Analyzing the AIR Language: A Semantic Web (Production) Rule Language [⋆]

Ankesh Khandelwal[1], Jie Bao[1], Lalana Kagal[2], Ian Jacobi[2], Li Ding[1], and James Hendler[1]

[1] Rensselaer Polytechnic Institute
Troy, NY 12180
{ankesh, baojie, dingl, hendler}@cs.rpi.edu
[2] Massachusetts Institute of Technology
Cambridge, MA 02139
{lkagal, jacobi}@csail.mit.edu

Abstract. The **A**ccountability **I**n **R**DF (*AIR*) language is an N3-based, Semantic Web production rule language that supports nested activation of rules, negation, closed world reasoning, scoped contextualized reasoning, and explanation of inferred facts. Each AIR rule has unique identifier (typically an HTTP URI) that supports reuse of rule. In this paper we analyze the semantics of AIR language by: i) giving the declarative semantics that support the reasoning algorithm, ii) providing complexity of AIR inference; and iii) evaluating the expressiveness of language by encoding Logic Programs of different expressivities in AIR.

1 Introduction

AIR was developed as an extension to N3Logic [4] to support accountable privacy protection in Web-based information systems [11]. While N3Logic supports scoped contextualized reasoning (SCR), nested graphs, and built-in functions, AIR additionally supports rule reuse, rule nesting and explanation of rule-based inference. AIR rules are encoded in N3[1], which extends the expressiveness of RDF with (i) the ability to use graphs as literal values, (ii) universal or existential quantification of variables in a graph and (iii) built-in functions and operators represented as RDF properties.

AIR is designed to provide detailed explanations for actions performed by AIR reasoner. AIR permits natural language descriptions to be added to actions, so that when an action is performed the description is included in the justification, by the reasoner. Since justifications can potentially reveal sensitive information from the knowledge base or expose bias in the rules, AIR also provides ways to customize explanations, e.g. hiding actions of certain rules.

[⋆] This work was supported in part by NSF Cybertrust award CNS-0831442, IARPA award FA8750-07-2-0031, AFOSR award FA9550-09-1-0152, and support to RPI from the Information Technology Alliance in Network and Information Science by U.S. ARL and the U.K. MoD under Agreement Number W911NF-06-3-0001

[1] http://www.w3.org/DesignIssues/Notation3.html

AIR rules have unique IDs (IRIs), which enables the rules to be part of the linked data cloud and reused by other rule definitions. Furthermore, AIR rules can be nested. These features are natural to model real-world rules and laws where conditions are often split into sequentially-activated rules, and rules are frequently reused.

AIR supports rule definitions in which Web resources, such as triple stores with SPARQL [18] end-points, can be objectively checked for patterns. Sometimes the data has an associated intensional component, defined through rules. For complete access to the data, AIR supports inclusion of certain inferences without allowing the rules in the intensional component to be applied to the entire input data. SCR is important for a Web rule languages because information on the Web is often assumed to be incomplete, and its correctness is subject to the trustworthiness of the source.

There are many rule languages and rule systems; some are more advanced while others are at prototypical levels. Liang et. al. give a nice overview of popular, and often advanced, rule systems in [15]. Examples of rule languages include N3Logic, NG [20], SILK [8], and SWRL [10], while some rule engines include Jess[2], Jena[3], and XSB[4]. These languages and systems have different levels of expressiveness and features.

Networked Graphs (NGs) supports the well-founded (WF) negation, XSB supports Negation as Failure (NAF), and Jess, a production rule system, supports non-logical negation. SILK supports WF negation, as well as classical negation. SWRL and Jena do not support negation. In comparison, N3Logic supports monotonic negation, and AIR supports non-monotonic negation, which is different from negation accepted by the systems above.

Rule Interchange Format (RIF) [14] has two major dialects – the Framework for Logic Dialects (FLD) and the Production Rule Dialect (PRD). Unlike PRD, actions in AIR cannot modify or remove facts, but they can add new production rules. FLD is an extensible framework for rule-based languages, and includes the Basic Logic Dialect (BLD), which corresponds to the definite Horn rules with equality and standard first-order logic semantics. AIR is neither more nor less expressive than BLD. Unlike BLD, function symbols are not supported in AIR, whereas AIR supports negation. SWRL is a function-free rule language limited to binary and unary predicates, and all its features, barring different-from, are covered by BLD. AIR is as expressive as SWRL, and also supports negation.

OPS/YES [21] extends the OPS5 [7] production rule system with many features including *incremental rule addition*, which allows matching in the actions. AIR has a similar notion of rule nesting. Other than AIR, of the systems mentioned above, only Jena supports (partially) nesting of rules.

In SPARQL, query patterns can be restricted to selective named graphs, and as a result NGs naturally supports SCR. Other than N3Logic, AIR, and NGs, SILK also supports SCR.

[2] http://www.jessrules.com/
[3] http://jena.sourceforge.net/
[4] http://xsb.sourceforge.net/

The Ontonova system [1] provides natural language explanation of proof trees for conclusions by the Ontobroker[5] through meta-inferencing. A similar mechanism for generating proofs for defeasible reasoning system DR-DEVICE is described in [3]. In contrast, the explanations for conclusions in AIR are generated by the reasoner itself, and the justification can be declaratively modified by changing rule definitions.

AIR's focus on explanation generation for Web reasoning makes it unique. Other distinguishing features include AIR's ability to treat AIR rules as part of linked data and the ability to match patterns against remote triple stores.

While AIR has a production rule syntax, it is limited to assertions of facts, and addition of rules. Neither can facts be removed from the current state of the world nor any procedural attachments may be called. Therefore, we can define its declarative semantics. In this work we define the model-theoretic semantics of AIR-programs, by defining the translation of an arbitrary AIR-program to a semantically equivalent stratified Logic Program (LP).

