APPROVAL SHEET

Title of Thesis: Integrated Development Environment for Policies

Name of Candidate: Anjali Bharat Shah
Master of Science, 2005

Thesis and Abstract Approved: ______________________
Dr. Timothy W. Finin
Professor
Department of Computer Science and
Electrical Engineering

Date Approved: ____________________________

Curriculum Vitae

Name: Anjali Bharat Shah.

Permanent Address: 4806 Westland Blvd, Apt A, Baltimore, MD 21227.

Degree and date to be conferred: Master of Science, May 2005.

Date of Birth: May 31, 1981.

Place of Birth: Mumbai, India.

Secondary Education: Mithibai College, Mumbai, 1998.

Collegiate institutions attended:
University of Maryland, Baltimore County, M.S. Computer Science, 2005.
Usha Mittal Institute of Technology, India, B.Tech. Information Technology, 2002.

Major: Computer Science.

Professional positions held:

Software Engineer I, Broadwing Corporation (July 2004 - Present).

Research Assistant, CSEE Department, UMBC. (May 2003 - June 2004).

ABSTRACT

Title of Thesis: Integrated Development Environment for Policies

Anjali Bharat Shah, Master of Science, 2005

Thesis directed by: Dr Timothy W. Finin
Professor
Department of Computer Science and
Electrical Engineering

There has been growing interest in the use of policy-based governing frameworks for management of a wide range of systems. These systems ranging from simple and static to increasingly complex and dynamic have demanding requirements that make the management of policies a complex task. Though tools have been developed to facilitate policy frameworks, there is not much work in policy development that meets the requirements of these policy-based environments. Some of these requirements include: (i) facility for both individual and rule-based policy specifications, (ii) ability to test policy conformance, and (iii) simplification of the inherently error-prone process of policy creation through the use of templates and well-designed interfaces.

In this thesis, we present RIDE, an integrated policy development environment that uses a wizard-based approach to provide a user-friendly and extensible graphical user interface. RIDE (Rei Integrated Development Environment) has been developed as a plug-in extension of Eclipse workbench and enables the development of policies specified in Rei, which is a declarative, machine-understandable policy specification

language. As our main goal was to facilitate policy development for distributed systems, we chose Rei specifications because Rei is grounded in a web ontology language that allows it to model different kinds of domain knowledge and it has been successfully deployed in several open, dynamic distributed environments.

RIDE attempts to meet the requirements of a wide range of policy-based environments for which existing tools provide little or no support. This is possible due to the unique combination of following features it supports: (1) simple, user-friendly interface offering valid input options through the entire process, (2) ease of management of domain information by offering the option of template creation, (3) automation of the complex and error-prone policy creation process by automatically generating user-defined policies in Rei, (4) ability to express individual as well as group policies, (5) ability to accommodate information spanning multiple domains by using ontologies to represent domain information, (6) policy creation over speech acts that are used for dynamic policy management, (7) support for creation and verification of policy test-cases for checking policy conformance, and (8) conformity to the principles outlined by Human Computer Interaction by performing iterative evaluation and refinement of the interface to make it as simple and useful as possible from user's perspective.

INTEGRATED DEVELOPMENT ENVIRONMENT FOR POLICIES

by

Anjali Bharat Shah

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2005

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr Timothy Finin and my thesis committee member, Dr Lalana Kagal of CSAIL, MIT for their invaluable advice, support and guidance. I would also like to express my earnest thanks to Dr Anupam Joshi for the help I have received from him throughout the span of my graduate studies. I am grateful to Dr Yelena Yesha for being a member on my thesis committee. Finally, I thank the staff members of graduate school and the Computer Science and Electrical Engineering department at UMBC for help with the administrative aspects of my thesis work.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Policy-based approaches have been widely employed in areas such as security, trust and management of networks and distributed systems. Recent research affirms their utility in more complex, open, dynamic distributed systems such as pervasive computing environments, semantic web, grid computing and multi-agent systems. Policies could be defined as rules that guide or control the behavior of entities in a system. Security policies generally define rules for access-control, authentication and integrity, or authorization of entities in a system. They restrict access to certain resources in an organization. For example, there could be a policy stating that no graduate student in UMBC can use the fax machine in the graduate school office. Management policies define the role of an individual in terms of his duties and rights. For example, a policy stating every graduate student in UMBC should display his ID card to school officials upon demand. The chief benefit from using a policy-based governing framework is that policies are separated from implementation of the managed system. This gives improved flexibility in changing the behavior of the system without changing its underlying implementation.

1.1 Overview of Problem Addressed and Thesis Contribution

The wide range of environments to which policy-based government is applicable makes management of policies a complex task. Some of the characteristics of these environments such as (i) a large number of clients, resources and services, (ii) inability to pre-determine all resources and clients, (iii) inability to identify all entities, and (iv) environments spanning several domains with diverse types of domain knowledge; underline the need for appropriate tool support to facilitate their management. Our work on Integrated Development Environment for policies uses Rei policy specification language to express policies created by the user through a graphical interface. Rei is a declarative, machine-understandable policy specification language [14]. It defines a policy as a set of rules describing deontic [16, 10] concepts of permissions, obligations, dispensations, and prohibitions. Associated with the language is a policy engine that can be used within the application domain to interpret and reason over policies, help resolve in case of conflicts between policies and answer queries related to policy making. There is considerable amount of work in the area of policy specification languages. However, compared to other policy languages, Rei is unique in that it has been successfully deployed and used in open, dynamic distributed environments. Such environments typically have more complex requirements that make it indispensable that the language be supported by user interface tools that hide much of its complexity from the users. It is the abstraction from the intricacies of policy specification using Rei that our work seeks to provide.

The primary focus of this thesis is the development of an integrated policy management environment that provides user-friendly, easily extensible graphical tools that automatically generate code for user-defined policies in a wide range of environments. Its main contribution is RIDE, which stands for **R**ei **I**ntegrated **D**evelopment **E**nvironment. RIDE toolkit consists of (i) a wizard-based intuitive, graphical user interface that is plugged into the Eclipse framework, (ii) its interface with the Rei engine that allows users to query the engine and test the correctness of the policies created through the wizard, and (iii) a namespace manager integrated into the wizard that has useful features like ease of management of domain independent information, use of ontologies to describe domain specific information and namespace templates to eliminate the need to re-enter frequently needed domain information. The domain-specific information represented using ontologies that are expressed in semantic web languages like OWL [2] and RDFS [24] offers several benefits. Firstly, policies can be created using entities and the context at different levels of abstraction, ranging from a specific instance to the most general class it inherits from. Secondly, it provides the extensibility required to incorporate different kinds of application-specific knowledge and the ability to accommodate information spanning several domains. This enhances the scalability and extensibility aspects of RIDE toolkit.

1.2 Thesis Outline

This thesis is structured as follows:

Chapter I provides an overview of the problem domain and our efforts in working on a solution for it.

Chapter II provides information on related work on user interfaces for policy languages.

Chapter III provides a comprehensive description of the problem domain, our contribution in solving it, and scope of the thesis.

Chapter IV provides a detailed description of the IDE design, framework, supported features along with the description of tools and concepts used.

Chapter V presents a summary of the work and identifies directions for future research.

Chapter 2

RELATED WORK

There has been considerable research in the area of policy specification languages and user interface support for them. However, a majority of these tools and their respective policy specification languages are tightly coupled to the environment. As a result of this, much of the complexity of the underlying implementation and mechanisms is exposed to the user. The principle factor affecting our choice of providing a wizard-based intuitive approach has been to address this issue. We provide abstraction from the underlying mechanisms and provide an easy-to-use, friendly environment. Although tools have been developed with features that aid right from policy creation up to its deployment and conflict resolution, their policy specification languages lack expressivity, which in turn limits their scope. On the other hand, our work lets users express security, management and even conversation policies [14]. It allows specification of individual and group based policies, which are auto-generated in OWL. This expressivity is gained through the choice of Rei as the policy specification language. Expressing them in OWL also enhances their interoperability. Kagal [14] gives a comprehensive coverage of research efforts in specification for policy languages and advantages of using Rei over them. In this chapter we discuss related work in the area of tool support for policy-based management.

The Platform for Privacy Preferences (P3P) is a standard developed by the World Wide Web Consortium (W3C) that enables websites to describe their privacy policies and allows browsers and other user agents to reason over these policies to decide whether they match the user's preferences [5]. It warns users of conflicts with the website's privacy policies, if any, and allows them to automate the acceptance or rejection of the latter's requests for information. This work tries to reduce Internet users' concerns about the personal information they reveal when visiting various websites. Websites specify their policies in P3P expressed using machine-readable XML [25] and users express their preferences using P3P Preference Exchange Language (APPEL) [1].

The IBM P3P Policy Editor provides a user-friendly interface, for creating a website's privacy policies. Though it helps organizations create machine-readable policies for their websites by reducing the complexity of this inherently error-prone task, it cannot be used to describe privacy policy for agents and services on the web. It does not allow much expressivity and is much more limited in its capabilities. Our work, on the other hand, can be used to create Rei policies for any kind of web entities such as resources, agents, services, and users. This is achieved by allowing the use of generic variables for actors and actions in policy creation. These variables could represent any entity or set of entities and any action or set of actions in the domain. Constraints can be attached to them to specify conditions under which the rules and policies apply. Also, unlike P3P policies, these policies are expressed in OWL and can be easily translated to other semantic web languages, which makes them semantically rich and suitable for the Semantic Web.

EPAL 1.1 [20] specifies enterprise-level privacy policies in terms of positive and negative authorization rights. It can be thought of as a policy language that specifies privacy rules in XML, which can be enforced by organizations to automate privacy management tasks. EPAL Editor is developed as an open source project that provides a GUI to enter information about policy rules, user category, action, data category, purpose, policy condition and obligation, where a user category is similar to a role and obligations are linked to rules implying that if the action is performed the obligation holds. All information in these fields has to be typed out by the user making the process tedious and less user-friendly. RIDE provides drop-down menus and dialog boxes populated with a list of valid values to offer possible input options to the user. The user can just click and choose the value to be entered in the corresponding field. This reduces the chance of errors and simplifies the process of policy creation. EPAL Editor does not benefit from the interoperability and scalability provided by ontology languages like RDFS [24], DAML+OIL [8] and OWL, because policies created are in XML. There is no support for describing domain-specific information, which restricts its usage considerably. Our work can be used for any environment ranging from pervasive computing systems to multi-agent systems with the provision it offers for entering domain-specific information. RIDE toolkit also allows users to create, modify and remove namespace templates, which are essentially means by which a user can aggregate domain-specific information and persist this information for future usage. This prevents the user from having to re-enter any such information that has already been stored.

Ponder is a declarative, object-oriented language for specifying policies in distributed systems [6]. It defines basic policies like access control policies and obligation policies to specify access control and management actions respectively. These are described by a binding between a set of subjects and a set of targets. It also defines composite policies, which provide the ability to group policies to reflect organizational structure or for reusability of common definitions, thereby allowing structured, reusable specifications. Ponder allows meta-policy specification to disallow simultaneous execution of conflicting policies or for specifying application specific constraints on sets of policies.

Ponder toolkit consists of a graphical domain browser, compiler framework, policy editor, management console tool and conflict analyzer for integrated policy management [7]. The graphical domain management tool allows browsing/updating policies and other objects in the domain. This is a useful feature, which is currently not supported by RIDE. Policy editor is used for creating/modifying policies. Though help is provided in the form of content assistance, users are expected to have a basic understanding of ponder programming language as they are expected to type out policies in the editor. On the other hand, RIDE has a user-friendly, wizard-based approach that automatically converts all user selections for rules and polices into their respective Rei specifications expressed using OWL. The user is not required to know either the specifics of Rei policy language or the semantic web language OWL. The compiler framework of Ponder toolkit transforms policies directly into XML or low-level representations suitable for the underlying system (by default Java code) that can be interpreted at runtime. This makes it tightly coupled to the environment. RIDE, on the other hand, generates all rules and policies in OWL to be reasoned over by the Rei Engine. It does not need to transform policies into any other representation, decoupling itself from the underlying environment. Their conflict analysis tool, integrated with the policy editor, is able to detect conflicts between policies of opposite modalities and detect inconsistencies between policies and external context [15]. This is another useful feature that is currently not supported by RIDE. However, it provides facilities for meta-policy specification over policies that are expected to conflict at run-time. Ponder provides a deployment model as well. After specifying the policy, it is compiled by the Ponder compiler into a Java class and then represented at runtime by a Java object. Authorization policies are enforced by Access Controller agents that allow or reject access requests to controlled target resources. Policy Management agents try to enforce obligation policies when the relevant events occur. Though Ponder provides the specification of the interfaces for these enforcement agents it does not provide any implementation.

Ponder toolkit is a well-developed tool support for integrated management of distributed environments. However, since Ponder policy language does not support run-time modification of policies, it limits Ponder toolkit from providing this feature. RIDE allows policy creation over speech acts that are use in Rei policy framework for run-time modification of policies. These include delegation, request, cancel and revocation [14] speech acts. A valid delegation leads to a new permission. Similarly, a revocation speech act nullifies an existing permission (whether policy based or delegation based) by causing a prohibition. An entity can request another entity for a permission, which if accepted causes a delegation, or to perform an action on its behalf, which if accepted causes an obligation. An entity can also cancel any previously made request, which leads to a revocation and/or a dispensation.

KAoS [22, 23] is a policy language based in OWL. It can be used to develop positive and negative authorization and obligation policies over actions. It allows conflict resolution and enforcement of policies within domains. It is similar to Rei and uses semantic web language OWL in representing domain information, application context and the policies. It provides the KAoS Policy Administration toolkit (KPAT) to facilitate policy management.

KPAT graphical user interface is organized as a set of views, namely, domain view, actor classes, policies, policy templates, policy disclosure, namespaces, configuration, ontology query and guard manager. Domain view allows creation and deletion of domains and policies. Actor class view shows the list of actor classes defined in the loaded ontologies and is used to create, modify, and delete policies for the selected actor class. Policies view shows a list of policies in the system and a hierarchy of policy sets. It can be used to discover and resolve policy conflicts, modify and remove policies, create, modify, and remove policy sets, and define membership in these sets. Policy Templates view shows the list of available policy templates, from which the user can create, modify or remove policies templates. Policy Disclosure view shows a list of policy disclosure queries that could be asked about the authorized or obligated actions by the policies in effect. Namespaces view allows loading ontologies, displaying a list of loaded ontologies and information about the selected ontology. Configuration view shows configuration of the Directory Service, which includes a list of loaded ontologies, domains, policies and policy sets, and user-defined classes and instances. Ontology Query view allows the user to query the ontology to ensure it is defined correctly. Guard Manager view shows the hierarchy of registered Guards, which are responsible for local policy enforcement.

