Gasping for AIR – Why we need linked rules and justifications on the Semantic Web*

Lalana Kagal[1], Ian Jacobi[1], and Ankesh Khandelwal[2]

[1] MIT CSAIL
Cambridge, MA 02139
{lkagal, jacobi}@csail.mit.edu
[2] Rensselaer Polytechnic Institute
Troy, NY 12180
ankesh@cs.rpi.edu

Abstract. The Semantic Web is a distributed model for publishing, utilizing and extending structured information using Web protocols. One of the main goals of this technology is to automate the retrieval and integration of data and to enable the inference of interesting results. This automation requires logics and rule languages that make inferences, choose courses of action, and answer questions. The openness of the Web, however, leads to several issues including the handling of inconsistencies, integration of diverse information, and the determination of the trustworthiness of data. AIR is a Semantic Web-based rule language that provides this functionality while focusing on generating and tracking explanations for its inferences and actions and conforming to Linked Data principles. AIR supports ***Linked Rules***, which can be combined and re-used in a manner similar to Linked Data. Additionally, AIR explanations themselves are Semantic Web data so they can be used for further reasoning. In this paper we present an overview of AIR, discuss its potential as a Web rule language by providing examples of how its features can be leveraged for different inference requirements, describe how justifications are represented and generated, and present an overview of AIR semantics.

1 Introduction

Though RDF Schema (RDFS) and the Web Ontology Language (OWL 1 & 2) provide some reasoning capability over Resource Description Framework (RDF) data, the application of Semantic Web technologies to e-government, business, policy management, workflow systems, and many other fields requires more expressive rule languages to capture the underlying system logic. Much of the Semantic Web's growth today has been in the form of Linked Data, which means that any Web rule language must be able to handle highly interconnected and spatially dispersed data. It must be able to dynamically traverse this web of data to find additional facts to support the conclusions of its reasoner. Furthermore, a Web rule language should expose its rules as Linked Data as well so that they can be re-used and combined in a similar manner.

* This work was supported in part by NSF Cybertrust award number CNS-0831442, IARPA award number FA8750-07-2-0031 and AFOSR award number FA9550-09-1-0152

Web rule languages must be able to cope with the problems that arise from the inherent openness of the Web. Reasoning over data on the Web can easily lead to logical inconsistencies as anyone can assert anything. For example, a reasoner could infer multiple subjects for the same value of an **Inverse Functional** property, **foaf:mbox_sha1sum**[3], caused by the incorrect copying of someone's Friend of A Friend (foaf) page, leading to a logical inconsistency. A Web rule language must be able to isolate the results of reasoning [19] to prevent them from causing inconsistencies in the global state. Another problem with Web systems involves the trustworthiness of data — what data to trust and what criteria to use for this decision. Different Web documents may be trusted with assertions about different data but not all data they contain. For example, a hospital site may be trusted with information about a potential virus outbreak but may not be trusted with respect to its inflation predictions. The ability to access only trusted RDF subgraphs from Web pages is useful in maintaining the quality of inference results. In order to trust inference results, deduction traces, or justifications as they are known, are also required [9, 10]. They provide detailed provenance information, including the data sources and the rules applied, to allow applications to evaluate the trustworthiness of a particular result through automated proof checking [3, 15]. Capturing and tracking this provenance information is another important property of Web rule languages.

AIR (**A**ccountability **I**n **R**DF) is a Semantic Web rule language that provides these features. It is represented in N3 and supports named rules, graphs, quantified variables and functions for accessing Web resources and SPARQL endpoints, and for cryptographic, string, and math operations. It also includes functions for objectively retrieving subgraphs from Web resources as well as functions for scoped contextualized reasoning.

AIR also provides ***Linked Rules***. Most rules, whether laws, security policies, business rules, or workflow plans, are rarely defined by a single entity or exist in a single document. They usually comprise of several rules that are defined and maintained by different entities. For example, a University policy depends on concepts and rules defined by the HR department, the different schools, as well as the labs within the schools. Additionally, rules often reference other rules, including those of other organizations. As an example, a hospital might want to know with whom a pharmaceutical company might share patient records it provides and might want to reason over the latter's rules before transferring patient information. AIR models this rule re-use and linkage by (i) enabling rules to be uniquely identified by Uniform Resource Identifiers (URIs) such that they can be spatially dispersed but combined during reasoning, (ii) allowing rules to be developed modularly using existing rules, and (iii) allowing rules to be executed from within other rules and their results to be queried. This conformance of AIR rules to Linked Data principles forms the basis of ***Linked Rules*** that provide a more natural way to think about and model real world rules, laws and policies on the Web.

AIR has been used in various projects to meet different rule-based inferencing requirements. It has been used for controlled information exchange in information sharing environments[4] where contextualized reasoning was very important. It has been used to secure access to Web resources [17] and SPARQL endpoints [6] based on the credentials of the user. AIR has also been used to analyze database queries [11] with respect to privacy

[3] `http://xmlns.com/foaf/spec/#term_mbox_sha1sum`

[4] http://dig.csail.mit.edu/2009/DHS-fusion/

policies that required ontology-based reasoning, the querying of large knowledge bases for factual information and expressive non-monotonic reasoning. Lastly, AIR has been used in accountability mechanisms for processing audit logs and looking for data usage outside of what was allowed by data usage policies [23]. All of the above projects made use of justifications, for deductions by the reasoner, for debugging and accountability purposes.

This paper is structured as follows: we start by describing a motivating scenario in Section 2 before providing an overview of the AIR language in Section 3. In section 4, we discuss AIR's support for justifications — how they are generated and the representation schema. The next section, Section 5, deals with an overview of the procedural and declarative semantics of AIR followed by a comparison to related work in Section 6. We conclude the paper with a summary and directions for future work in Section 7.

