Modular Natural Language Interfaces to Logic-Based Policy Frameworks

Jason Perry[a,b,*], Konstantine Arkoudas[a], Jason Chiang[a], Ritu Chadha[a], Daniel Apgar[c], Keith Whittaker[c]

[a]*Applied Communication Sciences, Piscataway, NJ 08854, USA*
[b]*Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA*
[c]*U.S. Army CERDEC, Aberdeen, MD, USA*

Abstract

Natural-language interfaces to policy-based frameworks for access control can facilitate deeper policy understanding and support the analysis and maintenance of policies. We have developed a translation module that maps sentences of Attempto Controlled English to predicates of many-sorted first-order logic, which can be directly used as policy rules and conditions in a logic-based policy framework. This translation achieves broader semantic coverage than previous work that uses ACE, by means of a novel lambda-calculus-based mapping that applies modern principles of compositional semantics, and by a set of semantic assumptions relevant to the use of natural language in describing access control policies. We also demonstrate a form of question answering that is supported natively by this translation.

Our design also demonstrates a modular architecture for rapid development of natural language interfaces to new policy domains, enabled by the automatic generation of domain lexicons from logical signatures. We initially developed the translation module for a vocabulary in the cognitive radio domain, and subsequently generalized the system and applied it to additional domains. The module and framework inter-operate with policies written in the XACML rule language.

1. Introduction

The specification and analysis of access control policies is vital to the deployment of computer systems in virtually every setting. The development of frameworks for formally specifying and reasoning about policies is a fruitful research area that promises to contribute to the usability, security, and stability of many systems. Even where automated policy enforcement systems are in use, access control policies are generally specified in natural language first. Ideally, policies should continue to have a natural language representation in the system, in order to aid human understanding and facilitate the analysis of the policies. Natural language interfaces for policy frameworks achieve these goals by allowing input and analysis of policy components as natural language sentences within the system itself.

There is no "one-size-fits-all" solution for adding natural language interfaces to software systems. Natural language processing technologies are not yet (and may never be) advanced enough to provide a fully automatic conversion of policy rules to a formal representation that is guaranteed to be correct, for rules expressed in unrestricted English. However, the domain of access control policies is one in which current methods of computational linguistics can significantly aid the process of policy development and analysis. An automated machine translation of English policy rules into logic, even for a closed subset of English, can provide vital feedback in the cycle of refining both the natural-language and formal expressions of policies until each of them precisely expresses the desired conditions, and it can facilitate communication between policy advisors and technical policy analysts.

The work described in this paper presents a modular system and methodology that allows natural language modules with rigorously specified semantics to be quickly adapted to any policy domain. An example policy domain, which will provide us with examples for the remainder of the paper, is that of university policies for assigning and viewing grades. This domain is taken from work on the Margrave policy engine [14], and will be useful for drawing comparisons between our work and existing policy tools.

1.1. Policy Rules and Constraints

In this work we consider *access control policies* of the form specified in [2], in which policies are comprised of a *rule base*, containing a list of the policy's access rules in logical form, and a *constraint base*, listing the relevant environmental constraints. Policy requests are described in terms of an *Action*, that is, the operation being requested, a *Subject* (the entity placing the request), and an *Object*, to which the action will be applied. Thus the rules and environmental constraints are also described using entities of these types. We are primarily concerned with translating a natural language sentence to the logical form of either a rule or a constraint from an access control policy.

*Corresponding author
Email addresses: `jperry@appcomsci.com` (Jason Perry), `karkoudas@appcomsci.com` (Konstantine Arkoudas), `jchiang@appcomsci.com` (Jason Chiang), `rchadha@appcomsci.com` (Ritu Chadha), `daniel.d.apgar.civ@mail.mil` (Daniel Apgar), `keith.d.whittaker.civ@mail.mil` (Keith Whittaker)

As a first example, say that a policy analyst drafts the following policy rule:

> *If the requester is a student and the action is an assign and the grades are external then the request is denied.*

The purpose of this rule is to prevent students from assigning externally visible grades. From this sentence, our translation module produces the following predicate, which is an one-variable lambda abstraction of a first-order formula:

```
(lambda (r)
  (if (and  (isStudent (subject r))
            (= (action r) assign)
            (isExternal (object r)))
      (deny r)))
```

Notice that we use equality to test whether the action is equivalent to the first-order constant "assign", while properties of the subject and object are tested with predicates. This reflects a model of access control in which the requested action is of one of a fixed predetermined set, while the subject and object may have an arbitrary and expandable set of testable properties that determine access rights.

In addition to rules, a policy may contain *environmental constraints*. Unlike policy rules, these are not written as conditionals but as simple declarative sentences that specify restrictions. One example of such a constraint is:

> *No action is a view and an assign.*

This will be translated into the following expression:

```
(lambda (r)
  (not (and (= (action r) view)
            (= (action r) assign))))
```

The condition can be seen as asserting the disjointness of the interpretation of the first-order constants "view" and "assign". The translation is an example of correct handling of negation and coordination by our module.

1.2. System Architecture

The main operation of the translation module is to take an English statement of a policy rule or external condition, and translate it into a sentence of first-order logic. The major components of the system are shown in Figure 1. The system represents a "classical" NLP pipeline. The preprocessor corrects spelling errors and tokenizes the input for the parser. The parser converts the sentence into a tree structure according to a set of grammar rules. The semantic mapper associates nodes in the tree with lambda calculus expressions that represent partial meanings. The beta-reducer reduces the composite expression for one entire sentence, resulting in a formula of first-order logic that represents the meaning of the policy rule or condition and can be directly interpreted by the policy engine.

The lexical knowledge utilized by the system is comprised of two components: a *domain lexicon* used by the syntactic parser, which contains part-of-speech and morphological information, and a *semantic lexicon* that associates word forms with known logical predicates. This second lexicon fulfills the role of a first-order logic signature, in specifying the vocabulary of a first-order language. Both of these lexicons contain domain-independent and domain-specific terms, which can be specified independently.

The system is written in the Standard ML programming language, except for the ACE Parser which is written in Prolog and is called as an external program. Each of the components of the system will be discussed in detail in subsequent sections.

1.3. Related Work and Contributions

This work builds on prior research in the use of natural language interfaces for policy engines. A proposal for the architecture of a natural language input processing tool as a part of an interactive "policy workbench" is given in [20]. Besides identifying the key component technologies of such a tool, this work describes a prototype for generating draft policies from natural language sentences. However, this prototype is based more on knowledge extraction than precise sentence analysis.

One policy management tool that has work published on natural-language rule entry is IBM's SPARCLE [10]. Like the above work, this also uses shallow parsing primarily to identify rule elements in natural language sentences.

More recently, one prominent policy language, Protune [8], has had a natural-language interface developed [12] and applied to the domain of privacy policies in social platforms [13]. We follow this work in using Attempto Controlled English (ACE) [15] as our input language, with a custom vocabulary. The features of ACE will be described in detail in section 3. The approach of [12] is to take the logical output of ACE analysis tools and map it to rules in the Protune policy language. This mapping has significant limitations that the present work addresses.