There has been an earlier work on translating Jess to and from Situated Ordinary LP (SOLP)[6] and Situated Courteous LP (SCLP) [9], showing interoperability of the two rule languages. However, Jess and AIR are very different rule languages. Jess, unlike AIR, supports procedural attachments and with certain restrictions on its features has been shown to support (stratified) NAF [9]. Not only is the notion of nesting of rules absent from Jess, AIR's negation is different from NAF. The authors are not aware of any other related works.

The remainder of the paper is organized as follows. Section 2 provides an overview of the AIR language and reasoning. In section 3, we define a new class of stratified LP - Positively Stratified Negatively Hierarchical LP (PSNHLP). In section 4, we define the declarative semantics, and provide the data and program complexities of AIR programs. In section 5, we show how LP rules and fairly expressive LPs, such as PSNHLP, can be encoded in AIR. We conclude in section 6 with a summary and discussion on future work.

There is a longer version of the paper [13][7], that includes review of standard terminologies used in Logic Programming, proofs of all the claims, and detailed comparison of AIR with other rule systems.

2 AIR Overview

The AIR production rule system has two components- the rule language and the rule engine (reasoner). The reasoner computes the closure, **AIR-closure**, for given facts, in N3, with respect to given AIR-program. An **AIR-program** contains one or more AIR-rules. The closure contains the initial facts and all the facts that can be deduced from it using the given rules. We will review the AIR

[5] http://ontobroker.semanticweb.org/

[6] http://sweetrules.projects.semwebcentral.org/sweetrules-overview-presentation-2005-04-24-v3.pdf

[7] http://tw.rpi.edu/proj/tami/AIR_Language_Formalization

```
A. RULE SET
# $m_1 \geq 0$, $m_2 > 0$
@forAll $< u\_var_1 >, \ldots, < u\_var_{m_1} >$ .

<setid> a air:RuleSet ;
        air:rule $< ruleid_1 >, \ldots, < ruleid_{m_2} >$ .

B. RULE
# $n_1, n_2, n_3, n_4 \geq 0, n_3 + n_4 > 0$
<ruleid> a air:BeliefRule ; # alternatively air:HiddenRule, air:EllipsedRule
         air:if { @forSome $< e\_var_1 >, \ldots, < e\_var_{n_1} >$ .
                  $s_1 < p_1 > o_1$ .
                  ...
                  $s_{n_2} < p_{n_2} > o_{n_2}$ . } ;
         air:then $< t\_action_1 >, \ldots, < t\_action_{n_3} >$ ;
         air:else $< e\_action_1 >, \ldots, < e\_action_{n_4} >$ .

C. ACTIVATING NEW RULE
<action> air:description (list_of_var_and_str) ;
         air:rule $< ruleid >$ .

D. ASSERTING A GRAPH PATTERN
# $n > 0$
<action> air:description (list_of_var_and_str) ;
         air:assert {$s_1 < p_1 > o_1$ .
                     ...
                     $s_n < p_n > o_n$ . } .
```

Fig. 1. Template for defining AIR-program

language, and introduce the algorithm for computing the closure in sections 2.1 and 2.2. The details about AIR justification are available at [12].

2.1 AIR Language

AIR rules are defined using following properties: `air:if`, `air:then`, `air:else`, `air:description`, `air:rule` and `air:assert`, according to the AIR abstract syntax[8]. A template for declaring an AIR-program is shown in Figure 1. This template will be used in the later sections. When we refer to s and o related by `p1.p2` we mean that `s p1 [ p2 o ]` holds.

The rules are of the form `air:if` *condition*; `air:then` *then-actions*; `air:else` *else-actions*. The condition is specified as a graph pattern, similar to the Basic Graph Pattern of SPARQL queries, which may be pattern matched against N3 graphs. Whenever the condition matches the current state of the world, i.e. the facts known or inferred so far, variables may acquire bindings and all the `then-action`s are fired. Otherwise all the `else-action`s are fired.

The rules are grouped under rule-sets. The rules nested directly under the `RuleSet` are referred to as the **top rule**s of the rule-set. A rule (**child rule**) nested under another rule (**parent rule**) via `air:then.air:rule` is said to be positively nested and a rule nested via `air:else.air:rule` is said to be negatively nested. A **chain** of rules is defined as a sequence of rules such that every

[8] http://tw.rpi.edu/proj/tami/AIR_Abstract_Syntax

```
@prefix air:<http://dig.csail.mit.edu/TAMI/2007/amord/air#>.
@prefix conf:<http://www.conf.org/ontology#>.
@prefix pol:<http://www.conf.org/policies/publication#>.
@prefix:<http://www.conf.org/policies/publication#>.

pol:PubInProcPolicy a air:RuleSet ;
    air:rule pol:CheckPub .

@forAll :PUBL .
pol:CheckPub a air:BeliefRule ;
    air:if { @forSome :PROC . <http://www.conf.org> conf:hasProceedings :PROC .
             :PROC conf:hasPaper :PUBL . } ;
    air:then [ air:description (:PUBL " published in this conference") ;
               air:rule pol:ChkNonCompl ],
             [air:rule pol:CheckAuth], [air:rule pol:CheckExempt].

pol:ChkNonCompl a air:BeliefRule ;
    air:if { :PUBL air:compliant-with pol:PubInProcPolicy . } ;
    air:else [ air:description ("the publication of " :PUBL " is questionable as
                                   it did not meet any of the two criteria") ;
               air:assert { :PUBL air:non-compliant-with pol:PubInProcPolicy } ] .

@forAll :AUTH .
pol:CheckAuth a air:BeliefRule ;
    air:if {:PUBL conf:hasAuthor :AUTH.<http://www.conf.org> conf:registeredBy :AUTH.};
    air:then [ air:description ("Author," :AUTH " registered for the conference");
               air:assert { :PUBL air:compliant-with pol:PubInProcPolicy } ] .

@forAll :REASON .
pol:CheckExempt a air:BeliefRule ;
    air:if { @forSome :COCHAIR, :EXEMPTION .
             :COCHAIR a conf:Co-Chair . :PUBL conf:hasFirstAuthor :AUTH .
             :EXEMPTION a conf:PublicationExemption ; conf:exemptedBy :COCHAIR ;
                   conf:exemptee :AUTH ; conf:reason :REASON . } ;
    air:then [ air:description ("the first author " :AUTH " was exempted by one of the
                                                    cochairs, because " :REASON);
               air:assert { :PUBL air:compliant-with pol:PubInProcPolicy } ] .
```

