Natural-Language Processing Support for Developing Policy-Governed Software Systems

James Bret Michael, Vanessa L. Ong*, and Neil C. Rowe
Naval Postgraduate School, Department of Computer Science
833 Dyer Road, Monterey, California 93943-5118
{bmichael|rowe}@cs.nps.navy.mil; surfn63@cnrf.nola.navy.mil

Abstract

Organizations are policy-driven entities. Policy bases can be very large and the relationships between policies can be complex. In addition, policy can change on a frequent basis. Checking for gaps in policy or analyzing the ramifications of changing policy is necessary to both identify and rectify gaps or unintended policy prior to the policy base being refined into requirements for a system. A policy workbench is an integrated set of computer-based tools for developing, reasoning about, and maintaining policy. A workbench takes as input a computationally equivalent form of policy statements. We have developed a prototype of a tool that maps natural-language policy statements to an equivalent computational form. In this paper we describe the architecture of a natural-language input-processing tool (NLIPT). It has an extractor, which generates a meaning list representative of the natural-language input; an index-term generator, which identifies the key terms used to index relevant policy schema in the policy base; a structural modeler, which structures a schema for input; and a logic modeler, which maps the schema to an equivalent logical form. We experimented with a prototype of the extractor which successfully parsed a sample of ninety-nine Naval Postgraduate School security policy statements with ninety-six percent accuracy.

1. Introduction

Organizations expect their members to adhere to both explicit and implicit (unwritten) policies. Adherence to an organization's policies can be difficult, especially when the policy base is large or there is much implicit policy. For example, the designers of the early versions of the Java Virtual Machine (JVM) embedded policy into the JVM, but the security policy corresponding to the low-level procedures (i.e., mechanism) in the JVM were not known to the users. Complex relationships among policies and conflicting policies can lead to errors in the interpretation, refinement (i.e., implementation of policy as procedures in information systems) and enforcement of policy. This complexity can be amplified when two or more organizations form a permanent or temporary coalition: a subset or the entire set of policies of each organization form a composed policy base, such as in the case of coalitions of United Nations security forces.

Ideally, an organization's policies would be stored in a computational form in a central repository. Users could search the repository for policies that are applicable to a given action or plan (i.e., a sequence of actions for reaching a goal state) within a specific context. Queries to the repository would be through an interface. In addition,

* Lieutenant Ong's current address is Commander Naval Surface Reserve Force, Code N63, 4400 Dauphine Street, New Orleans, Louisiana 70146-5100.

39th Int. Conf. on Techn. for Object-Oriented Lang. and Syst., IEEE Computer Society Press, (Santa Barbara, Calif., July 2001), 263-274.

authorized users could update the policy base to reflect changes in the organization's policy. Such an interface could be part of a larger system that we term a "policy workbench." A workbench is a suite of tools that serves as an expert database management system. A policy workbench could update policy and test the policy for gaps; by "gap" we refer to any type of error in policy, its refinement, or implementation. It could also map the policy to procedures, which are the mechanisms that implement policy. Thus, a policy workbench is intended to enable the user to represent, reason about, maintain, implement, and enforce policy [12].

A policy workbench could assist members of an organization to better understand and become more aware of policy, which is necessary for acting in a manner that conforms to policy. Usability of the interface is important. People are less likely to use a system that requires cumbersome structured input, no matter how spectacular its results. An alternative would be to interact with the workbench using a natural language. However, efficient automated processing necessitates converting the natural language into a computational form, usually expressible as well-formed formulae in a formal language [7].

In work reported here, we examine processes that map policies submitted in a natural language to formats suitable for further processing by a policy workbench. Correct semantic interpretation of the input is also important. Errors in interpreting policy could become embedded in the computational model of the policy, making the resulting model of dubious value for use by the tools. Processing inputs submitted in a natural language entails semantic interpretation of submitted input, mapping the interpretation to an equivalent computational form, identifying applicable existing policies, and submitting everything to the appropriate workbench tools for further processing such as consistency checking.

2. Computer support for policy

Policy serves as a guide in decision-making processes. Policies typically exist in a hierarchy, progressing from broad-spectrum policies at the top to more narrowly defined policies at the lower levels. Policies have both a domain and scope. The domain specifies the objects in the organization's environment. For instance, policy pertaining to computer security might include system administrators, user accounts, and passwords as part of the domain. Scope identifies the range of roles, obligations, and rights of objects within its domain. As an example, the scope of a system administrator's role might include review of audit trails but not procurement of new equipment.

