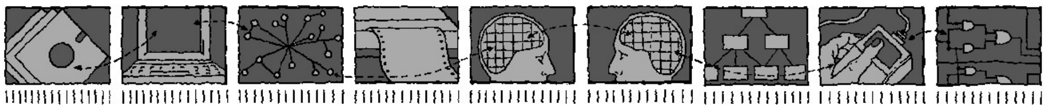
View metadata, citation and similar papers at core.ac.uk

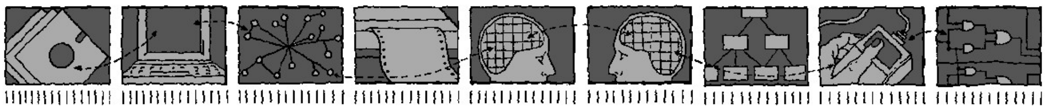
brought to you by CORE

provided by Stirling Online Research Repository



Department of Computing Science and Mathematics

University of Stirling



An Overview of Ontology Application for Policy-Based Management using POPPET

Gavin A. Campbell

Technical Report CSM-168

ISSN 1460-9673

June 2006

Department of Computing Science and Mathematics
University of Stirling

An Overview of Ontology Application for Policy-Based Management using POPPET

Gavin A. Campbell

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland
Telephone +44-1786-467421, Facsimile +44-1786-464-551

Email **gca@cs.stir.ac.uk**

Technical Report CSM-168

ISSN 1460-9673

June 2006

Abstract

The use of ontology to describe the key concepts and their interrelationships within a particular area has become a widely recognised and advantageous means of sharing information about the structure of knowledge within a domain. Ontologies provide a way of integrating structured data within an application.

This report provides an overview of how the ACCENT policy-based management system [1] was significantly re-engineered to utilise an ontology in place of previously hard-coded, domain-specific information within its user interface. In order to successfully integrate the ontology with the policy system, a new framework named POPPET was developed, responsible for parsing and querying ontological data. Although a substantial alteration in technical structure, the process vastly generalises the policy system, enabling adaptation for policy management within any custom domain.

An introduction to the concepts and motivation for ontology creation using OWL is presented, together with general background to the ACCENT policy system. A technical overview is then given covering the developed ontologies, the POPPET system design, and the policy system re-engineering process. Finally, a comparison is made between the new and old policy system structures, and the impact on system performance is evaluated.

Keywords: Knowledge-based systems, Policy, POPPET, Ontology, OWL, System Re-engineering.

Table of Contents

Table of Figures

1 Introduction

The notion of extensibility and component re-use plays a central role in software system development. When designing a system, consideration is given towards how easily it might be extended to implement new features or be utilised within a different context or environment. Often, while a large proportion of a system may be generic, application or user interfaces limit potential system reuse as they are hard-coded to serve a specific domain. In such cases manual reconfiguration can be a costly and frustrating process. Ideally, a system should be adaptable to suit a new application domain with the minimum of re-engineering.

This report presents an innovative solution to the issue of domain-dependent user interfaces. Motivated by a desire to reuse an existing policy-based management system across multiple domains, the concept of ontology is employed to radically generalise the system user-interface. Using an ontology to model knowledge associated with the application domain, previously hard-coded domain information can be extracted and the system altered to query the ontology instead. By specifying domain knowledge externally, the system may process details of any custom field expressed within a similarly structured ontology.

In the subsections that follow, an overview of the concept of ontology is given together with an introduction to the ACCENT policy-based system and the issues that necessitated its generalisation. Section 2 describes the technical development of the implemented ontology system known as POPPET, outlining the various tools and frameworks incorporated in its design. Section 3 provides a summary of the solution and evaluates its effectiveness.

1.1 Ontology Overview

The concept of ontology has been of great significance in the field of philosophy for a number of decades. In recent years, however, its use has exploded within the realm of computing – in particular the fields of AI, software agents, software development and, more specifically, the World Wide Web. An ontology can be defined as the terms used to describe and represent an area of knowledge, coupled with the logical relationships among these. It provides a common vocabulary to share information in a domain, including the key terms, their semantic interconnections and some rules of inference. Advantages of creating an ontology include the ability to share a common understanding of the structure of information in a particular domain, and the possibility to reuse this knowledge among software applications. In addition, an ontology enables separation of domain knowledge from common operational knowledge in a system.

A variety of specialised languages exist to define ontologies. OWL (The Web Ontology Language [9]) was the language chosen for the ontology development described in this report. The language is XML-based and was officially standardised by the World Wide Web Consortium (W3C) in February 2004. OWL was chosen primarily due to its recent standardisation, the benefits this brings in terms of available software tool support, and compatibility with existing and future industrial and academic projects. In addition, OWL provides a larger function range than any other ontology language to date.

Ontology documents expressed in OWL are intended for use in applications where ontological content must be processed rather than simply extracted and presented to the human eye. OWL was designed to combine and extend the customisable tagging of XML with the flexible data representation ability of RDF (the Resource Description Framework [15]), with a view to formally describing the semantics of terminology in a domain.

Using OWL, an ontology is created by defining various classes, properties and individuals. A class represents a particular term or concept in the domain, while a property is a named relationship between two classes. An individual is an instance or 'member' of a class, usually representing real data content within an ontology. Properties are applied to classes in the form of 'restrictions'. A property restriction describes an 'anonymous' class, that is, a class of all individuals that satisfy the restriction. In OWL, each property restriction places a constraint on the class in terms of either a value (class or data-type), or cardinality (number of values the property may be related to). The language also supports inheritance within class and property structures. A property restriction placed upon a class is automatically inherited by any of its subclasses. The Web Ontology Language Reference document [10] provides a complete description of all language constructs.

OWL supports the sharing and reuse of ontologies by means of ontology importation. Using this mechanism, all definitions of classes, properties and individuals within an imported ontology, may be used by the importing ontology. This provides an effective means to separately define generic and domain-specific policy facts.

The OWL language is broken down into three sub-languages that provide mounting strengths of expressiveness to meet the needs of different users and implementers. In descending order, the dialects are:

- **OWL Full:** The complete OWL language, OWL Full provides maximum expressiveness in an ontology. It permits all the syntactic freedom of RDF but gives no computational guarantee that statements will be logically inferable using existing Description Logic reasoners.

- **OWL DL (Description Logic):** Designed to provide complete computational compatibility with Description Logic reasoners, OWL DL contains the full range of OWL language constructs, but places certain restrictions on how they are used. The result is an extremely expressive sub-language that can be used in conjunction with existing reasoning systems.

- **OWL Lite:** The weakest dialect, providing only a subset of OWL language constructs, OWL Lite was designed for users requiring simple constraints and a class hierarchy. Additionally, tool support for OWL Lite ontologies is easier to implement, and the documents themselves are more compact. As OWL Lite is a condensed subset of OWL DL, it also offers compatibility with existing reasoning tools.

A further, formal definition of the differences between OWL dialects can be found in the OWL Semantics and Abstract Syntax guide [11]. The OWL ontologies described in this report conform to the OWL DL sub-language.

1.2 ACCENT Policy System Overview

The ACCENT policy-based management system [1] allows users to specify high-level policies for detailed preferences surrounding how they wish (Internet) calls to be handled. Although original system implementation was specialised for the domain of (Internet) call control, the core system architecture is sufficiently generic to support policy handling in any application area. The major components of the ACCENT system are encapsulated in a three-layer structure as shown in Figure 1.1.