KPAT and RIDE frameworks share a few similarities. Both approaches offer well-defined and user-friendly graphical interfaces for policy creation. They hide the complexity of their respective policy specification languages from the user and automatically generate policies using semantic web languages. Both approaches offer the notion of templates that can be created, modified or removed. These templates capture information most frequently or commonly required by the user. However, due to existence or absence of some features in either framework, there are both advantages and disadvantages of using one over the other. The advantages of KPAT are that it has features that can let users create domain-specific ontologies, browse, modify and delete them. RIDE does not provide this. We believe that this is not a required feature as Rei assumes all domain-specific ontologies could be created, browsed and modified outside the IDE using Protégé-2000 [21], SWOOP [12] and such other ontology editors. Secondly, KPAT GUI lets users browse through the created policy objects. Rei IDE stores rules, constraints and policy objects as OWL text files and has no GUI to browse through them. However, all text-based editing features are provided by the text editor in Eclipse. KPAT provides conflict detection between policies and facilitates policy enforcement, which RIDE does not support.

The advantages of using RIDE over KPAT for policy creation are that it lets users create policies over speech acts such as delegation, revocation, cancel and request as defined by Rei policy specification language. These allow policies to be dynamically modified. KAoS only models simple delegations thereby limiting KPAT that has no integrated support for dynamic policy management. Secondly, KPAT is a stand-alone and RIDE is developed as a plug-in for the Eclipse workbench. This allows it to be extended easily due to the extensible Eclipse framework. RIDE also provides support for formulating test-cases to test policy conformance, which is not offered by KPAT.

KeyNote [4] is a distributed trust management system that provides a specific language for representing policies, which govern the actions that principals (entities that can be authorized to perform actions) are authorized to perform. The language provides the semantics for describing actions, which are operations with security consequences that are to be controlled by the system and which are specified as a collection of name-value pairs. It is also used for specifying credentials, which allow principals to delegate authorization to other principals. In KeyNote, the application sends the engine a set of credentials, policies, public keys of the requester, and the set of actions. A KeyNote assertion is made of authorizer, licensees, conditions and signature. The licensees' field specifies the principals to which the authority is delegated and conditions' field is a set of tests of the action environment. Satisfying an assertion implies satisfying both the licensees and conditions fields and describes actions the holders of the public keys that signed that request are authorized to perform. As KeyNote requires all assertions, credentials and policy files to be specified in its native syntax, writing syntactically correct policy files can become a daunting task.

To overcome this problem, a policy toolkit called Policy-Editor [17] has been developed. It is based on the principle of using XML for policy specification in the form of an XML user policy file, which is transformed using an XML style sheet into the native KeyNote policy file format. Style sheets for transforming the policy file in XML to the KeyNote policy file format and a graphical web-based format are provided by the toolkit. It also consists of a Java-based graphical user interface, which has drop-down menus and dialog boxes collecting inputs from user to populate parameters of policy file.

Policy-Editor for KeyNote is an elegant solution in terms of using XML to hide the complexity of the KeyNote syntax, which is similar to our idea of auto-generating OWL files to hide the complexity of Rei syntax. However, KeyNote Policy-Editor is quite limited in terms of its capabilities as compared to our work. It can be used only to specify authorization policies, whereas RIDE not only lets users create authorization policies, but also obligation policies. RIDE is designed to use domain knowledge expressed using ontology languages. This makes it applicable to any kind of domain ranging from multi-agent systems to pervasive computing environments. Also, it is possible to specify constraints over attributes of requesters, actions, and the environment at different levels of abstraction through the graphical interface provided. This is not possible in KeyNote as they use a programming language for describing assertions. In KeyNote, delegation is controlled by a delegation depth and simple conditions on delegation. On the other hand, RIDE can be used to generate permissions to delegate as separate permissions. These permissions can have constraints specified not only on who can delegate, but also on whom they can delegate to.

Chapter 3

PROBLEM DESCRIPTION AND THESIS CONTRIBUTION

Policy-based governing frameworks for management of systems are gaining a lot of popularity. One of the significant benefits of using policy-based approach is improved flexibility in changing system behavior without changing its underlying implementation. Though a plethora of policy specification languages have been developed to address issues and requirements of fairly static environments such as security and trust management, management of network and distributed systems and privacy government, Rei is applicable also to open, dynamic distributed environments. Its usefulness is demonstrated by its successful deployment in a wide range of environments including pervasive computing environments, semantic web, grid computing and multi-agent systems. In order to facilitate policy management using Rei, need is evident to provide a policy management toolkit that hides the syntactic details of the language and simplifies the inherently error-prone and complex process of policy creation.

Main focus of this thesis is the development of an integrated policy management environment that provides easy-to-use graphical tools that automatically generate code for user-defined policies, provide abstraction from specifications of Rei policy language and means to verify correctness of the policies created.

3.1 Problem Domain Description

Management of policies can get quite complex in systems that are large-scale, open, dynamic, distributed or any combination of these. Though tools that have been developed to facilitate policy management, there is not much work done that singularly meets all the requirements of this wide range of environments that make use of policy-based approaches.

Characteristics of the environments to which policy-based government is applied outline features for policy management tools to support. Some of these requirements that highlight the inadequacies of existing tools are:

- Support for dynamic policy modification: For systems that are constantly changing due to variations in externally imposed constraints or environmental conditions, policy-based management could get more difficult. It is desirable in such cases to have a policy management toolkit that facilitates run-time modification of existing policies. This is required so that the system can constantly adapt itself to meet the new requirements. Using our tool it is possible to create policies over delegation, revocation, request and cancel speech acts that allow dynamic policy modification.

- Facilities for group policy specification: There could be a large number of resources and entities in the environments that make use of policy-based management. This could result in a potentially large number of policies governing the environment. This underlines the need for policy management toolkits to provide some features for defining policies for groups of entities. Our solution allows creation of group policies, which can be described over common characteristics of the entities and actions these entities want to perform over the resources.

- Ability to accommodate information spanning several domains: The environments under consideration generally span more than one domain and need to handle heterogeneous domain-specific information. It is very important that the policy management tool-support has the ability to handle any domain-specific information. Our tool is designed to accept domain knowledge expressed using ontology languages. This makes it easily extensible and usable for any domain-specific information without any change.

- Easy-to-use and easily extensible interfaces: Policy management frameworks that are applicable and extensible enough to support a wide range of environments usually have complicated policy specification languages and underlying mechanisms. A policy management toolkit that can hide all of this complexity from the user through a friendly user interface is highly desirable. It makes administration tasks simpler and reduces the chances of errors considerably. RIDE provides a wizard-based graphical user interface that is intuitive and user-friendly. It allows policy specification by providing a complete list of possible and valid values wherever applicable and automatically generates the OWL code for these policies.

  Constantly evolving systems with their changing requirements need the interfaces to be easily extensible. Our work is developed as an extension of the Eclipse framework and it is easily extensible.

- Comprehensive support for testing: With the possibility of creating a large number of policies and the ability to dynamically modify them; another important feature the user would want to have is to be able to verify the correctness of these policies. This is possible using RIDE. It has an interface with the Rei Engine that allows users to check the correctness of individual policy modules.

The novel aspect of our work on an Integrated Development Environment for Rei Policy Specification Language lies in the fact that it has been designed to support all these requirements. Yet, there has been no compromise on simplicity and ease-of-use aspects of the user interface.

3.2 Thesis Contribution

The following is a list of contributions made by this work:

1. The main contribution of the RIDE toolkit is the graphical wizard-based policy specification tool. It provides user interface to create rules, constraints, granting objects, policies and meta-policies. Rules are permissions, obligations, prohibitions and dispensations as described by Rei. For more detailed description on these, please refer to section 4.1. Constraints are specified over the entities and actions in the domain and allow evaluation of the conditions under which the rules apply. The wizard allows creation of granting objects to add additional constraints to existing rules. Policies are created by combining any number of existing rules, granting objects and a context. The context is a set of constraints used to define the environment under which the policy is applicable. RIDE allows security policies to be defined using permission and prohibition rules. Management policies can be defined using obligations and dispensations. It can also create conversation policies. The order in which speech acts occur is called a conversation. The order is a domain specific state and a log of the speech acts that have occured so far. If appropriate domain ontologies are developed to represent speech acts like reply, query, deny and the like, then RIDE can be used to create Rei conversation policies over them.

   A complete list of possible and valid input options for various fields involved in the process of rule, constraint and policy creation are presented to the user using drop-down menus and dialog boxes. The user can just click and choose the value to be entered in the corresponding field. This 'click-and-choose' feature simplifies the process considerably. These user-defined constructs are automatically converted into their appropriate OWL representations as per the specifications of Rei. This automates the laborious, complex and error-prone process of policy generation for any domain and keeps the user away from having to learn

OWL or Rei specifications. Individual and group policies could be created using the wizard. Its support for policy creation over speech acts facilitates dynamic policy modification.

2. The other contribution of RIDE toolkit is an integrated front-end to Rei Engine, providing an interface for querying and policy testing. It is a wizard that allows formulation of test-cases that verify correctness of the policies created through the graphical wizard-based policy specification tool. It involves querying the Rei Engine for an answer to the test-case and computes its result by comparing the answer from Rei Engine to the expected answer for the test-case.

3. As a part of the wizard-based policy specification tool, a namespace management tool is provided that allows users to enter domain-specific information. This is achieved by use of ontologies to specify domain knowledge. It also has support for creation, modification and removal of namespace templates. These templates are used to save and re-use namespace information. It is provided to make the tool user-friendlier. Namespaces are essentially URLs that are locators of the ontology files that capture the domain information. This makes our tool scalable and capable of handling heterogeneous information.

Design and implementation of this work build upon our experience gained through the earlier two approaches of developing a graphical user interface for Rei. The first approach involved creating a tab-widget plug-in extension for Protégé-2000 ontology editor. It provided such features as loading and browsing the domain-specific ontologies through Protégé-2000's existing interface and allowed creating policies, rules and meta-policies through features provided by our extension. Figure 3.1 and Figure 3.2 show some of its interfaces.

Though, this approach provided a simple and convenient way for specifying Rei policies, rules and meta-policies, it was not as intuitive and easy-to-use as wizard-based approaches generally are. Furthermore, it was also limited by the limitations in

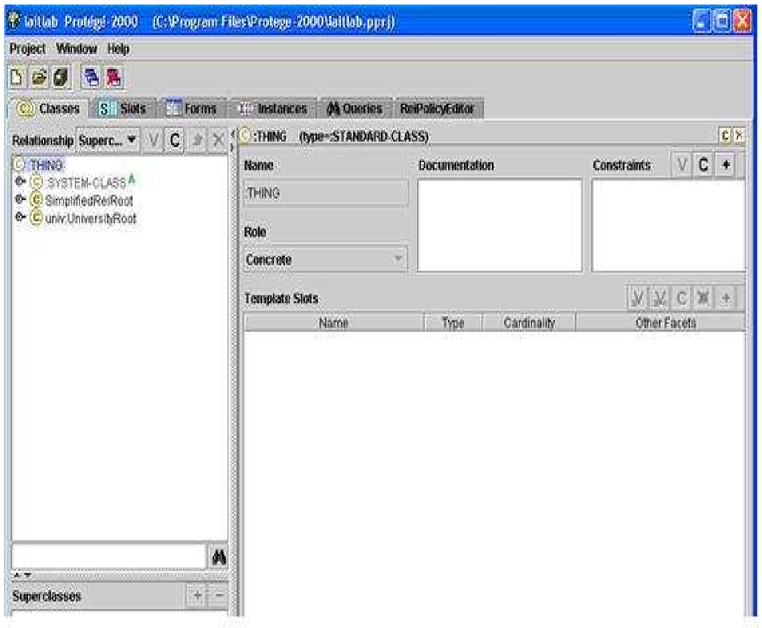


FIG. 3.1. Rei Policy Editor Using Protégé-2000

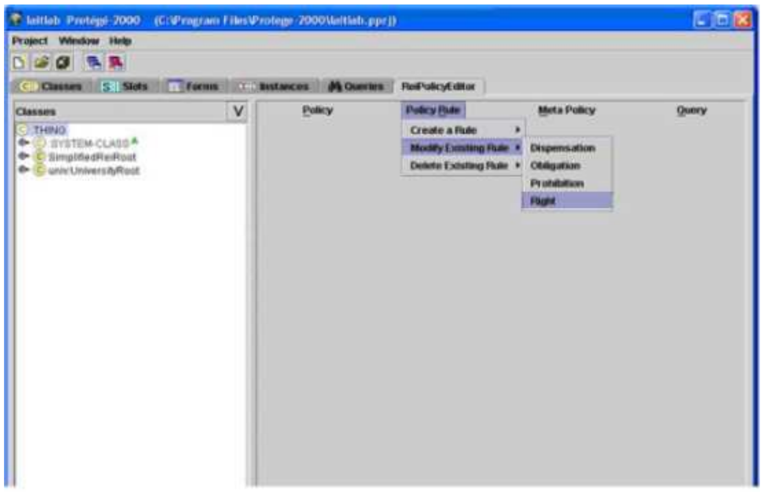
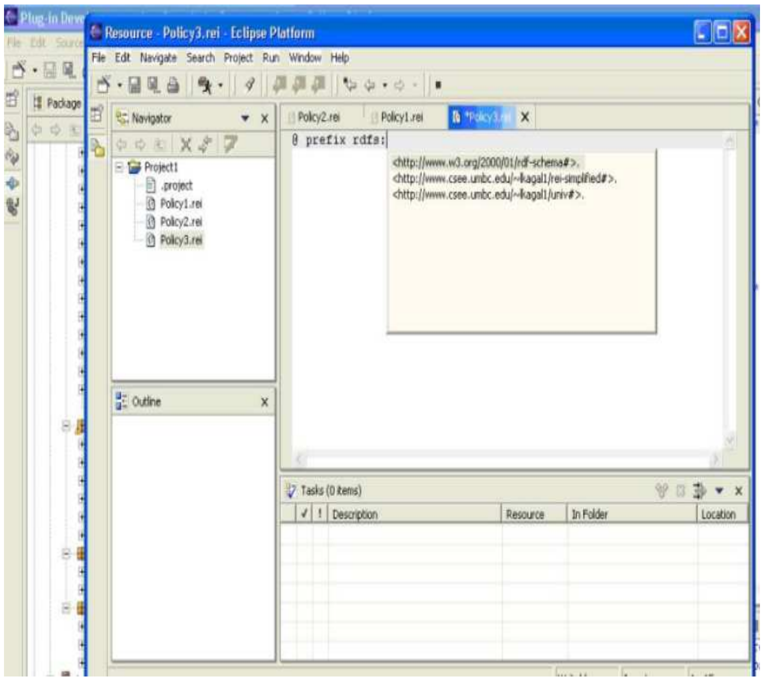


FIG. 3.2. Policy Creation Using Rei Policy Editor

design of the protégé-2000 ontology editor such as multiple dialog boxes opening up when asking for details of each field in the policy creation process that could confuse the user considerably.



```
@ prefix rdfs:
<http://www.w3.org/2000/01/rdf-schema#>.
<http://www.csee.umbc.edu/~lkagal1/rei-simplified#>.
<http://www.csee.umbc.edu/~lkagal1/univ#>.
```

FIG. 3.3. Rei Editor's Text Interface Showing Content Assist Feature

Our next approach was focused around developing a text-based editor for specifying policies in the Notation3 (N3) [3] representation. N3 is the XML syntax for RDF and makes policies more readable. Policies written in N3 can be easily translated into other semantic web languages like RDFS, OWL. The N3 text editor was based on providing content assistance and context information to the users while they type out Rei Policies. Figure 3.3 shows one if its interfaces.