We distinguish between meta-policy, goal-oriented policy, and operational policy. Meta-policy is policy about policy [6]; meta-structures can prove useful in indexing heterogeneous systems. An example of a meta-policy is "Any policy related to system access is a security policy." Goal-oriented policy states the desired outcome but gives little or no indication of how to obtain the outcome. An example of goal-oriented policy is "Passwords must be difficult to guess." Operational policy defines required actions but rarely identifies the goal. An example of an operational policy is "Passwords shall be changed every six months."

There has been much research in the area of formal representation of all aspects of policy. Ambiguities in natural-language statements used to represent policy can lead to several interpretations of the policy. Formal representation of policy can, to some ex-

tent, remove the ambiguity. Formally represented policies can be defined by axioms and reasoned about using automated systems [1].

2.1. Policy workbench

A policy workbench is an automated knowledge-based system comprised of a suite of tools designed to assist the user in the representation of policy; reasoning about the properties of policy such as consistency, completeness, soundness, and correctness; refinement of policy into systems; maintenance of policy; and enforcement of policy. Ultimately, however, a policy workbench's purpose is to facilitate adherence to policy.

Use of automation to maintain, reason about, refine, or enforce policy has been examined in several projects. In 1982 ZOG, a menu-based display system developed at Carnegie-Mellon University, was installed on the aircraft carrier *USS Carl Vinson*. It served as an aid in information management and decision-making in combat situations. Aspects of the ship's tasks, including policy and knowledge from subject-matter experts, were represented in the knowledge base [13].

Regulating Internet and network traffic policies has provided the impetus for many commercial-off-the-shelf (COTS) middleware releases. Although not necessarily a policy workbench, much of this middleware (such as intrusion-detection devices and firewalls) allows a network administrator to select predefined policies or generate their own to enforce an organization's network traffic policies.

The Internet Engineering Task Force (IETF) is developing a policy management architecture that will allow consistent recognition and enforcement of policy protocols. This includes a central policy repository, a common policy definition language, and a common policy object model. The goal is to allow consistent interpretation of protocol policy regardless of the device [15]. Formal languages, such as Ponder [2] and Path-based Policy Language (PPL) [14], provide for specifying policy about the management of networks and distributed systems.

2.2. Policy workbench architecture

Sibley, Michael, and Wexelblat propose an architecture for a generic policy workbench in [12]. The authors identify five user classes that should be accounted for in the design of a policy workbench:

1. The policy maker enters policy, maintains a current resource dictionary, confirms consistency of proposed policy statements, allows users to propose scenarios for feedback, and partitions policies into subsets as applicable and necessary.
2. The policy maintainer performs regression testing (testing changes in policy from a baseline against some criteria such as correctness or consistency) to ascertain the consequences of modifying policy. It distributes modified policies, in addition to performing configuration management and control tasks.
3. The policy implementer translates policy into procedures, maintains records of rule applications, and maintains a current account of relationships or linkages among policies.
4. The policy enforcer identifies violations of policies and recommends appropriate responses, checks procedures for consistency with policy, and provides authorizations for exceptions to policies.
5. The policy user analyzes the existing policy base via queries.

A policy workbench architecture has the following components:

1. "A *theorem and assertion analyzer* (entering and exercising policy) to check inputs stated as axioms and theorems."
2. "A *rule compiler-generator-interpreter* (selecting, merging and generating parts of systems) to produce an executable component of the system." (Although the proposed architecture for the workbench could support many data models, the examples in [12] are represented as conditional rules.) An example of an executable component is a procedure that models a proposed policy environment and provides a run-time scenario for user queries.
3. "An interactive *policy structurer and selector* (aiding in understanding and applying policy) to check what rules are applicable to a given situation and preprocessing the rules into pre- and post-conditions."

Policy input is accepted in a quasi-natural format that is checked for syntactic correctness, then mapped to a formal rule. The rule is submitted to a theorem prover that performs semantic evaluation of the rule to check consistency with policies in the database and to eliminate duplication. If the rule is acceptable, it is sent to the policy database. Conflicts or errors are reported back to the user for action.