At the base level there is the Communications System Layer which connects the system to the external environment. The hub of system processing occurs at the Policy System Layer which incorporates the Policy Server and data stores. At the top level is the User Interface Layer, through which the system is accessed by a user. Users may define and edit policies via the Policy Wizard [20] – a web-based interface largely customized to support the application domain. For a detailed explanation of the ACCENT system architecture refer to [18].

The system supports rule-based policies in event-condition-action (ECA) form. In relation to the concept of ECA, a policy rule broadly consists of three main components:

- A **trigger** set (events which potentially cause a policy to be executed)
- A **condition** set (contextual variables used to determine whether the triggers justify policy execution)
- An **action** set (output or resulting actions taken by the system upon policy execution).

A policy is eligible for execution on the occurrence of certain triggers it defines. When the policy system is informed of an event, applicable policies are retrieved and executed.

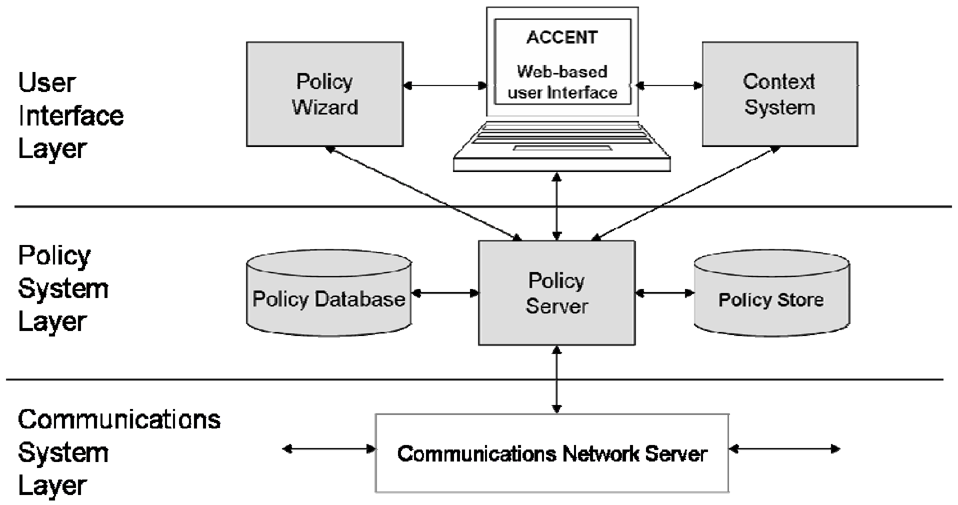


Figure 1.1 ACCENT Policy System Architecture

A comprehensive policy description language called APPEL [17] was designed to facilitate the creation of policies. APPEL comprises a core language schema which can be extended to support policy management for any given domain. In the original implementation, the language was extended to include capability for (Internet) call control and for call conflict resolution. Constructs of APPEL are incorporated in the policy wizard and associated user interface layer components. In addition, the policy wizard generates domain-specific content based on extensions to the language. In the original implementation, a substantial proportion of the policy wizard was hard-coded with information specific to (Internet) call control.

1.3 A Domain-Independent Ontology-Driven Policy System

As the existing policy system user interface was geared specifically towards call control, this presented a fundamental flaw in utilising the ACCENT system for policy-based control within a different application area. The solution was to extract domain-specific details and house them in an external ontology which could be used by the policy wizard. In addition to knowledge of the application domain, the ontology was required to contain details of the core APPEL policy description language.

The generic aspects of APPEL can themselves be encapsulated in an OWL ontology. From this, domain-specific extensions to the core language constructs can be defined through the creation of further ontology documents. Developing ontologies to describe the core and specialist structures of APPEL provides a method of separating generic language knowledge from application-specific knowledge. The policy wizard can then query ontology information in place of hard-coded details. This provides a more generalised system structure, easily adaptable to support policy management in a new domain.

To integrate an ontology with the policy system, a new framework was developed to provide the policy wizard with an API through which to query and retrieve domain information. This framework, named POPPET, is described in the following section.

2 Ontology System Design

The aim was to re-engineer the ACCENT system to incorporate ontological data in place of hard-coded domain knowledge. Technical development was undertaken in three stages:

- Ontologies were defined to separately model generic policy language constructs, additional policy wizard interface features, and information specific to the domain of call control. Combined, these three ontologies form a single, structured OWL document. This three-tier 'ontology stack' is explained in Section 2.1.
- A framework system was developed for parsing the ontology and building an interpretable model of its structure. In addition, a suitable API was designed to enable the policy wizard to query the stored ontology model. This framework is outlined in section 2.2, together with a description of the technologies and software packages it utilises.
- The ACCENT system policy wizard and associated user interface layer components were re-engineered, in order to remove hard-coded domain knowledge and gather it instead via queries to the ontology. Section 2.3 provides an overview of the alterations made.

Additionally, section 2.4 provides a worked-example of how to create a compatible ontology for a new domain and apply this to the policy wizard. The domain of call control is used as an example to illustrate the process.

2.1 OWL Ontology Stack

Domain-specific knowledge applicable to the policy system is captured within a single OWL ontology that imports separate ontologies describing the core policy language and common policy wizard interface features. This ensures domain knowledge is defined within a set structure which relates it to policy language terms. This interprets how the information may be processed, applied or displayed by the policy wizard. A domain-specific ontology is therefore constructed using a three-tier 'stack' or hierarchy of separate ontology documents, as shown in Figure 2.1.

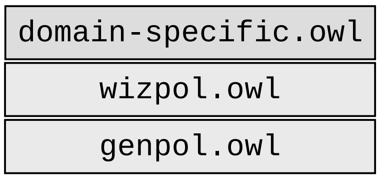


Figure 2.1 OWL Ontology Stack

At the base level, the `genpol` ontology describes the core constructs of the APPEL policy description language. This includes definition of key policy-related concepts such as Policy Document, Policy Variable, Policy Rule, Trigger, Condition and Action. Relationships between these concepts describe named associations, inheritance properties and cardinality restrictions. This ontology specifies a skeleton structure of ontology classes and properties which can be imported and extended within a domain-specific ontology.

The policy wizard incorporates a number of features which control and manipulate domain data prior to its display. Such features are not part of the policy language itself, but common and useful in any domain-specific ontology that is geared towards use with the policy system. This additional, wizard-related knowledge is defined in the `wizpol` ontology. `Wizpol` directly imports the `genpol` ontology, thus specialising the APPEL language for use with the policy wizard. A detailed description of `genpol` and `wizpol` is given in the technical report 'Ontology Stack for a Policy Wizard' [3].

The ontologies of `genpol` and `wizpol` are intended to be entirely reusable. Due to the recursive nature of the OWL import mechanism, a domain-specific ontology is required to import only `wizpol` – importation of `genpol` is inherently automatic. Once included, an ontology may extend the class hierarchy of the imported ontology structure to define additional sub-classes and properties together with applicable constraints. In particular, this includes the definition of specific trigger events, condition parameters and actions associated with the domain in question. The implemented domain-specific ontology for call control is described within the technical report 'Ontology for Call Control' [4].

Although a domain-specific ontology structure is specially geared towards use within the policy wizard, there is no restriction on the inclusion of additional non-policy related knowledge. The presence of the `genpol` and `wizpol` ontology structure ensures compatibility with the ACCENT system, but the same ontology may contain additional domain knowledge and an unlimited number of imported ontologies for use by other applications or agents.