As we developed the interface using this approach, we realized that it would not be as user-friendly as a wizard-based approach would. Also, it expected our users to be familiar with N3, which could have severely limited its usability.

Our efforts in building user interface to aid policy development and maintenance clearly emphasize our conformance of the principles and concepts outlined by Human

Computer Interaction [11]. These include requirements analysis to support design, prototyping, iterative evaluation and iterative refinement of the interface to make it as simple and useful as possible from user's perspective.

3.3 Scope of Thesis

This thesis primarily limits its scope to using Rei policy specification language to define policies. Design and implementation of RIDE are therefore influenced greatly by concepts defined by Rei.

RIDE allows creation of only a subset of all possible Rei constructs. Rei grounded in first order logic allows specification of a range of relations like role value maps using logic-like variables. For example, it is possible in Rei to describe *uncle of* and *same group as* relations. It also allows creation of complex constructs for entities in the domain such as objects with start or ending constraints that are not possible to express using RIDE's graphical interface. The latter, however, allows a majority of the Rei defined constructs to be specified easily.

The namespace management tool of RIDE requires users to input all the domain specific information using namespace URLs that correctly point to the ontology files created to capture all the required information. There is no support provided currently to create these ontology files containing domain terminology and instance data. We believe that this is not a severe limitation as there are several ontology editors available that could be used for this purpose. Also, RIDE expects the first namespace to point to the terminology file containing a class called "Action" that represents directly or through inheritance all domain actions. Entities (Actors) and resources in the domain are all classes other than the Action class and its sub-classes. Following the first ontology file, there could be any number of other ontology or instance data files.

Chapter 4

RIDE FRAMEWORK AND POLICY TOOLKIT

This chapter gives the relevant details of Rei policy specification language, Eclipse framework, Model-View-Controller architecture and Jena toolkit that contribute to the design and implementation of our work. It also discusses in detail, different features and functionalities provided by RIDE.

4.1 Rei Policy Specification Language

Rei is a policy specification language defined in OWL-Lite with general specifications for policies as well as mechanisms for policy verification, analysis and testing. Rei defines a policy as a set of rules describing deontic concepts like permissions, prohibitions, obligations and dispensations over actions in the environment with respect to the actor, the action and the context. It models speech acts for remote policy management like delegation and revocation that affect permissions and prohibitions, and request and cancel that affect obligations and dispensations. It defines meta-policies to resolve any conflict that may arise between enforced policies. It provides specifications and tools for policy analysis like use-case analysis where certain optimal situations are described and what-if analysis where a policy maker can test out changes to the policy and check their effect before committing them. All these concepts are specified using ontologies, called 'domain-independent ontologies', written in OWL. The use of the semantic language OWL enhances its interoperability and extensibility. Associated with the language is a policy engine that can be used within the application domain to interpret and reason over policies, help resolve in case of conflicts between policies and answer queries related to policy making.

A short description of each of these Rei specified concepts along with an example demonstrating their corresponding OWL representations follows. More detailed explanation of all Rei concepts and terms can be found in [14, 13].

4.1.1 Action

Actions in Rei allow capture of contextual information and are of two types: Domain Actions and Speech Acts. Rei Action ontology describes general properties, some of which being actor (performer of the action) and location (location where action occurred) inherited by its Domain Action and Speech Act subclasses.

Domain Action : This subclass includes additional properties for application specific actions. These include target (object on which action is performed), precondition (condition that must hold before action can be performed) and effect (condition that occurs after action is performed).

For example, where the domain is, say, CS department in some university and we want to specify an action that uses the printers in that department, the OWL representation would be as follows:

```
<owl:Class rdf:ID="CSPrinting">
  <rdfs:subClassOf rdf:resource="&action;DomainAction"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&action;target" />
      <owl:allValuesFrom rdf:resource="#CSPrinter" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Speech Act : This subclass specifies additional properties for dynamic and remote policy management. These include sender (specifies sender of the speech act), receiver (recipient of the speech act), condition (constraints added by the sender) and context (actual message, either action or deontic object to be sent). It is further subclassed into Delegate (leads to a new permission), Revoke (nullifies an existing permission by causing a prohibition), Request (specifies a request for a permission, which if accepted causes a delegation, or to perform an action, which if accepted causes an obligation) and Cancel (cancels any previously made request creating a revocation and/or a dispensation).

As an example of a speech act, consider a delegation action from Susan to Bob, giving him the permission to perform a printing action in the lab.

```
<action:Delegation rdf:ID="Del_SusanToBob">
  <action:sender rdf:resource="# Susan "/>
  <action:receiver rdf:resource="# Bob "/>
  <action:deontic>
   <deontic:Permission>
    <deontic:actor rdf:resource="# Bob "/>
    <deontic:action rdf:resource="#LabPrintingAction"/>
   </deontic:Permission>
  </action:deontic>
</action:Delegation>
```

4.1.2 Constraint

Constraints are used to define a set of objects that satisfy or possess certain property or a set of properties. It has two subclasses, SimpleConstraint and Boolean-Constraint.

Simple Constraint : It defines properties like subject (entity the constraint is about), predicate (the property of the entity) and object (value of the property).

The following is an example of a SimpleConstraint that defines that an action is of type CSPrinting action.

```
<constraint:SimpleConstraint rdf:ID="IsCSPrinting"
  constraint:subject="#var"
  constraint:predicate="&rdf;type"
  constraint:object="&univ;CSPrinting"/>
</constraint:SimpleConstraint>
```

Boolean Constraint : Rei allows SimpleConstraints and BooleanConstraints to be combined in pairs to form a BooleanConstraint. This pairing is achieved using the And, Or and Not operators. Properties defined are first (first constraint, either Simple or Boolean) and second (second constraint in the pair, either Simple or Boolean).

The following is an example of a Boolean Constraint that defines that the CSPrinting action can only be performed by graduate students.

```
<constraint:And rdf:ID="GradStudentConstraint">
  <constraint:first rdf:resource="#IsGraduateStudent" />
  <constraint:second rdf:resource="#IsCSPrinting" />
</constraint:And>
```

Where, the Simple Constraint ”IsGraduateStudent” is defined as:

```
<constraint:SimpleConstraint rdf:ID="IsGraduateStudent">
  <constraint:subject rdf:resource="#var" />
  <constraint:predicate rdf:resource="&rdf;type" />
  <constraint:object rdf:resource="&univ;Graduate" />
</constraint:SimpleConstraint>
```

4.1.3 Deontic Object

Deontic objects correspond to rules in the policy domain. This class as defined in the Rei ontology has properties such as actor (actor or set of actors over whom the rule applies), action (action or set of actions that the rule is described over) and constraint (conditions that must be satisfied to make the rule applicable). It has four subclasses: Permission, Prohibition, Obligation and Dispensation. Permission is used to specify an action (set of actions) allowed for an actor (set of actors). Prohibition describes an action (or set of actions) that the actor (or set of actors) is prohibited from performing. Obligation is used to specify an action (or set of actions) that the actor (or set of actors) is obliged to carry out. Dispensation is a waiver for an obligation. It is used to negate or free the actor from an obligation.

The following is an example that describes that Susan has the permission to perform CSPrinting action. Prohibition, obligation and dispensation can be similarly described.

```
<deontic:Permission rdf:ID="Permission_Susan">
  <deontic:actor rdf:resource="#Susan"/>
  <deontic:action rdf:resource="#CSPrinting"/>
</deontic:Permission>
```

4.1.4 Policy

A policy as defined by the Rei ontology consists of a list of rules (deontic objects) and a context (Simple or Boolean Constraint) to guide the behavior of entities in the policy domain. It can also include a set of conflict resolution specifications. This class defines properties such as context, grants, granting, defaultBehavior, defaultModality, metaDefault, rulePriority and imports. An explanation of each of these properties can be obtained from [14].

A simple example of a policy for the CS Department that states that all members of that department can perform CSPrinting action can be represented as follows:

