

Using Dependency Tracking to Provide Explanations for Policy Management

Lalana Kagal, Chris Hanson, and Daniel Weitzner
Decentralized Information Group
MIT Computer Science and Artificial Intelligence Lab
{lkagal, cph, djweitzner}@csail.mit.edu

Abstract

Explanations for decisions made by a policy framework allow end users to understand how the results were obtained, increase trust in the policy decision and enforcement process, and enable policy administrators to ensure the correctness of the policy. In our framework, an explanation for any statement including a policy decision is a representation of the list of reasons (known as dependencies) associated with its derivation. Dependency tracking involves maintaining the list of reasons (statements and rules) for the derivation of a new statement. In this paper, we describe our policy approach that (i) provides explanations for policy decisions, (ii) provides more efficient and expressive reasoning through the use of nested sub-rules and goal direction, and (iii) is grounded in Semantic Web technologies. We discuss the characteristics of our approach and provide a brief overview of the AIR policy language that implements it. We also discuss how relevant explanation information is identified and presented to end users and describe our preliminary graphical user interface.

1 Introduction

An explanation is a reason that justifies certain results or decisions. For policy frameworks, this justification becomes especially important because it provides 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 justifications 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 automated justification support so that policy administrators are not required to handle or manipulate these dependencies or justifications. We use a production rule system [19] as a reasoner and a *Truth*

Maintenance System (TMS) [9] 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 relevant portions of explanations in order to prevent the user from being overwhelmed by irrelevant data and rules. As justifications provided by the TMS are usually in the form of proof trees that may not be useful for end users, we have developed a user interface to interpret these results into a graphical user-friendly layout.

The AIR policy language, based on this approach, is 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.

The paper is structured as follows: we start by discussing details of our dependency tracking approach in section 2 and then describe our motivating scenario in section 3. In section 4, we discuss the AIR policy language and its constructs. The next two sections deal with how we generate and display explanations. We compare our research with other work in policy explanations in section 7 and conclude the paper with a discussion of our results and our future work in section 8.

2 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.
- It 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.

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.

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 of believed statements.

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.

3 Motivating Scenario: Decentralized Access Control

The DIG group has several online resources including pictures, papers, presentations, and proposals that are only accessible to members of the group and to people that members trust. The group has a webpage that lists its members using N3 [17], a representation of RDF (Resource Description Framework) [13]. An example of this group listing is provided below.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://dig.csail.mit.edu/data#> .

:DIG a foaf:Organization;
  rdfs:label "DIG";
  foaf:homepage <./>;
  foaf:logo <i/logo.png>;
  foaf:member :MLList;
  foaf:name "Decentralized Information Group" .