Fig. 2. Example AIR-program, describing the publication policy

rule barring the first in the chain is nested under the preceding rule. `t_action`$_i$s and `e_action`$_i$s, in Figure 1, are the `then-action`s and `else-action`s, respectively. Together they are referred to as `action`s. The actions can be annotated with natural language description through the `air:description` property. Note that the description can be defined using variables.

The existentially quantified variables may be declared within the condition. The universally quantified variables are declared outside of the rule. The scope of existentially quantified variable is the condition where it's declared, and of universally quantified variable is any rule chain with the first rule as a top rule.

The graph pattern asserted in the action is declared using `air:assert`, and the nested rule is declared using `air:rule` property. The asserted graph patterns cannot contain blank nodes or existentially quantified variables. When the nested rule is activated, an instance of the rule with known variable bindings substituted is created. When a rule with an `air:else` property is activated, its condition cannot contain unbounded universally quantified variables.

Traditionally, rule languages have been designed assuming that all rules will apply on all input data. However, this is not desirable

A. Data Structures

```
rule = (ruleid, condition, success, then-actions, else-actions)
then(else)-actions = list of child rules to be activated and graphs to be asserted
RB = set of active rules
FB = set of facts
SRF = queue of (rule, bindings) tuples
FRF = queue of rules
```

B. COMP-CLOSURE

```
INPUT : AIR-program, facts
OUTPUT : Closure of facts w.r.t. the AIR-program in FB
1. Initialize the data-structures
1.1. FB = facts
1.2. RB = $\phi$
1.3. SRF = empty queue
1.4. FRF = empty queue
1.5. FRF' = empty queue
2. For each top rule, of all RuleSets in AIR-program
2.1. add-to-rulebase(R)
3. while SRF and FRF is non empty
3.1. while SRF is non empty
3.1.1. f = SRF.dequeue(), i.e. remove first queue element and assign it to f
3.1.2. rule-fire(f.rule.then-actions, f.bindings)
3.2. while FRF is non empty
3.2.1. f = FRF.dequeue()
3.2.2. if f.rule.success is false
3.2.2.1. FRF'.enqueue(f)
3.3. while FRF' is non empty
3.3.1. f = FRF'.dequeue()
3.3.2. rule-fire(f.rule.else-action, {})
```

C. RULE-FIRE

```
INPUT : actions, bindings bndgs
OUTPUT : updated RB and FB
1. for action in actions :
1.1. substitute all variables in action which have bindings with bound values from bndgs
1.2. if action is a rule
1.2.1. add-to-rulebase(action)
1.3. if action is a graph
1.3.1. add-to-factbase(action)
```

D. ADD-TO-RULEBASE

```
INPUT : rule
OUTPUT : updated RB and (SRF or FRF)
1. if rule not in RB
1.1. add rule to RB
1.2. R.success = False
1.3. pattern match rule.condition against FB
1.4. If rule.condition matched a sub-graph
1.4.1. R.success = True
1.4.2. for each sub-graph that matched rule.condition
1.4.2.1. SRF.enqueue((rule, bindings))
1.5. If rule.condition did not match any sub-graph
1.5.1. FRF.enqueue(rule)
```

E. ADD-TO-FACTBASE

```
INPUT : graph (grounded graph pattern)
OUTPUT : updated FB, RB and SRF
1. let graph' = graph \ FB
2. if $graph' \neq \phi$
2.1. add triples in graph' to FB
2.2. for each rule in RB
2.2.1. pattern match rule.condition against FB
2.2.2. If rule.condition matched a sub-graph $g$ s.t. that $g \cap$ graph' $\neq \phi$
2.2.2.1. rule.success = True
2.2.2.2. for each sub-graph $g$ that matches the rule.condition and $g \cap$ graph' $\neq \phi$
2.2.2.2.1. SRF.enqueue((rule, bindings))
```

Fig. 3. The COMP-CLOSURE algorithm, and related data-structures and algorithms.

for a web rule language [4, 17], and therefore AIR supports SCR. The matching of a condition pattern can be scoped to web documents, RDF-stores with SPARQL end-points, or to the AIR-closure of some facts and AIR-programs using the `log:semantics.log:includes`, `sparql:queryEndPoints.log:semantics.log:includes` and `air:justifies` built-ins respectively. Note that by using `air:justifies`, a subset of rules can be applied to a subset of data, if desired, without risking the application of those rules to the entire input data. In addition to the built-ins for scoped reasoning, AIR supports most of the N3Logic built-ins[9] for cryptographic, math, string, list and time functions.