2 Motivating Use Case

To better understand how AIR's features may be used to solve complex problems, consider the following hypothetical scenario: Maury conducts genomic research at BigPharma, a pharmaceutical company. As part of a larger effort by BigPharma to develop new therapies for diseases of the lung, BigPharma has established a partnership with a number of hospitals including BigCity Hospital. The purpose of this relationship is to obtain the responsiveness of patients with certain disease expressions to particular medications. Maury performs a federated search over the SPARQL endpoints of these hospitals for patient response information.

BigCity receives this request but the information cannot be simply given to Maury, as BigCity, like all hospitals, restricts how patient information may be disseminated to third parties. Here, BigCity must determine whether or not dissemination of the requested patient information to Maury would violate any of its rules.

BigCity's rules state that patient information may only be disseminated to requesters from "pharmaceutical collaborators". Since Maury's credentials need to be verified by information on a particular SPARQL endpoint maintained by BigPharma, whatever language we implement must be able to query SPARQL endpoints and incorporate the results of queries into its rules. In this way, Maury may be determined to truly be a member of BigPharma, a pharmaceutical collaborator. Other rules might require patients to have consented to share their information and might need to query another SPARQL endpoint for this information.

Determining Maury's affiliation is not entirely sufficient for BigCity to send the data. Maury must also be able to prove that he is authorized to act in an appropriate role as defined by another BigCity rule. This rule defines role-based access control (RBAC) rules for all BigCity employees and states that only "Principal Researchers" can access patient response information. The rule also reasons over ontology mappings of collaborators. Maury must be able to prove that he is employed at BigPharma in a role equivalent to BigCity's "Principal Researcher". As the definitions in the RBAC rule are used in multiple rules, BigCity should be able to encode it as a separate rule which may be invoked from other rules. In practice, a rule language that satisfies BigCity's needs should be *recursive.*

Unfortunately, Maury is not working in an appropriate role and is denied access to the information. Although it is entirely possible that BigCity would not desire that Maury know why he was denied access in the first place, Maury may benefit from knowing exactly why he was not shown the material he requested. Thus, the rule language used should be able to encode and present justifications that correspond to the logic used. With such a justification, Maury will be able to determine not only how the decision against access was made, but he will also be able to verify the correctness of this determination.

3 AIR Overview

AIR is an extension to N3Logic [4] and has been structured to meet the justification and reusability requirements of Web information systems. Along with including the N3Logic features of scoped negation, scoped contextualized reasoning, nested graphs, and built-in functions, AIR also supports ***Linked Rules*** and is focused on generating useful ***justifications*** for all actions made by the reasoner. Like N3Logic, AIR is written in N3[5], which provides a human-readable syntax for a superset of RDF. N3 extends the RDF data model by allowing for the quantification of variables as URIs with the **@forAll** and **@forSome** directives. It also permits the inclusion of nested graphs by using curly braces to quote subgraphs.

AIR is made up of a set of built-in functions and two independent ontologies — the first is for the specification of AIR rules, and the second deals with describing justifications for the inferences made by AIR rules. The built-in functions allow rules to access Web resources, query SPARQL endpoints, and perform scoped contextualized reasoning, as well as basic math, string and cryptographic operations. While developing the rule ontology, we focused on capturing how real world rules and laws are written to allow them to be represented naturally in AIR. For the justification ontology, our focus was on re-usability of justifications and on automated proof checking. When given as input some AIR rules, defined in the AIR rules ontology, and some Semantic Web data, the AIR reasoner produces a set of inferences that are annotated with justifications, described in the justification ontology. The runtime input to AIR rules can be any RDF graph or an empty graph, if the rules only access Web resources.

All the examples in the paper are in N3. Please refer to `http://www.w3.org/2000/10/swap/Primer` for an overview of N3 and to the Appendix for the list of namespaces used in the paper.

3.1 AIR Rules

As illustrated in Figure 1, AIR rules are defined using the following properties: ***air:if***, ***air:then***, ***air:else***, ***air:description***, ***air:rule*** and ***air:assert***. Every rule is named with a URI, and they are grouped into ***air:RuleSets*** or nested under other rules. This nesting can happen either under the ***air:then*** property or the ***air:else*** property. The rules nested directly under the ***RuleSet*** are referred to as the top rules of the ruleset. A chain of rules is defined as a sequence of rules, such that every rule, barring the first in the chain, is nested under either the ***then*** or the ***else*** of the preceding rule. Figure 2 provides an example of nested rules. In this case, `:CheckCollaboratingPharma` only becomes active

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

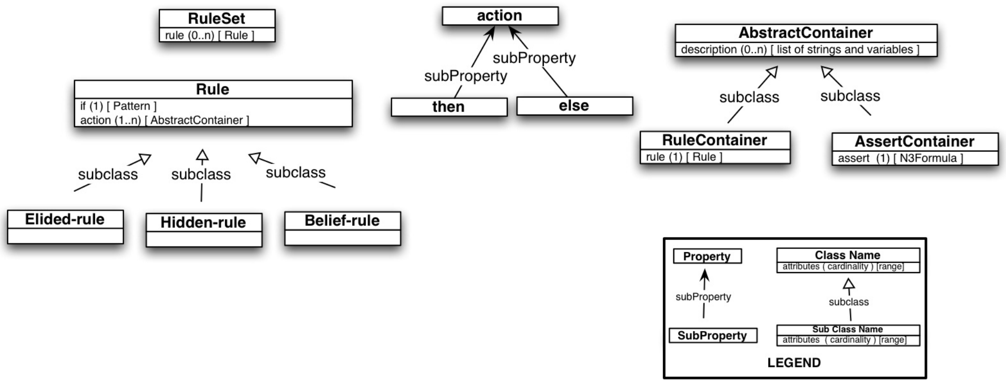

Fig. 1. AIR Rule Ontology

if the ***air:if*** of the parent rule, `:CheckQueryRule`, matches a pattern in the knowledge base. On the other hand, `:NonCollaboratorRequestProcessing` becomes active when the ***air:if*** of its parent rule fails.