One source of limitations in translation from natural language to policies is the formal representation of policy rules used in the policy framework. Protune uses logic programming as the basis for its rule format. Therefore, in translating from English to policy rules, there is a natural restriction to sentences that have a clear interpretation as Horn-clause forms. Specifically, the policy must have the form "If *condition*$_1$ and ... and *condition*$_n$ then *action*." Indeed, in the work in [12] only a subset of ACE that has a direct correspondence to rules of this form is translated.

Our work contributes significant advances to this state of the art. In the Athena Policy Engine [2], both policy rules and environmental constraints are expressed in full many-sorted first-order logic. First-order logic is a universal standard that is also highly human-readable. Other significant advantages given by FOL-based policy specification and reasoning are discussed in [2]. For the purposes of natural language interfaces, translating directly into FOL allows us to have a more flexible and yet principled interpretation of a larger subset of ACE. The only constraint on the form of policy rules is that they must be implications, while environmental constraints can be sentences of any form. The first-order logic representation also allows for treating negation more directly than in Prolog-style semantics. In sum, more English sentences can be understood "as is" as policy rules and constraints.

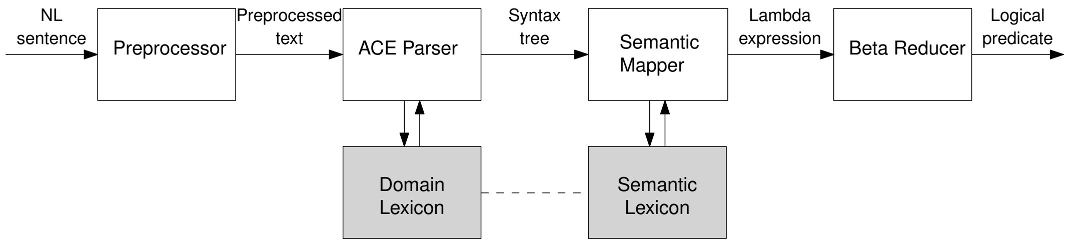


Figure 1: The translation module architecture.

To exploit the flexibility offered by first-order logic, we employ more sophisticated machinery for the translation. Rather than using the output of ACE semantic analysis tools (e.g., Discourse Representation Structures, SWRL), we develop a new mapping directly from ACE syntax trees to sorted first-order logic. Modern tools of computational semantics allow us to provide broader semantic coverage in a principled fashion. Specifically, using *continuation grammar* [5], we can declaratively specify any quantifier scope relations, where the existing ACE semantics only specifies one fixed scope interpretation for quantified sentences – namely, that the scope of a quantifier begins at its point of introduction and extends to the end of the coordinated sentence. Using the same approach, we also handle coordinated noun phrases, which is one example of a syntactic structure not supported in the prior Protune-based work [12]. To our knowledge, this is the first work to develop a new semantic mapping directly from ACE syntax, as opposed to transforming the output produced by ACE's semantic tools to suit some specific application.

As an additional contribution, we demonstrate how to make such a translation component general, modular and easily adaptable to new domains. We show how natural-language translators for new policy domains can be generated semi-automatically from the domain specifications used in the logical policy framework.

2. Typed Logic for Policy Domains

The flexible policy framework described in [2, 3, 4] is built on top of Athena [1], a logical framework that serves both as a functional programming language and as a theorem-proving system, using many-sorted first-order logic[19] as its core representation. Many-sorted first-order logic can be thought of as first-order logic with types. Athena's native support for manipulating sentences of this logic make it especially suited for defining policy domains, formulating policies, and reasoning about them. The policy engine built in Athena interfaces with SMT solvers.

Satisfiability modulo theories (SMT) [22] is a recent technology that can be viewed as a generalization of propositional satisfiability. An input to an SMT solver is a quantifier-free[1] formula with various interpreted and uninterpreted function symbols. The interpreted atoms of the formula come from background theories such as linear (integer and rational) arithmetic, inductive data types, uninterpreted functions with equality, the theory of lists, extensional arrays, fixed-size bit vectors, etc. The satisfiability of an input formula p is determined by these background theories along with the Boolean structure of p. An SMT solver will determine whether p is satisfiable, and if it is, it will also provide a satisfying model. Thanks to their ability to "understand" useful background theories, SMT solvers are enabling a wide array of applications that would have been either impossible or impractical with previous reasoning technologies [22]. The prior work [2], [4] describes how the logical policy representation allows such diverse reasoning tasks as request evaluation, consistency checking, and change impact analysis to be directly formulated as SMT problems.

The three types Subject, Object and Action described in section 1.1, as well as the type of Requests themselves, are declared as domains in Athena syntax as follows:

```
domains Request, Subject, Object, Action;
```

Entities from the Subject, Object and Action domains are considered as attributes of a Request entity, and they can be retrieved through the following accessor functions, which are specified for all policy vocabularies:

```
declare subject: [Request] -> Subject;
declare object : [Request] -> Object;
declare action : [Request] -> Action;
```

The square brackets are Athena's syntax for indicating argument types.

Both rules and constraints are specified as lambda abstractions of one variable, that is, as unary predicates. This variable is always of type `Request`, indicating that the topic of each policy rule and constraint is the requested action. Applying such a predicate to a logic variable of type Request produces a first-order sentence. This operation forms the basis of evaluating requests. This representation scheme also admits a straightforward translation of XACML-style policy rules to sorted first-order logic.

Now we consider the specific declarations that define a policy management system for a particular domain. For any given policy domain, the types Subject, Object, and Action refer to categories specific to that domain—for example, in the grading domain the Objects are the grades to be assigned. These can be represented by declaring type aliases for Subject, Object, and Action:

[1]Some SMT solvers accept quantified sentences, but these are typically removed using various heuristics [16].

```
define Requester  := Subject
define Grades  := Object
define Action := Action
```

(The name "Action" is suitable enough for this domain as it is.)

In this example, entities of the "Action" domain are the only ones referred to explicitly, and so we declare named constants for these:

```
declare assign, view, receive : Action
```

The properties of entities of the Subject and Object types can be tested by means of recognizer predicates such as the following:

```
declare isStudent : [Subject] -> Boolean
declare isFaculty : [Subject] -> Boolean
declare isInternal : [Object] -> Boolean
declare isExternal : [Object] -> Boolean
```

So an expression to state that the requester of an action is a student would look like this:

```
(lambda (r) (isStudent (subject r)))
```

These type declarations describe the domain-specific portion of the signature of the first-order language, and thus they essentially comprise the logical lexicon portion of the natural-language translation module. As we will see, the sorted logic that gives structure to policy formulas also aids in translating English sentences to policy rules. The types help in resolving ambiguities and inferring information that is specified only implicitly in the natural language form of a policy rule.

In the following sections we describe the algorithms and technologies that our system uses to translate an English sentence into a first-order policy rule or environmental constraint that is consistent with these declarations.