To be compatible with existing formal reasoning tools, both `genpol` and `wizpol` were designed to conform to the OWL DL sub-language. It was the view that these base ontologies, and the domain-specific ontologies which extend them, would define the structure of policy-related knowledge but not actual policy data. Such data is separate from the actual structure of the language or knowledge describing a domain. It is therefore defined, stored and processed independently by the policy system. For this reason, the developed ontologies contain no individuals or 'instances' of ontology classes. Each ontology applies constraints strictly to 'anonymous' classes. That is, relationships between classes are described in purely abstract terms.

Consequently, the ontologies are intended for static use by the policy system and should not be altered in any way during normal operation. This is in contrast to a dynamic knowledge base where actual data sets (class individuals) are continually modified in real time to reflect the changing nature or state of the domain.

2.2 POPPET System

POPPET (Policy Ontology Parser Program Extensible Translation) is a framework designed to integrate an OWL ontology with the ACCENT policy-based management system. The framework is responsible for building a model of a given ontology, and offers an API through which an external program may query this model. An overview of POPPET is given in Section 2.2.1. POPPET utilises a number of existing software frameworks and packages to parse and reason about an OWL ontology. Section 2.2.2 gives a description of these supporting technologies and an explanation of why they were adopted. Following the creation of an ontology model, the POPPET system provides an application with the means to query the model. This has been achieved using Remote Method Invocation (RMI), which enables the wizard to access the ontology model remotely. An explanation of how RMI is used and why it was chosen in this context is outlined in section 2.2.3.

2.2.1 Overview

The POPPET system bridges the gap between an ontology and the ACCENT policy system. Designed with maximum abstraction in mind, POPPET can be used to parse and model any given OWL ontology, allowing an application to access and query this model. It offers access to the model via a specially designed API.

The POPPET framework contains no specific, hard-coded knowledge of either the developed policy ontology stack or the policy wizard. The domain or data content of an ontology is entirely irrelevant to POPPET. Consequently, this gives obvious scope for its reuse and extension within additional contexts, should the policy language represented within the ontologies be adapted.

In summary, POPPET has the following responsibilities:

- Accessing an OWL ontology document from a given location.
- Parsing the ontology document and performing inference via an external reasoner.
- Constructing and storing a model of the parsed ontology.
- Providing an interface and methods (an API) to enable external applications to query the model.

The complete POPPET system architecture is shown in Figure 2.2. An entirely Java-based system, POPPET utilises an existing ontology parser called Jena [7] which offers extensive support for OWL ontology document parsing and model building. Jena also provides predefined methods to reason about an ontology via a compatible external inference engine. In order to construct an ontology model, an OWL document is first analysed by the reasoner to infer additional information about classes defined within it. The inference engine chosen for use within the POPPET system was Pellet [12].

The complete POPPET system is run as a stand-alone server application. When invoked, POPPET is passed the URL of an ontology to be read and modelled. Following successful ontology parsing, the model is then held until the POPPET system is stopped. Thus, it is available for any application to access and query. As explained in Section 2.1, policy ontology documents are entirely static in that their content describes the structure of domain knowledge, without any instances of actual policy data. Therefore, an ontology model is not intended to be modified during normal use by the policy wizard. For this reason, the time-consuming process of accessing, parsing, reasoning and constructing a model of the ontology need only be carried out once – when the POPPET server is started. The model must then be held in memory, prior to communication from an application. If the POPPET system did not operate in this way, a separate model would have to be constructed each time a query on the ontology was issued. As an indication of frequency, the policy wizard may issue an average of ten ontology queries in the process of constructing a single HTML form to the user. This would incur considerable overhead in processing time, leaving the user interface frustratingly slow to interact with.

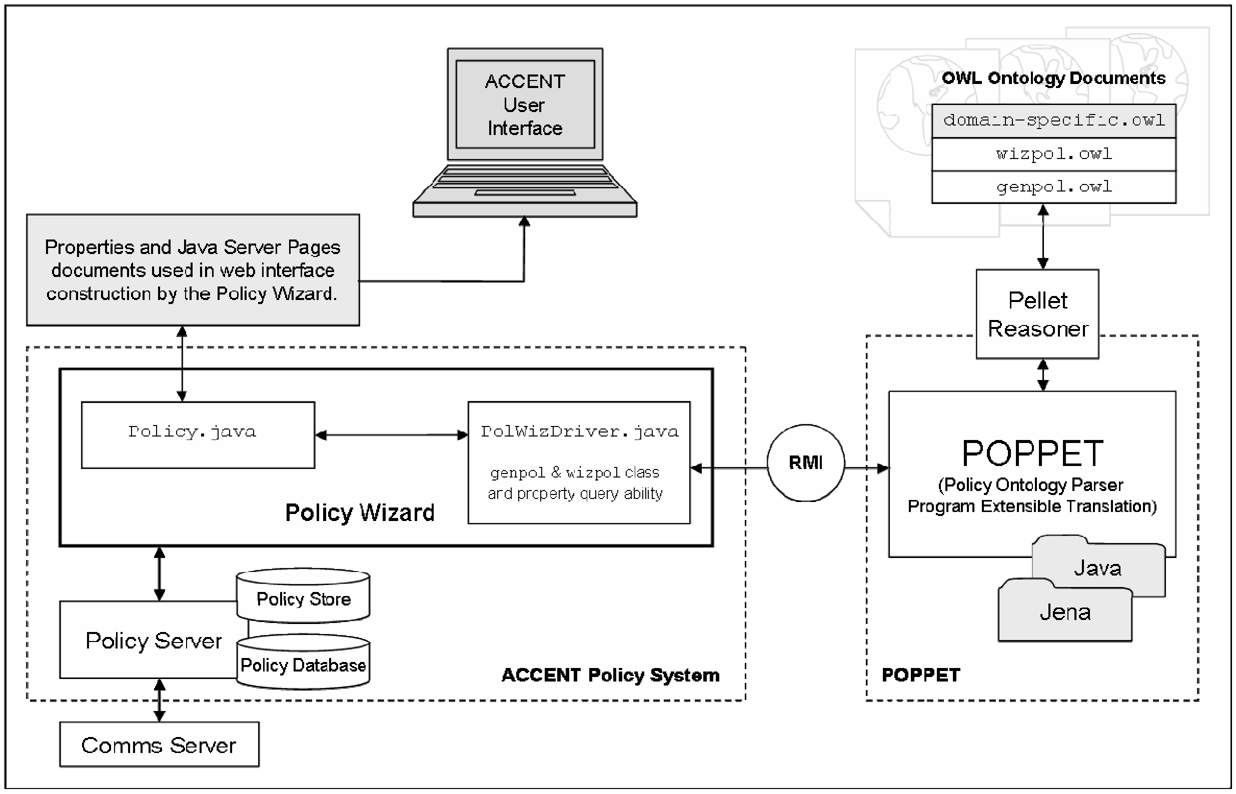


Figure 2.2 POPPET System Architecture

An overview of the Jena package and the Pellet reasoner is given in section 2.2.2. As the POPPET system exists as a stand-alone server application, communication with the policy system is provided via RMI, as described in section 2.2.3.