There are three kinds of rules in AIR — ***air:Belief-rule***, ***air:Hidden-rule*** and ***air:Elided-rule***. All rules are, by default, ***Belief-rules***. The descriptions and conditions of ***Belief-rules*** contribute to the overall justification. `:CheckQueryRule` in Figure 2 is an example of a ***Belief-rule***. In contrast, ***Hidden-rules*** and ***Elided-rules*** are used to modify the default justification. (Please refer to Section 4.2 for more information about justification generation and modification).

The conditions of a rule (for example, **:REQUEST a ex:Patient_Record_Request** in Figure 2) are defined as graph patterns which are matched against RDF graphs, similar to the Basic Graph Pattern (BGP) of SPARQL queries[6]. If the condition matches the current state of the world, defined as the facts known or inferred to be true so far, then all the actions under ***then*** are fired, otherwise all the actions under ***else*** are fired. The condition matches the current state if there is a subgraph of known facts that matches the graph pattern. This subgraph is referred to as the matched graph.

Existentially quantified variables may be declared within graph patterns by using the **@forSome** directive. Any universally quantified variables, quantified using **@forAll**, are declared outside of the rule. The scope of a existentially quantified variable is the graph pattern in which it is declared, whereas that of a universally quantified variable is any chain of nested rules.

Rules with conditions where some graph pattern must match the current state and others should not match the current state can be specified through the nesting of rules. The actions under ***then*** and ***else*** (together referred to as actions) are defined by an assertion pattern using the ***air:assert*** property, or a rule reference, using the ***air:rule*** property. All actions may be annotated with the natural-language description of the rule or action through the use of the ***air:description*** property.

[6] `http://www.w3.org/TR/rdf-sparql-query/#BasicGraphPatterns`

```
@forAll :REQUEST, :R, :CRED, :C, :G, :SE.

:BigCityDissemination a air:RuleSet;
    air:rule :CheckQueryRule.
:CheckQueryRule a air:Belief-rule;
    air:if {
        :REQUEST a ex:Patient_Record_Request;
                ex:requester :R;
                ex:credentials :CRED.
    };
    air:then [
        air:description ("Received patient record request, " :REQUEST " from " :R ".");
        air:rule :CheckCollaboratingPharma ;
    ].
:CheckCollaboratingPharma a air:Belief-rule;
    air:if {
         :CRED log:semantics :C.
         :C log:includes { :G foaf:member :R }.
         :BIGCITYDB log:includes { :G a ex:CollaboratingPharma ;
                                  ex:sparqlendpoint :SE. }.
    };
    air:then [
        air:description ( :REQUEST " might be from collaborating
        pharmaceutical company, need further verification from their sources.");
        air:rule bcc:CollaboratorRequestProcessing ] ;
    air:else [  air:rule  bnc:NonCollaboratorRequestProcessing ].
```

Fig. 2. Example AIR RuleSet: Following from the motivating use case, this rule checks if the request is from a collaborating pharmaceutical company. Maury's credential identifies his organization and BigCity checks whether it is a collaborating partner and if so, identifies its SPARQL endpoint. The input also includes the RDF graph of Maury's request, described at <http://bigcity.example.com/record_request_log.n3>

When the action is executed, the variables in an assertion pattern (for example, the variable `:CRED` in Figure 2) are substituted by their bindings and the pattern is asserted. If a rule reference is defined instead, an instance of that rule, created by substituting the variable bindings acquired so far, is activated. The variables in any ***air:description*** property are also instantiated, and the description is maintained by the reasoner.

Any asserted graph pattern cannot contain blank nodes or existentially quantified variables. When a rule containing an ***air:else*** property is activated, its condition cannot contain unbounded universally quantified variables.

Since AIR supports ***Linked Rules***, AIR rules may be identified by their URIs that allows them to be easily reused and developed modularly. For instance, `bc:CheckAppropriateRole` and `bnc:NonCollaboratorRequestProcessing` in Figure 3 are defined outside of their parent rule's document, and AIR semantics cause them to be included during the reasoning of their parent rule, `bcc:CollaboratorRequestProcessing`. The effect of reusing a rule is that all the rule chains starting with that rule are reused.

3.2 AIR Built-ins

AIR supports most N3Logic built-ins including those for cryptographic, math, string, list and time functions. ***:LEN math:notGreaterThan 6*** is an example graph pattern that uses the `math:notGreaterThan` built-in function. The subgraph of ***:LEN math:notGreaterThan 6*** matches if the value of `:LEN`, which is a variable, is found

```
@forAll :SPARQL, :RESULTS, :ORG, :ROLE.

bcc:CollaboratorRequestProcessing a air:Belief-rule;
    air:if {
        ("""
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT { ?G foaf:member """ :R """ . """ :R """ rdf:type ?ROLE . ?G foaf:name ?ORG . }
WHERE { ?G foaf:member """ :R """ .
        ?G foaf:name ?ORG.
        """ :R """ rdf:type ?ROLE . }
       """) string:concatenation :SPARQL .
        ( :SE :SPARQL ) sparql:queryEndpoint :RESULTS .
        :RESULTS log:includes { :G foaf:member :R . :G foaf:name :ORG.
                                :R rdf:type :ROLE . }.
    };
    air:then [
        air:description ( :R " has been confirmed to be employed at "
        :ORG ", which is an authorized collaborator.");
        air:rule bc:CheckAppropriateRole ;
    ];
    air:else [ air:rule bnc:NonCollaboratorRequestProcessing ].
```