Figure 2 shows an example AIR-program. It expresses the conference policy: any paper accepted in the conference should be published in the proceedings

[9] http://www.w3.org/2000/10/swap/doc/CwmBuiltins

```
@prefix colog:<http://www.conf.org/log#>. @prefix conf:<http://www.conf.org/ontology#>.
<http://www.conf.org>   conf:hasProceedings   colog:proc .
colog:proc              conf:hasPaper         colog:pub1 .
colog:pub1              conf:hasAuthor        colog:auth1 .
<http://www.conf.org>   conf:registeredBy     colog:auth1 .
```

Fig. 4. Example of input facts: sub-graph from log of conference activities.

only if one of the authors has presented it in the conference (`pol:CheckAuth`), or the first author has received an exemption from one of the co-chairs (`pol:CheckExempt`). The former condition is interpreted in practice by checking to see if one of the co-authors has registered for the conference. It is assumed that the conference activities are logged using the same vocabulary as that used in the rules. `pol:CheckPub` is the only top rule, and the three rules, `pol:ChkNonCompl`, `pol:CheckAuth`, and `pol:CheckExempt` are its child rules.

2.2 AIR Reasoning (The Procedural Semantics)

The AIR reasoner employs a RETE [6] based forward-chaining approach to compute the AIR-closure. Figure 3 describes the **COMP-CLOSURE** algorithm used for computing the AIR-closure. The data structures and algorithms described here are abstractions of the actual implementation [12]. For instance **FB** (Fact Base), which is shown to be a set of facts, is actually an efficiently indexed triple store, and **RB** (Rules Base), the set of active rules, is compiled into the RETE framework. The abstraction is sufficient for analyzing the language.

One cycle of step 3 of **COMP-CLOSURE** is referred to as a **stage**, i.e. steps 3.1 to 3.3 (3.3.1 and 3.3.2 included) constitute one stage of the algorithm. Furthermore, step 3.1 is referred to as a positive stage, denoted by **stage**$^+$ and steps 3.2 and 3.3 constitute a negative stage, denoted by **stage**$^-$. We will be using stage$_i$, stage$_i^+$ and stage$_i^-$ to refer to ith stage, stage$^+$ and stage$^-$, respectively.

In any given stage, successful rules are given priority over failed rules, and their `then-action`s are effected in stage$^+$ before failed rules fire. When all successful rules have fired, the world is temporarily closed and the `else-action`s of all the failed rules are fired in stage$^-$ with the belief that the conditions of all the failed rules are false. AIR reasoning enters the next stage once the failed rules have all fired. The **COMP-CLOSURE** algorithm computes the next stage until the fixpoint is reached, i.e. when both the **SRF** and **FRF** are empty.

Consider the input facts in Figure 4 and the example AIR-program from Figure 2. The AIR-closure contains one additional triple `colog:pub1 air:compliant-with pol:PubInProcPolicy`. In order to compute the closure, **FB** is initialized by triples in the input and `pol:CheckPub` is added to the **RB**. When the rule is added, its condition matches and it is added to the **SRF** with `pol:PUBL` bound to `colog:pub1`. When it fires, the three child rules are added to the **RB**. Only `pol:CheckAuth` succeeds and other rules are added to the **FRF**. The `pol:CheckAuth` is added to the **SRF** with `:AUTH` bound to `colog:auth1`. It fires next, and asserts the triple `colog:publ1` `air:compliant-with pol:PubInProcPolicy`. When that triple is added, it satisfies the `pol:ChkNonCompl`'s condition and the rule's success attribute is set to true. **SRF** is now empty and `pol:ChkNonCompl` is removed from **FRF**. `pol:CheckExempt` is then added to **FRF**. However, it doesn't have an `air:else` property. The fixpoint is reached at the end of first stage.

3 Positively Stratified Negatively Hierarchical Logic Programs

We will review definitions of *Ground*(R), the T_P *operator* and the *perfect model* semantics for stratified LPs, before defining PSNHLPs. *Ground*(R) refers to the set of all possible ground instances of R, with respect to given universe of constant symbols U, where R is a rule or an LP. T_P is the one-step induction operator (or immediate consequence operator) [2] associated with an LP P, defined by:

$$T_P(I) = \{A_0 \mid P \text{ contains a rule whose instantiation is } A_0 \leftarrow A_1, \ldots, A_n \text{ such that } I \models A_1 \& \ldots \& A_n\}$$

where I is the given set of atoms that are true. The powers of T_P are defined as : $T_P^{\uparrow 0}(I) = I$, $T_P^{\uparrow n+1}(I) = T_P(T_P^{\uparrow n}(I)) \cup T_P^{\uparrow n}(I)$, $T_P^{\uparrow \omega}(I) = \bigcup_{n=0}^{\infty} T_P^{\uparrow n}(I)$

The semantics of a (locally) stratified LP, P, is defined in terms of its *perfect model*, which we refer to by *PM*(P). P entails a ground atom A iff A belongs to *PM*(P). Let $P_1 \cup \ldots \cup P_n$ be the (local) stratification of P (*Ground*(P)). We can choose a stratification and assignment of levels to predicates such that the rule $A \leftarrow L_1, \ldots, L_m$ in P is in P$_i$ iff the level assigned to A is i, and we will assume that to be the case in rest of the paper. *PM*(P) $= M_n$ [19], which is defined by : $M_1 = T_{P_1}^{\uparrow \omega}(\phi)$, $M_2 = T_{P_2}^{\uparrow \omega}(M_1)$, $\ldots$ $M_n = T_{P_n}^{\uparrow \omega}(M_{n-1})$

When a rule contains a negative literal in the body, we will refer it as **negative rule**. All other rules are thus **positive rules**. An LP P is **PSNHLP** if there is an assignment of ordinal levels to predicates such that whenever a predicate appears in the body (negatively or positively) of a negative rule, the predicate in the head of that rule is of strictly higher level, and whenever a predicate appears in the body of a positive rule, the predicate in the head has at least that level. Similarly, an LP P is locally PSNHLP if such an assignment of ordinal levels is possible over ground atoms for *Ground*(P). Every hierarchical LP is PSNHLP, and every PSNHLP is stratified LP. This doesn't hold in opposite direction.

