Reasoning Strategies for Semantic Web Rule Languages

by

Joseph Scharf

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author . . . .
Department of Electrical Engineering and Computer Science
May 1, 2008

Certified by . . . .
Tim Berners-Lee
Senior Research Scientist, Computer Science and Artificial Intelligence Lab
Thesis Supervisor

Certified by . . . .
I Dunno
Professor, Mechanical Engineering
Thesis Reader

Accepted by . . . .
I Dunno
Chairman, Department Committee on Graduate Students

Reasoning Strategies for Semantic Web Rule Languages

by

Joseph Scharf

Submitted to the Department of Electrical Engineering and Computer Science
on May 1, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This is the text for the abstract.

Thesis Supervisor: Tim Berners-Lee
Title: Senior Research Scientist, Computer Science and Artificial Intelligence Lab

Thesis Reader: I Dunno
Title: Professor, Mechanical Engineering

Acknowledgments

This is the text for the Acknowldegments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Background

2.1 RDF Data Model

2.2 RDFS Reasoning

2.3 OWL

2.4 Rule Languages

2.4.1 N3Logic

2.4.2 SWRL

2.4.3 AIR

2.4.4 RIF ??

Chapter 3

AIR Reasoner

3.1 Introduction

Justifications are required to explain reasoning results and usually include the knowledge and rules used to infer these results. For policy reasoners, this justification becomes especially important because it provides important insights into the policy development and enforcement process. Policy administrators use these justifications to confirm the correctness of the policy and to check that the result is as expected. Users, on the other hand, mainly utilize them to check that the policy enforcement process works as it should and in the case of failed queries to figure out what additional information is required to get the correct result.

Our policy approach tracks dependencies during the reasoning process in order to provide integrated justification support where policy administrators are not required to handle or manipulate these dependencies or justifications. We use a production rule system as a reasoner and a Truth Maintenance System (TMS) as the dependency-tracking mechanism. Our reasoner has additional features for improved reasoning efficiency such as goal direction, which controls how much inferencing the reasoner does. The reasoner also supports the extraction of portions of explanations in order to prevent the user from being overwhelmed by irrelevant data and rules. As explanations are usually in the form of proof trees, we have developed a graphical justification user interface, which presents the explanation either in a Semantic Web rule language or in a graphical layout that highlights the result of the reasoning and allows the explanation to be explored.

Based on this approach, we propose a new policy language aimed at meeting policy compliance requirements of open, decentralized information infrastructures such as the World Wide Web and large enterprise systems. The policy reasoner must be able to search over a knowledge base as open as the Semantic Web, but must also be able to assert closure over some set of facts in order to reach a useful result. These open environments point to the need for flexible dependency tracking that gives users and administrators the most complete possible view of the inference as well as efficient ways of reasoning.

3.2 Overview

3.2.1 Dependency Tracking

A deductive reasoning system derives conclusions from previous deductions or premises by the application of deductive rules. For any given conclusion, it is useful to know the specific set of premises that it was derived from; this set is called the *set of dependencies* for the conclusion. *Dependency tracking* is the process of maintaining dependency sets for derived conclusions.

Some dependency-tracking mechanisms provide additional features. For example, a *Truth Maintenance System* (TMS) [?] keeps track of the logical structure of a derivation, which is an effective explanation of the corresponding conclusion. Another useful feature, also provided by a TMS, is the ability to assume and retract hypothetical premises.

There are several reasons why dependency tracking is useful for policy systems:

- The dependency set for a result provides a natural focus when trying to solve policy compliance problems.
- A dependency-tracking mechanism that tracks derivation structure can provide a concise explanation for a result. This is essential for confirming that a policy

is correctly modeled. It can also help identify situations where a policy is having unanticipated or undesirable consequences.

- A dependency-tracking mechanism that supports hypothetical premises can simplify analysis and design of policies. It can be used to test the response of a policy to a hypothetical situation. Or it can be used to construct an argument based on a supposed but as-yet unsupported belief.

We have chosen to use a TMS as the dependency-tracking mechanism for our project. The TMS provides considerable power in a very simple mechanism; its primary cost is the memory required to record the structure of a derivation. Although the TMS technology was invented in the 1970s, it is not well known outside the artificial intelligence community, and consequently there are no uses of this technology in policy systems of which we are aware.

One possible complaint about dependency tracking is that its use might clutter the description of a policy, obscuring its meaning. In practice, this is not usually a problem since most of the information needed to support dependency tracking can be inferred from the policy description. To see how this is done, we first need to define some terms.