```
<policy:Policy rdf:ID="CSDeptPolicy">
  <policy:actor rdf:resource="#var" />
  <policy:action rdf:resource="#CSPrinting" />
  <policy:context rdf:resource="#members_of_CSDept" />
  <policy:grants rdf:resource="#granting_right_to_print" />
</policy:Policy>
Where, the granting object "granting_right_to_print" is defined as:
<policy:Granting rdf:ID="granting_right_to_print">
  <policy:to rdf:resource="#var" />
  <policy:deontic rdf:resource="#right_to_print" />
</policy:Granting>
```

OWL representations of the Rei constructs described in examples above are provided solely as a means to facilitate understanding of the language specifications. The user, however, is not required to know them as they are automatically generated by RIDE.

Rei provides specifications called Meta-policies for conflict resolution. It provides default meta-policy specification and meta-policies over conflicting rules and policies. Current implementation of RIDE only supports meta-policy specification over conflicting policies. Rei also provides policy analysis tools to enable the development of consistent and valid policies. Current implementation of RIDE does not support this.

4.2 RIDE Framework Architecture

RIDE has been developed by extending the Eclipse framework to provide a wizard-based graphical user interface. It embodies the Model-View-Controller architecture in building this user interface. Wizard-user interaction to get inputs from the user and the automatic OWL code generation as the output are made possible by making use of the Jena toolkit. A description of relevance and contribution of these tools and concepts to the design and implementation of RIDE follows. We also describe the capabilities and features of RIDE toolkit.

4.2.1 Eclipse Platform

Eclipse Platform is an IDE that provides building blocks and a foundation for constructing and running integrated software-development tools [18]. It is open source, extensible and specialized for building integrated development environments. It has therefore been our choice in developing an IDE for policies. RIDE toolkit extends the Eclipse platform by wrapping itself into a pluggable component; called Eclipse plug-in. Eclipse runtime provides the necessary infrastructure to support the activation and operation of all plug-ins. RIDE is activated when a user invokes the "New" menu-item in the Eclipse workbench and chooses "Policy" to be the new resource to be created. This launches a wizard that takes users through the various steps in policy creation and at the end, automatically generates policy files written in OWL based on user selections and entries. Once a policy file is created, users can test policies by invoking a separate wizard. This wizard acts as an integrated front-end to the Rei engine and provides an interface for querying the engine and policy testing.

The complete User Interface (UI) has been developed using Standard Widget Toolkit (SWT) and JFace toolkit. This includes the wizards and the widgets these wizards contain for interacting with the users. SWT provides an OS-independent API for widgets and graphics. It is analogous to AWT/Swing in Java. It is implemented in a way that allows tight integration with the underlying native window system. JFace is a UI toolkit implemented using SWT. It provides API for implementing many common UI programming tasks. The wizard framework and dialogs in RIDE have been developed using the JFace API. All the widgets that appear on the wizard pages have been developed using the SWT API.

4.2.2 Model-View-Controller Architecture

Many GUI-based applications use the Model-View-Controller (MVC) architecture as a primary design pattern to present, manipulate, and store data for the end users. It was first developed using the Smalltalk programming environment for creating user interfaces. It decouples presentation from data and from operation on that data. Model, in the MVC architecture, stores data that defines the components in the GUI. View creates the visual representation of the components. This is done from the data in the model. Controller deals with the interaction of the user with the interface and modifies the model and/or view in response to user actions as necessary. In user interfaces, the views and controllers often work very closely together. For instance, the user interactions with a view require controller to make necessary modifications to the model, which in turn updates that particular view or any of the other views. For this reason the view and the controller are typically represented by a single composite object that corresponds with a view and an integrated controller. RIDE framework follows the MVC pattern in its design and implementation of the user interface.

Implementation primarily consists of the following classes that help realize the user interface as per the MVC paradigm.

- PolicyRuleModel.java : This class represents the model in the RIDE framework.

  In a wizard-based GUI, wizard pages essentially act as the views for user interaction. The RIDE policy creation wizard consists of the following two wizard pages.

- PolicyNamespacePage.java : This class represents one of the view-integrated controller pairs in the RIDE framework. It contains necessary methods for accepting namespace URLs from the user and storing them in the model for further computational requirements.

- PolicyCreationPage.java : This class represents one of the other controller-view pairs in the framework. It consists of nested views that appear as tabs on the wizard page. The respective classes that represent these tabs are ActorTab.java, DeonticLiteralTab.java, ActionTab.java and PolicyTab.java. These classes together handle user interactions for policy creation and update each other through the model based on user actions.

The policy test wizard consists of the following wizard page.

- PolicyUnitPage.java : This class represents a controller-view pair in RIDE framework and contains logic to allow users to test policy conformance. The interface allows test-case creation to verify if the individual policy units work as desired. It provides an interface to the Rei engine to achieve test-case verification.

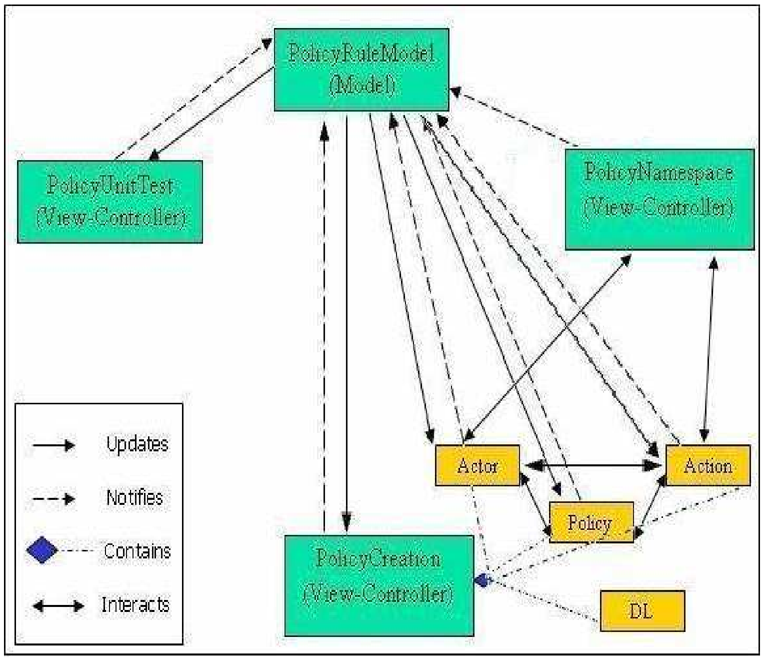


FIG. 4.1. RIDE Framework Using MVC Architecture

The way the model and views interact with each other is by establishing a subscribe/notify protocol between them. The model in our framework has setter methods to store data (user inputs) collected by the different views. It provides getter methods for access of this data by the different views. Views subscribe themselves as observers of the data objects stored within the model. The model in turn notifies them whenever there is any change in the data objects held by it. In response, the corresponding view affected by that data change gets an opportunity to update itself. This ensures that the view's appearance reflects the model's state.

The decoupling between the model and the views and subscribe/notify protocol existing between them depicts the behavior described by the Observer design pattern [9]. The intent of this design pattern is to define a one-to-many dependency between objects so that when object changes state, all its dependents are notified and updated automatically. This occurs without requiring the changed object to know the details of all the others. This is amply demonstrated in our design and implementation. For instance, PolicyNamespace wizard page, the Actor tab and the Action tab know nothing about each other. However, by interacting through the PolicyRuleModel they behave as though they do. When user enters namespaces in the namespace wizard page, the latter calls setter methods in the model to save this information. In response, the model immediately notifies the Actor and Action tab views, which are registered as observers of this data in the model. This ensures that any change in the namespace wizard page automatically reflects itself in the Actor and Action tab views. Multiple such dependencies exist between the different views in our framework as shown in Figure 4.1. The decoupling of the model and the different views also enables the RIDE framework to achieve greater flexibility and reuse. This is demonstrated by reusing the PolicyRuleModel class for the policy creation as well as the policy test wizards.

4.2.3 Jena Toolkit