Let $P_1 \cup \ldots \cup P_n$ be the stratification of a PSNHLP P, and let $P_i^+ \cup P_i^-$ is the partition of P$_i$ such that a rule from P$_i$ is in P$_i^+$ if it is positive and in P_i^- otherwise. Then, the **PSNH-stratification** of P is defined as $P = P_1^+ \cup P_1^- \cup \ldots \cup P_n^+ \cup P_n^-$. The predicates in the body of a rule in P_i^- have levels strictly smaller than i. The *PM*(P) $=$ M$_n$, where M$_n$ is defined by:

$M_1 = T_{P_1}^{\uparrow \omega}(\phi)$,
$M_2' = T_{P_2^-}^{\uparrow 1}(M_1)$, $M_2 = T_{P_2^+}^{\uparrow \omega}(M_2')$,

$\ldots$
$M'_n = T^{\uparrow 1}_{P_n^-}(M_{n-1}),\ M_n = T^{\uparrow\omega}_{P_n^+}(M'_n)$

such that $M_1 = T^{\uparrow\omega}_{P_1}(\phi)$, and $M_i = T^{\uparrow\omega}_{P_i}(M_{i-1})$ for $i > 1$. The former holds because $P_1 = P_1^+$ (i.e. $P_1^- = \phi$). The latter follows from following claim.

Claim 1: $T^{\uparrow\omega}_{P_i}(M_{i-1}) = T^{\uparrow\omega}_{P_i^+}(T^{\uparrow 1}_{P_i^-}(M_{i-1}))$

4 Declarative Semantics

The declarative semantics of AIR-program can be defined in terms of a semantically equivalent PSNHLP. We will first define τ, that translates AIR-program Δ to LP $\wp$. We will then show that $\wp$ is PSNHLP and that the AIR-closure of Δ and input facts κ, mapped as ground facts in $\wp$, is same as $PM(\wp)$. $\wp$ contains rules for predicates `t`, $\mathtt{t}_s$, `active_rule` and `cond`, with following meanings :

– `t(s, p, o)` represents N3 triple {`s p o`}.
– $\mathtt{t}_s$`(s, p, o, c, n)` represents true N3 triple {`s p o`}, in the `n`th stage of reasoning, in context `c`.
– `active_rule(ruleid, ?v`$_1$`, ..., ?v`$_m$`, c, n)` represents active AIR-rule, `ruleid`, in the `n`th stage, in context `c`. The exact instance of the rule is determined by the partial bindings of `?v`$_i$'s. `?v`$_i$'s are the universally quantified variables used in the chain of rules to which `ruleid` belongs. For example, the rule `pol:CheckAuth` has two variables `?PUBL` and `?AUTH`.
If, for example, `active_rule(pol:CheckAuth, colog:pub1, ?AUTH, c, 1)` atom is true, then an instance of the rule `pol:CheckAuth` is active at the 1st stage with `?PUBL` bound to `colog:pub1` in the context `c`. Here `?AUTH` is unbound and can be bound by any constant from U.
– `cond(ruleid, ?cv`$_1$`, ..., ?cv`$_m$`, c, n)` represents a satisfied condition of the AIR-rule, `ruleid`, in the `n`th stage, in context `c`. The `?cv`$_i$'s are universally quantified variables occurring in the rule's condition. The VAR_IN_COND function returns all these variables for given rule. If, for example, `cond(pol:CheckAuth, colog:pub1, colog:auth1, c, 1)` atom is true then the condition of `pol:CheckAuth` rule is true in 1st stage, in context c, when `?PUBL` and `?AUTH` are bound to `colog:pub1` and `colog:auth1` respectively.
– `succ` (successor) and `=` predicates are used with their usual meanings.

The **context** may be defined by a collection of N3 and/or AIR-program sources, and is identified with an ID. We may use unique predicate symbols to distinguish `active_rule` predicates for same rule in different rule chains.

$\tau(\Delta)$ is defined by conjunction of the rules returned from `TRANS(setid)` and `TRANS(ruleid)`, for every rule set and production rule in Δ, respectively. Figure 5 gives the definition of **`TRANS`**`(ele, ?n)` function, for `ele`'s in a built-in-free AIR-program. The `ele`s are described as per the template in Figure 1, and `?n` is optional argument.

```
A. TRANS(setid), where setid is a RULE-SET
1. ruleset(setid, c) .
2. for i in 1 to $m_2$
      active_rule(ruleid$_i$, ?u_var$_1$, ..., ?u_var$_{m_1}$, ?c, 1) $\leftarrow$ ruleset(setid, ?c) .
B. TRANS(ruleid), where ruleid is a RULE
3. cond(ruleid, VAR_IN_COND(ruleid), ?c, ?n) $\leftarrow$ t$_s$(MAP(s$_1$), MAP(p$_1$), MAP(o$_1$),
                 ?c, ?n), ..., t$_s$(MAP(s$_{n_2}$), MAP(p$_{n_2}$), MAP(o$_{n_2}$), ?c, ?n), ?n $\leq$ UB .
4. for i in 1 to $n_3$
      TRANS(t_action$_i$, ?n) $\leftarrow$ active_rule(ruleid, ?u_var$_1$, ..., ?u_var$_{m_1}$, ?c, ?n),
                           cond(ruleid, VAR_IN_COND(ruleid), ?c, ?n), ?n $\leq$ UB .
5. for i in 1 to $n_4$
      TRANS(e_action$_i$, ?m) $\leftarrow$ active_rule(ruleid, ?u_var$_1$, ..., ?u_var$_{m_1}$, ?c, ?n),
            not(cond(ruleid, VAR_IN_COND(ruleid), ?c, ?n)), succ(?m, ?n), ?n$\leq$UB.
C. TRANS(action, ?n), where action is ACTIVATING NEW RULE
6. active_rule(ruleid, ?u_var$_1$, ..., ?u_var$_{m_1}$, ?c, ?n)

D. TRANS(action, ?n), where action is ASSERTING A GRAPH PATTERN
7. t$_s$(MAP(s$_1$),MAP(p$_1$),MAP(o$_1$), ?c, ?n), ..., t$_s$(MAP(s$_n$),MAP(p$_n$),MAP(o$_n$),?c,?n)

E. TRANS({s p o}), where {s p o} is INPUT FACT
8. t$_s$(s, p, o, c, 1) .
```