3. Attempto Controlled English for Syntactic Parsing

Attempto Controlled English, or ACE [15], is described as "a formal language with English syntax." Its specification consists of an unambiguous grammar for a subset of English, and is integrated with a set of tools translating ACE sentences to various semantic forms. The Attempto Parsing Engine (APE) is a parser and set of semantic analysis modules conforming to the ACE specification.

The grammar used for parsing ACE contains 229 context-free production rules augmented with binary features to indicate grammatical number, gender, etc. ACE specifies a mapping of controlled English sentence constituents to well-known categories of phrase structure grammar, e.g., sentence (*S*), noun phrase (*NP*), verb phrase (*VP*). ACE handles declarative, imperative, and question sentences and auxiliary and modal verbs, but only present tense. The category of declarative sentences is named "specification." Conditional ("if-then") sentences are treated specially as their own subcategory of specifications, which is especially suitable to our purposes, since policy rules are expressed as conditionals. Figure 2 presents an example ACE syntax tree for the sentence "If the requester-type is Faculty and the action is Assign then the request is allowed."

ACE sentences can be read and understood by anyone who knows English. In order to write ACE sentences reliably, a small amount of training is necessary to navigate the syntactic restrictions that ACE imposes on English. For example, conjunctive coordination of adjective phrases or noun phrases is allowed, but disjunctive coordination is not; disjunctive coordination must be done at the VP or sentence leve. For example, if a policy author wishes to say "The grades are internal or external", it is necessary to write "The grades are internal or are external". This is to prevent ambiguity in the grouping of AND and OR, as it allows us to assume that OR always has lower precedence than AND.

The APE parser defaults to a closed lexicon but will optionally guess the part of speech of unknown words. APE allows user-defined lexicons that can augment the built-in lexicon or replace it entirely. Terms are added to a lexicon by writing a Prolog fact specifying the word, its base form and part of speech, along with any grammatical features required for the part of speech, such as number or gender. This collection of such rules corresponds to the box labeled "domain lexicon" in Figure 1. Our system works by augmenting the built-in lexicon with both domain-independent policy terms and domain-specific vocabulary.

We also use domain-specific lexicons to build in additional flexibility of expression without affecting the grammar directly, including common synonyms and misspellings. The preprocessor is also used to slightly loosen ACE's syntactic restrictions when we feel there is no danger of affecting the semantics; for instance, we allow optional commas in lists and between clauses by removing them in the preprocessing stage.

The APE parser for ACE is utilized in our system as an external program module. To parse an input sentence, we first tokenize and preprocess it, call the APE parser specifying the custom lexicon, and then read the parse tree into a data structure.

4. Semantic Analysis

Syntactic parsing is usually considered part of the "front end" of natural language analysis. Before the computer can process the content of the sentence in a meaningful way, a *semantic analysis* must be performed on the output of the parser. The remainder of this paper will be concerned with semantic analysis.

The ACE suite of tools provides its own set of semantic analysis tools that can translate the parse tree of an ACE sentence directly into first-order logic in the form of Discourse Representation Structures (DRS) [17]. However, our system does not make use of these; of the components provided in the ACE toolkit, we only use the syntactic parser, APE, and have developed our own semantic analysis code from scratch. Here we discuss our reasons for doing so.

The primary reason is that the most general possible translation of a sentence into logic is not necessarily the one best suited to any given application domain. The translation provided by the ACE tools cannot take advantage of any domain-specific

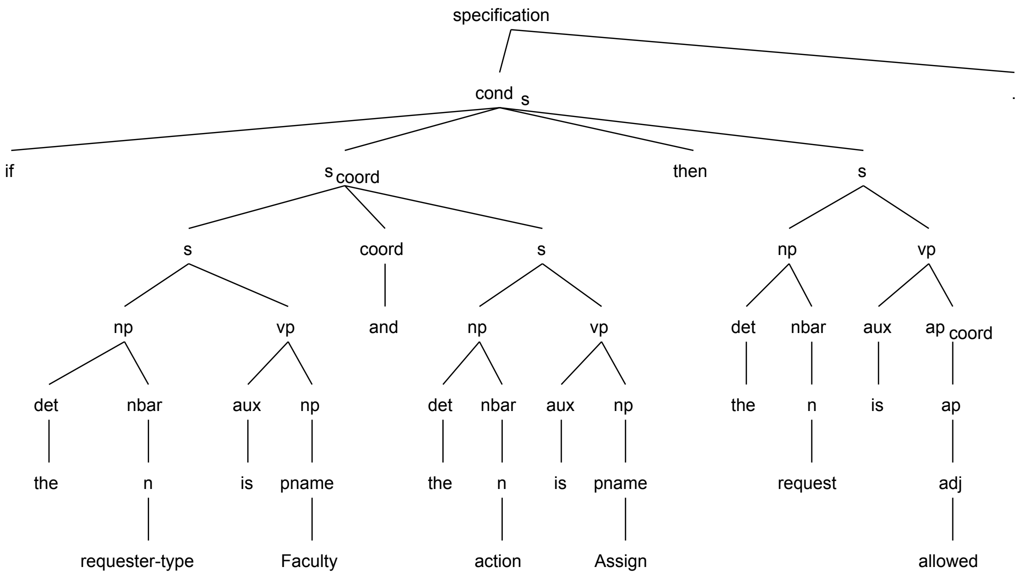


Figure 2: A sample ACE parse tree.

presuppositions in order to generate a more precise meaning. In the case of policy engines, the form of policy rules and the environments in which they operate are highly constrained. Since we know we are dealing with sentences that represent policy rules or constraints specified within a given framework, we can often infer more meaning from the syntax than the ACE tools can.

For example, APE can syntactically parse the sentence “The grades are external”, but if we try to obtain a FOL or DRS representation, the analysis module complains that the definite noun phrase “the grades” has no antecedent. However, we know that in this case “grades” must refer to an attribute of the requests that are to be tested by this policy rule. More specifically, our domain-specific lexicon tells us that “the grades” refers to the *object* of the request with which the sentence is concerned. So we can directly generate the concise logical form `(lambda (r) (isExternal (object r)))`. For this reason, we may in principle obtain broader interpretive coverage than a system that uses the output of APE’s semantic analyzers.

A second reason for developing our own semantic analysis is that the logical representations generated by APE contain unnecessary details that are not relevant to the policy domain—for instance, the analyzer represents the distinction between count and mass nouns explicitly in the logical output. This is not a shortcoming of ACE itself, but arises because the APE parser, being totally general, cannot know in advance which syntactic features are relevant for the desired semantics. So at any rate, we would not be able to use the output of APE’s semantic analyzers unmodified.

Other work has reflected the reality that the output of ACE tools often requires further processing for use in any given application. When Bernstein et al. [6] provide an ACE-based natural language query interface for ontologies, they take the DRS output of APE and use rewrite rules to map the discourse structures to the PQL query language. In our case, developing our own semantic mapping from scratch allows us to apply the well-studied principles of formal compositional semantics starting from Montague [21] more directly. We can even depart from the semantics specified by ACE when it is not a good match for the application. One instance of this relates to pronoun resolution. As discussed below, we found APE’s pronoun resolution algorithm a poor fit to sentences describing policy rules. Overall, our solution to the semantic translation problem is quite robust within the context of policy frameworks.