:MLList a rdf:List;
  rdf:first <http://dig.csail.mit.edu/People/RRS>;
  rdf:rest
    ( <http://www.w3.org/People/djweitzner/foaf#djw>
      <http://csail.mit.edu/~lkagal/foaf#me>
      <http://www.w3.org/People/Berners-Lee/card#i>
      <http://swiss.csail.mit.edu/users/cph/foaf.rdf#cph>
    ).
```

Figure 1. DIG member description in Turtle

The page identifies members by their Friend Of A Friend (FOAF) [10] pages. FOAF is a vocabulary for describing people, their email and physical addresses, projects they are working on, people they know etc. Authentication in this scenario is via OpenID [16]. OpenID is a decentralized authentication mechanism that associates users with URIs (Uniform Resource Identifiers) that they own and can modify. If users can prove that they own the URI, they are successfully authenticated. DIG members provide their OpenID URI (usually the same as their work homepages) in their FOAF pages.

When a user makes a request for a DIG resource, she specifies her OpenID URI. The OpenID component on the DIG web server authenticates the user and obtains this OpenID URI. The policy reasoner then checks whether the authenticated OpenID URI belongs to one of the members of the group or to anyone who is known (specified via `foaf:knows` on the member's FOAF pages) to any DIG member. The former is checked by extracting the FOAF pages of members from the DIG page and seeing if any of them state that their OpenID URI is the same as the requester. The latter is checked by reading in the FOAF pages

of the members and then reading the FOAF pages of people they know and checking if any of them have the same OpenID URI as that of the requester. If the OpenID URI matches, the user is given access and the resource is returned. If not, the user is denied access. The reasoner is able to provide a justification for why someone was given or denied access to a certain document.

This entire example can be accessed at the following URI

<http://dig.csail.mit.edu/2008/Papers/IEEE%20Policy/ex/>

4 AIR Policy Language

AIR (Accountability In RDF) is a policy language that exploits our dependency tracking approach. The policies are represented in Turtle [3], which is a human readable syntax for RDF, and include the quoting feature of N3Logic [17]. 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 [8]. 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 in Turtle outlining the derivation of the result from the inputs.

4.1 Language Overview

The AIR ontology comprises several classes and properties that are used to define rule-based policies. Please refer to Figure 2 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.

The following is an example of a policy, `:DecAccessPolicy`, which has 2 properties, `variable` and `rule`. The policy has three variables, `:REQ`, `:REQUESTER`, `:RESOURCE`, and consists of one rule, `:DAP-1`. The scope of the variables declared in `:DecAccessPolicy` is within the policy and the rules it contains. The scope of the variables declared in `:DAP-1` is `:DAP-1` itself. If a variable is bound before

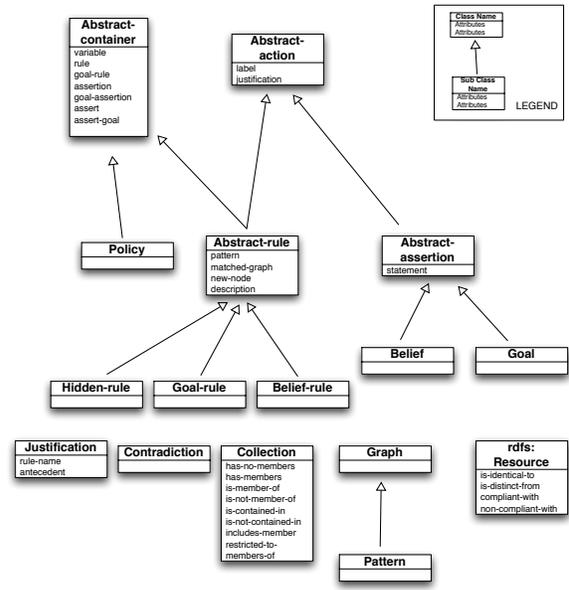


Figure 2. AIR ontology

a rule is invoked, then it is passed as a value and not as a variable. (Variables do not have to be uppercase, it is just a convention we use)

```

:DecAccessPolicy a air:Policy;
  air:variable :REQ, :REQUESTER, :RESOURCE;
  air:rule :DAP-1.

:DAP-1 a air:Belief-rule;
  air:variable :MEMBERLIST, :MEMBER.
  
```

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-container`s 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 atomic formulas in the RDF abstract syntax are called *triples* and they are analogous to one 3-place *holds(s, p, o)* predicate. A pattern is a set of triples containing variables in any one of the 3 places and matches a set of triples in an RDF graph. The matched-graph variable is bound to that set of triples at run-time. The matched-graph variable, justification, and label are optional. (More information about matched-graph and justification properties is provided in Section 5.2.) If justification is omitted means the default justification is used.

For example, the following belief rule `:DAP-2` has two variables, a label titled “Decentralized Access Control

Rule2”, a pattern, and an assertion. The pattern consists of 4 triples, where the last two triples have the same subject :MEMBER. The assertion is a single triple stating that the request is compliant with the the policy.