The theorem and assertion analyzer also accepts queries regarding policy statements. Accepted in natural language, queries are first submitted to an extractor and translator module, which converts the query to an appropriate computational form. The translation is then processed in a fashion similar to direct policy input with the exception that the policy database is not updated. Rather, the query response is directed to the user.

The rule compiler-generator-interpreter allows the operation of a simulated system determined by user procedural inputs or scenario requests. Policy changes can essentially be seen "in action." The policy structurer and selector tool finds policies represented as preconditions and postconditions in the policy database that are applicable to a given input (e.g., scenario or direct policy input). It does so by finding commonalities in policy statements. This information is updated in the resource dictionary. The "understanding module" in conjunction with regression testing would allow the user to discern the effects of policy changes. The understanding module of the policy selector and structurer identifies the relationships between a policy and other components in the database. This tool could also aid in the development of the exceptions that are required for a policy set.

The focus of this paper is on a natural-language input-processing tool (NLIPT) for a policy workbench.

3. Previous work on natural-language processing of policy

One approach to policy specification is to require that the users of a policy workbench articulate policy or queries about it in a quasi-natural language "until a more friendly user interface is developed" [12]. Then a software engineer would translate this language into a formal language. But the policy base of an organization can be quite large, making the process of manual translation into an acceptable format almost insurmountable. And since policy bases are typically not static, frequent updates may be required, placing a further burden on the users. Furthermore, manual translation of policy is an error-prone process, as was demonstrated in experiments reported in [7]. A better policy workbench would accept input in natural language and store formal models of the expressed policy. A variety of previous work has addressed this.

3.1. The British Nationality Act as a logic program

[11] explores the feasibility of using logic statements to represent part of The British Nationality Act of 1981 and mechanically determine the consequences of the Act when applied to test cases. The system represents part of the Act using extended Horn clauses implemented as a Prolog program using APES, a Prolog-based expert system shell developed by Sergot and Hammond. The collection of clauses is an axiomatic theory, which can be mechanically analyzed by theorem provers; Prolog serves as a limited-purpose theorem prover. The shell queries a user to dynamically input facts as required.

This work followed a top-down, goal-directed manual approach when formalizing the Act. They postponed the representation of lower-level concepts until high-level concepts were refined. They addressed vague concepts such as "good character" by assuming that vague concepts always applied when generating answers. Modification and restructuring of previously formalized concepts was made as needed when later sections of the Act refined earlier concepts.

A closed-world assumption implemented negation as failure (i.e., anything which is not known is assumed to be false). Double negation, if treated classically, would cancel out, but this was not the intent of the Act for all cases. The authors treat negative information as part of the input from the user, though this approach has drawbacks. Counterfactual conditional statements were not adequately handled by extended Horn clause logic, so additional rules were written to handle them. Finally, discretionary clauses (i.e., clauses that give an authority the discretion to modify application of other sections of the Act) were handled by generating two clauses: one for the standard case and one for the discretionary case.

This approach requires considerable revision work when the policy base is large, so it would not scale. Modification or restructuring of rules could easily get out of hand as the number of formalized statements increased. Moreover, the closed-world assumption, while convenient for domains where all cases are specified, would not apply to most real-world policy.

3.2. INCAS: A legal expert system for contract terms in electronic commerce

[16] developed an automated expert system that provides advice on the use of Incoterms, thirteen terms used in legal trade contracts that stipulate which party (buyer or seller) is responsible for arranging and paying for transport and arranging the required documents for the transport. INCAS is a Prolog-based system that defines Incoterms to the user, reasons using the Incoterms knowledge base to advise on queried scenarios, and proposes the optimal Incoterm for both buyer and seller given their obligations. It uses formal specification of the Incoterms in Prolog, manually derived from the International Chamber of Commerce guidelines. A graphical user interface allows the user to view the INCAS response to a query along with the assumptions used to derive the conclusion when applicable. Users can change the assumptions to refine the conclusion and rerun the query. Users can also introduce hypothetical assumptions to generate responses to what-if scenarios.