Based on user inputs in the namespace view of the policy creation wizard as described in the earlier section, the RIDE framework makes use of the Jena toolkit to populate some of the widgets in the Actor and Action views. The Jena semantic web toolkit is a Java Application Programming Interface (API) and software toolkit for manipulating Resource Description Framework (RDF) models [19]. Jena Ontology API provides the necessary means for accessing and manipulating ontologies represented in RDF-S or OWL semantic web languages.

Namespace URLs entered by the user essentially point to the domain dependent ontologies expressed in RDF-S or OWL. Using the API provided by Jena toolkit, these ontologies are accessed and parsed to extract all the necessary information from them. This information includes a list of actors in the domain, which populates a drop-down menu for the actor's field in the Actor tab of the policy creation page. A list of all possible domain actions and speech acts to populate a drop-down menu for the action field in the Action tab of the policy creation page. A list of property-value pairs defined in the domain ontologies is also extracted to populate similar drop-down menus for the constraint's field in the Actor and Action tabs. This field enables creation of constraints on actors and actions in the domain. The information extracted from these domain-dependent ontology files using Jena toolkit to populate the drop-down menus for the fields in several views is a means to offer a list of valid and possible input options to the user. This information is persisted by the model and updated when there is any change in the ontologies loaded through the namespace view.

4.3 RIDE Policy Toolkit

RIDE Policy toolkit is a wizard-based graphical, user-friendly and extensible tool support for policy management in a wide range of environments. Its main goal is to be able to singularly meet the demanding requirements of the wide range of application domains that could benefit from policy-based management. In doing so, it has however made no compromise on ease-of-use and simplicity aspects of the user interface.

RIDE allows creation of rules based on the notion of permissions, obligations, revocations and dispensations as described by Rei. It allows creation of constraints, which are described over actors and actions in the domain. Using RIDE, policies can then be easily defined as contextually constrained rules in the domain. Policy creation over speech acts for dynamic policy management, comprehensive support for testing and ability to express policies over groups of actors and actions in the domain are among some of the other useful features of RIDE toolkit.

This section discusses the policy creation process using RIDE toolkit in detail. It covers specifics of tools that RIDE toolkit comprises of. The separation and division of the RIDE toolkit into these tools is only logical based on the functionalities each tool offers. Physically, each tool is a part of the wizard-based user interface.

4.3.1 Namespace Manager

Policy management using RIDE requires domain specific information to be represented using ontologies in OWL or RDF. Conceptually, the ontologies used in policy creation and management are classified as domain dependent (or domain specific) and domain independent. Domain dependent ontologies capture domain related information such as classes of actors, actions and resources, specific instances of these classes, as well as properties and their values associated with these classes. The domain independent ontologies include those defined by Rei and some standard ontologies defined by W3C for semantic web languages. All ontologies are referenced by RIDE using namespaces expressed as prefix, URL pairs. URLs are used to fetch the ontology files from their respective locations and prefixes are used in OWL code automatically generated by the toolkit.

Given this importance of namespaces in policy development, RIDE toolkit provides comprehensive namespace management support through its namespace manager tool. Its layout reflects the conceptual separation of ontologies into domain dependent and domain independent ontologies.

Properties of the namespace manager tool that describe its functionalities follow. A demonstration of the tool at work is also provided with the help of screenshots.

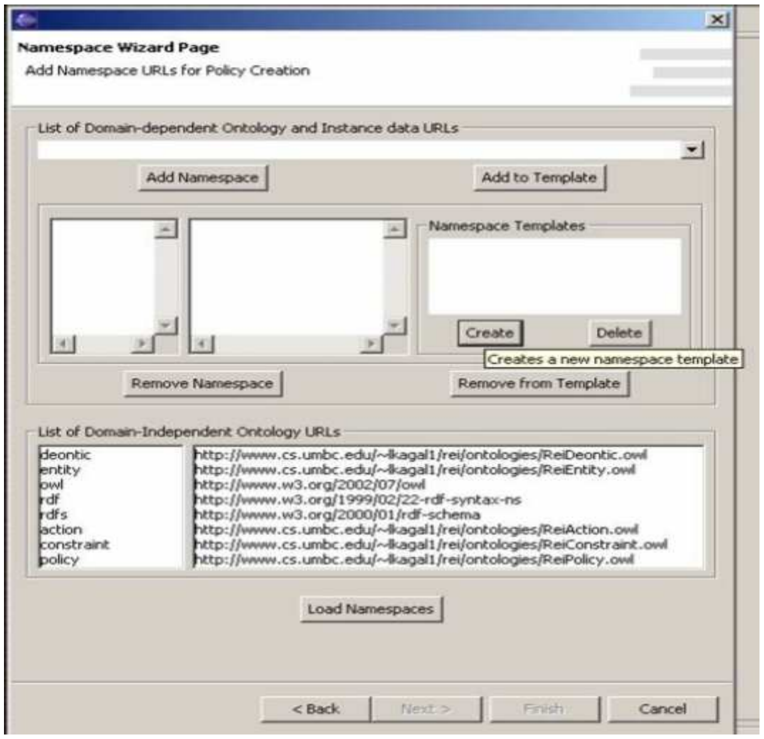
Ease of Management of Domain Independent Information : Namespace manager furnishes the necessary information about domain independent ontolgies to keep the user away from having to enter this information. Rei defined namespace prefixes include policy, metapolicy, action, entity, constraint, deontic and analysis.

Namespace prefixes defined by W3C include those for RDF, RDF-S and OWL semantic web languages. A list of all these prefixes and their respective URLs is always displayed on the manager's wizard page and is non-editable. Any namespace a user might need beyond the ones provided by this list, could be entered using widgets provided to enter domain dependent ontologies.

Namespace Loading using Templates : Since, domain related information cannot be presupposed and also as a means to allow greater flexibility in modifying domain specific information as per user's requirements; namespace manager expects users to enter this information. Entering namespace information can get tedious due to long declarative URL descriptions. To facilitate this process, namespace manager offers the notion of namespace templates.

Namespace templates could be thought of as holders of namespaces (prefix and URL pair information) belonging to a particular domain for which the template was created. Thus, if a user needs to enter information spanning several domains, a template could be created to represent each domain. Any number of such templates can be created or deleted using the namespace manager. Also, any number of namespaces can be added to or deleted from the existing templates. Once templates are created with namespaces stored under them, namespace manager takes care of persisting this information and maintaining its correct hierarchy. This information does not have to be re-entered and is reloaded during any subsequent usage of the wizard. Therefore, information is entered once, but can be modified any number of times and the reloaded information always reflects the last modification made.

For example, a user can create a template named University representing university domain related information. Another template named CSDepartment representing a specific department in the university can be created. Namespaces can be added to their respective templates. Figures 4.2 through 4.5 show the process of template creation, template deletion, namespace addition to a template and namespace deletion from a template using namespace manager.



Namespace Wizard Page

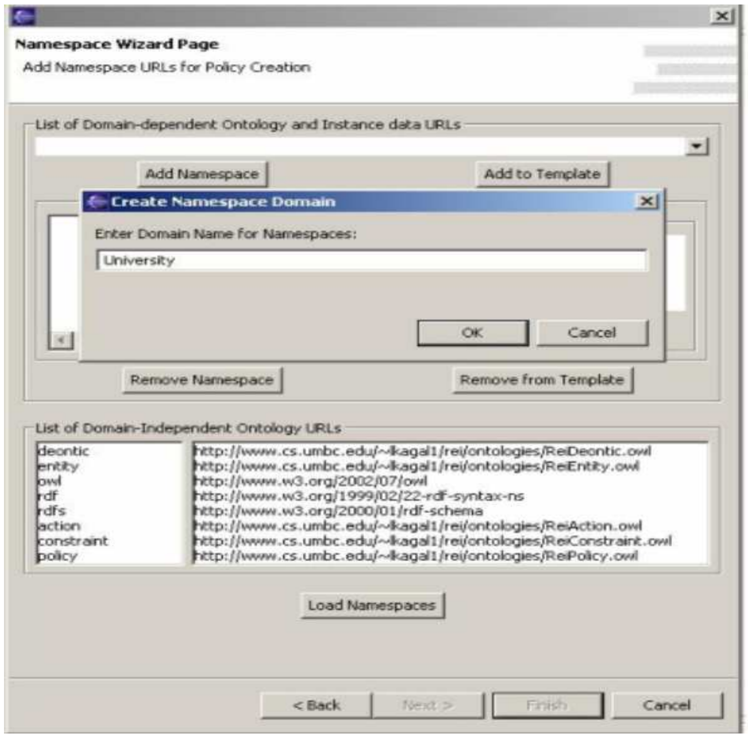
Add Namespace URLs for Policy Creation

List of Domain-dependent Ontology and Instance data URLs

List of Domain-Independent Ontology URLs

deontic	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiDeontic.owl
entity	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiEntity.owl
owl	http://www.w3.org/2002/07/owl
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns
rdfs	http://www.w3.org/2000/01/rdf-schema
action	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiAction.owl
constraint	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiConstraint.owl
policy	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiPolicy.owl

FIG. 4.2. Namespace Manager



Namespace Wizard Page

Add Namespace URLs for Policy Creation

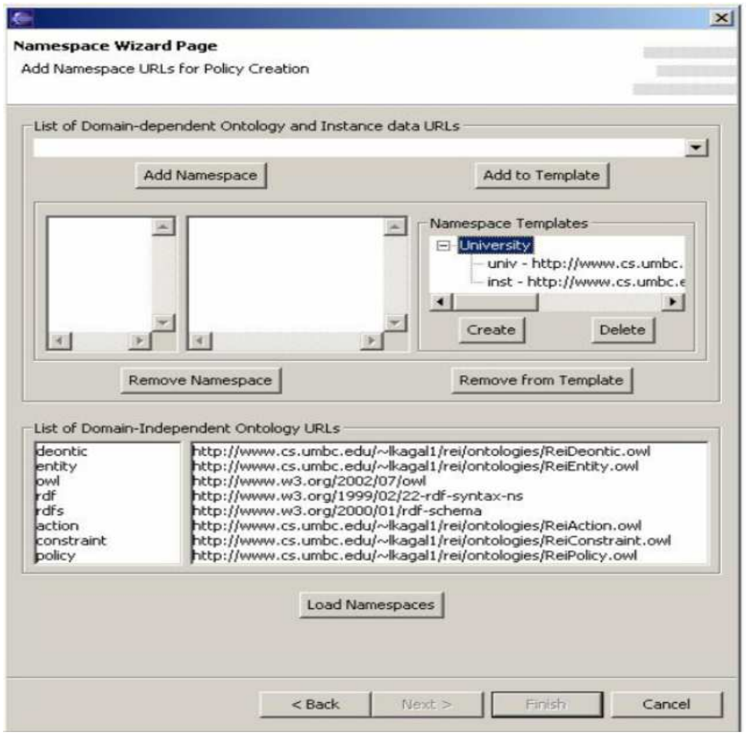
List of Domain-dependent Ontology and Instance data URLs

Enter Domain Name for Namespaces: University

List of Domain-Independent Ontology URLs

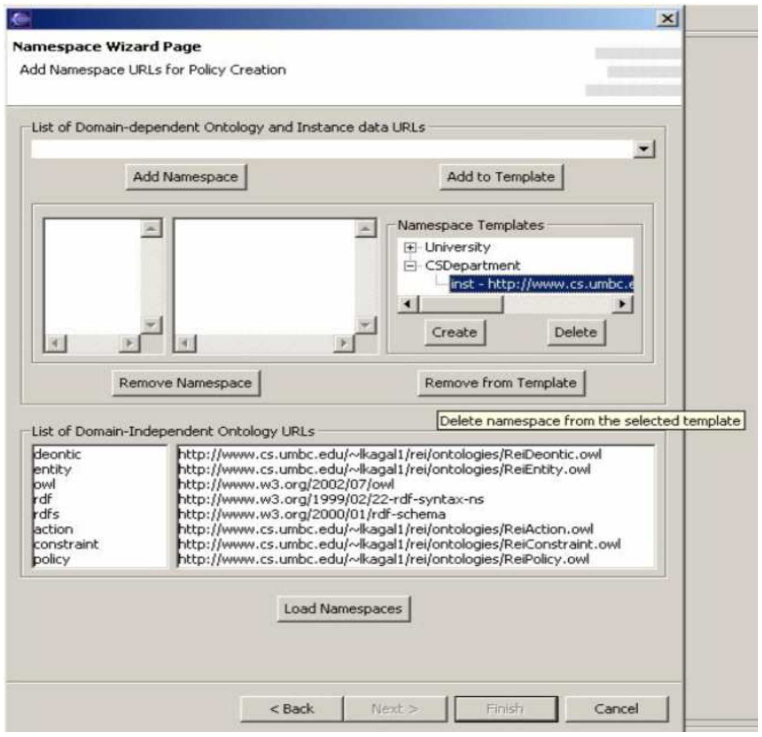
deontic	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiDeontic.owl
entity	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiEntity.owl
owl	http://www.w3.org/2002/07/owl
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns
rdfs	http://www.w3.org/2000/01/rdf-schema
action	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiAction.owl
constraint	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiConstraint.owl
policy	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiPolicy.owl

FIG. 4.3. Template Creation Using Namespace Manager


Namespace Wizard Page
Add Namespace URLs for Policy Creation
List of Domain-dependent Ontology and Instance data URLs