2.2.2 Supporting Technologies

An open-source Java-based framework, Jena [7] was designed for use in building semantic web applications. It provides programmatic support for a variety of ontology-related languages, including RDF [15], RDFS [16], SPARQL [19] and OWL. Due to the strong OWL support offered by Jena, it was an obvious choice of parser for the POPPET system. Jena provides support to access, parse and build a scalable model of a given ontology document. Methods are provided to search the model for specific classes and properties, and query the restrictions and constraints placed upon them. Use of Jena saved considerable time in developing a parser for OWL.

An inference engine is necessary to derive additional statements that a straight-parsed OWL document model does not explicitly express. Although Jena includes its own rule-based inference engine, it is intended to be of general use in parsing documents written in any ontology language, and is not specifically designed for use in conjunction with OWL. Pellet [12] is an open-source Java-based reasoner, specifically geared towards OWL DL (Description Logic) compliant ontologies. Due to its specific support for OWL, the use of Pellet was preferred over the use of Jena's own built-in reasoner, as it is guaranteed to recognise and correctly interpret all OWL DL constructs.

Pellet has been tested against the Jena package, and its use is promoted in Jena documentation. This compatibility between Pellet and the Jena framework was another motivating factor in its choice over other available external reasoning engines.

2.2.3 RMI Communications Interface

The POPPET system uses Remote Method Invocation (RMI) [6] to interact with applications and permit them to query a given ontology model remotely. RMI enables an object running on one Java Virtual Machine (JVM) to remotely manipulate an object running on a different JVM (another context or physical host), by way of a set of publicly defined methods.

Using RMI as an interface to POPPET has several advantages. Firstly, it provides a secure method of object access using an established technique. RMI also allows the POPPET server to be run on a different host machine from the policy wizard, enabling greater flexibility in policy system set-up. Using such a distributed architecture ensures the policy system remains independent of the system used to access an ontology. This option is also favoured against housing the POPPET system directly alongside the policy wizard within a single Java Servlet context on Tomcat [2].

While RMI is an effective and secure option, its use is at the expense of a slight overhead to access and process queries on a remote object. Network access incurs a time delay when processing and displaying responses to a user. However, in practice, this delay was found to be acceptable, having no detrimental effect on the operational efficiency of the policy wizard.

2.3 Policy System Re-engineering

In its previous existence, the ACCENT policy wizard contained a large proportion of hard-coded knowledge of the call control domain. This knowledge was directly displayed to a user wishing to define policies. In order to generalise the wizard so it may generate a user interface specific to any custom domain, all details related to call control were replaced by relevant method calls to retrieve data from an external ontological source. To this end, the re-engineering process targeted only system components directly associated with the policy wizard. Remaining components, including the policy server, policy store, policy database and additional context systems, were unaffected.

To illustrate the re-engineering process, section 2.3.1 explains the alterations made to the original system, describing the process involved in displaying a policy condition page as an example. Section 2.3.2 summarises extensions made to the policy wizard.

2.3.1 Policy Wizard Alteration

The heart of policy wizard operation lies within Java code that processes requests via a web-based user interface. JSP-based, it creates a response through communication with the policy server (to retrieve stored policies and policy-related data) and constructing HTML output tailored to the domain.

Previously, the domain knowledge embedded within the logic was generated using hard-coded strings. The process of re-engineering involved removing all hard-coded domain-specific knowledge and replacing it with appropriate method calls to retrieve data from an external source. Note the data source itself is irrelevant to the wizard – it simply calls a method to retrieve information.

To demonstrate the new ontological approach, consider the process of selecting an option to view available conditions for a policy. The process can be described as follows:

- The user, running the policy wizard in a Web browser, selects the condition section of a policy rule by clicking on hyperlink text. This generates a request to the server hosting the policy wizard, causing `EditCondition.jsp` to be executed.

- The JSP within the policy wizard utilises Java code in the package `uk.ac.stir.cs.accent.wizard`. This includes the classes `Common.java` and `Policy.java`. The `Common` class handles communication with the underlying policy database and policy store, as well as HTML formatting. The `Policy` class generates the HTML page body, incorporating domain-specific policy options gathered from the ontology. During JSP execution, JavaScript code is generated to validate form input. The `Common` class generates the HTML page header and footer.

- The `Policy` class constructs the page body. It is passed session information including the detected 'user-level' of the user requesting the page. In particular, domain-specific policy information is handled in the method `FormCondition`. This method constructs and outputs HTML, using data gathered from the ontology via `PolWizDriver.java`. Essentially, policy condition data is retrieved using the following methods:

    - `PolWizDriver.getFullConditionCategoryList()`
      Returns a list of condition category names. For call control, there are four categories: 'address', 'amount', 'description' and 'time'.

    - `PolWizDriver.getParameterListFromCategory (category_name, user_level)`
      Returns a list of condition parameters associated with the category name passed, which are valid for the specified user-level.

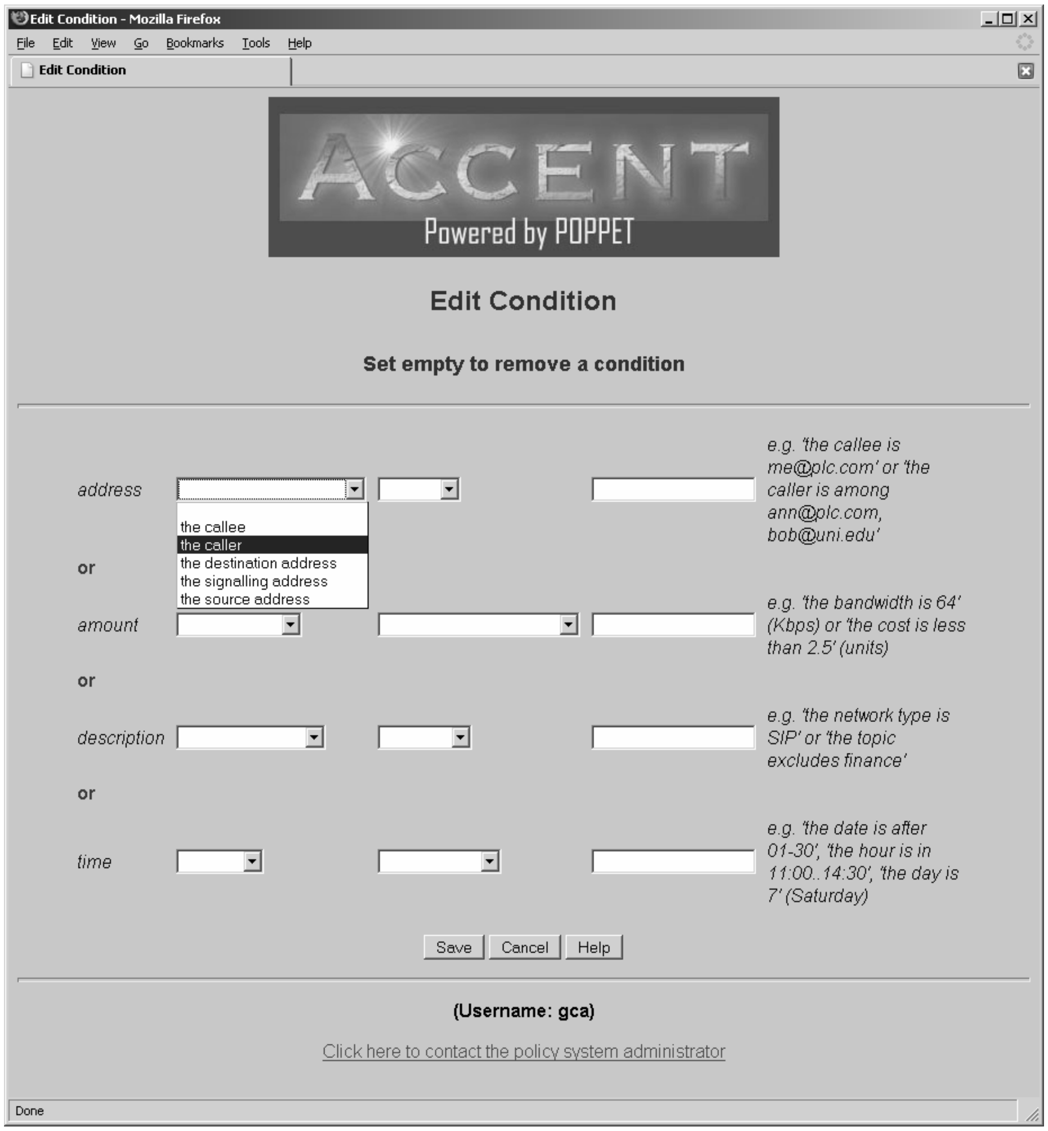
    - `PolWizDriver.getOperatorListFromCategory (category_name, user_level)`
      Returns a list of condition operators associated with the category name passed, which are valid for the specified user-level.