```
:DAP-2 a air:BeliefRule;
air:label "Decentralized Access Control Rule2";
air:variable :MEMBERLIST, :MEMBER;
air:pattern {
  :DIG foaf:member :MEMBERLIST.
  :MEMBER air:in :MEMBERLIST.
  :MEMBER a foaf:Person;
  foaf:openid :REQUESTER.
};
air:assert { :REQ air:compliant-with :DIGPolicy }.
```

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 :RuleA 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 is also a member of the containing class :V2”.

```
:RuleA 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 and provide *goal direction*. On the other hand, if :RuleA 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.

```
:RuleA 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, sub-rules, and alternatives. When the pattern of a rule matches, its assertions get fired and its sub-rules become *active*. 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.

In the following example, :sub-rule only becomes active after the pattern of :containing-rule has been matched and similarly :sub-sub-rule only becomes active after the pattern of :sub-rule has been matched.

```
: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 alternative is a rule that becomes active if the pattern of the containing rule fails. This `alt` property is used to assert closure over some set of facts. Consider the following example. If the pattern of :RuleB matches, then the assertion fires, otherwise the alternative, :RuleC becomes active.

```
:RuleB a air:Belief-rule;
air:variable :MEMBER;
air:pattern {
  :MEMBER air:in :MEMBERLIST.
};
air:assert { :MEMBER foaf:member :DIG };
air:alt [ air:rule :RuleC ].

:RuleC a air:Belief-rule;
...
```

An assertion appearing in the action of a rule consists of a set of triples containing variables and literals. It is similar in appearance to a `pattern`. When the pattern of the rule is matched, the assertion statement with bound variable values is added to the set of beliefs.

In the following example, an assertion is associated with :RuleD. When the rule’s pattern matches, the statement is asserted as a belief. The variables :MEMBERLIST and :DIG are bound before the rule is invoked.

```
:RuleD a air:Belief-rule;
air:variable :MEMBER;
air:pattern {
  :MEMBER air:in :MEMBERLIST.
};
air:assert { :MEMBER foaf:member :DIG } .
```

After the pattern of the above rule matches, assume that :MEMBER is bound to <http://people.apache.org/~oshani/foaf.rdf> and :DIG is bound to <http://dig.csail.mit.edu/data#DIG>, then the triple that is asserted is

```
<http://people.apache.org/~oshani/foaf.rdf> foaf:member
<http://dig.csail.mit.edu/data#DIG>
```

AIR provides a small library that implements some simple RDFS [6] and OWL [2] 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, `:DecAccessPolicy` has a rule, `:DAP-3`, which when matched, asserts that the request is compliant with the policy. If the pattern does not match, then the alternative rule, `:DAP-4` becomes active.

```
:DecAccessPolicy a air:Policy;
  air:variable :REQ, :REQUESTER, :RESOURCE;
  air:rule :DAP-1, :DAP-3.

:DAP-3 a air:BeliefRule;
  air:variable :MEMBER, :FOAF-REQ;
  air:pattern {
    :MEMBER air:in :MEMBERLIST.
    :MEMBER foaf:knows :FOAF-REQ.
    :FOAF-REQ foaf:openid :REQUESTER.
  };
  air:assert { :REQ air:compliant-with :DIGPolicy };
  air:alt [ air:rule :DAP-4 ].
```

5 Explanation Generation

Our reasoner provides three ways in which to identify and extract relevant information from justification trees generated by the TMS; (i) hidden rules, (ii) explicit justifications, and (iii) rule descriptions.

5.1 Hidden rules

One problem with the dependency-tracking mechanism is that it can record *too much* information. It has no way to distinguish between deductions that are interesting and those that should be omitted. For example, suppose there is a rule that says “Students have access to group resources” and the statement “GradStudents is a subclass of students” is believed. Then while proving that a student has access to some resource, there will be a rule to deduce the sub-class relationship, but it won’t be 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-rule`, 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.

