty universities Jess returned an exception related to garbage collection. At this point the performance of Jess suffered a drastic slowdown, but still managed to terminate.

Regarding the performance slowdown face the original Jess LUBM results, there is the possibility that it was caused due to the KB in our implementation being formed by only 4 different templates: atoms, membership formulas, subclass formulas and frames. In the Jess LUBM tests in OpenRuleBench, for each different class there is a specific template, which in principle makes the pattern matching faster, as there are less facts to search. We

suspect this was the cause for having worse results in our Jess RIF-PRD implementation.

Finally, regarding the XSB implementation, in the queries 2, 9 and 14, the performance is drastically worse than query 1. The query 1 was able to finish before the established time limit in the tests of one, five and ten universities. In the twenty and fifty universities tests, the execution was interrupted with an out of memory exception.

The reason behind the slow performance of the three queries is the incremental tabling mechanism of XSB, that in those three queries (that fire far more rules than query 1) the amount of facts to table is huge, causing an overall performance slowdown.

5.4 Yield RIF comparison

In this section, we will compare our Jess RIF-PRD implementation with the Yield RIF project Jess implementation, since Yield RIF Jess transformation is publicly available[5].

The Yield RIF Jess implementation provides a partial RIF-PRD translation to Jess, as it does only support quanti?ed rules, where our implementation supports quanti?ed rules, conditional action blocks and unconditional action blocks. Furthermore, our implementations supports all conditional formulas, whereas Yield RIF translation uniquely supports the And conditional formula. However, Yield RIF supports the Equal atomic formula that is not supported by our implementations. Regarding RIF-PRD actions, we have implemented in our three transformations the assert, retract and modify actions, whereas Yield RIF only supports assertion of atoms.

5.5 Conclusions

In this section we detailed the evaluation and benchmarking of our three RIF-PRD implementations. We were able to implement almost every feature of RIF-PRD (with few exceptions) in our three transformations. Though, as the results of the benchmarking have shown us, the RIF-PRD implementation in iASP and XSB approaches is hardly viable, as the LUBM tests with a larger number of rules failed to present a satisfiable performance in both approaches. On the other hand, Jess benchmarking results show that it is a viable option for implementing RIF-PRD, with satisfiable results in the 1, 5, 10 and 20 universities.

[5] http://yieldrif.appspot.com/script/5003

6

Conclusion

The driving force of this dissertation was to provide a better understanding of the RIF-PRD dialect, by presenting three consumer implementations of RIF-PRD using different approaches.

The aim of RIF-PRD is to provide a solution to the interchange of rules between production rules systems, as the development of a standard production rules language for production rules is not viable. If a standard production rules language was developed, some (or probably all of them) of the current production rules systems would not be able to process the standard language, as the existing production rules systems have huge differences between themselves.

Being RIF-PRD an interchange format, every production rules system can implement a translation from its own internal language to RIF-PRD and translate RIF-PRD language to its own language, then accomplishing the interchange of rules between production rules system.

In this work we presented an overview of production rules systems, including the popular matching algorithm RETE, an overview of the syntax and semantics of RIF-PRD and Answer Set Programming. The major contribution of this work were the three RIF-PRD consumer implementations, using different paradigms, production rules (Jess), iASP (iClingo) and logic programming (XSB).

The iASP consumer implementation of RIF-PRD was the most relevant in this work. It was based in a complete declarative specification [DAL10], that was the base for the whole work in this thesis. This declarative specification offers a more user friendly approach to RIF-PRD, and was the first complete declarative specification of RIF-PRD. This implementation was developed using iClingo, an ASP system that provides an incremental mode, which we used to simulate the stateful computation of RIF-PRD semantics.

Furthermore, we improved the declarative specification by introducing enhancements that improve the conflict resolution stategy, the incremental process of the KB, and management of the information in the KB.

The production rules consumer implementation of RIF-PRD provided a more real alternative, as RIF-PRD is oriented toward production rules systems. For this implementation we used the Jess, a production rules system written in JAVA that has its own markup language (JessML). In addition, using the Jess JAVA API we were also able to develop a JAVA implementation of the *rif:forwardChaining* conflict resolution strategy, for better understanding of the default RIF-PRD conflict resolution strategy.