  The first call returns a list of condition categories. In turn, parameters and operators for each category are retrieved. Passing the user-level as an argument also ensures returned information is suitably refined.

- The `PolWizDriver` class connects to the POPPET server via an RMI interface, to gather relevant domain knowledge from an ontology. The POPPET server processes remote calls on its ontology model and returns the requested data. The `PolWizDriver` passes the query results back to the `Policy` class.

  Ultimately, this query process produces a set of string literals, which can be translated into natural language prior to being embedded appropriately within HTML. The existing implementation hard-coded lists of condition parameters, operators, and their relevant categories. This required an alternative, explicit list of strings for each user-level. Needless to say, this method was highly inflexible, with policy data treated as fixed text with no associated semantic knowledge, or notion of significance from bare HTML. Using the re-engineered, ontology-based approach, the same condition parameter and operator lists may be constructed dynamically via method calls to the POPPET system. The detected user-level is passed to POPPET as a parameter argument, ensuring the returned data set is strictly relevant. This reduces the size and complexity of the policy wizard, and renders it fully domain-independent.

- The policy wizard server sends the complete HTML page to the user. An example condition page for the call control domain is shown in Figure 2.3. There are four condition categories, each containing associated lists of condition parameters, condition operators and a condition value field. The user selects from one category only, choosing a named parameter and operator from the option lists and entering a corresponding condition value. As shown, an option list contains data gathered from the ontology, which has been translated into natural language. In Figure 2.3, the condition parameter option list for the “address” category is actively displayed. Following data selection, Figure 2.4 shows a valid condition entry. The natural language representation of this condition reads “if the caller is kjt@cs.stir.ac.uk”.


Edit Condition - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Edit Condition
ACCENT
Powered by POPPET
Save
Cancel
Help
Click here to contact the policy system administrator
Done

Edit Condition

Set empty to remove a condition

address: the callee / the caller / the destination address / the signalling address / the source address — *e.g. ‘the callee is me@plc.com’ or ‘the caller is among ann@plc.com, bob@uni.edu’*

or

amount — *e.g. ‘the bandwidth is 64’ (Kbps) or ‘the cost is less than 2.5’ (units)*

or

description — *e.g. ‘the network type is SIP’ or ‘the topic excludes finance’*

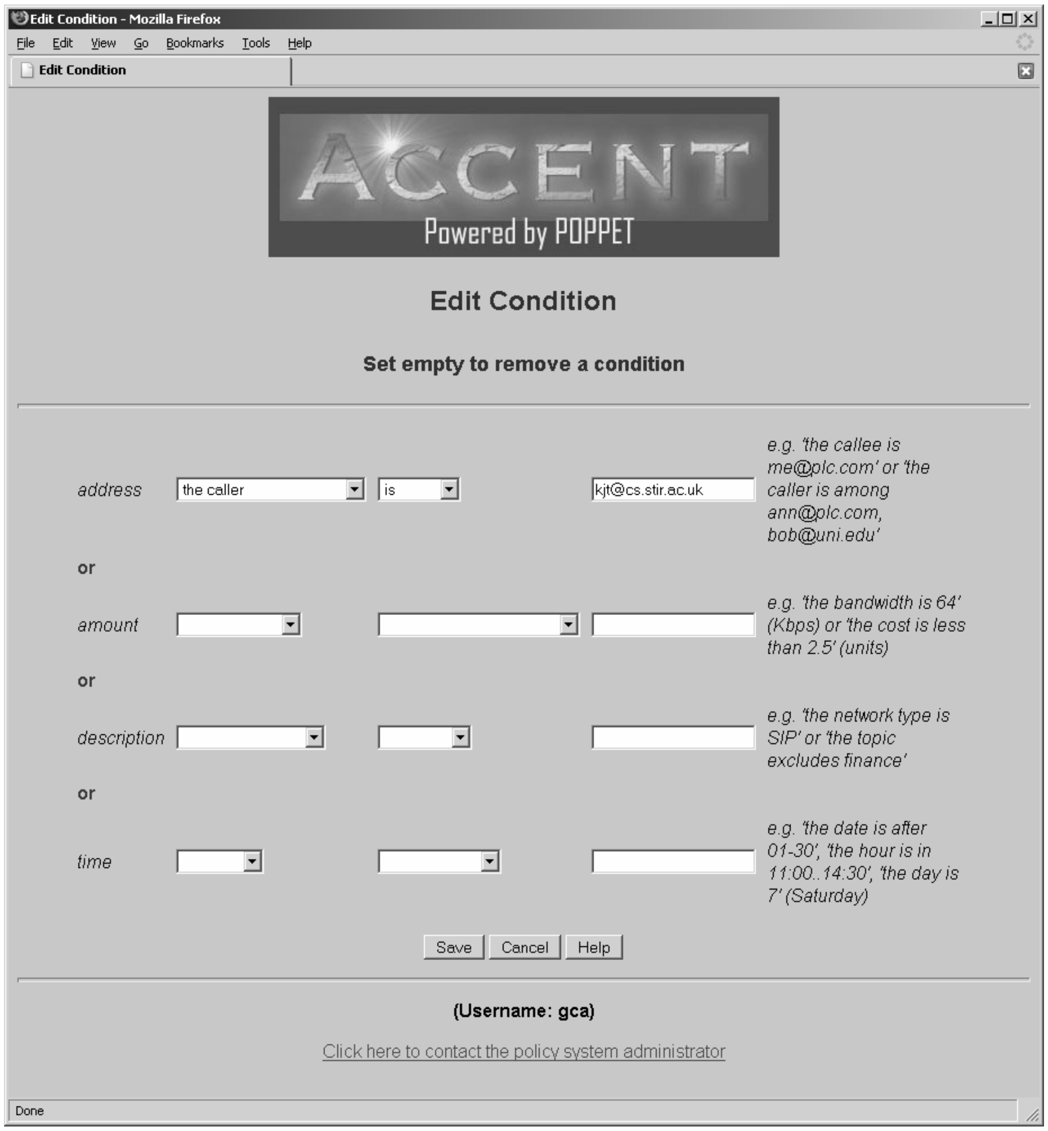
or

time — *e.g. ‘the date is after 01-30’, ‘the hour is in 11:00..14:30’, ‘the day is 7’ (Saturday)*

(Username: gca)

Figure 2.3 Policy Wizard Edit Condition Screen

- Once the user has selected the desired condition constraints, the condition is processed and appended to the policy by clicking the ‘Save’ button. When the button is selected, client-side JavaScript validates the form. If the user has failed to select a parameter or operator, or the entered condition value is of incorrect format or numerical range, an error message is displayed. When the data entered is detected to be valid, the details are sent to the policy wizard server. A similar cycle of JSP, Java, JavaScript and ontology model interaction then takes place to generate the page response to show the updated policy rule.