Fig. 3. Querying a SPARQL endpoint Within AIR Rules: An AIR rule that may be used to query the SPARQL endpoint to discover if Maury is indeed employed by the organization he claims to represent and to retrieve his role there. There will be a similar rule that checks if patient consent has been given.

to be less than or equal to 6. Arguments to N-ary functions and built-ins are declared using an **rdf:List**. RDF graphs in Web documents can be accessed using the built-in function ***log:semantics***, and subgraph matching functionality is provided by the ***log:includes*** property. The ***log:semantics*** and ***log:includes*** properties allow rules to extract certain RDF graphs from Web documents making them useful in *trust management* as different Web documents may be trusted with assertions about certain data but not all data. Accessing only trusted RDF subgraphs from Web pages is useful in maintaining the quality of inference results. The ***log:notIncludes*** property is used to check if a graph is not included in another graph (with closed world semantics).

As seen in Figure 3, SPARQL queries can be executed from within AIR rules using ***sparql*** built-ins. SPARQL CONSTRUCT queries may be sent to SPARQL endpoints using the ***sparql:queryEndpoint*** property assertion, and subgraph patterns may be searched for in the graph returned by the endpoint.

AIR's ability to incorporate the contents of SPARQL queries meets the first need of the motivating use case; BigCity may query back to BigPharma's SPARQL server to verify Maury's identity and role. The rule in Figure 3 includes a SPARQL CONSTRUCT query to determine Maury's affiliation and extract a graph that may be used in additional rules. If this graph contains a desired triple **:G foaf:member :R .**, it would allow for the conclusion that Maury is actually a member of BigPharma as his credentials claim.

The ***air:justifies*** property can be used to check if the execution of some external AIR rules (AIR-closure) against some Semantic Web data produces (RDF-entails) a certain RDF graph. The results of these external rules are not directly included into the current state but can be queried selectively. Together, the ***log*** and ***air:justifies*** built-ins provide *scoped contextualized reasoning.* Figure 4 provides an example of the use of ***air:justifies*** in order to prove that Maury performs an appropriate role, as defined in a separate

```
@forAll :LOG, :RULE, :FUNCTION .

bc:CheckAppropriateRole a air:Belief-rule;
    air:if {
        ( { :R a ex:Requester; rdf:type :ROLE. :G foaf:member :R. }. :RESULTS)
               log:conjunction :LOG.
        <http://bigcity.example.com/bcrbac.n3> log:semantics :RULE.
        ( ( :LOG ) ( :RULE ) ) air:justifies
            { :R air:compliant-with bcpol:BigCityRBAC.
              :R ex:authorized_role :FUNCTION . } .
    } ;
    air:then [
        air:description ( :R " is in an authorized role " :FUNCTION
                          " as defined by BigCity RBAC." );
        air:assert { :REQUEST air:compliant-with :BigCityDissemination. } ;
    ] ;
    air:else [
        air:description ( :R " performing function " :ROLE
                          " but is not in a permissible role
                            as defined by BigCity RBAC." );
        air:assert { :REQUEST air:non-compliant-with :BigCityDissemination. } ;
    ] .
```

Fig. 4. Recursive Execution of AIR Rules: ***air:justifies*** may be used to recursively call rule sets, as in this example where `BigCityRBAC` is executed from within `CheckAppropriateRule` and ***log:includes*** is used to check the results of its execution.

rule, `bcpol:BigCityRBAC` (located at <`http://bigcity.example.com/bcrbac.n3`>). AIR thus meets the second need of our hypothetical use case: that BigCity should be able to reuse and execute rules defined in other documents.

4 AIR Justifications

Upon finalizing its reasoning results, the AIR reasoner produces a justification that contains sufficient information to understand the decisions and actions made by the reasoner and to debug the rules, if needed. We have developed a justification ontology based on basic Proof Markup Language (PML) concepts [18]. The AIR operational semantics (as described in Section 5) also define how this justification is generated by the reasoner. As AIR justifications themselves are N3 data, they can be consumed by other reasoners including AIR to evaluate the *quality* and *trustworthiness* of the results based on the rules and data sources used.

4.1 Justification Ontology

The AIR justification ontology extends certain PML [18] concepts as shown in Figure 5. We use the PML-Lite vocabulary[7] that represents a subset of PML and is modeled as Events, which can be conveniently used for representing the AIR reasoning steps. The AIR justification ontology consists of three main classes — ***pmll:Event***, ***pmlj:Mapping*** and ***pmll:Operation***. The ***pmll:Event***s may be categorized into ***airj:BuiltinAssertion***, ***airj:ClosingTheWorld***, ***airj:ClosureComputation***,

[7] PML-Lite - http://tw.rpi.edu/proj/tami/PML-Lite

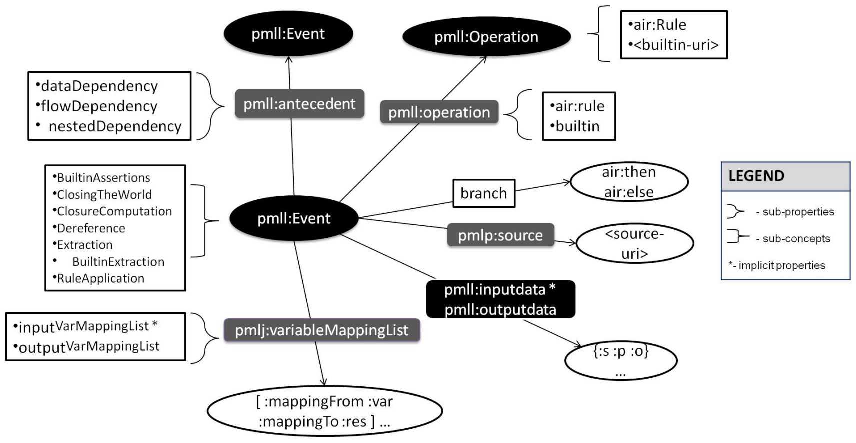

Fig. 5. AIR Justification Ontology: PML concepts are enclosed in solid boxes and ovals, sub-concepts and sub-properties are depicted in the boxes adjacent to the concept and property labels, respectively.

airj:Dereference, ***airj:Extraction***, and ***airj:RuleApplication*** events depending on the operation performed.