Add Namespace
Add to Template
Namespace Templates
University
univ - http://www.cs.umbc.
inst - http://www.cs.umbc.e
Create
Delete
Remove Namespace
Remove from Template
List of Domain-Independent Ontology URLs
Load Namespaces
< Back
Next >
Finish
Cancel

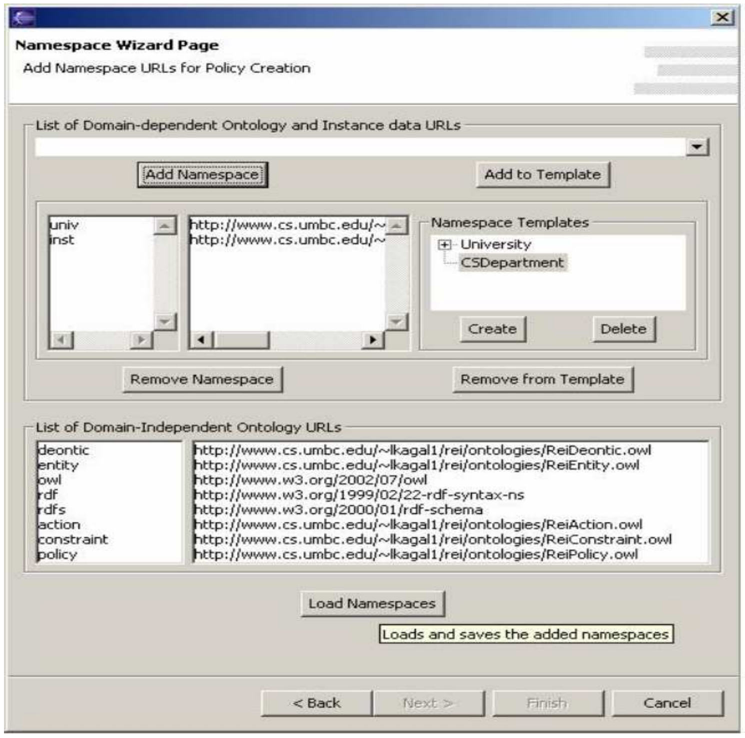
deontic	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiDeontic.owl
entity	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiEntity.owl
owl	http://www.w3.org/2002/07/owl
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns
rdfs	http://www.w3.org/2000/01/rdf-schema
action	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiAction.owl
constraint	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiConstraint.owl
policy	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiPolicy.owl

FIG. 4.4. Namespace Addition to Template


Namespace Wizard Page
Add Namespace URLs for Policy Creation
List of Domain-dependent Ontology and Instance data URLs
Add Namespace
Add to Template
Namespace Templates
University
CSDepartment
inst - http://www.cs.umbc.e
Create
Delete
Remove Namespace
Remove from Template
Delete namespace from the selected template
List of Domain-Independent Ontology URLs
Load Namespaces
< Back
Next >
Finish
Cancel

deontic	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiDeontic.owl
entity	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiEntity.owl
owl	http://www.w3.org/2002/07/owl
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns
rdfs	http://www.w3.org/2000/01/rdf-schema
action	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiAction.owl
constraint	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiConstraint.owl
policy	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiPolicy.owl

FIG. 4.5. Namespace Deletion from Template



Namespace Wizard Page

Add Namespace URLs for Policy Creation

List of Domain-dependent Ontology and Instance data URLs

univ	http://www.cs.umbc.edu/~
inst	http://www.cs.umbc.edu/~

List of Domain-Independent Ontology URLs

deontic	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiDeontic.owl
entity	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiEntity.owl
owl	http://www.w3.org/2002/07/owl
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns
rdfs	http://www.w3.org/2000/01/rdf-schema
action	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiAction.owl
constraint	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiConstraint.owl
policy	http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiPolicy.owl

Loads and saves the added namespaces

FIG. 4.6. Direct Namespace Loading

Direct Namespace Loading : There is also provision for entering namespace information of domain dependent information without adding it to templates. This serves user's need to include domain specific information that need not be persisted. This possibility can arise if user expects frequent changes in the namespace URL information and prefers such information be added anew each time. The namespace manager processes domain dependent information entered through both the means, directly and through templates to compute information for the next set of wizard pages. However, it expects namespace information be provided by at least one of the two means and does not allow the user to proceed to the next wizard page if this information is missing. Figure 4.6 demonstrates these features of namespace manager tool.

4.3.2 Policy Creation User Interface

After entering namespaces using the namespace manager tool, the next step in policy creation process using RIDE toolkit is creation of rules, constraints and granting objects, which combine together to form policy objects. Policy creation wizard launches the policy creation page after the user clicks 'Next' on the namespace management page. The earlier consisting of nested Actor, Deontic Literal, Action and Policy tabbed pages provides a comprehensive and user-friendly policy creation interface.

Using the tabbed pages of the interface, policy creation is an incremental process that entails prior creation of rules, constraints and granting objects. A description of creation process of each of these constructs follows.

Rules

Rule creation begins by selecting an actor on the Actor tab page from a drop-down list consisting of actors in the domain. An action can be similarly selected from the Action tab page using a drop-down list that contains all the domain actions. This involves rule creation with a specific instance of actor and a specific instance of action (e.g. 'Susan' is obliged to perform 'AStudentPrintAction'). If the user wishes to create rules not containing specific instances of actors or actions, but over classes of actors or actions to create group policies (e.g. 'All GraduateStudents' are permitted to perform 'PrintingActions'), it is possible without much change in the rule creation process. The only change is that entries for actor or action fields on their respective pages should be strings not selected from drop-down lists on these pages. Internally, OWL code for the rules generated by RIDE toolkit uses 'var1' as the string to denote group of actors and 'var2' as the string to denote a group of actions. The Deontic Literal tab page provides a drop-down list of deontic literals or modalities to be chosen from for the rule under construction. The selection made from this list decides if the chosen actor or group of actors on the Actor tab page can, cannot, must or need not perform the chosen action or group of actions on the Action tab page. Based on this selection, the rule under construction would respectively be termed a permission, prohibition, obligation or dispensation as defined by a deontic object in Rei.

After making these selections from the Actor, Deontic Literal and Action tabbed pages, the rule creation process can be completed with the 'Add Rule' button on the Actor tab page. This will add the rule to a list of user-defined rules stored in the model, which is described in detail in section 4.2.2. There is also provision for deleting rules using the 'Delete Rule' button that deletes selected rule(s) from the list maintained by the model. Figures 4.7 through 4.10 show the rule creation process using policy creation user interface of RIDE toolkit.

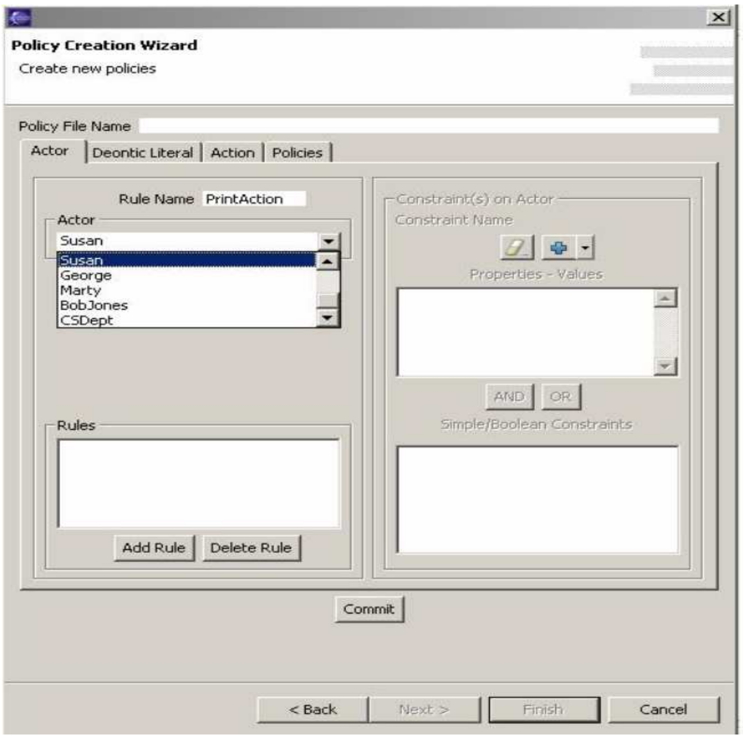


FIG. 4.7. Actor Selection for Rule Creation

Speech Acts

Speech acts, as defined by Rei, are created to dynamically modify already existing rules. RIDE allows speech acts to be easily created using a process very similar to

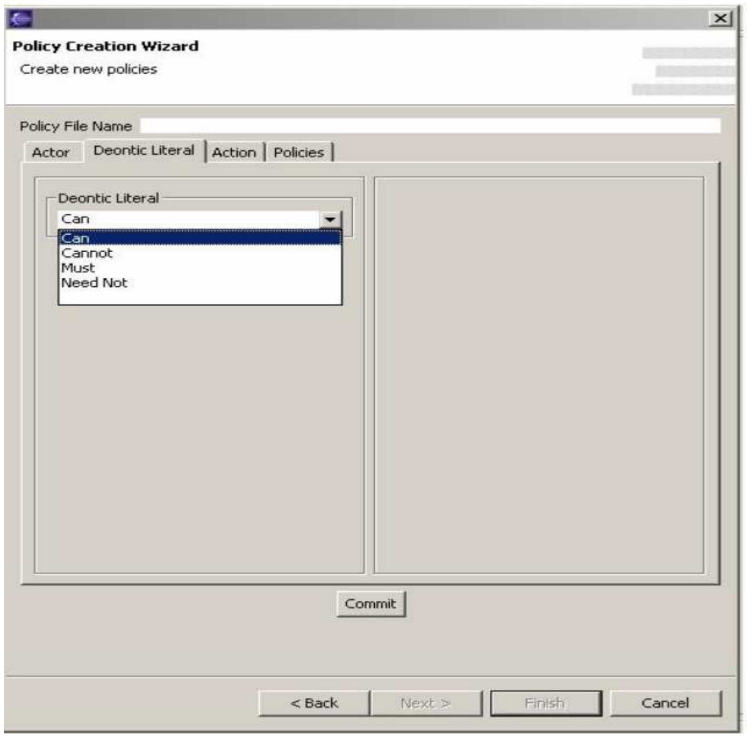


FIG. 4.8. Modality Selection for Rule Creation

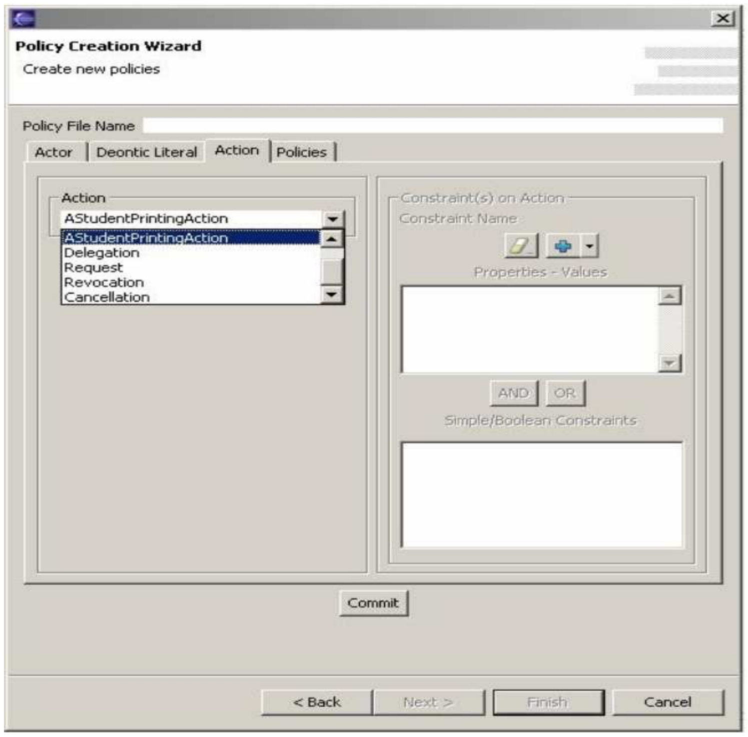


FIG. 4.9. Action Selection for Rule Creation

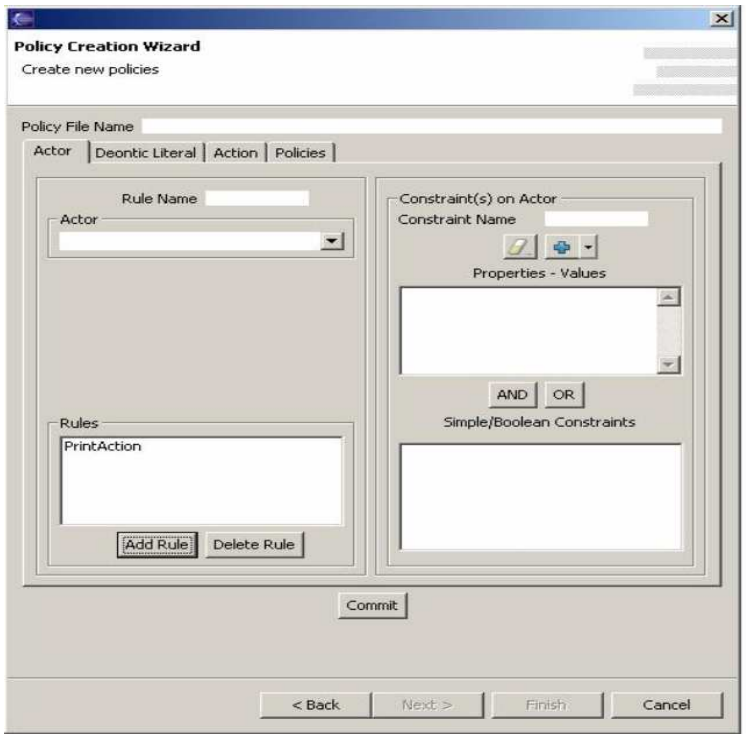
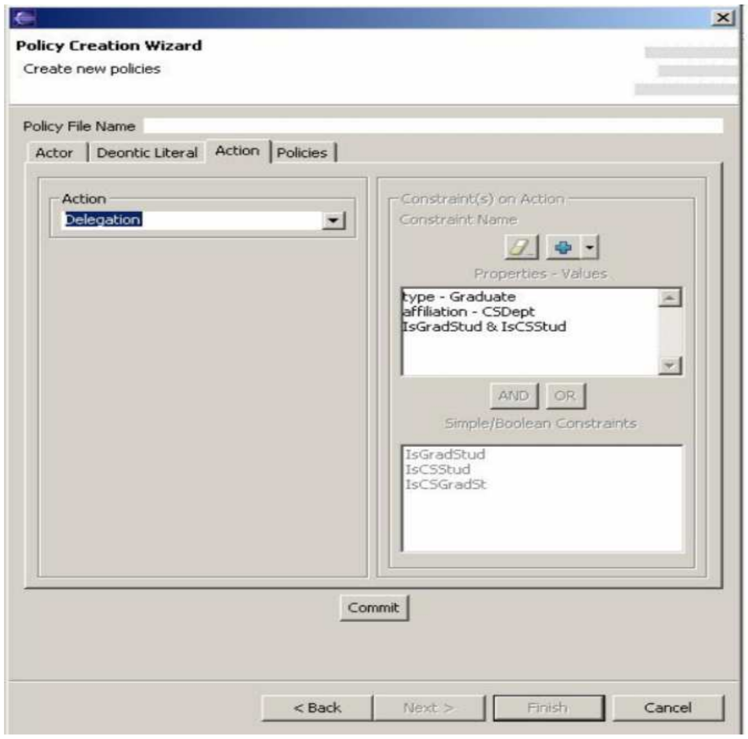


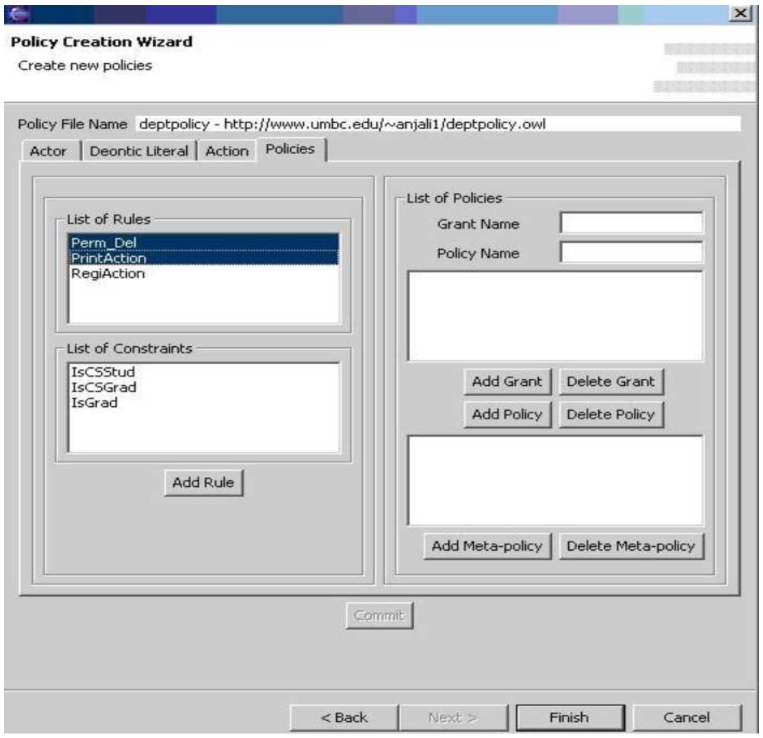
FIG. 4.10. Completion of Rule Creation Process



type - Graduate
affiliation - CSDept
IsGradStud & IsCSStud

IsGradStud
IsCSStud
IsCSGradSt

FIG. 4.11. First Step in Speech Act Creation Process



Policy File Name deptpolicy - http://www.umbc.edu/~anjali1/deptpolicy.owl

Perm_Del
PrintAction
RegiAction

IsCSStud
IsCSGrad
IsGrad

FIG. 4.12. Second Step in Speech Act Creation Process

that of rule creation. The construction of a speech act is divided into two halves. The first half concentrates on defining the sender of the speech act, the type of speech act being sent and modality selection to indicate if the sender has the permission, obligation, dispensation or prohibition to send the speech act. Sender of the speech act is selected from the Actor page by following a process identical to selecting an actor in the rule creation process. The modality selection for the speech act is similarly done from the Deontic Literal page as described in the rule creation process. The only variation is while making a selection on the Action page. The drop-down list on this page contains not only a list of domain actions but also offers four other options to create speech acts. These options are delegation, revocation, cancel and request. One amongst these options is selected and the selection determines the type of speech act being sent. This partial speech act construct is added or can be deleted using the 'Add Rule' and 'Delete Rule' buttons on Actor page.