Fig. 5. TRANS(ele, ?n) definition, where `ele` is defined according to the template in Figure 1

The **MAP**(`val`) function maps `val` to `?val` or `val` depending on whether `val` is declared as a variable or not. `UB` is the upper bound on number of stages required for computing the fix-point for given AIR-program and initial facts. `UB` is finite because assertions can use only fixed IRIs (i.e. no blank nodes are allowed in the asserted graph), and every fact and rule instance is asserted at most once. `UB` is polynomial in the size of input facts and the maximum depth of nesting in input AIR-program [13]. `?n` $\leq$ `UB` ensures that the closure for $\wp$ is finite. Additional rules that hold in every $\wp$ are shown in Figure 6.

The triples in κ are translated as facts into $\wp$, in translation step (TS) 8. The triples asserted and the rules activated when the rule condition is satisfied are true from the same stage (TS-4), whereas they are true only from next stage when the rule condition failed (TS-5). The definition in Figure 5 can be extended to handle the built-ins for SCR, namely `air:justifies`, `log:includes`, and `log:notIncludes` [13].

For example, TRANS(`pol:ChkNonCompl`) yields following rules :

- t$_s$(?PUBL , `air:non-compliant-with`, `pol:PubInProcPolicy`, ?c, ?m) $\leftarrow$
  active_rule(`pol:ChkNonCompl`, ?PUBL, ?c, ?n), succ(?n, ?m), ?n $\leq$ UB,
  not(cond(`pol:ChkNonCompl`, ?PUBL, ?c, ?n)) .
- cond(`pol:ChkNonCompl`, ?PUBL, ?c, ?n) $\leftarrow$
  t$_s$(?PUBL,`air:compliant-with`, `pol:PubInProcPolicy`, ?c, ?n) .

9. active_rule(?x$_1$,...,?x$_m$, ?c, ?n)←active_rule(?x$_1$,...,?x$_m$, ?c,?m), ?m≤?n, ?n≤UB.
10. t$_s$(?s, ?p, ?o, ?c, ?n) ← t$_s$(?s, ?p, ?o, ?c, ?m), ?m ≤ ?n, ?n ≤ UB .
11. t(?s, ?p, ?o) ← t$_s$(?s, ?p, ?o, c, ?n), ?n ≤ UB .

Fig. 6. Additional rules that hold in every $\wp$.

Claim 2: $\wp$ is locally PSNHLP.
Proof. Let the assignment of levels to ground atoms for all predicates except `t`, `succ`, and = be equal to the value of their last term. For example `t`$_s$`(s, p, o, c, n)` is assigned level `n`. Let predicates `t`, `succ`, and `=` be assigned levels `UB`, 1, and 1, respectively. This assignment gives a local PSNH-stratification of $\wp$. □

We can replace the atom `t`$_s$`(s, p, o, c, n)` by `t`$_{s_n}$`(s, p, o, c)`. We can do so for other atoms with `t`$_s$, `active_rule` and `cond` predicate symbols, by introducing `t`$_{s_n}$, `active_rule`$_n$ and `cond`$_n$ predicate symbols, for all $n \leq$ `UB`. Although the TS-9, TS-10 and TS-11 rules will expand O(`UB`^{2}) times and the rest O(`UB`) times, changes to TSs in Figures 5 and 6 are straightforward. Then the resulting $\wp$, say $\wp'$, is a PSNHLP as against being just locally PSNHLP.

Let, $Ground(\wp) = \wp_1^+ \cup \wp_1^- \cup \ldots \cup \wp_{\mathrm{UB}}^+ \cup \wp_{\mathrm{UB}}^-$, be the local PSNH-stratification of $\wp$. We can show semantic equivalence, denoted by $\sim$, between the steps for computation of $PM(\wp)$ and the AIR-closure of Δ and κ. $T \sim s$ is used to denote the direct correspondence between extensions of predicates `active_rule` and `t`$_s$, inferred after the application of T operator during the computation of $PM(\wp)$, and the rules activated and triples asserted after completion of stage s of **COMP-CLOSURE** execution.

Claim 3: $T_{\wp_1^+}^{\uparrow\omega} \sim stage_1^+$. For $i > 1$, $T_{\wp_i^-}^{\uparrow 1} \sim stage_{i-1}^-$ *and* $T_{\wp_i^+}^{\uparrow\omega} \sim stage_i^+$.

Let $\aleph$ be the N3 graph obtained by taking the N3 representation of extensions of the predicate `t` in $PM(\wp)$.

Claim 4: The AIR-closure of Δ and κ is same as $\aleph$.
Proof. From claim 3 it follows that for all i $\geq$ 1, $M_i \cap \aleph$ and the triples in **FB** after the completion of $stage_i^+$ are the same. Since the fix-point is reached by the end of `UB`th stage, $M_{UB} \cap \aleph$ and the triples in the AIR-closure of Δ and κ are the same. However, $PM(\wp) = M_{UB}$. □

The above results (claims 3 and 4) hold for $\wp'$ as well [13]. We have seen that $\wp$ can be viewed as a stratified LP (PSNHLP), and we have shown above the direct correspondence between the steps for computing $PM(\wp)$ (equivalently $PM(\wp')$) and those for computing AIR-closure of Δ and κ. Therefore, complexity results for AIR follow from those for stratified programs [5]. The **data complexity** is the complexity of computing the model when the rules-base is fixed and the fact-base is an input. The **program complexity** is the complexity of computing the model when the fact-base is fixed but the rules-base is an input.