Our reasoner is a kind of *production-rule system* in which the *condition* of a rule is a pattern to be matched against a set of believed statements. When the pattern matches, the rule's *action* is performed. Typically the action asserts new beliefs, causing them to be added to the set.

When something is added to the belief set, it is associated with a *justification* for its belief. In the case of a simple statement of fact, there is a trivial justification that the statement is an assumption. A derived statement has a justification based on the inputs used to make the derivation.

In such a simple rule system, all of the dependency information is implied by the rules themselves. If a new belief is asserted by a rule's action, then its justification is the set of statements that matched the rule's pattern. More precisely, we believe the asserted statement if and only if we believe every one of the matched statements.

Additionally, the justification records an identifier for the rule; this identifier together with the matched statements provide all the relevant information about the particular deduction step just performed. Typically these deduction steps build on one another, resulting in a tree-like justification structure for any given belief, in which the belief is the trunk of the tree and the assumptions are the leaves. This tree structure is a complete explanation of the support for the belief.

Our reasoner is not quite so simple. It has additional features, such as goal direction, for which the correct dependency structure is not always inferrable from the rule. To handle such situations, we provide a means to write explicit justifications. In practice, this is used only occasionally, and has little impact on the overall readability of a policy description.

Finally, one problem with this dependency-tracking mechanism is that it can record *too much* information. It has no way to distinguish between deductions that are interesting and that should appear as part of an explanation, and those that should be omitted. For example, suppose the statement "Bob has a sister named Alice" is believed and that some rule has a pattern "?X has a brother named ?Y". A rule can be written to deduce "Alice has a brother named Bob" so the pattern can match, but that deduction step won't be very interesting for most end users.

Our system provides a simple means for policy authors to elide uninteresting deduction steps from explanations. When writing a rule that makes such a deduction step, the author declares it as *hidden*, and any deduction made by that rule will not appear in the resulting explanation. This distinction is under the control of the policy author, so the consequent hiding can be tailored to the users in the policy domain.

Let's reconsider our example to see how this works. Assign the following names:

- A: "Bob has a sister named Alice"
- B: "Alice has a brother named Bob"
- R_1: rule with pattern "?X has a brother named ?Y"
- R_2: rule that deduces B from A

In the absence of any hiding, the result of rule R_1 will be shown as derived from B, and B in turn will be shown as derived from A via R_2. However, if R_2 is marked *hidden*, then the result of rule R_1 will be shown as derived directly from A, while B won't be shown at all. A person reading the explanation will see that R_1 was triggered by A, which is semantically obvious even though syntactically incorrect.

3.2.2 AIR policy language

AIR (**A**ccountability **I**n **R**DF) is a policy language that exploits our dependency tracking approach. The policies are represented in Turtle [**?**], which is a human readable syntax for RDF. AIR constructs allow policy writers to explicitly control how the reasoning happens by invoking rules according to pattern matches and are based on AMORD constructs [**?**]. AMORD is a production-rule system that features pattern matching, dependency tracking, nesting of rules, and goal direction. The combination of these features provides expressive power (pattern matching and rule nesting), efficient execution (goal direction), and integrated explanations (dependency tracking).

AIR consists of an ontology and a reasoner, which when given a set of policies and data in Turtle, attempts to compute compliance of the data with respect to the policies. Each computed compliance result has an associated explanation outlining the derivation of the result from the inputs.

The AIR ontology comprises several classes and properties that are used to define rule-based policies. Please refer to Figure 3-1 for an overview of the AIR classes, properties, and their relationships.

There are two top-level classes in AIR: `Abstract-action` and `Abstract-container`. `Policy` is a subclass of `Abstract-container`. The `Abstract-container` class has properties for defining variables, belief rules, goal rules, belief assertions, and goal assertions that `Policy` inherits. Variables are scoped to the container they appear in. For example, in the following declaration the scope of variable `:REQ` is within `:DIGPolicy` and the rules it contains including `:rule1` and `:rule2`. The scope of variable `:MEMBER` declared in `:rule1` is `:rule1` and the scope of variable `:MEMBER` and