The next half of the process begins after the partially constructed speech acts on the Actor page are committed and they appear under the rules' list on the Policy tab page. This list also contains some existing rules. Any such rule that needs to be modified dynamically is selected along with one of the partially constructed speech acts and a new rule is created out of the two selections. This completes the construction of the selected speech act. A constraint can optionally be added to the newly created speech act rule. The already existing rule that was chosen appears as a nested rule inside the speech act's OWL code generated by RIDE. The other partially constructed speech acts in the list can also be similarly completed. Figures 4.11 and 4.12 show two steps of speech act creation process. Figure 4.11 shows only the Action page as selections on Actor and Deontic Literal pages are similar to rule creation process.

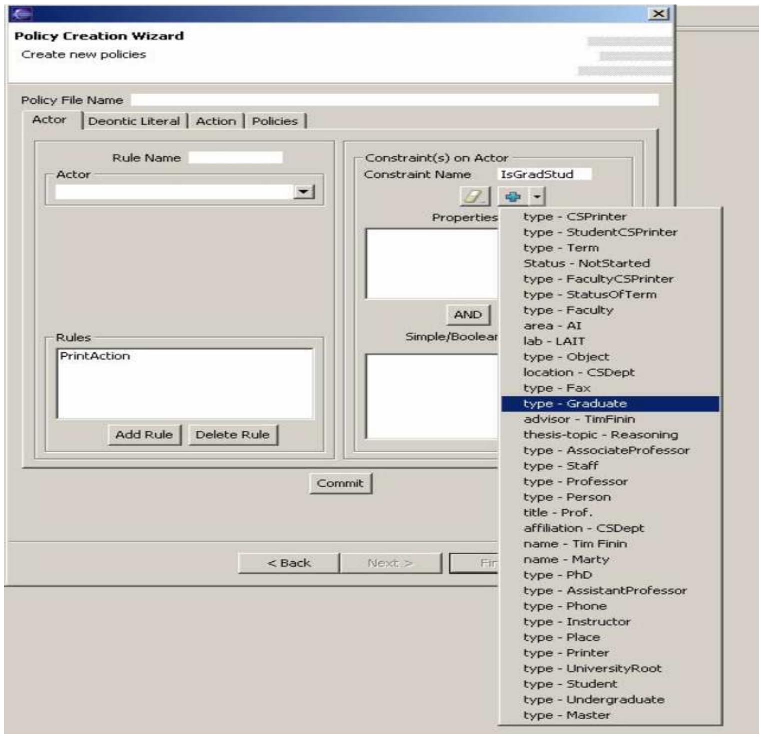
Constraints

Constraint creation using RIDE is made simple by providing a drop-down list of property-value pairs. For the Actor tab page, this list is constructed by extracting properties and their corresponding values defined by the domain specific ontologies over classes of actors and resources. For the Action tab page, it is similarly constructed by extracting properties and their values defined over classes of actions.

Simple constraint over an actor/resource can be created by selecting a property-value pair from the drop-down list in the constraints' section on the Actor tab page. Simple constraint over an action can be similarly created from the Action tab page. These constraints are added to and can be deleted from the constraints' list maintained by the model and displayed on the Actor and Action tab pages. The OWL code generated for the simple constraint has the subject field populated with the variable, 'var1' if the constraint is over an actor/resource and 'var2', if over an action. The predicate and object fields are respectively populated with the property and its value from the selection.

Boolean constraint creation is very straightforward once two or more simple constraints are created. The first Boolean constraint can be created by selecting two simple constraints present in the constraints' list of Actor/Action pages and combining the two using And button or Or button. Following this, any number of Boolean constraints can be created by selecting two simple, one simple and one Boolean or two Boolean constraints from the list. The list is constantly updated to include all constraints created by such user interactions.

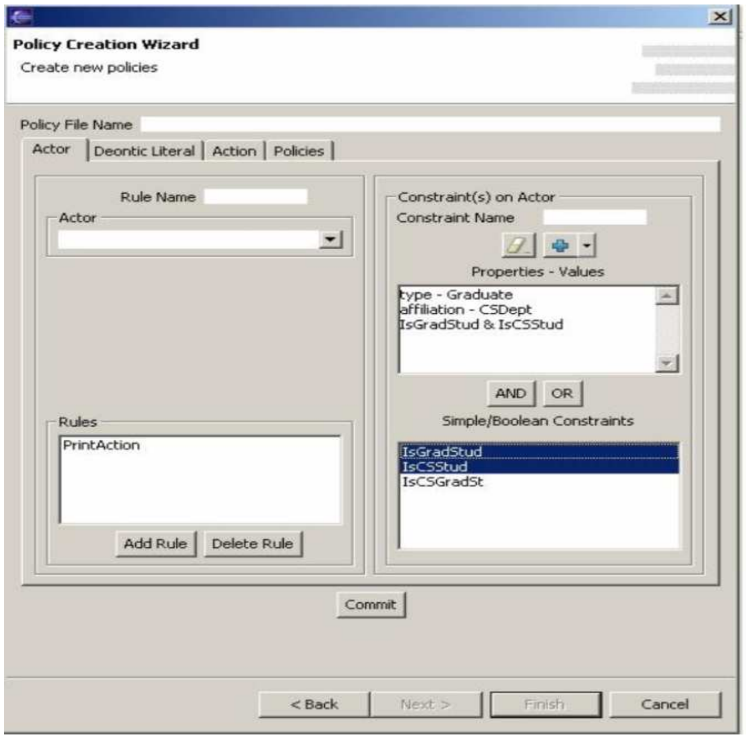
Simple and Boolean constraints created by following these processes can then be committed so that they appear on the Policy tab page. On this page, a constraint can be combined with a rule to define constraints for actors/actions in that rule. This makes the rule applicable to only those actors/actions that satisfy the condition set by the selected constraint. Likewise, these constraints can also be combined with speech acts to specify a condition to be satisfied for it to be sent. They can be combined with granting objects and policies for similar reasons. Figures 4.13 and 4.14 respectively show the process of simple as well as Boolean constraint creation over actors and actions in the domain.



Constraint Name IsGradStud

Rules: PrintAction

FIG. 4.13. Simple Constraint Creation Process



type - Graduate
affiliation - CSDept
IsGradStud & IsCSStud

PrintAction

IsGradStud
IsCSStud
IsCSGradSt

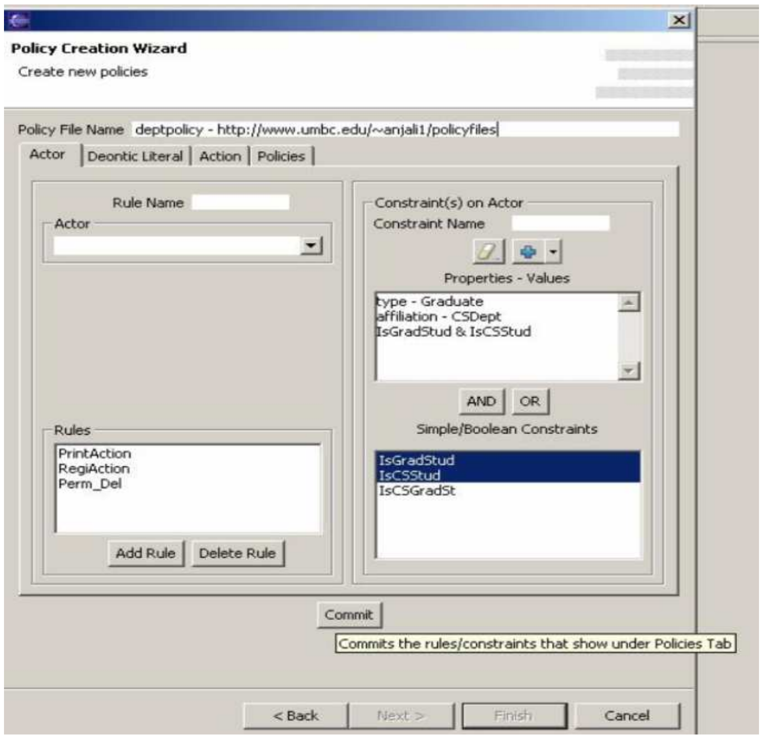
FIG. 4.14. Boolean Constraint Creation Process

Granting Objects

A granting adds a set of constraints to an existing rule to form a new rule. This allows reuse of rules (deontic objects) in different policies with varied constraints and actors. It also helps modularize the policy creation process. From the list of pre-defined rules and constraints on the Policy tab page, the user can select a rule and any number of constraints to create a granting object. Figure 4.15 shows the process of committing rules and constraints to create granting objects and policies and complete the process for speech act construction. Figure 4.16 shows the process of creating granting objects.

Policies

The two main types of Rei policies that can be created using RIDE's policy creation user interface are individual and group policies. Individual policies are created from rules defined over specific instances of actors and actions. Group policies can



Policy File Name deptpolicy - http://www.umbc.edu/~anjali1/policyfiles

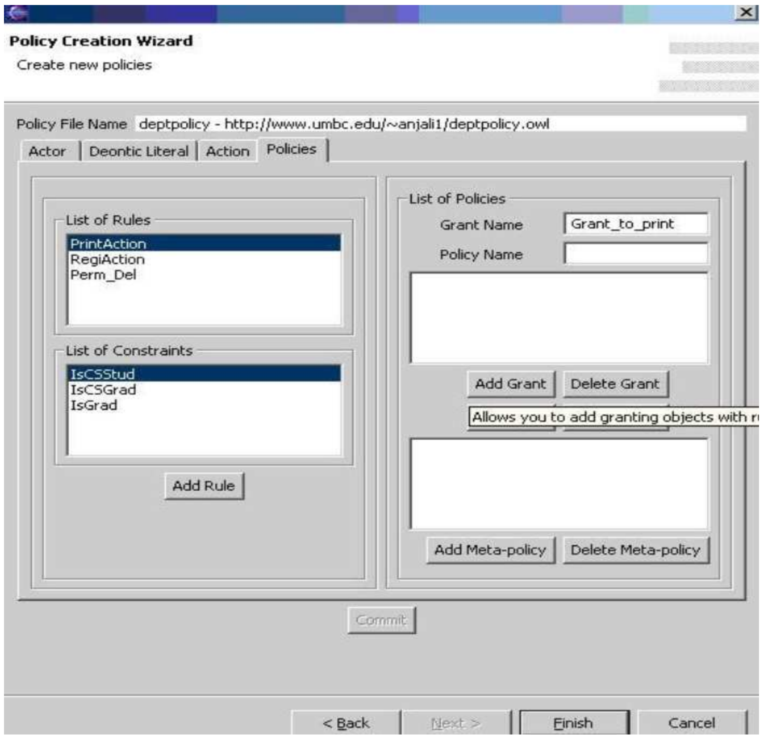
Rules:
PrintAction
RegiAction
Perm_Del

Properties - Values:
type - Graduate
affiliation - CSDept
IsGradStud & IsCSStud

Simple/Boolean Constraints:
IsGradStud
IsCSStud
IsCSGradSt

Commits the rules/constraints that show under Policies Tab

FIG. 4.15. Committing Rules/Constraints



Policy File Name deptpolicy - http://www.umbc.edu/~anjali1/deptpolicy.owl

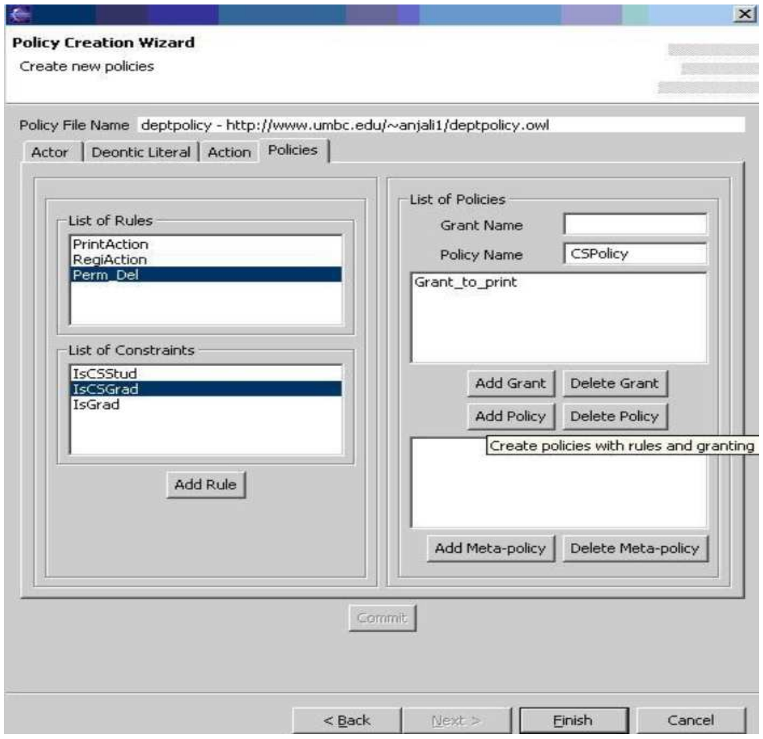
List of Rules:
PrintAction
RegiAction
Perm_Del

List of Constraints:
IsCSStud
IsCSGrad
IsGrad

Grant Name Grant_to_print

Allows you to add granting objects with r

FIG. 4.16. Granting Object Creation Process


Policy Creation Wizard

Create new policies

Policy File Name deptpolicy - http://www.umbc.edu/~anjali1/deptpolicy.owl

PrintAction
RegiAction
Perm_Del

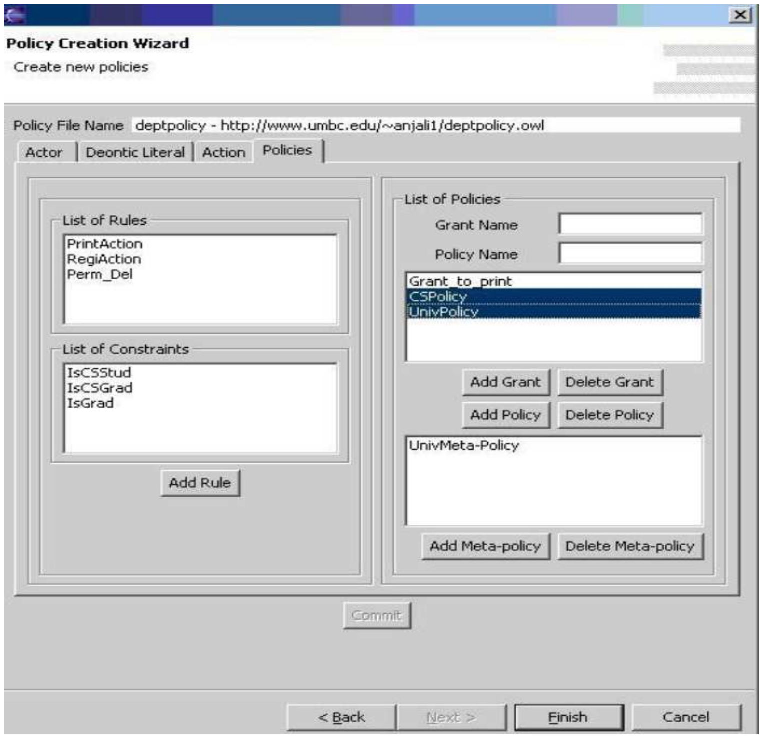
IsCSStud
IsCSGrad
IsGrad

Grant Name

Policy Name CSPolicy

Grant_to_print

FIG. 4.17. Policy Creation Process


Policy Creation Wizard

Create new policies

Policy File Name deptpolicy - http://www.umbc.edu/~anjali1/deptpolicy.owl

PrintAction
RegiAction
Perm_Del

IsCSStud
IsCSGrad
IsGrad

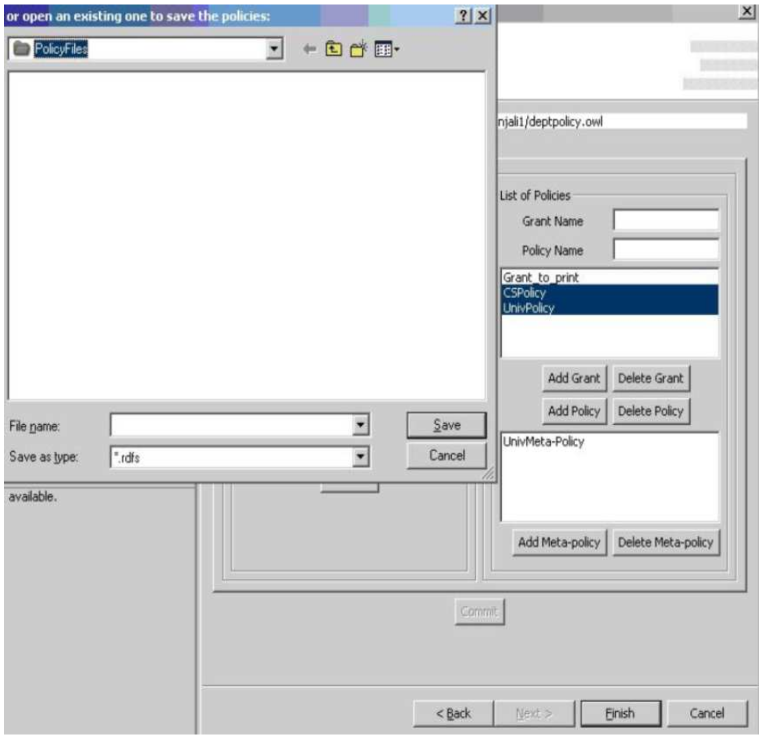
Grant Name

Policy Name

Grant_to_print
CSPolicy
UnivPolicy

UnivMeta-Policy

FIG. 4.18. Meta-policy Creation Process


or open an existing one to save the policies:

PolicyFiles

File name:

Save as type: *.rdfs

available.

njali1/deptpolicy.owl

List of Policies

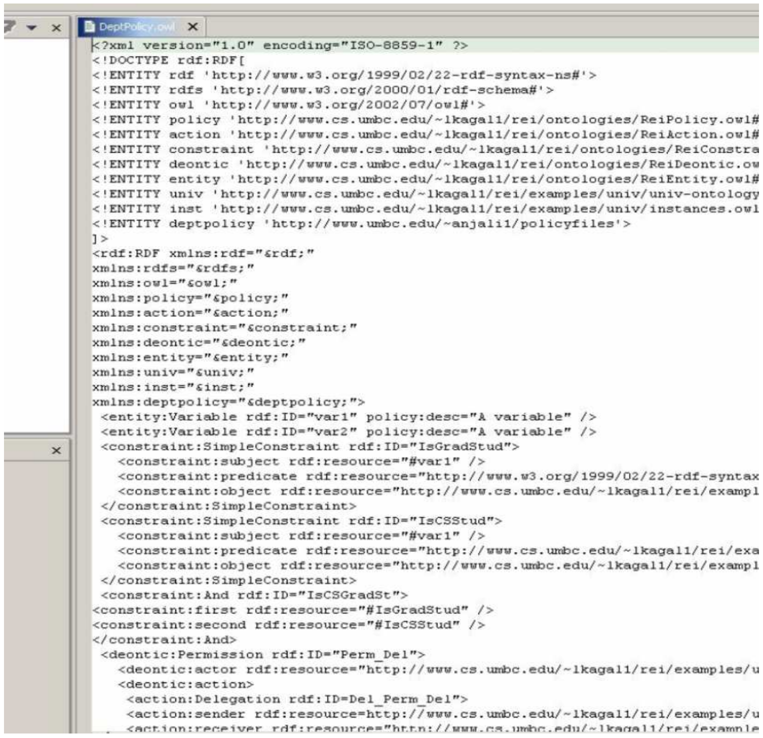
Grant Name

Policy Name

Grant_to_print
CSPolicy
UnivPolicy

UnivMeta-Policy

FIG. 4.19. Policy File Creation Process



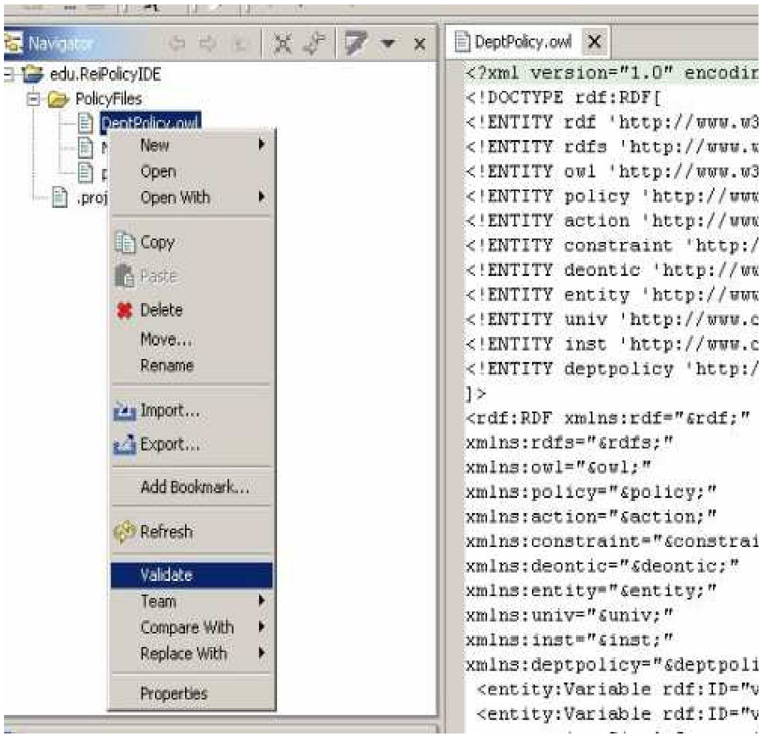
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE rdf:RDF[
<!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
<!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
<!ENTITY owl 'http://www.w3.org/2002/07/owl#'>
<!ENTITY policy 'http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiPolicy.owl#
<!ENTITY action 'http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiAction.owl#
<!ENTITY constraint 'http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiConstra
<!ENTITY deontic 'http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiDeontic.ow
<!ENTITY entity 'http://www.cs.umbc.edu/~lkagal1/rei/ontologies/ReiEntity.owl#
<!ENTITY univ 'http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/univ-ontology
<!ENTITY inst 'http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl
<!ENTITY deptpolicy 'http://www.umbc.edu/~anjali1/policyfiles'>
]>
<rdf:RDF xmlns:rdf="&rdf;"
xmlns:rdfs="&rdfs;"
xmlns:owl="&owl;"
xmlns:policy="&policy;"
xmlns:action="&action;"
xmlns:constraint="&constraint;"
xmlns:deontic="&deontic;"
xmlns:entity="&entity;"
xmlns:univ="&univ;"
xmlns:inst="&inst;"
xmlns:deptpolicy="&deptpolicy;">
 <entity:Variable rdf:ID="var1" policy:desc="A variable" />
 <entity:Variable rdf:ID="var2" policy:desc="A variable" />
 <constraint:SimpleConstraint rdf:ID="IsGradStud">
   <constraint:subject rdf:resource="#var1" />
   <constraint:predicate rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax
   <constraint:object rdf:resource="http://www.cs.umbc.edu/~lkagal1/rei/exampl
 </constraint:SimpleConstraint>
 <constraint:SimpleConstraint rdf:ID="IsCSStud">
   <constraint:subject rdf:resource="#var1" />
   <constraint:predicate rdf:resource="http://www.cs.umbc.edu/~lkagal1/rei/exa
   <constraint:object rdf:resource="http://www.cs.umbc.edu/~lkagal1/rei/exampl
 </constraint:SimpleConstraint>
 <constraint:And rdf:ID="IsCSGradSt">
<constraint:first rdf:resource="#IsGradStud" />
<constraint:second rdf:resource="#IsCSStud" />
</constraint:And>
 <deontic:Permission rdf:ID="Perm_Del">
   <deontic:actor rdf:resource="http://www.cs.umbc.edu/~lkagal1/rei/examples/u
   <deontic:action>
    <action:Delegation rdf:ID=Del_Perm_Del">
    <action:sender rdf:resource=http://www.cs.umbc.edu/~lkagal1/rei/examples/u
    <action:receiver rdf:resource="http://www.cs.umbc.edu/~lkagal1/rei/example
```