The Incoterms domain has many instances where defeasible reasoning is involved. Defeasibility means that rules can be superseded by another rule or fact. The authors address defeasibility by incorporating exception predicates into the rules and adopting the closed-world assumption. Exceptions to exceptions are also addressed in a similar

fashion. INCAS performs symbolic processing on strings and does not use any semantic constructs. A user is required to provide the data concerning the situation for which the query has been formulated. It can also accommodate correcting or otherwise modifying assumptions used in deriving a conclusion to generate a new conclusion. No data has been provided about the difficulty of development and the approach; scalability is a problem since manually formalizing policy is difficult.

3.3. SACD: A system for acquiring knowledge from regulatory texts

[9] developed a Prolog system SACD capable of generating a knowledge base from regulatory text by analyzing the text's logical structure. SACD is specifically designed to work with prescriptive text, especially the normative that describe instructions and characteristically contain modal operators. Most regulatory text, such as policies and legal manuals, are prescriptive in nature. The authors use portions of the National Building Code of Canada for analysis.

Regulatory text can typically be segregated into three layers: (1) the macrostructure layer, corresponding to headings, titles, chapters, and sections; (2) the microstructure layer, the logical content of the text featuring the expressions that identify conditions, exceptions, modalities, and references; and (3) the domain layer, domain-specific information that belongs to neither of the other two layers. SACD initially uses context-free macrostructure and microstructure text grammars to parse the input text. The grammars have multiple entry points and behave like chart parsers, the classic bottom-up approach to parsing with a context-free grammar. Macrostructure analysis detects the presentation elements; microstructure analysis uses modal operators (e.g., "may," "must," "cannot"), conjunctions, internal references, and punctuation in order to identify relevant objects in the domain and applicable deontic rules. Deontic rules characterize the modal object to which they refer and require modal logic. The knowledge base eventually contains an object-type hierarchy, object descriptions, rule specifications which indicate the modality and characterize the related object, relationships between the data structures, and the relationships between the structures and the text provisions. Data structures are represented using Prolog predicates. A "domain specialist" creates the object-domain hierarchy, partly domain-specific, and resolves anaphoric references.

SACD was used on a subset of the National Building Code of Canada (NBC). Of 100 provisions evaluated, the macrostructure analysis took roughly five minutes overall. The microstructure analysis averaged five seconds per sentence depending on the complexity. A simple expert system checked situations against the provisions of the code. This approach requires well-structured input text, and many policies are well structured. However, the system may not be suitable for handling queries to the policy workbench, where we need recognition of meta-textual structures and the refinement of rules based on cross-references. SACD requires a lot of user interaction; this makes scalability a concern. Also, the user must have an intimate knowledge of the input text to correctly generate the domain hierarchy, which makes it subject to personal interpretation.

3.4. Knowledge extraction from text using machine learning

[3] investigates natural-language processing to extract knowledge from technical expository texts in the MaLTe system. MaLTe operates with a minimum of a priori

knowledge and learns as it goes along. It extracts from the personal income tax law as described in *Revenue Canada 1991*. Additional knowledge required to resolve ambiguities and inconsistencies and to define synonyms are elicited from the user as needed.

Facts are generalized automatically from the examples into the higher-level concepts found in the narrative. In doing so, implicit knowledge is made explicit and a hierarchical domain (for the text) is generated. The authors propose absorption, an operator used in inductive logic programming, to achieve this abstraction, but over-generalization is a danger. Related facts obtained from the examples are aggregated. The constants in the facts are generalized to variables and the aggregation method becomes a rule. To handle texts with nested concepts, explanation-based generalization (EBG) integrates the applicable rules into one that makes the most useful features of the concepts explicit. This will operationalize the rule, that is, make it a procedure. But an acceptable operationality criterion must be determined, and EBG requires a complete domain-knowledge base. MaLTe is not autonomous; users must supply missing facts, correct mistakes, and address synonyms. The actual extent of the interaction required for a complete knowledge base for some domain is uncertain.

The authors chose a text about small engines to test the system. The system was able to generate correct analyses of inputs it had never seen before by using partial matching on the semantic patterns it had in its knowledge base. Imperfect parses did not necessarily result in no knowledge acquired. The average user time to determine a correct relationship proposed by HAIKU decreased over the course of the experiment. As the knowledge base grew, the user made fewer corrections to the proposed inferences. But very large bodies of policy may prove onerous to the user if much action is required on every input.