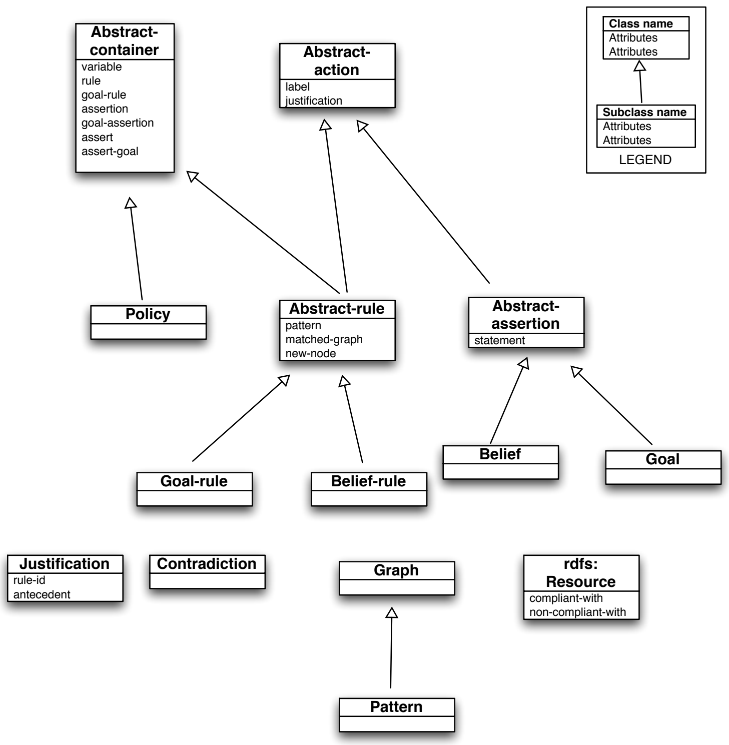

Figure 3-1: AIR ontology

`:FOAF-REQ` declared in `:rule2` is `:rule2`. If `:REQ` is bound before `:rule1` or `:rule2` are invoked then it is passed as a value and not as a variable. (Variables do not have to be uppercase.)

```
:DIGPolicy a air:Policy;
   air:variable :REQ, :REQUESTER,
           :RESOURCE, :MEMBERLIST;
   air:rule :rule1, :rule2.


:rule1 a air:Belief-rule;
   air:variable :MEMBER.


:rule2 a air:Belief-rule;
   air:variable :MEMBER, :FOAF-REQ.
```

An `Abstract-rule` is a subclass of both `Abstract-container` and `Abstract-`

`action` and its subclasses are `Goal-rule` and `Belief-rule`. Instances of `Goal-rule` match statements that are asserted as goals, and are used sparingly in typical applications. The `rule` property applies to all `Abstract-containers` and is used to attach rules to policies as shown in the example above.

A rule consists of a pattern, a matched-graph variable, a justification, a label, and zero or more actions. The matched-graph variable, justification, and label are optional. Omitting the matched-graph variable means there's no direct reference for the matched graph, which often isn't needed. Omitting the justification means a default justification is used, also the usual case. For example, the following belief rule declares a variable and has a label and a pattern.

```
:rule1 a air:Belief-Rule;
   air:variable :MEMBER;
   air:label "Member access";
   air:pattern {
     :MEMBER air:in :MEMBERLIST.
     :MEMBER a foaf:Person;
     foaf:openid :REQUESTER.
   }.
```

Belief rules implement forward-chaining deduction, while goal rules provide a means to limit the application of rules (and consequently the amount of computation performed). In the following example, a belief rule `:ruleB` has a nested goal rule `r1` that controls the introduction of sub-class type inference. The outer rule fires whenever a sub-class relationship is believed, causing `r1` to be enabled. `R1` is applied when there is a goal to show that some resource is a member of the class `:V2`, enabling the belief rule `r2`. `R2` implements the implication "if a resource is a member of a subclass `:V3`, it's also a member of the containing class `:V2`".

```
:ruleB a air:Belief-rule;
   air:variable :V1, :V2, :V3, :V4;
   air:label "sub-class implication";
   air:pattern {
      :V3 rdfs:subClassOf :V2.
   };
   air:goal-rule [       # sub-rule r1
      air:pattern { :V1 a :V2. };
      air:rule [         # sub-rule r2
         air:pattern { :V1 a :V3. };
         air:assert { :V1 a :V2. };
      ];
   ].
```

The purpose of the goal rule r1 is to limit the deductions made by the system. If `:ruleB` were rewritten as a belief rule, as below, it would make all possible deductions of this kind, whether they were needed or not. The use of a goal rule instead limits the deductions to those actually asked for (specified as goals) rather than for every possible deduction.

```
:ruleB a air:Belief-rule;
   air:variable :V1, :V2, :V3;
   air:label "sub-class implication";
   air:pattern {
      :V3 rdfs:subClassOf :V2.
      :V1 a :V3.
   };
   air:assert { :V1 a :V2. }.
```