The AIR reasoner computes the closure of input facts with respect to a given AIR ruleset. An ***airj:ClosureComputation*** event captures this operation. The events of all other types are all part of some ***airj:ClosureComputation*** event.

The input to an AIR ruleset may be contained in a set of N3 or RDF documents. These documents, on the web or on a local machine, are dereferenced to retrieve their RDF representation. This step is encoded as an ***airj:Dereference*** event. In the example in Figure 6, the document located at <`http://bigcity.example.com/record_request_log.n3`> is dereferenced to get the RDF graph `_:g1` in the `_:g0`.

Apart from the input triples, there are RDF triples that are tautologically true. Some of these are built-in assertions. In practice, built-in triples are created dynamically. However, we abstract this process and represent it through ***airj:BuiltinAssertion*** and ***airj:BuiltinExtraction*** events. The output of an ***airj:BuiltinAssertion*** event is assumed to be the graph that contains all the true built-in assertions (potentially unbounded in number) for the built-in function specified by the ***airj:builtin*** property. The assertions needed to match rule conditions are then extracted from this output in an ***airj:BuiltinExtraction*** event. The ***airj:Extraction*** event from which ***airj:BuiltinExtraction*** is derived, is used to encode the step where a subgraph is obtained from a graph. The input to airj:BuiltinExtraction event is also left implicit, but may be determined by the ***airj:BuiltinAssertion*** event on which the extraction has an ***airj:dataDependency***.

In the example in Figure 6, to match the condition of `:CheckCollaboratingPharma` it was required that the graph `:BIGCITYDB` include (among others) the triple **<http://bigpharma.example.com/> a ex:CollaboratingPharma**. This was tested using the ***log:include*** built-in. Using the abstract event `_:g2`, we get all ***log:include***

```
_:g0 a airj:Dereference ;
     airj:source <http://bigcity.example.com/record_request_log.n3> ;
     airj:outputdata _:g1 .

_:g2 a airj:BuiltinAssertion ;
     airj:builtin log:includes .

# Note, we use :BIGCITYDB as a short-hand for the graph bound to :BIGCITYDB in
# :CheckCollaboratingPharma

# In practice, :BIGCITYDB would be replaced with what the graph was bound to, rather
# than remaining a URI.
_:g3 a airj:BuiltinExtraction ;
     airj:dataDependency _:g2 ;
     airj:outputdata { :BIGCITYDB
       log:includes {
         <http://bigpharma.example.com/> a ex:CollaboratingPharma ;
           ex:sparqlendpoint <http://bigpharma.example.com/sparql> } } .

# Other built-ins are asserted similarly...

_:g4 a airj:RuleApplication ;
     air:rule :CheckCollaboratingPharma ;
     airj:branch air:then ;
     airj:dataDependency _:g0, _:g3 ;
     # More dependencies would be here to justify the
     # other built-ins in :CheckCollaboratingPharma.
     airj:outputVariableMappingList (_:mapping1 _:mapping2) . # and so on

_:mapping1 a pmlj:Mapping ;
           airj:mappingFrom :G ;
           airj:mappingTo <http://bigpharma.example.com/> .

# More mappings like this would be generated for each and every mapping
# bound in the particular rule application.
```

Fig. 6. An AIR Justification: Part of the justification of Maury's employment at BigPharma as based on the rules provided in Figures 2. Note that for reasons of convenience, all events generated have been assigned to blank nodes (e.g. `_:g0`), but could just as easily be assigned to automatically generated dereferenceable URIs in a local justification log.

triples that can be true, and extract the one relevant for reasoning in the `_:g3` event. Thus, `_:g3` has a data dependency on `_:g2`.

The ***airj:ClosingTheWorld*** event refers to a step of reasoning where triples other than the ones in the input or those inferred so far, and rules other than those that are active, are believed not to be true. An ***airj:ClosingTheWorld*** event has data and/or flow control dependencies on all prior ***airj:RuleApplication*** events. These dependencies are represented through ***airj:dataDependency*** and ***airj:flowDependency*** properties.

In Figure 7, another part of the justification begun in Figure 6, an ***airj:ClosingTheWorld*** event, `_:g11`, is used to close the world to decide `bc:CheckAppropriateRole` so that it may follow the ***air:else*** action. This allows the reasoner to conclude that Maury is not performing a valid function that would justify releasing the patient records from BigCity. Note that event `_:g11` has an ***airj:dataDependency*** on all previous ***airj:BuiltinExtraction*** and ***airj:Dereference*** events which built the now-closed knowledge base.

The ***airj:RuleApplication*** event represents a rule firing event. It is linked to the rule that fired with the ***air:rule*** property. When a nested rule is activated, the known variable

```
_:g11 a airj:ClosingTheWorld ;
      airj:dataDependency _:g0, _:g4 . # And others in the same way.

_:g12 a airj:RuleApplication ;
      air:rule bc:CheckAppropriateRole ;
      airj:branch air:else ;
      # ...
      airj:dataDependency _:g11 ;
      airj:outputdata { <http://bigcity.example.com/record_request_log.n3#mauryRequest>
        air:non-compliant-with :BigCityDissemination . } .
```

Fig. 7. Justifying Maury's rejection: Closing the world to assert ***air:else***.

bindings are passed from the parent rule. The ***airj:RuleApplication*** event for a nested rule has a special flow dependency, referred to as an ***airj:nestedDependency***, on the ***airj:RuleApplication*** event where the parent rule fired. The input variable bindings for the nested rule are implicit and are the same as the output variable bindings of the parent.