4.1. Lambda calculus-based mapping of syntax trees

The mapping of syntax trees to logical forms is done according to the method pioneered by Richard Montague [21], which we will proceed to describe. We do not utilize the full equipment of Montague’s intensional logic, but take an approach based on first-order logic similar to that described in Blackburn and Bos’s computational semantics textbook [7].

In Montague semantics, the sentence constituents found in the parse tree, such as noun phrases and verb phrases, are considered to have lambda calculus expressions for their denotations. To derive the interpretation of a sentence from its parse tree, nodes in the tree are mapped to lambda calculus formulae, and then these “semantic attachments” are propagated up the tree by means of function application. Finally, the root of the tree will be associated with a single lambda expression. If the mapping specified is consistent, beta-reducing this expression results in a closed first-order formula (excepting, in our implementation, a single outermost lambda abstraction) that represents the meaning of the sentence.

In the case of a parse tree derived using a context-free grammar, it is often practical to associate a combination rule with each syntax production in the grammar. This allows us to use combination rules beyond just left-to-right function application.

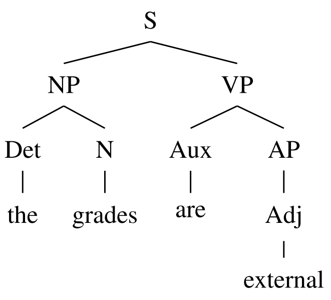
For example, we can associate the operation of function application with the grammar production for predication (the composition of a noun phrase and verb phrase to form a sentence), and we write the production followed by its composition rule in braces:

$$S \rightarrow NP\ VP \ \ \{(VP\ NP)\} \tag{1}$$

In writing the combination rule, the nonterminal symbols are used to refer to their associated lambda expressions (derived lower in the tree.) So to construct the meaning of a sentence consisting of an *NP* and a *VP*, we apply the noun phrase as an argument to the verb phrase.

Montague's truth-conditional semantics specifies a set of types for the denotations of sentence constituents. The type e represents entities, and the type t stands for truth values. These types constrain the ways constituents can be combined, so that only constituents with well-defined meanings can be constructed. Valid declarative sentences are those where the applications result in an expression of type t. In the above example, if the type of a noun phrase denotation is e, then the type of a verb phrase denotation can be $e \rightarrow t$, and the result of the application $(VP\ NP)$ is indeed an expression of type t. These types are not represented explicitly in the system but provide the principles for constructing lambda expressions for the grammar rules.

As an example of the mapping process, note the syntax tree of the sentence "the grades are external":



For purposes of clarity, unary syntax productions that carry no semantic content in this translation have been omitted. Here are the semantic attachments for the applicable syntax productions:

S → NP VP	$\lambda r.$ (NP VP)
NP → Det N	(Det N)
N → "grades"	(object r)
Det → "The"	$\lambda n.\lambda c.(c\ n)$
VP → Aux AP	(Aux AP)
Aux → "are"	$\lambda o.\lambda s.(o\ s)$
AP → Adj	$\lambda w.$ (Adj w)
Adj → "external"	isExternal

These rules show that predication, modification by adjectives and auxiliary verbs, and complementation (pairing of verbs with their grammatical objects) are basically all done by function application. Also notice that the auxiliary "are", being a transitive verb, takes two arguments, the first representing the object of the sentence and the second representing the subject. The domain-specific part of this mapping is to determine that the adjective "external" has the denotation of a predicate named "isExternal", and that "grades" refers to the Object attribute (not grammatical object) of the request variable. The variable r is reserved for the request entity, which is always bound in the outermost lambda abstraction of the sentence.

The form of the expression for the determiner "the" requires some explanation. For now, note that it has no semantic effect on the noun it modifies, and only acts to place its first argument in the argument position of an application, and its second argument in the operator position. This is necessary because the expression for S → NP VP applies the NP to the VP, rather than the other way around as one might expect. This allows us to model the sophisticated semantic effects of determiners such as quantifiers and negatives, as described below in the discussion of quantification.

The un-reduced lambda form for the entire sentence is

$$(\lambda r.((\lambda n.\lambda c.(c\ n)\ (\text{object}\ r))\\ (\lambda o.\lambda s.(o\ s)\ \lambda w.(\text{isExternal}\ w))))$$

which reduces, as we hoped, to

$$\lambda r.(\text{isExternal}\ (\text{object}\ r)).$$

Our mapping does not cover the entire range of ACE Syntax, but it covers all major forms of declarative sentences and includes possessives, prepositional phrases, and coordination, providing significant flexibility in expressing policy rules and constraints. Our mapping also covers numerical relations, for policies involving numerical reasoning. (The cognitive radio domain, described in [3], is notable for its extensive use of numerical attributes.) Approximately 90 mapping rules are described at present, several of which cover multiple syntax productions due to pattern-matching. If the user enters a sentence that is syntactically parseable by APE but that the semantic module does not know how to interpret, the analysis module will produce an informative error message.

We now discuss the more advanced issues addressed in the lambda-calculus based semantic analysis.

The role of types. As mentioned above, in the policy framework, the entities of concern are divided into types Request, Subject, Object, and Action. We do not incorporate these types into the lambda calculus itself; to do so would make it more difficult to combine expressions flexibly (basically, we would have to implement a polymorphic calculus.) Instead, we implement a non-formal method for deriving type information for domain-specific words, using the logical lexicon. The logical lexicon is simply a mapping of words to the predicate name or formula that each denotes, plus a grouping of these predicates into the four types. The semantic analysis module contains a procedure for determining the type of the entity represented by an NP "after-the-fact", by analyzing the structure of the NP and performing a lookup in the logical lexicon.

The type information is used in the translation module is as an additional source of information for distinguishing between identical syntactic forms that require different logical representations in the policy engine. For instance, this is needed for the mapping of the grammar rule *VP → Aux NP*, where the Auxiliary verb is "is" or an equivalent. As stated in section

1.1, in the policy engine, the members of type Action, such as "read" or "assign" in the grading domain, are named explicitly by first-order constants, while members of the Subject and Object types typically have their properties tested by predicates, such as "isFaculty()" or "isExternal()".[2] Therefore "The action is Assign" should have a logical form using the equality relation, `(= assign (action r))`, while "The requester is Faculty" should be translated to `(isFaculty (subject r))`. This distinction cannot be made from syntax alone, but is easily handled when we look up the types corresponding to the nouns "assign" and "faculty" in the logical lexicon. The auxiliary verb also generates the equality relation in the case when numeric attributes are used.