4. Our natural-language processing architecture

Our proposed natural-language input-processing tool both maps policy to an object-oriented "schema" and answers questions about the schema. [7] shows object-oriented modeling of policy appears to produce fewer structuring errors than a non-object-oriented approach. An object-oriented schema will then control the axiomatization of the policies. This does require rephrasing of some policy statements to explicitly refer to constructs of the schema before formalizing the statement, but this will not be burdensome.

Our approach (Figure 1) is to translate all user input—policies, queries, and scenarios—into equivalent first-order predicate calculus. Key terms are extracted and sent to the policy-element identifier tool, which identifies applicable elements in the policy base. The inability to find any match could suggest an error message or an automatic modification. The input is then mapped to the retrieved policy information, feedback returned to the user, and possibly changes made to the information. The policy-element identifier is an extension of the policy-selector tool proposed in [12]. The policy-element identifier differs from the policy-selector tool in that it would retrieve the structural schema of the pertinent items, rather than the computational form.

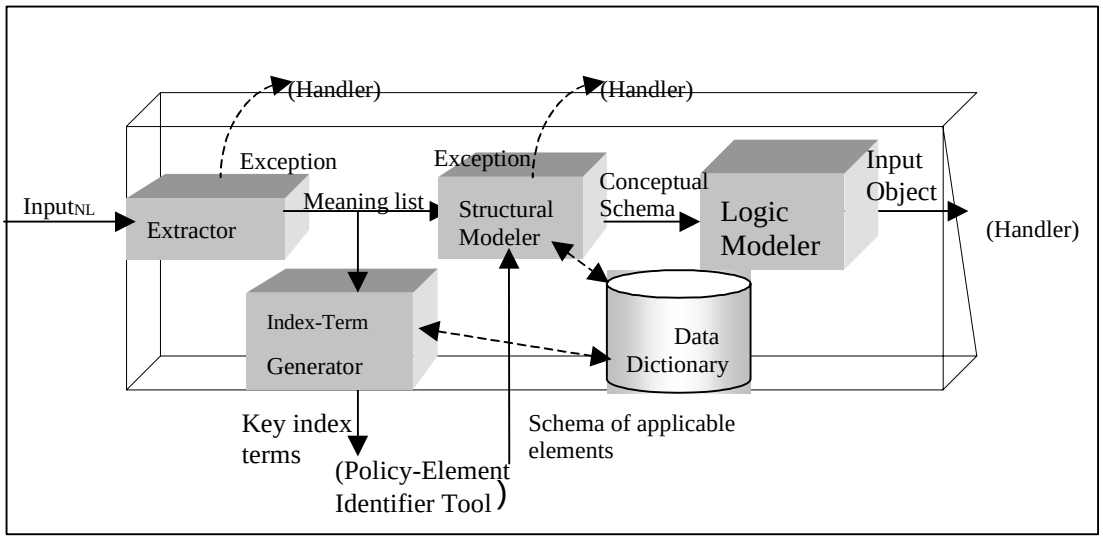

Figure 1. Proposed architecture of natural-language input processing tool

The extractor generates a meaning from each important word of the natural-language input. This meaning list should for a start identify the subject, object, and attributes of all actions. For example, "All passwords must be at least eight characters in length" could have the meaning list:

[subject(passwords), subjectquantifier(all), object(characters), objectmodifer(eight), objectquantifier(at least), subjectattribute(length), action(modal(must), verb(be))]

Accurate semantic interpretation requires a lexicon, grammar rules for natural language (usually mostly context-free rules with context-sensitive extensions), and semantic rules for connecting meanings of individual words.

The index-term generator extracts the terms from the meaning list most likely to find a relevant match in the policy base. Verbs, subjects, objects, and attributes are tried. For the example they would be subject(passwords), object(characters), attribute(length), and verb(be). The root form of each candidate term is obtained using synonym lists, misspelling analysis, and morphological simplifications. We can weight the index terms to maximize the likelihood of selecting relevant schema in the policy base. Subject terms should have high weights, original inputs should weigh more than synonyms, and weighting should also increase with the rarity (i.e., infrequency of use) of the word.

The structural modeler analyzes retrieved schema. The schema will identify implicit facts and hierarchical relationships. If no applicable schema can be found in the policy base, the structural modeler can either proceed (as for assertions of new policy) or generate an exception. For the example, a policy-element identifier should find schema that state the following for our example:

A valid password is issued to an authorized user, an employee, to allow the user to logon. The user must logon to obtain access to the system. Access is granted when a valid password is entered to complete a logon.