When the condition of the rule is satisfied, variable bindings may be acquired, and the ***air:then*** actions are effected. Otherwise, the ***air:else*** actions are effected. Which actions are effected is declared using the ***air:branch*** property. The variable bindings are acquired upon pattern matching of the rule condition against the fact base, and declared using the ***airj:outputVariableMappingList*** property. Any triples asserted when the rule fires are declared using the ***airj:outputdata*** property and the event is annotated with natural-language description specified by the ***air:description*** property. ***airj:RuleApplication*** events also may have data dependencies on other events. The condition can be satisfied by triples from more than one input log, or by triples asserted in some of the prior ***airj:RuleApplication*** events may. These events also have flow control dependencies on prior ***airj:ClosingTheWorld*** events.

In the example given in Figure 6 and its continuation in Figure 7 we observe two ***airj:RuleApplication*** events, `_:g4`, using the rule `:CheckCollaboratingPharma`, and `_:g12`, using `bc:CheckAppropriateRole`. Neither rule application was a result of a nested rule, so no ***airj:nestedDependency*** triples were generated. `_:g4` followed the ***air:then*** path, and depended on information generated by dereferencing the log in `_:g0` and a built-in asserted in `_:g3`. `_:g12`, in contrast, followed the ***air:else*** path by depending on closing the world in `_:g11`. This particular action asserted non-compliance, which is indicated by the ***airj:outputdata*** property.

By manually or automatedly tracing the output data and rule dependencies from each ***airj:BuiltinExtraction***, ***airj:Dereference***, or ***airj:RuleApplication*** event, we may determine the order in which rules were applied and what data was used. This also allows for the construction of meaningful, human-readable justification traces and interfaces [10].

4.2 Justification Generation

AIR supports justification generation for every action taken by the reasoner. When an action is taken (either a graph pattern asserted or a rule activated), the action is annotated with the identity of the rule and with either the matched subgraph or a list of components known to be true under the closed world assumption. The operational semantics, described later in Section 5, define how the justification is generated by the reasoner. The default justification for an AIR conclusion is constructed by taking the union of annotations for the conclusion and the rule in which the conclusion was asserted.

Though knowing the rules and facts from which a conclusion is derived is useful, it does not describe what the rule was attempting to do. In order to provide natural-language explanations, we allow ***air:descriptions*** to be added to ***air:actions***. These descriptions are English sentences and can contain variable values. The ***air:description*** property is a list instance, where list items are enclosed in brackets and separated by commas. Each list item can either be a string enclosed in quotes or a quantified URI variable. During the reasoning process, each variable is replaced by its current value and inserted into the description string. For example, the ***description*** of `:CollaboratorRequestProcessing` from Figure 3 is a list consisting of two variables, `:R` and `:Q` and two strings — *has been confirmed to be employed at* and *which is an authorized collaborator*.

```
...
@forSome _:g11 .
_:g11 a airj:RuleApplication ;
        # an air:flowDependency may be given.
        airj:outputdata { <http://bigcity.example.com/record_request_log.n3#mauryRequest>
        air:non-compliant-with :BigCityDissemination . } .
...
```

Fig. 8. Justification of Elided Rule: Justification for Maury's rejection does not contain the ***Elided-rule***, `:CheckAppropriateRole`

Sometimes, the default justification can be unwieldy or very revealing, and needs to be modified to hide trivial or sensitive information. AIR provides mechanisms to *declaratively modify justifications* generated by default. Rules in AIR may be declared to be ***air:Hidden-rules*** or ***air:Elided-rules*** to suppress justifications for certain actions. The detailed justifications for actions executed when an ***Elided-rule*** fires are suppressed, and only the natural-language description is provided. The justification for actions executed when a ***Hidden-rule*** or its descendants fire are suppressed completely. This flexibility to control the level of details, at a rule-based granularity, helps the rule writers to adjust the justification so that sensitive information is not revealed and so explanations are not overly verbose. Nesting of rules can be used advantageously to split a rule's conditions across multiple rules when parts of the graph pattern refer to sensitive (or insignificant) information, such that rules with sensitive conditions may be elided or hidden.

From our use case, Maury must be able to prove that he is authorized to act in an appropriate role in order to access information from BigCity. It is entirely possible that BigCity would desire that Maury not know what roles are allowed access, in order to prevent later exploitation. In this case, `bc:CheckAppropriateRole` from Figure 4 could be declared to be an ***Elided-rule*** such as ***bc:CheckAppropriateRole a air:Elided-rule.*** As illustrated in Figure 8, the justification for Maury's rejection would not contain information about `:CheckAppropriateRole`.

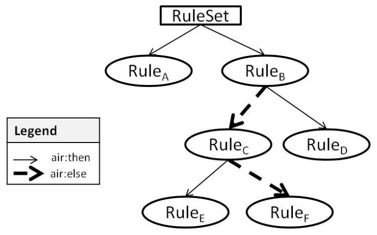

Fig. 9. Example of AIR Rule Nesting

5 AIR Semantics

The procedural semantics of AIR describe how AIR rules fire and how inferences are made. The AIR reasoner applies forward chaining reasoning to compute the closure of AIR rules over the input data. When AIR rules fire, their actions, substituted with known variable bindings, are effected. As a result, new rules may be added to the rule base, i.e. new rules may be activated, and/or facts may be deduced. Initially the rule base contains all top rules in the ruleset, and the fact base is made of the input facts. The rules in the rule base are said to be active. The active rules whose conditions match the current state of the world (fact base) are referred to as successful rules, whereas those active rules whose conditions have no match are called failed rules.