For example, if we wanted to hide the sub-class rule and its justification from the overall justification trace generated, would simply declare it to be of type `Hidden-rule`

```
:Sub-Class-Implication a air:Hidden-rule.
```

5.2 Explicit Justifications

The TMS provides a tree-like justification structure for every belief (asserted or inferred) in our policy reasoner. In certain cases, such as goal direction, the correct dependency structure is not always inferable from the rule. In other cases, we might want to modify the dependency structure to provide customized justifications. To handle such situations, we provide a means to write explicit justifications and to override the default justification provided by the TMS. This is done by using the `assertion` property associated with rules. This property is composed of two components, `statement`, which is the set of triples being asserted, and `justification`, which is the explicit justification that needs to be associated with the statement. The value of the `justification` property has to be an instance of the `Justification` class. The `Justification` class consists of two properties `rule-id` and `antecedent`. The `rule-id` can be set to the name of the rule that the assertion is to be attributed to and the `antecedent` is a list of matched graphs that would act as the premises. It is possible to obtain the matched graphs of rules by using the `matched-graph` property of `Rules` with a variable.

Consider the example below. A rule, `:DAP-1`, has a nested rule, `:DAP-2`, inside it. The default justification for the assertion of `:DAP-2` would be both the rules and their matched graphs.

```
:DAP-1 a air:BeliefRule;
  air:variable :REQ, :REQUESTER, :RESOURCE, :MEMBER,
    :MEMBERLIST;
  air:pattern {
    :REQ a air:Request;
    foaf:openid :REQUESTER;
    air:resource :RESOURCE.
    :DIG data:owns :RESOURCE.
  };
  air:rule :DAP-2.

:DAP-2 a air:BeliefRule;
  air:pattern {
    :DIG foaf:member :MEMBERLIST.
    :MEMBER list:in :MEMBERLIST.
    :MEMBER a foaf:Person;
    foaf:openid :REQUESTER.
  };
  air:assert{ :MEMBER permitted :RESOURCE }.
```

In order to change this justification structure and show that a completely different rule, `:SomeOtherRule` is used, the `matched-graph`, `:G1`, for the new rule is obtained. This `matched-graph` is used as the `antecedent` and the rule name, `:SomeOtherRule`, is used as the `rule-id` in the assertion of `:DAP-2`. This overwrites the original

<pre> @prefix : <http://dig.csail.mit.edu/data#> . @prefix foaf: <http://xmlns.com/foaf/0.1/> . @prefix air: <http://dig.csail.mit.edu/TAMI/2007/amord/air#> . @prefix tms: <http://dig.csail.mit.edu/TAMI/2007/amord/tms#> . @prefix yosi: <http://dig.csail.mit.edu/People/yosi#> . :DAP_1 tms:justification tms:premise . :DAP_3 tms:description (:Req2 " is a request made by a requester with openid, " <http://auth.mit.edu/syosi> ", for DIG resource " <http://dig.csail.mit.edu/proposals/nsf.tex/>); tms:justification [tms:antecedent-expr [a tms:And-justification; tms:sub-expr :DAP_1, {DIG :owns <http://dig.csail.mit.edu/proposals/nsf.tex/> . :Req2 a air:Request; air:resource <http://dig.csail.mit.edu/proposals/nsf.tex/> foaf:openid <http://auth.mit.edu/syosi> . }]; tms:rule-name :DAP_1] . :Req2 air:compliant-with :DIGPolicy . </pre>	<pre> { :Req2 air:compliant-with :DIGPolicy . } tms:description ("The requester with openid, " <http://auth.mit.edu/syosi> ", is known to a DIG member, " <http://dig.csail.mit.edu/People/RRS>); tms:justification [tms:antecedent-expr [a tms:And-justification; tms:sub-expr :DAP_3, {<http://dig.csail.mit.edu/People/RRS> air:in :MemberList; foaf:knows yosi:YES . yosi:YES foaf:openid <http://auth.mit.edu/syosi> . }]; tms:rule-name :DAP_3] . } { <http://dig.csail.mit.edu/People/RRS> air:in :MemberList; foaf:knows yosi:YES . yosi:YES foaf:openid <http://auth.mit.edu/syosi> . DIG :owns <http://dig.csail.mit.edu/proposals/nsf.tex/> . :Req2 a air:Request; air:resource <http://dig.csail.mit.edu/proposals/nsf.tex/> ; foaf:openid <http://auth.mit.edu/syosi> . } tms:justification tms:premise . </pre>
1	2