Suppose no schema defines a valid password, but our new policy states that the length must be at least eight characters. Linkages to the term "valid" should be made to the schema, and we should infer that passwords have a minimum length and are comprised

of characters. This can be done via the lexicon reference through recognition that the adverbial phrase "at least" is a minimum constraint and that the term "characters" is not a unit of measurement but an object. The term "be" allows the connection to composition. Quantifiers must also be inferred and their scope determined. Further, appropriate generalizations should be made. For our example, in our example the generalization should be made that length is actually the size of the password:

password(applies_to: all, property (size (minimum(eight)), composition(characters))

Finally, a logic modeler uses the schema developed by the structural modeler to generate a first-order predicate logic representation of the input. It must determine quantifiers and their scope. Other inferences must be made as appropriate; for our example, the logic modeler should infer that "composition" from the schema signifies a "part_of" relationship between "character" and "password." The extractor will need to express meanings using modal logic. Modal logic [4] is important in formalizing policy because policy is typically prescriptive in nature and indicates obligations, permissions, and interdictions. A modal is a special marker of verb tenses in English appearing as an "auxiliary" before the verb. Examples of modals are "can," "should," "must," and "may." Modal logic includes reasoning about such constructs; it augments first-order logic with modal quantifiers on sentences. It has several subareas, like deontic logic concerned with obligation, permission, and interdiction. Our example finally becomes:

$\forall_X$ *(password(X)* $\rightarrow$ *(*$\exists_S$*(*$\forall_C$ *(character(C)* $\wedge$ *part_of(C,X))* $\rightarrow$ *member(C,S)))* $\rightarrow$ *(*$\exists_N$ *size(S,N)* $\wedge$ $N \geq 8$*)))))*

5. Implementation of a natural-language extractor

5.1. Program design

We now describe a prototype natural-language extractor tool that we implemented. Input policy statements were initially submitted to an English tagger. This significantly reduced the complexity of the extractor program, which used the part-of-speech tags to more easily identify the key items for the meaning list. The tagger used was a syntactic partial parser, LT CHUNK, developed by the Language Technology Group of Edinburgh, United Kingdom. Its output assigns a part-of-speech tag to each word or symbol of the input and it brackets key multi-word syntactic units such as noun phrases. LT POS, a component used in LT CHUNK, assigns part-of-speech tags to words and symbols using hidden Markov models using maximum-entropy probability estimators [5]. It contains a tokenizer, a morphological classifier, and a morphological disambiguator. LT POS achieves between ninety-six and ninety-eight percent accuracy at correctly assigning POS-tags when all the domain words are in the lexicon. Noun groups and verb groups that it recognizes are also bracketed.

We converted the tagged output via a Java program to a format suitable for manipulation via Prolog. A Prolog extractor program generates a meaning list using the tags and basic grammar rules. Prolog is advantageous to use in natural-language processing because of its automatic backtracking. This program approximates a finite-state grammar developed to cover sentences in the test corpus while remaining as general as possible. A full context-free grammar is more desirable, but proved to be impractical in the time available to conduct this research.

The algorithm for the extractor was designed to capitalize on the typical policy structure as described in [8] as follows. For a given sentence or phrase, the main steps are:

- Identify the first verb group that is not part of a clause or phrase. Verb groups with modals are preferred over those without.
- Segment the input into the front scope (the input left of the verb group), the verb group, and the back scope (the remainder of the input).
- Find the verb group's subject and its modifiers (adjectives, determiners, quantifiers, prepositional phrases, and participial phrases) in the front scope of the input. The subject should be in the last noun group that is not part of a phrase or clause. Quantifier modifiers on the subject are especially important to note.
- Find the verb, its modals, and its modifiers (adverbs, other auxiliaries, and embedded objects) in the verb group.
- Find the verb group's object and modifiers (as with the subject and its modifiers) in the back scope of the input. The object should be in the first noun group that is not part of a phrase or clause.
- Construct a meaning list listing the subject(s), the subject modifiers, the object(s) and modifiers, and the verb(s) and its modifiers including modals.