The third consumer implementation we developed was the XSB implementation, whose aim was to compare the incremental tabling mechanism of XSB with the incremental mode of iClingo.

Finally, we compared the three implementations by using LUBM, providing results to show whether the implementations are viable or not.

Regarding the results of our three implementations, we concluded that though it is possible to implement RIF-PRD using iASP and XSB Prolog, these solutions are not viable for large sets of data, as they quickly ran out of memory, due to the large KB that was accumulated at each incremental step in iClingo, and in the XSB due to the incremental tabling, which caused a major slowdown to the execution. As to Jess, we concluded that it is a viable option for implementing RIF-PRD, providing a competitive approach to rule interchange as demonstrated by its benchmarking performance .

In future work, more built-in actions can be added to the implementations of RIF-PRD. Furthermore, specific implementation enhancements can be perform, such as optimizing the conflict resolution strategy in the declarative specification, optimizing the retract actions in Jess, and optimizing the tabled predicates in XSB.

Also, for the three consumer implementations of RIF-PRD there is the producer implementation still to be developed, enabling the full interchange of rules between the three implementations.

Finally, the integration of RIF-PRD with ontologies is a desirable feature, as RIF-PRD is a web-aware language.

Bibliografia

[BK09] Harold Boley e Michael Kifer. RIF overview. W3C working draft, W3C, Outubro 2009. http://www.w3.org/TR/2009/WD-rif-overview-20091001/.

[Bru10] Jos De Bruijn. RIF rdf and OWL compatibility. W3C recommendation, W3C, Junho 2010. http://www.w3.org/TR/rif-rdf-owl/.

[DAL10] Carlos Viegas Damasio, Jose Julio Alferes, e Joao Leite. Declarative semantics for the rule interchange format production rule dialect. In *9th International Semantic Web Conference (ISWC2010)*, November 2010.

[Doo95] Robert B. Doorenbos. *Production matching for large learning systems*. Tese de Doutoramento, Pittsburgh, PA, USA, 1995. UMI Order No. GAX95-22942.

[FH] E. Friedman-Hill. Jess, the rule engine for the java platform.

[For90] Charles L. Forgy. Expert systems. chapter RETE: a fast algorithm for the many pattern/many object pattern match problem, pÃ¡g. 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[GGKS11] M. Gebser, T. Grote, R. Kaminski, e T. Schaub. Reactive answer set programming. In J. Delgrande e W. Faber, editores, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pÃ¡g. 54–66. Springer-Verlag, 2011.

[Gia] J. Giarratano. Clips user's guide. Available at http://clipsrules.sourceforge.net/documentation/v630/ug.htm.

[GKK+08a] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, e S. Thiele. A user's guide to gringo, clasp, clingo, and iclingo. Unpublished draft, 2008. Available at URL http://downloads.sourceforge.net/potassco/guide.pdf?use_mirror=.

[GKK+08b] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, e Sven Thiele. Engineering an incremental asp solver. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pÃ¡g. 190–205, Berlin, Heidelberg, 2008. Springer-Verlag.

[GKK+08c] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, e Sven Thiele. Engineering an incremental ASP solver. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pÃ¡g. 190–205, Berlin, Heidelberg, 2008. Springer-Verlag.

[GL88] M. Gelfond e V. Lifschitz. The stable model semantics for logic programming. In *Proceeding of the Fifth Logic Programming Symposium*, pÃ¡g. 1070–1080, 1988.

[GPH05] Yuanbo Guo, Zhengxiang Pan, e Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158 – 182, 2005. Selcted Papers from the International Semantic Web Conference, 2004 - ISWC, 2004.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5:199–220, June 1993.

[HPdSM10] Gary Hallmark, Adrian Paschke, e Christian de Sainte Marie. RIF production rule dialect. W3C recommendation, W3C, Junho 2010. http://www.w3.org/TR/rif-prd.

[Kay07] Michael Kay. XSL transformations (XSLT) version 2.0. W3C recommendation, W3C, Janeiro 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[LFWK09] Senlin Liang, Paul Fodor, Hui Wan, e Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pÃ¡g. 601–610, New York, NY, USA, 2009. ACM.

[Lif08] Vladimir Lifschitz. What is answer set programming? In *AAAI*, pÃ¡g. 1594–1597, 2008.