Figure 3. Proof tree generated by TMS

justification tree of assertion of “:MEMBER permitted :RESOURCE”.

```

:SomeOtherRule a air:BeliefRule;
air:matched-graph :G1;
air:pattern { A X C };
air:assert { B A C }.

:DAP-2 a air:BeliefRule;
air:pattern {
:DIG foaf:member :MEMBERLIST.
:MEMBER list:in :MEMBERLIST.
:MEMBER a foaf:Person;
foaf:openid :REQUESTER.
};
air:assertion [
air:statement { :MEMBER permitted :RESOURCE };
air:justification [
air:rule-id :SomeOtherRule;
air:antecedent :G1
]
].

```

5.3 Rule Descriptions

The earlier mechanisms cause the default justification tree to be modified and the justification to be customized. 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 like explanations, we allow descriptions to be added to rules. These descriptions are English sentences and can contain variable values. The `description` is an optional property of rules and 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 variable. During the reasoning process, each variable is replaced by its current value and inserted into the description string.

Consider the rule below, `:DAP-3`. It has a description

containing four list items - two strings and two variables, `:REQUESTER` and `:MEMBER`

```

:DAP-3 a air:BeliefRule;
air:variable :MEMBERLIST, :MEMBER, :FOAF-REQ;
air:pattern {
:MEMBER air:in :MEMBERLIST.
:MEMBER foaf:knows :FOAF-REQ.
:FOAF-REQ foaf:openid :REQUESTER.
};
air:description ("The requester, whose openid URI
is" :REQUESTER ", is known by a
DIG member, " :MEMBER);
air:assert { :REQ air:compliant-with :DIGPolicy }

```

If the `:REQUESTER` variable is bound to `http://auth.mit.edu/syosi` and the `:MEMBER` variable is bound to `http://dig.sail.mit.edu/People/RRS`, then the description of the rule generated at run-time is

The requester, whose openid URI is `http://auth.mit.edu/syosi`, is known by a DIG member, `http://dig.sail.mit.edu/People/RRS`

6 Justification User Interface

The AIR reasoner produces explanations in the form of proof trees in Turtle. Please refer to Figure 3 for a justification generated by the reasoner for the DIG policy scenario. The justification is for a request, `:Req2`, made by *Yosi*, a friend of one of DIG members for a protected resource, `http://dig.csail.mit.edu/proposal/nsf.tex`. The reasoner finds that `:Req2` is compliant with the DIG policy because the requester is known to one of the DIG members, `http://dig.sail.mit.edu/People/RRS`. The conclusion in the proof is outlined by a box.

As these proof trees might be incomprehensible to end users, we have developed a graphical justification user interface in Tabulator [18], a Semantic Web browser. This