Bracketed noun and verb phrases found by the tagger (some, but not all of noun and verb phrases present) are a great help in this process. Conjunctions at several syntactic levels are sought and handled, ranging from compound-sentence structures to compound verb groups to compound nouns, verbs, adjectives, and adverbs. Appositives set off by commas or parentheses are sought and extracted. Infinitive groupings are also recognized and can serve as subject, object, or adverbial.

Subordinate clauses are also identified; phrases and clauses are parsed using the split algorithm mentioned earlier. While it is relatively easy to identify the beginning of a phrase or clause, identifying the end is another matter. To address this problem, phrases and clauses are consecutively extracted from the rear of the scope fragment. We identify the last occurrence of a word that could begin a phrase or clause (preposition, pronoun, or adverb) and attribute the words following it as part of the phrase or clause.

Here is an example from the extractor.

- Input in natural-language format:

Information of questionable value to the general public must be evaluated before worldwide dissemination to assess the risk to the DoD.

- LT CHUNK tagger output:

[Information_NN] of_IN [questionable_JJ value_NN] to_TO [the_DT general_JJ public_NN] < must_MD be_VB evaluated_VBN > before_IN [worldwide_JJ dissemination_NN] < to_TO assess_VB > [the_DT risk_NN] to_TO [the_DT DoD_NNP] ._.

- Meaning list produced by the extractor:

[[main_subject_Group([subject(information), relationship(of), subj(value), modifier(value,questionable), relationship(to), subj(public), determiner(public,the), modifier(public,general)]),

main_object_Group([relationship(before), obj(dissemination), modifier(dissemination,worldwide), obj(risk), obj(to_assess), determiner(risk,the), relationship(to), obj(dod), determiner(dod,the)]),

main_verb_Group([modal(must), passive(must,be), verb(must,evaluated)])]

5.2. Testing the extractor implementation

The test corpus was ninety-nine policy statements pertaining to web page content at the Naval Postgraduate School; see [10] for both a listing of the policies and the code listing for the extractor program. Many of the policy statements were in a prescriptive format; they had a modal grouping as well as well-defined front and back scopes. However, quite a few statements were expository, free-form with the intent to clarify a point. Some statements were bulleted items that were dependent clauses or phrases. Many of the statements contained technical constructs such as Uniform Resource Locator (URL) addresses.

The following statistics were tabulated for test runs: (1) whether the meaning list (ML) correctly identifies main subject(s), verb(s), and objects(s) based on the natural-language input; (2) whether the meaning list is incorrect but correctly identifies main constructs of input based on the tagger and intermediate output; and (3) whether the meaning list is incorrect. Of the generated facts, an average of ninety-six percent were correct. Table 1 summarizes the results for full sentences (excluding lists), counting a parse as correct if every fact generated was correct. Full sentences contained twenty-two words and generated eighteen facts on the average (since some words were combined in the meaning list).

Table 1. Results from testing the extractor component

Number of input statements	Number of completely correct meaning lists	Number incorrect meaning lists from a correct parse based on tags	Number incorrect meaning lists due to extractor	Number incorrect meaning lists due to intermediate programs	Number incorrect meaning lists due to LT CHUNK
99	**27**	**21**	**42**	**2**	**7**

The extractor program should be modified to verify and correct the tagger output as needed. Use of a context-free grammar to verify part-of-speech assignments by LT CHUNK would alleviate much of the problem. For example the tagger incorrectly identifies “Command” as a verb in “Command Web sites shall contains links to the following sites.” The tagger also fails to bracket properly some of the more ambiguous noun phrases, and a better parser could recognize these.

The tagger failed to recognize URL addresses, which can be resolved by improving morphological analysis. We ignored in testing the bulleted parts of the input corpus; LT CHUNK was able to identify lists, but the necessary handling of ellipsis for them must be added. The program also had difficulty identifying the subject when more than one verb group applied to that subject.

6. Conclusions and future research

Our goal in designing our NLIPT is to shift, as much as possible, the burden of formalizing policy statements from the user of a policy workbench to the computer-based tool. Delegation of this task to NLIPT is advantageous when large policy bases are involved, the relationships between policies are complex, or the user of the policy workbench is not well versed in formal methods. Even if a user of a policy workbench is knowledgeable enough to formalize policy, the manual process itself is error-prone.