Quantification, Coordination, Negation. In the above discussion of the types used in Monatgue grammar for semantic composition, noun phrases were considered to have type e, for 'entity'. In fact, in Montague's system and ours as well, the type of NP denotations is not e but $(e \to t) \to t$. This is an instance of *type-raising* that allows us to correctly handle noun phrases containing quantifiers, as described in Montague's seminal 'PTQ' work [21]. We generalize the lambda calculus denotations of all NPs to this form, having them take an argument, so that both quantified and non-quantified NPs can be treated uniformly. This type-raising technique is developed futher, for all constituent types and not just NPs, and given a principled analysis with new applications in Barker and Shan's work on continuation grammar [5], [23].

For policy rules and conditions, only universally quantified sentences are allowed, and in fact the first-order quantifier is not inserted when translating an English expression with quantifiers. For example, "Every request whose requester is Faculty is allowed" is translated as

```
(lambda (r)
  (if (isFaculty (subject r))
      (allow r)))
```

This makes sense because in the policy engine, sentences are never interpreted over the entire universe of requests but are predicates that are applied to individual requests.

Though the conditional expression is generated without the first-order quantifier, this still requires us to model the wide scope-taking behavior of quantifiers, since the conditional must take scope over the whole sentence. The ability of a constituent to take wide scope is modeled in the lambda calculus rules by expressions that take their own *continuation* as an argument. In the field of functional programming languages, a continuation is the "default future" of a computation, represented as a function, allowing control flow to be manipulated by means of calling the continuation. In semantic interpretation, the continuation is the function that accomplishes the interpretation of the remainder of the sentence. Providing a sentence constituent with access to its own continuation allows that constituent to take wide scope over the remainder of the sentence when needed. A *continuation grammar* is one in which the lambda calculus expressions of the semantic attachments specify such access.

This accounts for the form of the the determiner "the" shown above. Here we show the semantic attachment (in braces) for another determiner, "every":

$$\text{Det} \to \text{“every”} \;\; \{\lambda n.\lambda c.n \to c\} \tag{2}$$

An NP containing this quantifier takes wide scope over the VP by taking the VP as an argument, here shown by c. This necessitates reversing the order of application from that shown in section 4.1; to compose a sentence, the NP must be applied to the VP. For this reason the expression for *all* NPs, not just those representing quantifiers, must be altered to take its continuation as an argument. For NPs that represent simple entities (such as proper nouns), the denotation is written $\lambda c.(c \text{ noun})$. This behavior can be seen as the NP taking the VP as an argument and then, in turn, placing itself in the argument position of the VP.

The mapping for "every" also shows that it is not necessary to write a conditional sentence in English to specify a policy rule, even though the logical form of a policy rule must indeed be a conditional. In many cases it may be more concise and natural to specify a rule as a universally quantified sentence. But we could just as well have written, "If the requester is Faculty, the request is allowed." and it would be translated identically. This is an instance of the grammatical flexibility that we gain by handling quantifiers in a principled fashion. Further examples can be seen in the policy translation in the Appendix.

Coordination is another language phenomenon that causes scope displacement. In English, coordination can occur between almost any type of constituent; in particular, we can coordinate NPs as well as VPs. Recall the example sentence "No action is a view and an assign." It was written as a NP coordination, but it could equivalently be written as a VP coordination, "No action is a view and is an assign." Both of these forms should be translated into the same logical form. To see how this is handled, note the lambda calculus denotation of an NP "and"-coordination:

$$\text{NP}_{\text{coord}} \to \text{NP}_1 \text{ “and” } \text{NP}_2 \;\; \{\lambda c.(\text{and } (\text{NP}_1\ c)\ (\text{NP}_2\ c))\} \tag{3}$$

By the use of continuations, wherein the NP is a function that receives its context as an argument, the NP can distribute its coordination across its context. In the case that the NP is in subject position, we can simply apply it to the VP as shown above. However, to allow an *object* NP to take scope over the the entire sentence, as in "No action is a view and an assign," we need to modify the attachment describing application of a (grammatical) object NP to a transitive verb as follows:

$$\text{VP} \to \text{VT NP} \;\; \{\lambda c.(\text{NP } \lambda n.(\text{VT } n)\ c)\} \tag{4}$$

Generating an outer abstraction over c, where c is applied to the object NP, is what allows the object NP to take scope over the whole sentence. If the NP does not take wide scope by manipulating its continuation, this form is equivalent to applying the verb to an object NP representing an entity. This form is used even when the VT is an auxiliary verb, as in the above example.

[2]Note that the types "Subject" and "Object" used in the policy engine do *not* correspond to grammatical subjects and objects.

In addition to an NP coordination, this sentence also contains a quantifier, the negative determiner "No", representing universal quantification over a negated sentence. Negations can also take wide scope, and so we have the following continuized form for the negative determiner:

$$\text{Det} \rightarrow \text{"no"} \quad \{\lambda n.\lambda c.(\text{not } c\ n)\} \tag{5}$$

In this case we also need to ensure that the negative takes wide scope over the conjunction. But this is already taken care of because the subject NP is given the place that can take widest scope. Applying the reduced denotation of the NP to the reduced VP, the sentence is:

$$\lambda r.(\lambda v.(\text{not } v\ (\text{action } r))\ \ \lambda c.(\text{and } (=\ c \text{ view})\ (=\ c \text{ assign}))),$$

which reduces to

$$\lambda r.(\text{not } (\text{and } (=\ (\text{action } r) \text{ view})\ \ (=\ (\text{action } r) \text{ assign}))),$$

which is equivalent to the form shown previously in code. These forms embody the rule of surface scope for quantifier ambiguity resolution.

4.2. Pronoun Resolution

An important semantic problem not addressed by the the lambda calculus-based mapping is pronoun and anaphora resolution, which is a long-studied and difficult research problem in computational linguistics. Our system needs to handle basic pronoun usage in order to specify policy rules in natural-sounding English, for example "If a transmission's power is above 900 then it is denied." The most prevalent NLP solutions to anaphora resolution are computationally and representationally complex. To utilize such a solution would be overkill for a domain where we are primarily dealing with single sentences, and it could not be implemented in a short time.

The ACE semantic analysis tools make use of a basic algorithm for resolving pronouns, which is used in ACE's mapping to first-order logic and DRS. Even though we are only using the ACE tools for syntactic parsing, it is still possible to take advantage of the resolution capability by means of the software's "paraphrase" feature. The paraphrase module of the ACE tools takes an input sentence and generates a new English sentence or sentences with anaphora resolved by means of variables. The generated paraphrases make use of a smaller range of syntactic constructions than the parser can recognize, which gives the potential for simplifying the problem of mapping syntax trees to logical forms. As an example, if we input the sentence (with no pronouns)

> *If the requester of the request is Faculty then the request is allowed.*

Then we will obtain the paraphrase

> *If a requester of a request X1 is Faculty then the request X1 is allowed.*

The first thing we may notice is that the parser has changed "the requester" to "a requester" and the first "the request" to "a request". When the parser encounters a definite article that does not refer to an explicit antecedent, rather than assume the existence of a unique entity (which in general text is too strong a presupposition) it simply substitutes the indefinite. This is not a concern to us, since in our translation to policy rules both the definite and indefinite articles map to the identity function.