AIR reasoning is performed in stages. In any given stage, the successful rules are given priority over failed rules and their then-actions are effected before failed rules fire. When all successful rules have fired the world is temporarily closed and the else-actions of all the failed rules are fired simultaneously 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 AIR closure is computed in a finite number of stages [13].

In order to illustrate rule nesting, we consider an arbitrary nesting shown in Figure 9. Initially, i.e. before entering stage 1, only Rule$_A$ and Rule$_B$ are in the rule base. Then, if the condition of Rule$_B$ is satisfied, Rule$_D$ would be active in stage 1. However, if in stage 1 Rule$_B$ doesn't succeed, then Rule$_C$ will be added to the rule base after the world is closed for stage 1, and will be active from stage 2. Now, in stage 2, if Rule$_C$ succeeds then Rule$_E$ will become active in stage 2. Otherwise Rule$_F$ will be active from stage 3. Note that if Rule$_B$ succeeds in later stages, say stage 3, then Rule$_D$ will also be active (in addition to Rule$_C$) from stage 3 onwards.

The declarative semantics of an AIR program are defined through translation to stratified Logic Programs [20]. The AIR-closure computation is polynomially-complete in data-complexity and exponentially-complete in program-complexity, where data complexity is the complexity of computing the closure when the program is fixed and the facts are input and program complexity is the complexity when the facts are fixed and program is input [13].

Nesting introduces an ordering of rules that can be leveraged to encode fairly expressive Logic Programs (LP) in AIR, such as Positive LP and a special class of stratified LP - Positively Stratified Negatively Hierarchical LP [13]. Furthermore, SPARQL SELECT and CONSTRUCT queries may be encoded in AIR and executed by the AIR reasoner [13].

6 Related Work

There are many rule languages and rule systems, some advanced and others at prototypical levels. Examples of Web rule languages include N3Logic [4], NG [21], SILK [8] and SWRL[8], while some rule engines are Jess[9], Jena[10] and XSB [5]. Liang et. al. give a nice overview of popular, and often advanced, rule systems [16].

Though N3Logic supports monotonic negation, AIR supports non-monotonic negation. In comparison, NGs supports the well-founded (WF) negation, XSB support negation-as-failure, and Jess supports non-logical negation. SILK supports WF non-monotonic negation as well as classical negation. Nesting of rules is unique to AIR amongst the systems mentioned above. OPS/YES [22] extends the OPS5 [7] production rule system with many features including that which allows matching in the actions. This is referred to as incremental rule addition and AIR has a similar notion of rule nesting.

Amongst the above systems, only AIR provides justifications. The Ontonova system [1] provides natural-language explanation of proof trees for conclusions by the Ontobroker, through meta-inferencing. The meta-inferencing rules are defined for different rule instantiations and applied over the logs generated by Ontobroker during the inferencing process. The logs contain instantiated rules that were successfully applied and led to the derivation of an answer. The defeasible reasoning system, DR-DEVICE, has a similar mechanism for generating proofs[2]. Defeasible rules can be translated to XSB rules, and interpreted by XSB. The XSB trace is processed and tagged with a proof schema. In the context of OWL, a subset of the ontology that is sufficient for OWL entailment to hold is treated as the justification for the entailment and algorithms exist to derive the minimal such subset [9, 12]. Unlike the above systems, AIR supports syntax and semantics to provide for the alteration of rule definitions, addition of natural-language descriptions and hiding of portions of the justification.

Contextualized reasoning is important for Web rule languages because information on the Web is often assumed to be incomplete or inconsistent, and its correctness is subject to the trustworthiness of the source. [19] gives a formal definition of LPs with context and scoped negation. AIR borrows its contextualized reasoning aspects from N3Logic. Other than N3Logic and AIR, SILK supports contextualized reasoning. In SPARQL queries, the query patterns can be restricted to selective named graphs, and as a result NGs naturally supports contextualized reasoning.

The 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. However, because of the former, AIR negation can be termed logical. RIF-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. AIR supports negation unlike BLD, but it does not support function symbols that BLD does. Further, AIR has a rich set of built-ins. SWRL is a function-free rule language limited to binary and unary predicates, and all its

[8] http://www.daml.org/rules/proposal/
[9] http://www.jessrules.com/
[10] http://jena.sourceforge.net/

features, barring different-from, are covered by RIF-BLD. Therefore, AIR is at least as expressive as SWRL. As mentioned earlier, PLPs can be encoded in AIR and therefore OWL 2 RL[11] inference rules can be encoded in AIR and used concurrently with other rules.

Though several rule languages provide a subset of AIR's features, AIR is unique in its support of ***Linked Rules***, its focus on justification (both generated by default and declaratively modified), and its ability to objectively query the contents of Web resources (both Web pages and SPARQL endpoints). For a more detailed comparison of AIR with other rule languages, rule engines and rule systems, please refer to our technical report [13].

7 Summary and Future Work

We have found that AIR's expressiveness and functionality allow it to easily capture real world rules and policies while leveraging (Semantic) Web data and protocols. It supports ***Linked Rules*** so AIR rules can be developed and re-used in a manner similar to Linked Data, provides functions for scoped contextualized reasoning, and provides justification for its inferences that can be used to evaluate the trustworthiness of its results. Though our use case demonstrates how AIR can be used in collaborative environments, AIR has also been used for information sharing[12], for access control and policy management [10, 17], to secure SPARQL endpoints [6], to check the compliance of queries against privacy policies [11] and as an accountability mechanism for checking whether audit logs comply with data usage policy [23]. Thus far, our focus has been on studying the rule requirements of open Web information systems, designing appropriate features in the rule language, and implementing AIR-based systems. Moving forward we will work on handling conflicts between rules and on enabling default behavior when none of the conditions of a ***RuleSet*** match. We are also interested in understanding the performance and scalability of AIR and plan to use or extend existing benchmarks [16].