Our extractor prototype often correctly identifies the subject, object, attributes, and the verb. However, the program remains to be refined so that it can correctly handle more and more complex statements of policy. Modifying the program to use a full context-free grammar instead of ad hoc rules could increase the robustness of the program. Future research includes further refinement of the other three components of the NLIPT. We used commercial software tools in our extractor; more use of them could reduce the policy workbench development time. Other research will explore the interaction or degree of coupling of the natural-language interface with the other policy-workbench tools. Research is being conducted at the Naval Postgraduate School on the automated testing of policy, and the testing tools may place tool-specific requirements on the natural-language interface.

Acknowledgements

This work was supported by a grant from the Naval Postgraduate School Institutionally Funded Research Program. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

References

[1] Cholvy, L. and Cuppens, F. Analyzing consistency of security policies. In *Proc. IEEE Symp. on Security and Privacy*, IEEE (Oakland, Calif., May 1997), 103-112.

[2] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. The Ponder specification language. In Sloman, M., Lobo, J. and Lupu, E. C., eds., *Lecture Notes in Computer Science, No. 1995: Proc. Internat. Conf. on Policies for Distributed Systems and Networks*, Springer-Verlag, Berlin, (Bristol, Eng., Jan. 2001), 18-38.

[3] Delannoy, J. F., Feng, C., Matwin, S., and Szpakowicz, S. Knowledge extraction from text: Machine learning for text-to-rule translation. In Brazdil, P. B, ed., *Lecture Notes in Computer Science, No. 667*: *Proc. Machine Learning Text Analysis Workshop of the European Conf. on Machine Learning*, Springer, Berlin (Vienna, Aust., Apr. 1993), 1-7.

[4] Garson, J. W. Modal logic. In Zalta, E N., ed., *The Stanford Encyclopedia of Philosophy*. Spring 2001 ed.; http://plato.stanford.edu/entries/logic-modal

[5] Grover, C., Matheson, C., and Mikheev, A. TTT: Text Tokenisation Tool. Language Technology Group, Edinburgh, Scot.

[6] Michael, J. B., Sibley, E. H., and Wexelblat, R. L. A modeling paradigm for representing intentions in information systems policy. *Proc. First Workshop on Information Technologies and Systems*, Massachusetts Institute of Technology Sloan School of Management (Cambridge, Mass., Dec. 1991), 21-34.

[7] Michael, J. B., Sibley, E. H., Baum, R. F., and Li, F. On the axiomatization of security policy: Some tentative observations about logic representation. In Thuraisingham, B. M. and Landwehr, C. E., eds., *Database Security, VI: Status and Prospects*. Elsevier Science (North-Holland), Amsterdam, 1993, 367-386.

[8] Moulin, B. and Rousseau, D. Automated knowledge acquisition from regulatory texts. *IEEE Expert* 7, 5 (Oct. 1992), 27-35.

[9] Moulin, B. and Rousseau, D. SACD: A system for acquiring knowledge from regulatory texts. *Computers Elect. Engin.* 20, 2 (Mar. 1994), 131-149.

[10] Ong, V. L. An architecture and prototype system for automatically processing natural language statements of policy. Master's thesis, Naval Postgraduate School, 2001.

[11] Sergot, M.J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P., and Cory, H. T. The British Nationality Act as a logic program. *Comm. ACM* 29, 5 (May 1986), 370-386.

[12] Sibley, E. H., Michael, J. B., and Wexelblat, R. L. Use of an experimental policy workbench: Description and preliminary results. In Landwehr, C. E. and Jajodia, S., eds., *Database Security, V: Status and Prospects*. Elsevier Science (North-Holland), Amsterdam, 1992, 47-76.

[13] Sloane, S. B. The use of artificial intelligence by the United States Navy: Case study of a failure. *AI Magazine* 12, 1 (Spring 1991), 80-92.

[14] Stone, G. N. A path-based network policy language. Ph.D. dissertation, Naval Postgraduate School, 2000.

[15] Strassner, J., Ellensson, E., and Moore, B., eds. Policy framework core information model. Internet Engineering Task Force: Network Working Group, Internet Draft draft-ietf-policy-core-schema-02.txt, Feb. 1999.

[16] Tan, Y.-H. and Thoen, W. INCAS: A legal expert system for contract terms in electronic commerce. *Decision Support Systems* 29, 4 (Dec. 2000), 389-411.