Returning to the issue at hand, we see that the paraphrase operation has assigned variable *X1* to the request entity and given the second mention of the request the same referring expression. These variables are a part of the ACE grammar, and so this paraphrase can then itself be parsed into a syntax tree that contains the variables. Then we can use the ACE variables directly in the logical form that we generate, and so we automatically have a form with correct variable references.

However, we cannot rely on the ACE paraphraser to generate variables for us in every case. No variable is generated when there is only a single reference:

> *If the requester is a student then the request is denied.*
>
> *If a requester is a student then a request is denied.*

Since the logical form of a sentence always requires at least one variable, relying on ACE variables may make it more difficult to use a uniform approach to generating logical forms from syntax trees. Moreover, the pronoun resolution algorithm used by the ACE tools is a poor fit for this problem domain; it always takes the most recent noun as the antecedent. The following example paraphrase shows how this is often inappropriate for policy rules:

> *If the requester of the request is a faculty then it is allowed.*
>
> *If a requester of a request is a faculty X1 then the faculty X1 is allowed.*

The pronoun "it" is here clearly meant to refer to the request itself, but the ACE tools resolve it to the subject of the request, i.e. "the faculty". For these reasons, the solution of relying on the variables generated by these paraphrases to resolve anaphora was dropped. However, rewriting methods are well attested in NLP applications such as machine translation, and the examples we have shown give some flavor of its potential. In particular, this paraphrase rewrite is "safe" for ACE toolkit's default interpretation, because it preserves the semantics of the sentence as defined by ACE.

A sufficient solution to resolving anaphora in policy rules and conditions turned out to be simpler than expected. In the realm of access-control policies, the sentences we deal with often have a "fixed referent" flavor, in which the topic of every sentence is the action request to be evaluated. In general, the request is the only entity in a rule that is referred to more than once. So we assume that any impersonal pronouns and uses of the word "request" (or one of its synonyms) always refer to one and the same request, namely the referred to by the lambda variable in the logical form. So, for the above example, in our system it is sufficient to input *"If the requester of the request is Faculty then the request is allowed."*

Of course, "requester of the request" is still awkward. But the "fixed referent" property also allows us to interpret sentences in which the reference to the request is left implicit. So we can even enter the sentence *"If the requester is Faculty then it is allowed."* and the translation module will produce what is presumably the intended logical form:

```
(lambda (r)
  (if (isFaculty (subject r))
      (allow r)))
```

Note that it is still possible to specify environmental constraints which do not refer to the request variable at all. For example, the constraint "A view is not an assign" is translated to the expression:

```
(lambda (r)
  (not (= view assign))),
```

which is correct within the logical framework's representation.

5. Generating Domain-Specific Lexicons from Type Declarations

As described above, the domain-specific portion of the translation module is comprised of two components: the syntactic lexicon, which gives part-of-speech and other grammatical features of words, and the logical lexicon, which gives the name of the logical predicate or constant to be used for a term, as well as the type that an entity described by that term would have.

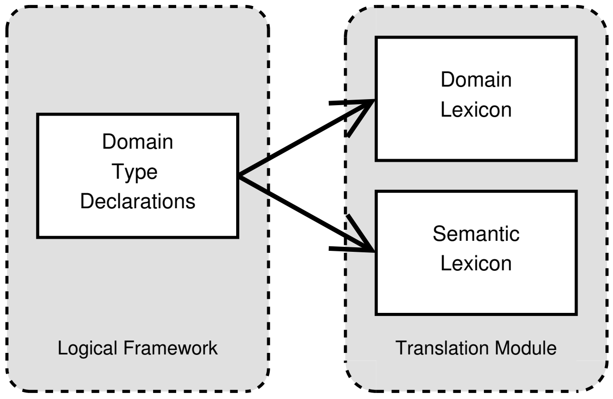


Figure 3: Syntactic and semantic lexicons derived from type declarations.

The amount of lexical knowledge that needs to be produced to adapt the system to a new domain is relatively small. The process of doing so can be streamlined even further, by generating both lexical components automatically from a single source of information—the domain type declarations used in the policy framework (see Figure 3.) As stated in section 2, the type declarations of the policy domain are in fact a logical-form lexicon, or signature. These also often contain sufficient information to provide "sensible default" word forms and parts of speech for the syntactic lexicon.

As shown in section 2, there are three categories of type definitions and declarations that must be specified in the policy framework for each domain. We call these *Type Aliases*, *Action Types*, and *Property Testers*. For each of these kinds of declaration, there is a uniform mapping to entities in the syntactic and logical lexicons.

The domain-specific Type Aliases for Request, Subject, Object, and Action are declared as nouns in the syntactic lexicon, and in the logical lexicon, accessor functions of the corresponding names can be inserted. For example, "requester" is logically "`(subject r)`".

The Action Types are inserted in the syntactic lexicon as nouns, and are mapped to their identical names in the logical lexicon, as they correspond to entity types.

The Property Tester predicates, which are all named beginning with "is", e.g., "isStudent" or "isTA", are inserted in the syntactic lexicon as nouns by removing the "is" prefix. The logical lexicon maps this noun back to the "is" predicate name.

Table 1 shows the correspondence between sample type declarations in the policy engine, the syntactic lexical entries, and the logical lexicon's mappings.

Thus a simple script can generate "draft" lexicons from the type information. We say "draft" because of course, the language produced by this process is not always completely natural. The CONTINUE policy has a predicate named "hasSubmittedReviewForResPaper". Ideally, the parser would recognize a phrase of the form "The reviewer has submitted a review for the paper" as an instance of this predicate, but this would require modifying not only the lexicon but the grammar itself. In most cases, however, the work required to adapt the lexicon to a new domain is much simpler—for example, adding synonyms and plural forms. The automatic generation of the draft lexicon can be seen as providing a starting point for this work, from which the lexicon can be further tailored.

6. Natural Language Translation for Question Answering

This translation approach, coupled with the SMT-solving policy framework, can also provide a straightforward implementation of natural language question answering regarding policies. This is an example of a closed-domain question answering system in the lineage of the classic LUNAR system [24].

In the policy framework, a request is considered "complete" if it specifies constraints for the Subject, Object, and Action. A question can then be seen as an *incomplete* request, one with missing constraints. Question sentences translated into a logical form can be answered simply by calling the solver or theorem prover on that form, so that the prover generates satisfying values for the missing constraints.