References

1. J. Angele, E. Moench, S. Staab, and D. Wenke. Ontology-based query and answering in chemistry: Ontonova @ project halo. In *in Proceedings of the Second International Semantic Web Conference (ISWC2003). 2003*, pages 913–928. Springer Verlag, 2003.
2. N. Bassiliades, G. Antoniou, and G. Governatori. Proof explanation in the dr-device system. In *RR'07: Proceedings of the 1st international conference on Web reasoning and rule systems*, pages 249–258, Berlin, Heidelberg, 2007. Springer-Verlag.
3. L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 81–95, May 2005.
4. T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, and J. Hendler. N3logic: A logical framework for the world wide web. *Journal of Theory and Practice of Logic Programming*, 2007.
5. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[11] http://www.w3.org/TR/owl2-profiles/#OWL_2_RL
[12] http://dig.csail.mit.edu/2009/DHS-fusion/

6. M. Cherian, L. Kagal, and E. Prud'hommeaux. Policy mediation to enable collaborative use of sensitive data. In *The Future of the Web for Collaborative Science (HCLS/WWW2010/Workshop)*, April 2010.

7. C.L. Forgy. The OPS5 user's manual. Technical report, Carnegie-Mellon University, Dept. of Computer Science, 1981.

8. B. Grosof, M. Dean, and M. Kifer. The silk system: Scalable and expressive semantic rules. In *8th International Semantic Web Conference (ISWC2009)*, October 2009.

9. M. Horridge, B. Parsia, and U. Sattler. Laconic and precise justifications in owl. In *In Proc. of ISWC-08, volume 5318 of LNCS*, pages 323–338, 2008.

10. L. Kagal, C. Hanson, and D. Weitzner. Using dependency tracking to provide explanations for policy management. In *IEEE Policy 2008*, 2008.

11. L. Kagal and J. Pato. A policy-awareness architecture for preserving privacy based on semantic policy tools. *IEEE Security and Privacy Sp Issue on Privacy Preserving Sharing of Sensitive Information (PPSSI)*, 2010.

12. A. Kalyanpur, B. Parsia, M. Horridge, and E. Sirin. Finding all justifications of owl dl entailments. In *ISWC'07/ASWC'07: Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, pages 267–280, Berlin, Heidelberg, 2007. Springer-Verlag.

13. A. Khandelwal, J. Bao, I. Jacobi, L. Ding, and L. Kagal. Analyzing the air language : A semantic web rule language. Technical report, Dept. of Computer Science, Rensselaer Polytechnic Institute, 2010.

14. M. Kifer. Rule interchange format: The framework. In *RR '08: Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems*, pages 1–11, Berlin, Heidelberg, 2008. Springer-Verlag.

15. V. Kolovski, Y. Katz, J. Hendler, D. Weitzner, and T. Berners-Lee. Towards a policy-aware web. In *In Semantic Web and Policy Workshop at the 4th International Semantic Web Conference*, 2005.

16. S. Liang, P. Fodor, H. Wan, and M. Kifer. Openrulebench: An analysis of the performance of rule engines. In *18th International World Wide Web Conference (WWW2009)*, April 2009.

17. C. man AuYeung, L. Kagal, N. Gibbins, and N. Shadbolt. Providing access control to online photo albums based on tags and linked data. In *AAAI Spring Symposium on Social Semantic Web: Where Web 2.0 Meets Web 3.0*, March 2009.

18. D. L. McGuinness, L. Ding, P. P. da Silva, and C. Chang. Pml 2: A modular explanation interlingua. In *AAAI 2007 Workshop on Explanation-aware Computing*, 2007.

19. A. Polleres, C. Feier, and A. Harth. Rules with contextually scoped negation. In *Proc. 3 rd European Semantic Web Conf. (ESWC2006*, pages 332–347. Springer, 2006.

20. T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. pages 193–216, 1988.

21. S. Schenk and S. Staab. Networked graphs: a declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 585–594, New York, NY, USA, 2008. ACM.

22. M. Schor, T. Daly, T. Lee, and B. Tibbitts. Advances in rete pattern matching. In *Fifth National Conference on Artificial Intelligence*, 1986.

23. D. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. Sussman. Information accountability. pages 82–87, June 2008.

A APPENDIX: Namespaces

In this paper, we refer to the namespaces defined in Figure 10.

```
@prefix air: <http://dig.csail.mit.edu/2009/AIR/air#>.
@prefix airj: <http://dig.csail.mit.edu/2009/AIR/airjustification#>.
@prefix log: <http://www.w3.org/2000/10/swap/log#>.
@prefix math: <http://www.w3.org/2000/10/swap/math#>.
@prefix string: <http://www.w3.org/2000/10/swap/string#>.
@prefix sparql: <http://www.w3.org/2000/10/swap/sparqlCwm#>.

@prefix pmlj: <http://inference-web.org/2.0/pml-justification.owl#>.
@prefix pmlp: <http://inference-web.org/2.0/pml-provenance.owl#>.
@prefix pmll: <http://inference-web.org/2.0/pml-lite.owl#>.

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.

@prefix ex: <http://bigcity.example.com/ont#>.
@prefix bcc: <http://bigcity.example.com/bccrule#>.
@prefix bnc: <http://bigcity.example.com/bcrule#>.
@prefix bc: <http://bigcity.example.com/bigcitydissemination#>.
@prefix bcpol: <http://bigcity.example.com/bcrbac#>.
@prefix : <http://bigcity.example.com/bigcitydissemination#>.
```

Fig. 10. Namespaces Used