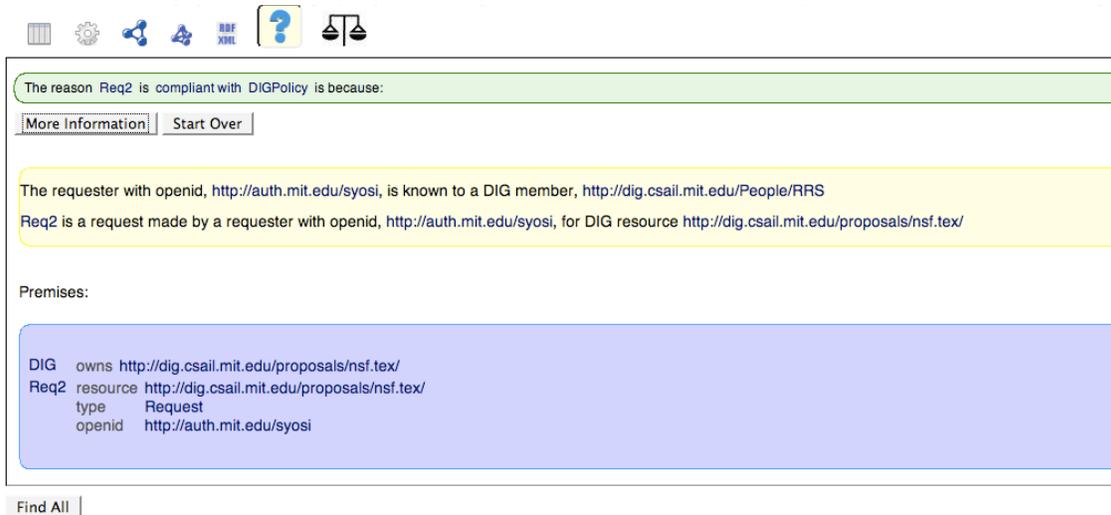


Figure 4. Justification UI

allows users to view the explanation provided by the reasoner in different ways: (i) in a simple rule language, N3, and (ii) in a graphical layout that highlights the result of the reasoning and allows the explanation to be explored. Please refer to Figure 4 for the graphical view of the explanation in Figure 3. The **More Information** button provides a way for the user to step through the proof starting from the most relevant information (in this case facts that Yosi is a friend of RRS who is a member of the DIG group) and working backwards to the top of the tree (a request was made for a DIG resource). The top bar contains the description of the rule that generated the assertion and the bar below contains premises (matched-graph) of that rule. We are still working on improving the user interface to make it more intuitive to users.

7 Related Work

Policy Languages

In recent years there have been several efforts to develop expressive policy languages using Semantic Web technologies for a variety of application domains including network management, Web services, and privacy. These include languages such as KAOs [5] and Rei [14]. KAOs policies are OWL descriptions of actions that are permitted (or not) or obligated (or not). This limits the expressive power, but allows the classification of policy statements, enabling conflicts to be discovered from the rules themselves. Another advantage that KAOs has is that if policy descriptions stay within OWL-Lite or OWL-DL, then the computation is decidable and has well understood complexity results. On the other hand, languages such as Rei and AIR allow for rules

to be defined over attributes of classes in the domain including users, resources, and the context. Though they are expressive, they lack well defined semantics.

AIR is more expressive than either KAOs or Rei. However, Rei includes tools for policy analysis and speech acts for dynamic policy modification, both of which AIR lacks. Conversely, AIR is focused on generating explanations for policy decisions, which neither KAOs nor Rei are capable of.

Explanation Generators & Languages

The explanation framework in [4] provides natural language explanations for questions about policy decisions including explanations for failed results. In their framework, 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 their proof language in the proof are then substituted with natural language structures. One problem with this approach is that abductive reasoning is known to produce logically invalid results that can be confirmed inductively [11] and as the proof generation process is different from the query process, it could infer an explanation that is different from that was 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, however, the proof generation happens along with the inference and is not a separate process so our reasoner tracks the rules and statements that were actually used to infer the statement (policy decision). Also, by explicitly handling failed results or un-

matched cases using the `alt` construct, our reasoner is able to provide explanations for failed (or in-compliant) policy decisions without revealing all possible unmatched rules.

Both WhyNot [7] and the Know system [1] 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.

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 our proofs are converted to PML a user could manipulate them in Inference Web (IW), a framework for displaying and manipulating proofs defined in Proof Markup Language (PML) [15]. IW concentrates on displaying proofs whereas our approach is mainly about generating these proofs.

8 Conclusion and Future Work

Having prototyped a number of different scenarios with AIR, we believe that the language is expressive enough to support policy compliance in a variety of environments. In the future, we plan to address the engineering challenges of scalable, efficient reasoning over large, distributed knowledge bases. We also see a number of UI and HCI-related research challenges to be met including a higher-level representation of the language to make authoring easier and UI designs to help with rules authoring and presentation of dependency relationships. Finally, we will integrate policy aware capabilities into Semantic Web data browsers.