The action of a rule consists of a set of assertions and sub-rules. A sub-rule appearing in the action of a containing rule is initially *inactive*, meaning it is not eligible for matching. When the containing rule's pattern matches and its action is performed, the sub-rule becomes *active* and its pattern will be matched as needed.

```
:SomePolicy a air:Policy;
```

```
  air:rule [
    air:label "containing rule";
    air:pattern { ... };
    air:rule [
      air:label "sub-rule";
      air:pattern  { ... };
      air:assert { ... };
      air:rule [
        air:label "sub-sub-rule";
        air:pattern { ... };
        air:assert { ... }
      ]
    ]
  ] .
```

An assertion appearing in the action of a rule can be either a belief or a goal. If it is a belief, the asserted statement can be matched against belief rules and if it is a goal the asserted statement can be matched against goal rules. In the following example, a belief assertion is associated with `:rule2`, so when the rule's pattern matches, the statement is asserted as a belief. Variables `:MEMBERLIST`, `:REQUESTER`, and `:DIG` are bound before the rule is invoked.

```
:rule2 a air:Belief-rule;
   air:variable :MEMBER;
   air:label "Member access";
   air:pattern {
     :MEMBER air:in :MEMBERLIST.
     :MEMBER a foaf:Person;
       foaf:openid :REQUESTER.
   };
   air:assert {
     :MEMBER foaf:member :DIG
   } .
```

AIR provides a small library that implements some simple RDFS [?] and OWL [?] deduction rules. For example, if a relation R is declared to be transitive, and the belief set contains xRy and yRz, the library can deduce xRz. The library provides a number of generally useful deductions, and will be augmented with new rules as the need arises.

The properties dealing with policy compliance are `compliant-with` and `non-compliant-with`; they specify that the subject is or is not compliant with the object policy. For example, the following policy has a single rule, which when matched asserts that the request is compliant with the policy.

```
:rule6 a air:Belief-Rule;
  air:variable :MEMBER, :FOAF-REQ;
  air:label "Member referral access";
  air:pattern {
    :MEMBER air:in :MEMBERLIST.
    :MEMBER foaf:knows :FOAF-REQ.
    :FOAF-REQ foaf:openid :REQUESTER.
  };
  air:assert {
    :REQ air:compliant-with :DIGPolicy
  } .
```

3.3 RETE AIR Reasoner

3.4 TREAT AIR Reasoner

3.5 TMS

3.6 Proof Generation

3.7 Related Work

The explanation framework in [?] provides natural language explanations for questions about policy decisions including explanations for failed results. The explanation generation process is separate from the regular query process and uses abductive reasoning, a method of inferring which hypothesis best explains the facts, to obtain a proof. Parts of the proof language in the proof are then substituted with natural language structures. Abductive reasoning is known to produce incorrect results and as the proof generation process is different from the query process it could infer an explanation that is different from that inferred by the query process. Another problem is that the explanation for failed policy results (e.g. why-not) ends up being the list of all rules that the user did not match. This means that the user has to sort through a lot of potentially irrelevant information and the disclosure of these additional policy rules could also be a privacy risk. In our approach, the proof generation happens along with the inference and is not a separate process. We track the rules and statements that were actually used to infer the statement (policy decision). Also, by explicitly handling failed results or unmatched cases using the `alt` construct, it is possible to provide explanations for failed (or incompliant) policy decisions without revealing all unmatched rules.

Our proofs could be easily converted to generic proof representation formats such as PML for display and analysis. PML is a general proof language or ”proof interlingua” that is used to describe proof steps generated by different kinds of reasoning engines. Once written in PML a user could Inference Web (IW) is a framework for displaying and manipulating proofs defined in Proof Markup Language (PML) [?]. IW concentrates on displaying proofs whereas our approach is mainly about generating these proofs.

Both WhyNot [?] and the Know system [?] focus explicitly on failed queries and try to suggest changes to the knowledge base that will cause these queries to succeed. Our justification approach is more general and allows failures to be captured in policies so explanations can be provided for both successful and failed policy decisions.

Chapter 4

Cwm Reasoner

4.1 Introduction

4.2 Overview

4.3 Forward Chained Reasoner

4.3.1 Technique

4.3.2 Builtins

4.4 Backward Chained Reasoner

4.5 SPARQL query engine

4.6 Proof Generation and Checking

Chapter 5

Related Work

Chapter 6

Contribution

Chapter 7

Conclusion