Edit Condition - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Edit Condition
ACCENT
Powered by POPPET

Edit Condition

Set empty to remove a condition

address	the caller	is	kjt@cs.stir.ac.uk	e.g. 'the callee is me@plc.com' or 'the caller is among ann@plc.com, bob@uni.edu'
or				
amount				e.g. 'the bandwidth is 64' (Kbps) or 'the cost is less than 2.5' (units)
or				
description				e.g. 'the network type is SIP' or 'the topic excludes finance'
or				
time				e.g. 'the date is after 01-30', 'the hour is in 11:00..14:30', 'the day is 7' (Saturday)

Save Cancel Help

(Username: gca)

Click here to contact the policy system administrator

Done

Figure 2.4 Policy Wizard Valid Condition Selection

While the bulk of domain-specific knowledge was limited to the Java code just described, a number of supporting wizard components also required alteration. In particular, although the majority of JSPs used are generic, a small subset relies on embedded JavaScript to handle domain-specific validation of user input. Currently, these are modified in order to tailor validity checking to a new domain, but could potentially be generated from the ontologies. For ease of current extension, template JSP's have been developed for use in a new domain application. These template files give clear guidance as to the minimum alteration required to permit basic input form validation.

In addition, the policy wizard encompasses a number of internal property files, used primarily for mapping and translating policy terms to natural language for display on-screen. The policy wizard supports a multi-lingual user-interface environment, which is achieved using a mapping file corresponding to each supported language. These configuration files must be adapted for each domain. Template properties files were created to enable easier implementation of new domain terms in a policy.

2.3.2 Policy Wizard Extension

Aside from the re-coding of existing components, the policy wizard was extended to include a dedicated driver `PolWizDriver.java`. The driver is responsible for forming a connection via RMI to access and query the ontology model object held by POPPET, and handles all communication between the policy wizard and the POPPET system. It contains hard-coded knowledge of the `genpol` and `wizpol` ontology structures, including URI strings of specific ontology classes – such as top-level policy language structures of `Action` and `TriggerEvent`. Provided a domain-specific ontology has imported these ontologies and defined policy-related domain knowledge as extensions to their structures, the domain knowledge can be successfully retrieved. Requests and responses between the driver class and POPPET take the form of primitive data structures, including lists of Strings and Integer values. It is important to note that the means through which an ontology document is processed and queried is entirely abstract from the point of view of both the driver and the wider policy wizard. The next section provides an overview of how to customise the approach for a new domain using call control as an example.

2.4 Policy Wizard Customisation

Using a specially designed ontology, the policy wizard can be customised for any given domain. In this section a worked-example is given of the process involved in applying the new ontology-driven approach to a new domain. The domain of call control is used as an example field.

To summarise, the main steps involved in customising the policy wizard are:

- Create a new OWL ontology, define its namespace URI, and import the generic policy ontologies of `genpol` and `wizpol`.
- Create named triggers, conditions and actions.
- Define categories of trigger, condition parameters and actions.
- Place restrictions on triggers, conditions, operators, actions and other policy components to associate categories
- Define further non-policy related knowledge.
- Configure POPPET to access the new ontology.
- Configure the policy wizard.

The process is described in the following sub-sections.

2.4.1 Domain-specific Ontology Creation and Ontology Importing

Domain-specific ontologies are created using the policy ontology stack explained in section 2.1. For an overview of the generic ontologies `genpol` and `wizpol`, refer to the technical report 'Ontology Stack for A Policy Wizard' [3]. For a detailed description of the devised call control ontology, used as an example in this section, refer to [4].

Protégé [13] is a widely used tool used to create and manage ontologies. It provides a graphical user interface framework through which to define and edit ontology documents, and supports automated reasoning capability via any external Description Logic compatible reasoning engine, such as RacerPro [14]. The use of Protégé is recommended when developing a domain-specific ontology for the policy wizard.

The first step in creating an ontology for a new domain is to create a fresh OWL document and assign it an appropriate namespace.

- A recommended approach is to create a new OWL/RDF project using Protégé. To do this, the new ontology must be given a name and namespace URI. For call control, the OWL ontology file name is `callcontrol.owl`, while the namespace URI is defined as:

`http://www.cs.stir.ac.uk/schemas/callcontrol.owl`

- To extend the core policy language the new ontology must import the generic ontology `genpol` and its policy wizard-specific extension `wizpol`. Using Protégé, only the `wizpol` ontology needs to be directly imported due to the nature of the OWL importation method. As `wizpol` itself imports `genpol`, there is no need to import `genpol` separately – its location is automatically inferred and included within the new ontology. `Wizpol` is located at:

  `http://www.cs.stir.ac.uk/schemas/wizpol.owl`

  The recommended namespace prefix is simply "wizpol". After importing `wizpol`, the new, domain-specific ontology has the power to utilise any class or property defined in the generic policy ontologies.

When Protégé opens the project, it locates and imports the content of `genpol` and `wizpol` automatically. The new ontology is now ready for customised domain extension.

2.4.2 Domain-specific Triggers, Condition Parameters and Actions

Using imported `wizpol` classes and properties, the ontology can be extended to define triggers, condition parameters and actions specific to the new domain. The process of defining triggers, condition parameters and actions is identical. As an example, the process of defining a trigger is as follows:

- Firstly, create a subclass of `wizpol:NamedTriggerEvent`. For call control, the class `ConnectIncomingCall` was defined to describe an incoming call connection event.
- Optionally, enter a description under the `rdfs:comment` attribute tag, to describe the purpose of the class for use internally when browsing the ontology.
- Create a new label property annotation using the tag `rdfs:label`. Labels are used by the POPPET system to integrate and match ontology classes. This is a way of maintaining separation of OWL ontology class names (the unique name of an ontology class) from the trigger name acknowledged by the policy system. In addition, the label text is listed in policy wizard mapping property files, where it is used to translate ontology data into natural language terms for screen display.

This process just described can be repeated for all triggers, condition parameters and actions. Classes are created under `wizpol:NamedTriggerEvent`, `wizpol:NamedCondParam` and `wizpol:NamedAction` respectively.

2.4.3 Categorisation of Triggers, Condition Parameters and Actions

Categories are used to group triggers, conditions and actions for display purposes. A category is created as a subclass of `wizpol:ClassCategorisation`. Specifically, categories related to triggers, conditions and actions are expressed by defining subclasses of `wizpol:TriggerCategory`, `wizpol:ConditionParamCategory` and `wizpol:ActionCategory` respectively. The process of category definition is identical for triggers, conditions and actions.

- Two trigger categories were defined for the call control domain: `CallTriggerCategory` and `AvailabilityCategory`, as shown in Figure 2.5. As all ontology class names must be unique the word 'trigger' was included in the Call category name to distinguish it from a call category used under ActionCategory. Crucially, each category class must have a label property annotation under the tag `rdfs:label`. In this example, label text used was 'call' and 'availability'.