FIG. 4.20. Policy File showing Auto-generated OWL Code

be defined using rules created over a class of actors and actions that are constrained by the associated conditions. The latter is useful in large-scale distributed environments where stating individual permissions and obligations is time consuming. For example, if the domain ontology includes a Student class that has a name, address and affiliation, a policy can be defined for all students whose 'affiliation' property is set to 'UMBC'.

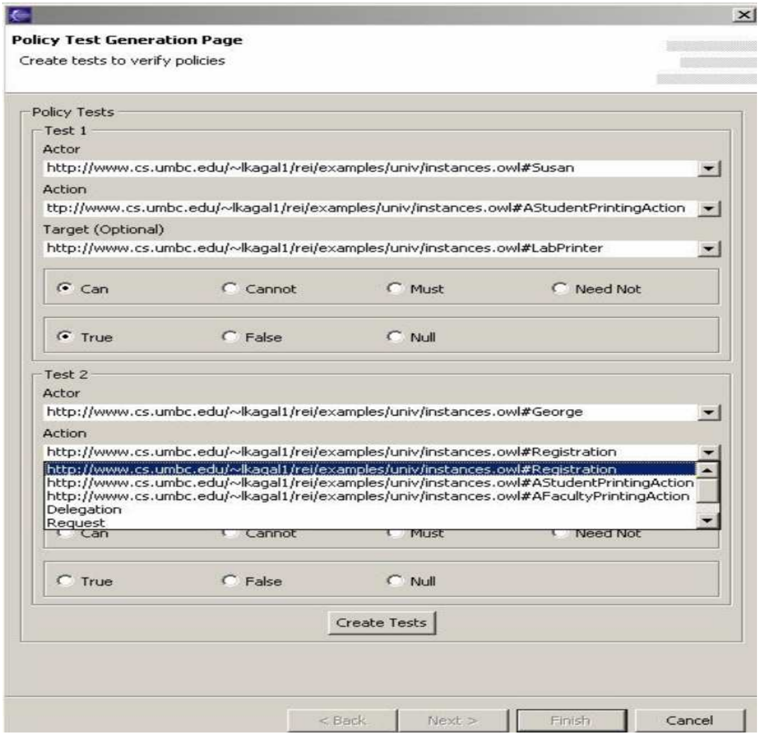
Rules and constraints created using Actor, Deontic Literal and Action tab pages can be committed following which they appear on the Policy page. Granting objects can be created on this page using the process described above. Rules, constraints and granting objects, once created, make policy creation a very simple process. The user can select the rules to be a part of policy under construction from the rules' list. Similarly, in order to specify the context under which the policy to be created is applicable, a constraint can be selected from the constraints' list. Any number of granting objects under the policies' list can be selected to be included within the policy and finally on triggering the 'Add Policy' button, a policy with the name entered in the Policy Name text field is added to the policies' list. The OWL code generated for a policy will show all the rules, constraints and granting objects selected to form it. A user can create as many policies as needed using this process.

Meta-policies can be created over conflicting policies by choosing them from the policies' list. The policy and meta-policy creation process will be completed when user hits the 'Finish' button on the wizard. This will open a file dialog box to enter the name of the policy file that will be stored in the local file system. This policy file contains the auto-generated OWL code for all the rules, constraints, granting objects, speech acts, policies and meta-policies present on the Policy tab page. Figures 4.17 and 4.18 show the policy and meta-policy creation processes. Figure 4.19 shows the policy creation wizard in the process of a policy file creation. Figure 4.20 shows the policy file generated by the wizard.



```
<?xml version="1.0" encodir
<!DOCTYPE rdf:RDF[
<!ENTITY rdf 'http://www.w3
<!ENTITY rdfs 'http://www.w
<!ENTITY owl 'http://www.w3
<!ENTITY policy 'http://www
<!ENTITY action 'http://www
<!ENTITY constraint 'http:/
<!ENTITY deontic 'http://ww
<!ENTITY entity 'http://www
<!ENTITY univ 'http://www.c
<!ENTITY inst 'http://www.c
<!ENTITY deptpolicy 'http:/
]>
<rdf:RDF xmlns:rdf="&rdf;"
xmlns:rdfs="&rdfs;"
xmlns:owl="&owl;"
xmlns:policy="&policy;"
xmlns:action="&action;"
xmlns:constraint="&constrai
xmlns:deontic="&deontic;"
xmlns:entity="&entity;"
xmlns:univ="&univ;"
xmlns:inst="&inst;"
xmlns:deptpolicy="&deptpoli
 <entity:Variable rdf:ID="v
 <entity:Variable rdf:ID="v
```

Fig. 4.21. Policy File Validation



Policy Test Generation Page

Create tests to verify policies

Policy Tests

Test 1

Actor: http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#Susan

Action: ttp://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#AStudentPrintingAction

Target (Optional): http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#LabPrinter

[✓ Can] Cannot Must Need Not

[✓ True] False Null

Test 2

Actor: http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#George

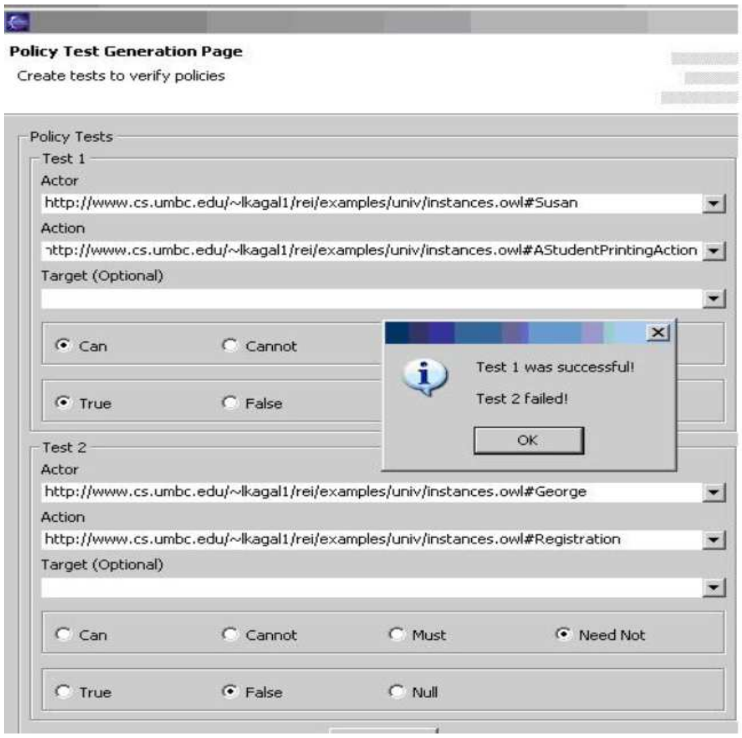
Action: http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#Registration

http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#Registration
http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#AStudentPrintingAction
http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#AFacultyPrintingAction
Delegation
Request

Can Cannot Must Need Not

True False Null

Fig. 4.22. Policy Test-Case Creation Process



Policy Test Generation Page

Create tests to verify policies

Policy Tests

Test 1

Actor

http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#Susan

Action

ıttp://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#AStudentPrintingAction

Target (Optional)

[✓ Can] Cannot

[✓ True] False

Test 1 was successful!

Test 2 failed!

OK

Test 2

Actor

http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#George

Action

http://www.cs.umbc.edu/~lkagal1/rei/examples/univ/instances.owl#Registration

Target (Optional)

Can Cannot Must [✓ Need Not]

True [✓ False] Null

FIG. 4.23. Policy Test-Case Results

4.3.3 Policy Test-case Creation Interface

The idea of unit testing, borrowed from Software Engineering, is to provide some infrastructure in order to make it easier for a programmer to determine that individual program modules perform correctly according to specification. We have extended this idea to test the correctness of individual policy units.

Policy files created by the user can be used to launch the policy test wizard. This wizard provides a simple, user-friendly interface for policy test-case creation. Each policy file created by the user using the policy creation wizard also has another file with the same name, but with an added '.model' extension, which is automatically generated by the wizard. The lists of actors, actions and targets formed by the policy creation wizard during that specific policy file creation are stored in this file. Using this file, identical lists of actors, actions and targets are created to populate drop-down lists for the fields of policy test-case creation page of the policy test wizard.

It is used internally by the wizard and hidden from the users. From the drop-down list, users can make selections to create policy test-cases. Target selection is optional. They must also select if the chosen actor can, cannot, must or need not perform the selected action. Test-case creation is completed by specifying a 'true', 'false', or 'null' answer expected for it. Two such test cases can be created at a time. Once these test cases are loaded, the policy test wizard uses its interface with the Rei policy engine to obtain an answer for the loaded test-cases. The answers from the Rei engine are compared with the answers the users had expected and results are returned.

Test-cases correspond to the rules that the enforced policies consist of. Rei engine reasons over all these policies and rules and returns an answer. If the returned answer matches the one expected by the user created test-case, it verifies that the rules, which in fact are individual policy units are correct and consistent. In this manner, every rule created by the user can be put to test. Testing the correctness of these rules automatically ascertains the consistency of the policy formed out of those rules. This helps the development of consistent and valid policies. Figures 4.21 and 4.22 show policy test-case creation and Figure 4.23 shows the results of policy test-cases obtained by querying the Rei engine.

Chapter 5

CONCLUSION AND FUTURE WORK

In this chapter we provide a summary of our work and describe its contributions. We also discuss possible directions for future research.

5.1 Conclusion and Contributions

The characteristics of the environments to which policy-based government is applied outline features for policy management tools to support. Though tools have been developed to facilitate policy management, there is not much work in policy development that meets all the requirements of this wide range of environments that make use of policy-based approaches. In this thesis, we have presented our work, which helps in that direction by meeting these requirements that highlight the inadequacies of existing tools.

We have described the framework and features supported by RIDE, the Rei integrated policy development environment. It consists of a wizard-based, graphical user interface that is plugged into the Eclipse framework and an interface with the Rei engine for policy test-case verification. RIDE allows creation of rules based on Rei's notion of permissions, obligations, revocations and dispensations. It allows creation of constraints, which are described over actors and actions in the domain. Policies can be easily created using these rules and constraints. It also allows meta-policy creation over conflicting policies.

RIDE's user-friendly and extensible interface, support for policy creation using

using speech acts to facilitate dynamic policy management, integrated namespace manager to facilitate domain information specification, ability to express policies over groups of actors and actions in the domain, automation of the laborious, complex and error-prone process of policy generation for any domain and abstraction from the details of Rei specifications and OWL contribute towards the novel aspects of our work. Also, since the domain-specific information is required to be in semantic web languages like OWL, RDFS and the policies are generated in OWL, its scalability and extensibility aspects are enhanced.

5.2 Future Work

As per Rei specifications, meta-policies can be defined over conflicting rules in a policy and can also be used to specify default behavior. Current implementation of RIDE supports meta-policy creation over conflicting policies, but it does not support meta-policy over rules or default meta-policy creation. It will be useful if future work can address this issue.

Other possible directions for future research include (i) extension of RIDE's user interface to support creation and manipulation of domain ontologies, (ii) its extension to support a graphical domain browser to view all policies in the domain and the way they relate to each other, and (iii) the ability to create and modify policies using such a browser that automatically detects inconsistencies between policies arising out of such creations and modifications.

REFERENCES

[1] A P3P Preference Exchange Language 1.0 (APPEL 1.0). http://www.w3.org/TR/P3Ppreferences/.

[2] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference, W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-ref/, 2004.

[3] Tim Berners-Lee. Notation3. http://www.w3.org/2000/10/swap/Primer, 2003.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System Version. Internet RFC 2704, September 1999., 1999.

[5] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle. Platform for privacy preferences 1.0 Specification. http://www.w3.org/TR/P3P/, 2003.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Policy Workshop 2001*, volume LNCS 1995 of *1*, Bristol, U.K., January 2001. Springer-Verlag.

[7] N. Damianou, N. Dulay, E. Lupu, M. Sloman, and T. Tonouchi. Tools for Domain-based Policy Management of Distributed Systems. In *Network Operations and Management Symposium 2002*, pages 213–218, Florence, Italy, April 2002.

[8] DARPA. DARPA Agent Markup Language + Ontology Inference Layer (DAML+OIL). http://www.daml.org, 2001.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Pearson Education.

[10] Georg Henrik von Wright. A Note on Deontic Logic and Derived Obligation. In *Mind*, 1956.

[11] T. T. Hewett. The role of iterative evaluation in designing systems for usability. In *Proceedings of the Second Conference of the British Computer Society, human computer interaction specialist group on People and computers: designing for usability*, pages 196–214. Cambridge University Press, 1986.

[12] Maryland Information and Network Dynamics Lab Semantic Web Agents Project. A hypermedia-based featherweight owl ontology editor. http://www.mindswap.org/2004/SWOOP/.

[13] Lalana Kagal. Rei : A Policy Language for the Me-Centric Project. Technical report, HP Labs, September 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-270.html.

[14] Lalana Kagal. *A Policy-Based Approach to Governing Autonomous Behavior in Distributed Environments.* PhD thesis, University of Maryland Baltimore County, Baltimore MD 21250, September 2004.

[15] Emil C. Lupu and Morris Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering*, 1999.

[16] J. Meyer and Roel Wieringa. Deontic logic: A concise overview. In *Deontic Logic in Computer Science, pp. 3-16, Chichester: John Wiley and Sons*, 1993.

[17] R. Mohan, E. Levin, and C. E. Irvine. An editor for adpative xml-based policy management of ipsec. In *Proceedings of the 19th Computer Security Applications*

Conference, Las Vegas, NV, IEEE Computer Society, December 2003, pages 276–285, 2003.

[18] Object Technology International, Inc. Eclipse Platform Technical Overview. http://www.eclipse.org/, 2003.

[19] HP Labs Semantic Web Programme. Jena2 - A Semantic Web Framework. http://jena.sourceforge.net/.

[20] M. Schunter and C. Powers. The Enterprise Privacy Authorization Language (EPAL 1.1). http://www.zurich.ibm.com/security/enterprise-privacy/epal/, 2003.

[21] Stanford Medical Informatics. Protégé-2000 project. http://protege.stanford.edu/.

[22] A. Uszok, J. Bradshaw, P. Hayes, R. Jeffers, M. Johnson, S. Kulkarni, M. Breedy, J. Lott, and L. Bunch. DAML reality check: A case study of KAoS domain and policy services. In *International Semantic Web Conference (ISWC 03). Sanibel Island, Florida*, 2003.

[23] A. Uszok, J. M. Bradshaw, R. Jeffers, M. Johnson, A. Tate, J. Dalton, and S. Aitken. Policy and Contract Management for Semantic Web Services. In *AAAI Spring Symposium, First International Semantic Web Services Symposium*, 2004.

[24] W3C. Resource Description Framework. W3C Recommendation, http://www.w3.org/TR/rdf-schema/, 2002.

[25] W3C. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, http://www.w3.org/TR/REC-xml, 2004.