The ACE parser already supports several types of question syntax, including *Wh*-questions. Consider a *Wh*-question such as the following: "Which requests whose requester is Faculty and whose action is View are allowed?" The desired answer to such a question is a set of objects that satisfy property constraints, with the "whose" clauses serving to specify those constraints. Our software translates such a sentence into the logical form of a conjunction of the constraints:

Athena Declaration	ACE Lexicon Entry	Logical Lexicon Entry
`define Requester := Subject`	`n_sg(requester, requester, neutr)`	`("requester", (subject r))`
`declare assign : Action`	`n_sg(assign, assign, neutr)`	`("assign", assign)`
`declare isFaculty : [Subject] -> Boolean`	`n_sg(faculty, faculty, neutr)`	`("faculty", isFaculty)`

Table 1: Sample mappings from logical type declarations to lexicon entries.

```
(lambda (r)
  (and (and (isFaculty (subject r))
            (= (action r) view))
       (allow r)))
```

The same question type can be posed using "is-there" syntax. For example, the sentence "Is there a request that is allowed and whose requester is Faculty and whose action is View?" will be translated into the same logical form.

The answers to these questions are then obtained by applying the predicate (rule procedure) to a single logic variable, producing a first-order formula, and calling the SMT solver on the formula, with the solver configured to produce all satisfying assignments. The solver's output will consist of attribute values for which the formula is satisfied–in the case of policies, values for which an instantiated access request would be accepted. For the above example, if Faculty members are allowed to view both internal grades and external grades, the solver would return two request objects, each of which has all the attributes as given in the question, and one with the missing "grades" constraint as `(isExternal (object r))`, the other with `(isInternal (object r))`.

This formulation of question answering only allows us to handle queries where the topic is a request. However, this is the key type of question that must be handled in order to implement a system of *negotiated permissions* such as that described in Bonatti et al. [9], in which requesters seek to determine a set of parameters that would cause their requests to be accepted.

7. Evaluation

To evaluate the flexibility of our translator and ease of adaptation to new domains, separate versions of the translator were instantiated for three distinct policy domains, each of which had previously developed policy rules and conditions. These are: the grading domain previously mentioned, conference management as modeled in the CONTINUE policy described in [14], and dynamic spectrum access for cognitive radios, described in [2, 4].

The following are the steps we followed in building each new translator, which we consider a "natural" workflow for porting the translation module to a new domain:

- Specify the domain type signature for use by the policy framework.
- Use a script to automatically generate the initial domain-specific lexicons from the type signature.
- Write the rules in natural language, and attempt to parse them with the new translation module.
- Refine the lexicon entries by hand so that the rules can be expressed in more natural vocabulary.
- Test the semantic correctness of the translated formal representations of the rules and conditions.

Developing the domain-specific lexicon required a very small amount of work in each case as only the lexical resources for a small vocabulary had to be adapted, as described in section 5. For the three policy domains, we verified that the logical forms produced were accepted as correctly typed predicates in the policy engine and were logically equivalent to the original policy rule. For the CONTINUE policy set, we developed a XACML-to-FOL translator in order to compare the FOL output of the translator to the original XACML form of the policy.

The complete set of rules and constraints for the grading domain is given in the Appendix.

8. Discussion

The advances in this work can be seen to derive from two key observations. The first is the flexibility and generality of the lambda calculus and first-order logic. Lambda-calculus-based compositional analysis has been a mainstay of computational semantics [7]. It provides high flexibility plus straightforward integration with syntactic parsing. First-order logic remains one of the most versatile knowledge representations for natural language, and gains clarity by use of types. The advent of efficient SMT solvers has made reasoning over first-order representations a much more practical possibility when the domain is finite or has a decision procedure.

The only danger in using the untyped lambda calculus lies in its high flexibility. Without a disciplined approach to meaning composition, it is easy to become mired in "grammar hacking" when one tries to expand the coverage beyond trivial sentences, and end up with a solution that does not generalize cleanly. Continuation grammar [5], [23] provides exactly such a discipline. To our knowledge, ours is the first work in the area of policy frameworks to use the continuation semantics paradigm to handle more sophisticated scoping phenomena for broader semantic coverage.

The second observation is that the implementation of semantic theories in natural-language interfaces can and should make assumptions relevant to the application domain. The most obvious employment of this principle is in simplifying the task of generating domain-specific lexicons; by constraining the meanings of words to a predefined context, we help bring lexical ambiguity under control. Also, the "fixed referent" feature of policy rules is what made coreference problems satisfactorily solvable, and in fact gave higher precision for this application than a technology that attempts to resolve anaphora in a general setting. Though work in solving these "deep semantic" problems in uncontrolled text is worthwhile in its own right, we believe that for the foreseeable future, solutions in deployed systems must be developed using an application-informed semantic theory, rather than in full generality. As we have seen, this does not in any way prohibit the use of generalized tools and representations, such as the lambda calculus and first-order logic.

9. Future Work and Conclusion

One obvious expansion of this work is to implement generation, that is, translating logical-form rules and constraints back to natural language sentences. However, this feature is not immediately urgent in the current application, since once a policy rule, condition, or query is given as input in natural language, this natural language sentence can be stored along with the logical representation and presented again in subsequent human interaction with the system. A potentially enlightening user study would be to see whether the variations produced by regenerating natural language from logical forms could be used to help analysts clarify the meaning of policy rules.

We plan to implement a more advanced question-answering interface, while continuing to expand the grammar coverage. A method for generating explanations of denied requests based on outputting policy rules is planned.

We have considered developing a graphical tool that makes it easier to input grammatically correct ACE sentences. The study documented in [18] confirms that users are able to author rules of significantly higher quality when using a UI with assistance than when entering natural language unguided. One such tool already existing for ACE is the predictive editor included with the AceWiki system [12]. (The Protune system also includes a command-line interface for entering ACE sentences.) We have found the predictive editor somewhat unwieldy to use because it involves choosing words in sequence from long lists of possible candidates. If the wrong function word is chosen, the user gets "stuck" and must back up to try different structures. A more advanced predictive editor is stated to be in development by the authors of [12]. In our perception, an optimal solution would be an auto-corrector or a tool that offers "advice" while typing, though admittedly such a system would be much more difficult to implement.

Though we used controlled natural language for this project, it is not strictly necessary to do so. The semantic mapping described here could be applied just as readily to the output of a wide-coverage statistical parser such as the Collins PCFG parser [11]. In this case, since the parse trees produced by such a parser do not follow a concise grammar, a mapping would be developed to handle the subset of parse trees most relevant to specifying knowledge in the realm of access policies, and generate an error message or request for clarification otherwise.

Natural language interfaces for policy frameworks facilitate better policy authoring, transparency, and maintenance. We have shown how to implement a semantically principled translator with an existing controlled-language parser that can quickly be adapted to new domains. It is our hope that this work will contribute to the adoption of modular architectures for adding natural-language interfaces to policy frameworks.

Appendix A. Complete Translation of the *Grades* Policy

To show the flexibility of the translation, multiple English sentences are given for some rules. The rules are labeled by the names given in [14].