Claim 5: AIR-closure computation is PTime-complete in data-complexity and ExpTime-complete in program-complexity.

```
@forAll :X, :Y .
:r a air:BeliefRule ;
     air:if { :X :conn :Y .} ;
     air:then [ air:rule :b1 ], [air:rule :b2 ] ;
     air:then [ air:rule :rAux ] .

:rAux a air:BeliefRule ;
     air:if { :X :aux :Y } ;
     air:else [ air:assert { :X :good-conn :Y . } ] .
```

```
:b1 a air:BeliefRule ;
     air:if { :X a :Infected . } ;
     air:then [ air:assert { :X :aux :Y } ] .

:b2 a air:BeliefRule ;
     air:if { :Y a :Infected . } ;
     air:then [ air:assert { :X :aux :Y } ] .
```

Fig. 7. Rewriting an example LP rule, in AIR : Connection between X and Y is good if both X and Y are not infected.

5 AIR and Logic Programming

Logic programming is a very popular declarative method of knowledge representation and programming. Like LPs, we define rules in AIR. However, AIR rules have different semantics from the LP rules, and they can be nested under one another. Different nesting of the same rules may yield (semantically) different AIR programs. In this section we investigate logic programming through AIR.

An LP rule is said to be **safe** if every variable occurring in the head of the rule or in a negative literal in the body also occurs in a positive literal in the body. We can encode any safe LP rule $A \leftarrow A_1, \ldots, A_n, \ \mathit{not}(B_1), \ \ldots, \ \mathit{not}(B_m)$ in AIR by rewriting it as:

$$
\begin{aligned}
&A \leftarrow A_1, \ldots, A_n, \ \mathit{not}(\texttt{Aux}) \ \ .\\
&\mathit{Aux} \leftarrow B_1 \ \ .\\
&\ldots\\
&\mathit{Aux} \leftarrow B_m \ \ .
\end{aligned}
$$

Aux is a new predicate and its arguments are all the variable terms that appear in the literals of the rule's body. For example following rule : `good-conn(?X, ?Y)` $\leftarrow$ `conn(?X, ?Y),` *not*`(infected(?X)),` *not*`(infected(?Y)).`, can be expressed as AIR rule as shown in Figure 7. Note that the predicate symbol `S` is translated to `:S`, and variable symbol `?S` is translated to `:S`, as quantified by an `@forAll` directive. We have translated binary predicates to property assertions, and the unary predicate to type assertion. Higher arity N-ary predicate $p(t_1, \ldots, t_n)$ can be encoded in N3 as $\{$`(` $t_1 \ldots t_n$`) p true`$\}$.

The nesting of rules impacts the order in which the rules are fired, and they may be nested properly to get the desired semantics. Since for Positive LPs (PLPs) the order in which the rules fire does not matter, the rules in PLP can be translated into AIR and included as top rules in an AIR program to get a semantically equivalent AIR program. When a logic language (e.g., LP) can be encoded in AIR with unchanged semantics, we will say that it can be **losslessly rewritten** into AIR.

We have seen that PLPs can be losslessly rewritten into AIR. OWL 2 RL[10] inference rules are all positive, and therefore they can be encoded in AIR and

[10] http://www.w3.org/TR/owl2-profiles/#OWL_2_RL

```
:rs a air:RuleSet ;
        air:rule : $r^+_{1_1}$, . . . , : $r^+_{1_{n_1}}$ ;
        air:rule : $r^-_{2_1}$, . . . , : $r^-_{2_{n_2^-}}$ ;
        GET_RULE_PROPERTY(2) .

, where GET_RULE_PROPERTY(j) function is defined recursively as:
when $j < n$:
        air:rule [
                air:if { owl:Thing rdfs:subClassOf owl:Nothing . } ;
                air:else [ air:rule : $r^+_{j_1}$ ], . . ., [ air:rule : $r^+_{j_{n_j^+}}$ ] ;
                air:else [ air:rule : $r^-_{j+1_1}$ ], . . ., [ air:rule : $r^-_{j+1_{n_{j+1}^-}}$ ] ;
                air:else [ GET_RULE_PROPERTY($j+1$) ] ]
when $j = n$ :
        air:rule [
                air:if { owl:Thing rdfs:subClassOf owl:Nothing . } ;
                air:else [ air:rule : $r^+_{j_1}$ ], . . ., [ air:rule : $r^+_{j_{n_j^+}}$ ] ]
```

Fig. 8. Rewriting PSNHLP in AIR.

used concurrently with other AIR rules. Next we show that PSNHLPs can be losslessly rewritten into AIR.

Claim 6: PSNHLPs can be losslessly rewritten into AIR.
Proof. Let $P = P_1^+ \cup P_1^- \cup \ldots \cup P_n^+ \cup P_n^-$ be the PSNH-stratification of the PSNHLP P, and each P_i^- has n_i^- rules and P_i^+ has n_i^+ rules. Let the k^{th} rule in stratums P_i^+ and P_i^- be encoded in AIR with IDs $:r^+_{i_k}$ and $:r^-_{i_k}$ respectively. Then the nesting of rules shown in Figure 8 yields a lossless translation of P. This nesting can be generated programmatically for any PSNHLP P. `owl:Thing rdfs:subClassOf owl:Nothing` is a tautologically false statement.

The rules in P_{i+1}^- and P_i^+ are activated after $stage_{i-1}$. Since all the ground facts for predicates of level less than $\overline{i+1}$ in $PM(P)$ are inferred before $stage_i^-$, rules in P_{i+1}^- do not fire incorrectly in $stage_i^-$. □

Since non-recursive datalogs are PSNHLPs, they can be losslessly rewritten into AIR. The same can be said about SPARQL queries.