9 Acknowledgements

This work was sponsored by NSF Cybertrust Award #0524481 and DTO NICECAP Award #FA8750-07-2-0031. We would like to thank our colleagues Hal Abelson, Tim Berners-Lee, Jim Hendler, Deborah McGuinness, Oshani Seneviratne, Yosi Scharf, Gerry Sussman, and K. Krasnow Waterman for their participation in this project and their comments on the paper.

References

- [1] A. Kapadia and G. Sampemane and R. H. Campbell. Know why your access was denied: regulating feedback for usable security. In *11th ACM conference on Computer and Communications Security*, pages 52–61, 2004.
- [2] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference, W3C Recommendation. <http://www.w3.org/TR/owl-ref/>, February 2004.
- [3] D. Beckett. Turtle - Terse RDF Triple Language. <http://www.dajobe.org/2004/01/turtle/>.
- [4] P. A. Bonatti, D. Olmedilla, and J. Peer. Advanced Policy Explanations on the Web. In *European Conference on Artificial Intelligence (ECAI)*, 2006.
- [5] J. Bradshaw, A. Uszok, R. Jeffers, N. Suri, P. Hayes, M. Burstein, A. Acquisti, B. Benyo, M. Breedy, M. Carvalho, D. Diller, M. Johnson, S. Kulkarni, J. Lott, M. Sierhuis, and R. V. Hoof. Representation and reasoning about DAML-based policy and domain services in KAoS. In *2nd International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS2003)*, 2003.
- [6] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. <http://www.w3.org/TR/rdf-schema/>, February 2004.
- [7] H. Chalupsky and T. Russ. Whynot: Debugging failed queries in large knowledge bases. In *Fourteenth Innovative Applications of Artificial Intelligence Conference (IAAI-02)*, pages 870–877, 2002.
- [8] J. de Kleer, J. Doyle, J. Guy L. Steele, and G. J. Sussman. AMORD Explicit Control of Reasoning. *SIGPLAN Not.*, 12(8):116–125, 1977.
- [9] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, November 1979.
- [10] Friend of A Friend (FOAF). <http://xmlns.com/foaf/0.1/>.
- [11] F. H.R. Abductive reasoning as a way of worldmaking. *Foundations of Science*, 6:361–383(23), 2001.
- [12] A. C. Kakas, R. Miller, and F. Toni. An Argumentation Framework of Reasoning about Actions and Change. In *Logic Programming and Non-monotonic Reasoning*, pages 78–91, 1999.
- [13] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation 10 February 2004. <http://www.w3.org/TR/rdf-concepts/>, 2004.
- [14] Lalana Kagal and Tim Finin and Anupam Joshi. A Policy Based Approach to Security for the Semantic Web. In *2nd International Semantic Web Conference (ISWC2003)*, September 2003.
- [15] D. L. McGuinness and P. Pinheiro. Explaining Answers from the Semantic Web: the Inference Web Approach. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):397–413, October 2004.
- [16] Openid. <http://openid.net/>.
- [17] Tim Berners-Lee and Dan Connolly and Lalana Kagal and Jim Hendler and Yosi Scharf. N3Logic: A Logical Framework for the World Wide Web. *Journal of Theory and Practice of Logic Programming (TPLP), Special Issue on Logic Programming and the Web*, 2008.
- [18] Tim Berners-Lee and Y. Chen and L. Chilton and D. Connolly and R. Dhanaraj and J. Hollenbach and A. Lerer and D. Sheets. Tabulator: Exploring and Analyzing linked data on the Semantic Web. In *SWUI06 Workshop at ISWC06*, 2006.
- [19] D. A. Waterman and F. Hayes-Roth. *Pattern-Directed Inference Systems*. Academic Press, Inc., Orlando, FL, USA, 1978.