Appendix A.1. Policy Rules

Pr1: "If the requester is a student and the action is an assign and the grades are external then the request is denied."

```
(lambda (r)
  (if (and (isStudent (subject r))
           (and (= (action r) assign)
                (isExternal (object r))))
      (deny r)))
```

Pr2: "If the requester is Faculty and the action is Assign then the request is allowed." / "Every request whose requester is Faculty and whose action is Assign is allowed."

```
(lambda (r)
  (if (and (isFaculty (subject r))
           (= (action r) assign))
      (allow r)))
```

Pr3: "If the action is an assign and a receive and the grades are external then the request is denied."

```
(lambda (r)
  (if (and (and (= (action r) assign)
                (= (action r) receive))
           (isExternal (object r)))
      (deny r)))
```

Pr4.1: "If the grades are internal and the requester is a TA and the action is an assign or is a view then the request is allowed."

```
(lambda (r)
  (if (and (isInternal (object r))
           (and (isTA (subject r))
                (or (= (action r) assign)
                    (= (action r) view))))
      (allow r)))
```

Pr4.2: "If the requester is a TA and the grades are external, and the action is an assign or is a view, then the request is denied."

```
(lambda (r)
  (if (and (isTA (subject r))
           (and (isExternal (object r))
                (or (= (action r) assign)
                    (= (action r) view))))
      (deny r)))
```

Appendix A.2. Environmental Constraints

"No action is a view and an assign."

```
(lambda (r)
  (not (and (= (action r) view)
            (= (action r) assign))))
```

"No requester is Faculty and a student."

```
(lambda (r)
  (not (and (isFaculty (subject r))
            (isStudent (subject r)))))
```

"If a requester is a TA then the requester is a student."

```
(lambda (r)
  (if (isTA (subject r))
      (isStudent (subject r))))
```

"A view is not an assign."

```
(lambda (r)
  (not (= view assign)))
```

Acknowledgements

The research reported in this paper was performed in connection with contract number W15P7T-08-C-P213 with the U. S. Army Communications Electronics Research and Development Engineering Center (CERDEC). The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the U. S. Army CERDEC, or the U. S. Government unless so designated by other authorized documents. Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] K. Arkoudas, Athena, http://proofcentral.org/athena/, 2012.

[2] K. Arkoudas, R. Chadha, J. Chiang, An application of formal methods to cognitive radios, in: Proceedings of DIFTS 2011 (Design and Implementation of Formal Tools and Systems), a workshop of FMCAD 2011 (Formal Methods in Computer Aided Design), Austin, TX, USA, pp. 3–13.

[3] K. Arkoudas, R. Chadha, J. Chiang, An efficient policy engine for dynamic spectrum access, in: Proceedings of the Fourth International Conference on Cognitive Radio and Advanced Spectrum Management (CogART 2011), Barcelona, Spain.

[4] K. Arkoudas, S. Loeb, R. Chadha, J. Chiang, K. Whittaker, Automated policy analysis, in: Proceedings of the 2012 IEEE International Symposium on Policies for Distributed Systems & Networks, Chapel Hill, NC, USA, pp. 1–8.

[5] C. Barker, Continuations and the nature of quantification, Natural Language Semantics 10 (2002) 211–242.

[6] A. Bernstein, E. Kaufmann, N.E. Fuchs, J. von Bonin, Talking to the semantic web – a controlled english query interface for ontologies, in: 14th Workshop on Information Technology and Systems, pp. 212–217.

[7] P. Blackburn, J. Bos, Representation and Inference for Natural Language. A First Course in Computational Semantics, CSLI, 2005.

[8] P. Bonatti, D. Olmedilla, Driving and monitoring provisional trust negotiation with metapolicies, in: In 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005, IEEE Computer Society, 2005, pp. 14–23.

[9] P.A. Bonatti, G. Antoniou, M. Baldoni, C. Baroglio, C. Duma, N. Fuchs, A. Martelli, W. Nejdl, D. Olmedilla, J. Peer, V. Patti, N. Shamheri, The rewerse view on policies, in: In Proc. of the ISWC Semantic Web Policy Workshop (SWPW).

[10] C. Brodie, C.M. Karat, J. Karat, An empirical study of natural language parsing of privacy policy rules using the sparcle policy workbench., in: L.F. Cranor (Ed.), SOUPS, volume 149 of *ACM International Conference Proceeding Series*, ACM, 2006, pp. 8–19.

[11] M. Collins, Head-driven statistical models for natural language parsing, Computational Linguistics 29 (2003) 589–637.

[12] J.L. De Coi, N.E. Fuchs, K. Kaljurand, T. Kuhn, Controlled English for reasoning on the Semantic Web, in: F. Bry, J. Małuszyński (Eds.), Semantic Techniques for the Web — The REWERSE Perspective, volume 5500 of *Lecture Notes in Computer Science*, Springer, Berlin / Heidelberg, Germany, 2009, pp. 276–308.

[13] J.L. De Coi, P. Kärger, D. Olmedilla, S. Zerr, Using natural language policies for privacy control in social platforms, in: ESWC'09 1st International Workshop on Trust and Privacy on the Social and Semantic Web (SPOT2009), CEUR-WS.org, Heraklion, Greece, 2009.

[14] K. Fisler, S. Krishnamurthi, L. Meyerovich, M. Tschantz, Verification and change impact analysis of access-control policies, in: International Conference on Software Engineering (ICSE).

[15] N.E. Fuchs, K. Kaljurand, T. Kuhn, Attempto Controlled English for Knowledge Representation, in: C. Baroglio, P.A. Bonatti, J. Małuszyński, M. Marchiori, A. Polleres, S. Schaffert (Eds.), Reasoning Web, Fourth International Summer School 2008, number 5224 in Lecture Notes in Computer Science, Springer, 2008, pp. 104–124.

[16] Y. Ge, L. Moura, Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories, in: Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09, pp. 306–320.

[17] H. Kamp, A theory of truth and semantic representation, in: J. Groenendijk, T.M.V. Janssen, M. Stokhof (Eds.), Truth, Interpretation and Information: Selected Papers from the Third Amsterdam Colloquium, Foris Publications, Dordrecht, 1984, pp. 1–41.

[18] C.M. Karat, J. Karat, C. Brodie, J. Feng, Evaluating interfaces for privacy policy rule authoring., in: R.E. Grinter, T. Rodden, P.M. Aoki, E. Cutrell, R. Jeffries, G.M. Olson (Eds.), CHI, ACM, 2006, pp. 83–92.

[19] M. Manzano, Extensions of first-order logic, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1996.

[20] J.B. Michael, V.L. Ong, N.C. Rowe, Natural-language processing support for developing policy-governed software systems., in: TOOLS (39)'01, pp. 263–275.

[21] R. Montague, The proper treatment of quantification in ordinary english, in: Approaches to Natural Language, volume 49, Dordrecht, 1973, pp. 221–242.

[22] L.D. Moura, N. Bjørner, Satisfiability Modulo Theories: Introduction and Applications, Communications of the ACM 54 (2011) 69–77.

[23] C.C. Shan, Linguistic Side Effects, Ph.D. thesis, Harvard University, 2005.

[24] W.A. Woods, Progress in natural language understanding: an application to lunar geology, in: Proceedings of the June 4-8, 1973, national computer conference and exposition, AFIPS '73, ACM, New York, NY, USA, 1973, pp. 441–450.