[LT84] J. W. Lloyd e R. W. Topor. Making prolog more expressive. *The Journal of Logic Programming*, 1(3):225 – 240, 1984.

[MMP10] Stella Mitchell, Leora Morgenstern, e Adrian Paschke. Rif test cases. World Wide Web Consortium, Working Draft WD-rif-test-20100622, June 2010.

[PNM+07] Mark Proctor, Michael Neale, Bob McWhirter, Kris Verlaenen, Edson Tirelli, Alexander Bagerman, Michael Frandsen, Fernando Meyer, Geoffrey De Smet, Toni Rikkola, Steven Williams, e Ben Truit. Drools, 2007.

[PRH+10] Adrian Paschke, Dave Reynolds, Gary Hallmark, Harold Boley, Michael Kifer, e Axel Polleres. RIF core dialect. W3C recommendation, W3C, Junho 2010. http://www.w3.org/TR/rif-core/.

[RRS+99] I.v. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, e David S. Warren. Efficient access mechanisms for tabled logic programs. 1999.

[SS] Terrance Swift e David S.Warren. The XSB system version 3.3 volume 1: Programmer's manual.

[SW94] Terrance Swift e David S. Warren. Efficiently implementing SLG resolution:. 1994.

[Syr] Tommi Syrjänen. Lparse 1.0 user's manual.

[VGRS91] Allen Van Gelder, Kenneth A. Ross, e John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:619–649, July 1991.

[You] J. You. Answer set programming (asp): A new paradigm for knowledge representation and constraint programming. Available at http://webdocs.cs.ualberta.ca/ you/courses/621/LecNotes/asp.ppt.

RIF-PRD Implementation tests

A.1 RIF-PRD usermade test cases

In this section we will present some tests cases we created to evaluate our implementations. For better readability, the tests are presented in RIF-PRD presentation syntax. All tests are valid RIF-PRD documents.

A.1.1 Chaining of Exists conditional formula

```
Document (
Group rif:forwardChaining (

(* rule1 *)
   Do (
      Assert( object1[status -> gold] )
      Assert( object2[status -> silver] )
      Assert( object3[status -> gold] ))

(* rule 2 *)
   If Exists ?dVar1 ?dVar2
      ( And (?dVar1[status -> ?dVar2]
        Exists ?dVar3
           ( And (?dVar1[?dVar3 -> silver])))
   Then Do (Assert(finish("true"))))
```

A.1.2 All retracts

```
Group rif:forwardChaining (

(* rule1 *)
   Do (
      Assert( object1[status -> gold] )
      Assert( object2[status -> silver] )
      Assert( object2#object )
      Assert( object3[status -> platinum] )
    )

(* rule 2 *)
   If And(
          Exists ?dVar1  ?dVar2 (
             And ( ?dVar1[status -> gold]
           ?dVar2#object ))
         object3[status -> platinum] )

   Then Do (
       Retract ( object2 )
       Retract ( object1 status )
       Retract ( object3[status -> platinum] ))))
```

A.1.3 INeg conditional formula

```
Group rif:forwardChaining (

(* rule1 *)
   Do(
      Assert( object1[status -> gold] )
      Assert( object2[status -> silver] )
    )

(* rule 2 *)
   Forall ?dVar1 ?dVar2 (
        If ( And (
           ?dVar1[status -> ?dVar2]
      Not ( op(?dVar1 ?dVar2) )))

   Then Do ( Assert( finish(true) ))))
```

A.1.4 Or conditional formula

```
Group rif:forwardChaining (

(* rule1 *)
   Do (
      Assert( object1[status -> gold] )
      Assert( object2[status -> silver] )
    )

(* rule 2 *)
   Forall ?cust
      (If ( Or (
         ?cust#customer
         ?cust[isCustomer -> no]))
      Then Do ( Assert( finish(true) ))))
```

A.1.5 Action variable declaration

```
Group rif:forwardChaining (

(* rule1 *)
   Do(
      Assert( obj1[status -> gold] ))

(* rule 2 *)
    Do (
      (?aVar obj1[status -> ?aVar])
      (?nVar New())
      Assert( ?nVar[isNewVar -> yes] )
      Assert( ?obj2[status -> ?aVar])))
```