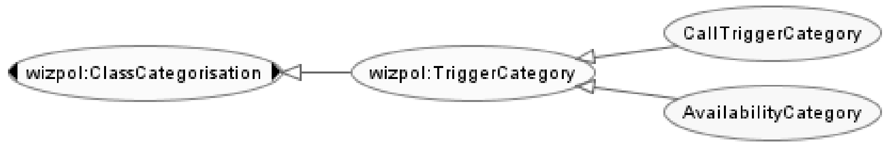


Figure 2.5 Call Control Trigger Categories

- For inference purposes, subclasses of `genpol:TriggerEvent`, `genpol:ConditionParameter`, `genpol:ConditionOperator` and `genpol:Action` must be created to represent each category.

  Continuing the call control trigger example, the two trigger categories are defined as subclasses of `genpol:TriggerEvent`, named `CallTriggerEvent` and `AvailabilityTriggerEvent`, as shown in Figure 2.6. Each class must have an associated label annotation which is identical to that defined for the subclasses of `wizpol:TriggerCategory`. For example, `CallTriggerEvent` must have the same label text as the class `CallTriggerCategory`. In this case, the label of both classes is 'call'.

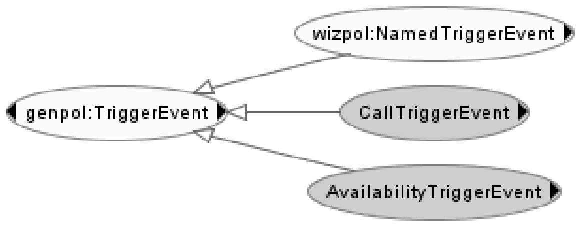


Figure 2.6 Call Control `TriggerEvent` Subclasses

- Assign the property restriction `wizpol:hasCategory` to every trigger, condition parameter and action class. This property associates each class with the relevant category defined under `wizpol:ClassCategory`. This property restriction must also be assigned to category subclasses. For example, the trigger category `CallTriggerEvent` is assigned the restriction:

  ```
  wizpol:hasCategory some CallTriggerCategory
  ```

2.4.4 Required Property Restrictions

Essentially, the domain ontology must apply restrictions to each trigger, condition parameter and action, to specify associated user-levels and arguments. In addition, every trigger must include restrictions to specify the various conditions and actions permitted to be associated with it. An outline of required property restrictions is given below.

- Firstly, assign user level restrictions to each named trigger, condition parameter and action. The restriction should be placed along the property `wizpol:hasUserLevel`. There are four user levels. If a class is to be associated with more than one user-level, a separate restriction must be placed on the class for each. For example, the `LogEvent` action in call control is associated with Admin, Expert and Intermediate users.

- Triggers are defined to have a related set of condition parameters and actions. Each trigger must specifically define associated conditions and actions using the properties `genpol:hasPermissibleParameter` and `genpol:hasPermissibleAction`. A separate restriction must be defined to link each parameter and action individually.

- Triggers and actions may have a number of associated arguments. For example, the `NoAnswer` trigger has an argument `TimePeriodTriggerArg`. The argument represents a specific user-defined value in a policy. Arguments are defined as follows:
    - Create a class to represent the argument as a subclass of either `genpol:ActionArgument` or `genpol:TriggerArgument`.
    - Place a restriction on the appropriate named trigger or action using the property `genpol:hasTriggerArgument` or `genpol:hasActionArgument` respectively.
    - For the same property, define a cardinality restriction to describe the number of arguments associated with the trigger or action. For the `NoAnswer` trigger, there is only one argument. The cardinality restriction is therefore:

      ```
      genpol:hasTriggerArgument exactly 1
      ```

2.4.5 Further Class and Property Extensions

In addition to the essential class and property extensions just described, a domain-specific ontology can specify data type information, unit type information, and define certain triggers and actions to be of internal use to the policy system.

- Data type information can be defined by creating subclasses of `wizpol:DataType`. The property `wizpol:hasDataType` can be used to place restrictions on `genpol:TriggerArgument` and `genpol:ActionArgument` subclasses, to enable the type of associated data to be queried. This is a crude way of defining data types in OWL ontologies, which may be improved should customised data-typing be incorporated within future revisions of the OWL specification.
- Unit type information can be defined by creating subclasses of `wizpol:UnitType`. The property `wizpol:hasUnitType` can be used to place restrictions on relevant trigger or action arguments to associate them with a particular unit of measurement. In call control, the `NoAnswer` trigger has an argument `TimePeriodTriggerArg`. The time measurement is seconds, defined by the class `SecondsUnitType`.
- The wizpol ontology defines the class `wizpol:InternalUse`. The property `wizpol:hasInternalUse` can be used to place restrictions on triggers and actions which are of internal use to the policy wizard. For example, in call control, a number of triggers refer to the state of internal system variables such as `AddressPresent` and `AddressAvailable`.

2.4.6 Additional Non-Policy Related Domain Information

A domain-specific ontology is encouraged to contain additional class and property information, out-with the imported class hierarchy of `genpol` and `wizpol`. Although such information cannot be used directly within the current policy wizard, it provides wider definition of the domain in question, offering high-level, operational knowledge which is useful to gain a broad overview of the domain.

2.4.7 POPPET Configuration

The POPPET server can parse and model any compatible policy ontology document. POPPET is initiated at the command line by specifying the URL of the ontology document to be read as an argument. A second argument must also be specified to indicate whether test ontology parsing data should be output during start-up. Test output includes a list of all classes and properties within the ontology model and a selection of test class and restriction queries by POPPET. The test facility is useful to obtain a visual indication that the ontology data parsed by POPPET is accurate. Should these initial tests fail, it may hint at errors in the form of the ontology – namely that it does not conform to the designed policy ontology stack. The following command invokes POPPET for the call control ontology:

```
java  PoppetServer  http://www.cs.stir.ac.uk/schemas/callcontrol.owl  true
```

The above command initiates the POPPET server and points it to the location of the ontology to be modelled. The second argument "true" will result in output of test data during setup. Should the specified ontology location be an invalid URL, POPPET will issue an error and abort any parsing attempt. Additionally, the `rmiregistry` must be running in order to register the ontology model. Similarly, if the `rmiregistry` is not detected, POPPET will issue an error during set-up.

2.4.8 Policy Wizard Configuration

The wizard utilises a number of Java property files to specify literal language representations of screen-displayable policy components. Two files require alteration:

- `mapping.properties    (…\WEB-INF\lib\)`
- `wizard.properties     (…\WEB-INF\lib\<locale_dir>)`

The `mapping.properties` file maps domain-specific terms to policy wizard terms. The `wizard.properties` file maps triggers, conditions, actions and other policy wizard constructs, with corresponding literal language translations. The wizard supports a number of different languages, including English, US English, Canadian French, French and German. There must be an equivalent `wizard.properties` file for each language. To correctly match policy terms with ontology classes, this file contains a natural language representation for each ontology class label. For example, the call control trigger class `NoAnswer` has an `rdfs:label` annotation in the ontology bearing the text 'no_answer'. Using the two properties files, this is translated as follows:

The `mapping.properties` file contains an entry:

```
no_answer              policy.noanswer
```

Within the `en-gb` (British English) directory, the `wizard.properties` file contains an entry:

```
policy.noanswer        a call is not answered
```

A different `wizard.properties` file must be constructed for each language supported so that terms may be tailored for each dialect. For ease of implementation, templates of both these files are available for immediate domain customisation. The `rdfs:label` text defined for triggers, condition parameters, actions and all other domain specific ontology classes must be listed within these files. If a label mapping is not present within the properties files, the wizard will fail to translate into natural language terms and output a label instead. In the example above, the wizard would display 'no_answer' instead of 'a call is not answered'.