Claim 7: SPARQL SELECT *and* CONSTRUCT queries, without the query modifiers like ORDER BY and LIMIT, can be losslessly rewritten into AIR, and executed by the AIR reasoner.
Proof sketch. SPARQL SELECT *and* CONSTRUCT queries, without the query modifiers, can be translated to a non-recursive Datalog, Π_Q, with NAF under the answer set semantics [16]. Using claim 6 we can say that Π_Q can be losslessly rewritten into AIR. □

We have seen that PSNHLPs can be losslessly rewritten into AIR. However, more general stratified LPs cannot be translated into AIR. For example, the following rules cannot be losslessly rewritten into AIR for arbitrary facts of predicates P, Q, and S:

P(x, z) ← P(x, y), P(y, z), *not*(Q(x, z)) .
R(x, y) ← S(x, y), not(P(x, y)) .

However, that can be resolved through a simple extension of AIR, whereby **RuleSet**s can be nested – `:rs`$_1$ `air:hasHigherPriority :rs`$_2$ – with the following meaning: the rules nested under **RuleSet** `:rs`$_2$ are activated only after the fixpoint is reached for the rules under `:rs`$_1$. Note that the rules under `:rs`$_1$ remain active. The corresponding change in reasoning algorithm is straightforward. This extension has some similarity with salience in Jess.

Claim 8: Stratified LPs can be losslessly rewritten into AIR extended with the nesting of **RuleSet**s.

6 Conclusions and Future Work

In this paper we have analyzed AIR, a rule language for the Semantic Web. AIR supports non-monotonic negation, and we have provided a declarative semantics that supports this negation by translating AIR programs to a specialized class of (locally) stratified LPs – PSNHLPs. While AIR does not support well-founded negation and is less expressive than other rule systems, its ability to construct explanations, declaratively manipulate them, and its support for scoped contextualized reasoning (SCR) make it sufficiently unique and useful for Web reasoning. As AIR is a language for the Web, SCR is especially relevant. AIR also allows for the nesting of rules; this allows for the segmentation of the conditions of a rule so that only part of them are revealed in the justifications. We have shown that nesting can also be leveraged to order rules and therefore encode fairly expressive LPs such as PSNHLP. Stratified LPs can be encoded in AIR with an incremental modification to AIR. Finally, we have shown that SPARQL queries may be executed by the AIR reasoner.

We plan to investigate ways to increase the expressiveness of AIR in the future. This is a twofold task. We have shown that PSNHLP can be encoded in AIR, but do not think that this is the most expressive LP that can be expressed in AIR. It may also be possible to investigate an alternative AIR reasoning algorithm in which actions that fire under the assumption of a failed condition are retracted when the rule's conditions match later on. Furthermore, with a well-defined LP translation of AIR program, it is possible to develop goal-based query answering mechanisms for these programs. We are also interested in investigating contextualization more closely.

References

1. J. Angele, E. Moench, S. Staab, and D. Wenke. Ontology-based query and answering in chemistry: Ontonova @ project halo. In *Proceedings of the Second International Semantic Web Conference (ISWC 2003)*, 2003.

2. K. R. Apt. Logic programming. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics.* 1990.
3. N. Bassiliades, G. Antoniou, and G. Governatori. Proof explanation in the dr-device system. In *Proceedings of the 1st international conference on Web reasoning and rule systems (RR 2007)*, 2007.
4. T. Berners-lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler. N3logic: A logical framework for the world wide web. *Theory Pract. Log. Program.*, 8(3), 2008.
5. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *IEEE Conference on Computational Complexity*, 1997.
6. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1), September 1982.
7. C. L. Forgy. Ops5 users manual. In *Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University*, 1981.
8. B. N. Grosof. Silk: Higher level rules with defaults and semantic scalability. In *Proceedings of the 3rd International Conference on Web Reasoning and Rule Systems (RR2009)*, 2009.
9. B. N. Grosof, M. D. Gandhe, and T. W. Finin. Sweetjess: Inferencing in situated courteous ruleml via translation to and from jess rules. In *Proceedings of the ISWC 02 International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2003.
10. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml. Technical report, W3C, 2004.
11. L. Kagal, C. Hanson, and D. Weitzner. Using dependency tracking to provide explanations for policy management. *IEEE International Workshop on Policies for Distributed Systems and Networks*, 2008.
12. L. Kagal, I. Jacobi, and A. Khandelwal. Gasping for air - why we need linked rules and justications on the semantic web. In *Under review at the International Semantic Web Conference (ISWC 2010)*, 2010.
13. A. Khandelwal, J. Bao, L. Kagal, I. Jacobi, L. Ding, and J. Hendler. Analyzing the air language: A semantic web rule language. In *Technical Report, Department of Computer Science, Rensselaer Polytechnic Institute*, 2010.
14. M. Kifer. Rule interchange format: The framework. In *Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems (RR 2008)*, 2008.
15. S. Liang, P. Fodor, H. Wan, and M. Kifer. Openrulebench: an analysis of the performance of rule engines. In *Proceedings of the 18th international conference on World wide web (WWW 2009)*, 2009.
16. A. Polleres. From sparql to rules (and back). In *Proceedings of the 16th international conference on World Wide Web (WWW 2007)*, 2007.
17. A. Polleres, C. Feier, and A. Harth. Rules with contextually scoped negation. In *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006*, 2006.
18. E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf. Technical report, W3C, 2006.
19. T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming.* 1988.
20. S. Schenk and S. Staab. Networked graphs: a declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In *Proceeding of the 17th international conference on World Wide Web (WWW 2008)*, 2008.
21. M. I. Schor, T. Daly, H. S. Lee, and B. Tibbitts. Advances in rete pattern matching. In *AAAI*, 1986.