Currently, the wizard requires JavaScript handing to be customised separately. For example, in call control, the action argument 'add_medium' has a limited set of allowed values – namely 'Audio', 'Video' or 'Whiteboard'. Presently, these options cannot be gleamed from the ontology and must be coded directly within JavaScript in `EditAction.jsp` to successfully validate user input. Template JSP's have been designed for `EditAction.jsp`, `EditCondtion.jsp` and `EditTriggger.jsp`. These provide minimal JavaScript skeleton code, which can be extended and customised for a new domain.

3 Conclusion

Faced with the challenge of generalising the existing ACCENT policy system to support policy creation for multiple application domains, the concept of ontology was employed to externally define domain knowledge and re-engineer the system to eliminate hard-coded domain-specific references. To integrate the developed ontology with the policy wizard, the POPPET framework was developed to model an ontology and provide an interface for the wizard to query the knowledge stored within it.

While this solution significantly enhanced the policy system, Section 3.1 considers the impact such alteration has had upon system usability and evaluates the overall technical design. Section 3.2 provides an overview of immediate and future extensions to the policy wizard and the POPPET framework.

3.1 Design and Performance Evaluation

In comparison with the original system, the improved policy wizard maintains all functionality of the previous implementation and promotes complete domain independence. The layout of the interface and validation of user input has also been preserved. Therefore, in terms of functionality, the new system does not sacrifice any previous features.

In general, the system design is sufficiently distributed and generic that individual components may be altered in isolation. For example, as the POPPET system encapsulates all ontology parsing and reasoning functions, it is entirely feasible to modify the parser or reasoning engine, from Jena or Pellet respectively, to another existing or proprietary application, without affecting the policy wizard or developed ontologies. Similarly, alterations to the policy wizard handling methods, or the web technologies behind the user interface, can be made with no impact on the methods used to parse or model the ontology. Such extensibility is extremely useful for future adaptation of the system for new domains. As a result of component distribution, ontology integration is invisible to the end user or indeed the core policy wizard. The wizard driver is unaware of the methods used to access an ontology document or where these may be physically located.

From the user's perspective, the only noticeable difference from the previous system is in the speed with which page requests are displayed. A slight overhead is incurred in the time taken to process and display pages in the policy wizard, mainly attributed to the use of RMI. As the original implementation contained hard-coded data, any alteration immediately introduces a degree of delay. Bearing this in mind, and considering the level of processing involved in querying an ontology, the response time is believed to be acceptable. Certainly, the delay is not so great as to either frustrate the user or render the interface unusable. Typically, the maximum time taken to display a requested page was between two and three seconds. The delay is however dependant on the speed and state of network connectivity. Should the wizard fail to establish a connection with the POPPET system at any point, rather than present an incomplete page response – including skeleton option lists, incorrectly formatted data or missing page sections – an error message is displayed in place of the regular page body content.

Potential solutions to performance issues associated with processing time delay include re-housing the POPPET system within the same servlet space as the core policy wizard components or using an alternative method of communication to RMI. However, these options would reduce the benefits of distributing the systems. The use of RMI allows the POPPET server to be placed on a different host to the policy wizard, allowing for flexibility in how the system may be deployed. The argument for RMI was explained in Section 2.2.3.

3.2 Scope for Future Work

The motivating reason behind replacing hard-coded domain information with dynamically retrieved ontology data was to generalise the system so it may be adapted easily for use in multiple domains. With a successful system in place to achieve this, the work described in this report is of immediate use in tailoring the ACCENT system towards a new domain. Specifically, an ontology may be defined that captures the new domain based on the ontology stack outlined in Section 2.1. The new ontology can be modelled using POPPET and interpreted by the policy wizard. Following application within a new domain, the developed tools and techniques may be revised in light of changing interface features or modifications to the base policy language ontologies.

In terms of improving the current system, there are several areas which could be given further thought. When developing the present solution, focus was placed on the design of a reusable ontology framework and the removal of embedded domain knowledge within the policy wizard, rather than the speed and efficiency of processing. Therefore, the implemented architecture may be altered to improve general operational efficiency. In particular, steps could be taken to reduce the time taken to reduce the access time overhead to query and relay ontological data. While the use of RMI has clear advantages in terms of providing secure resource distribution, the access time overhead may be shortened if a simpler, more centralised solution was implemented. In addition, local caching of common ontology queries would reduce the number of remote method calls and speed up processing.

Additionally, improvements could be made to further generalise and automate the policy wizard with regard to customised JavaScript validation of form input and particular internal property files. Currently, a small number of JSPs contain JavaScript code used to perform validation checking on form data input. A more efficient and generic method could be to allow the ontology to specify the exact nature of data values allowed within policy form input. The developed ontologies currently have the power to specify the data type of a value, but due to present limitations of the OWL DL sub-language there is no standard method to support data typing in a similar way to that of XML schema. Should this issue be resolved in the near future, this would be an obvious extension to the system.

Similarly, there are a number of internal property mapping files within the policy wizard that are responsible for natural language translation of policy terms. It would be possible to store language translations within the ontology, although the external files do provide a simpler means through which to easily maintain and update these mappings. In comparison to the structure of the developed policy ontologies, such mapping information may also be considered as a data component and perhaps something that should be kept separate. It is, however, a topic for further consideration.

3.3 Conclusion

Through successful ontology integration, the ACCENT policy-based system now has the power to provide users with a means to define and edit policies in multiple domains, and is no longer limited to call control. Furthermore, complete functionality of the original system has been preserved – the only noticeable change to a user being that the speed of operation is marginally slower due to the introduction of complex ontological integration. Overall, the techniques and systems developed are viable for immediate reuse in tailoring the policy system to support policy-based management in a new application area.

References

[1] ACCENT Policy-based system. Project home page: http://www.cs.stir.ac.uk/accent. June 2006.

[2] Apache Tomcat. Home page: http://tomcat.apache.org/, June 2006.

[3] Campbell, G.A., *Ontology Stack for a Policy Wizard.* Technical Report CSM-169, June 2006.

[4] Campbell, G.A., *Ontology for Call Control*. Technical Report CSM-170, June 2006.

[5] Generic Policy Language Ontology document (`genpol.owl`). Located online at URL: http://www.cs.stir.ac.uk/schemas/genpol.owl, June 2006.

[6] Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi/, June 2006.

[7] Jena: A Semantic Web Framework for Java. http://jena.sourceforge.net/, June 2006.

[8] Ontology for Call Control (`callcontrol.owl`). Located online at URL: http://www.cs.stir.ac.uk/schemas/callcontrol.owl, June 2006.

[9] OWL: The Web Ontology Language. http://www.w3.org/2004/OWL/, May 2006.

[10] OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/, June 2006.

[11] OWL Web Ontology Language Semantics and Abstract Syntax. http://www.w3.org/TR/owl-semantics/, June 2006.

[12] Pellet OWL DL Reasoner. http://www.mindswap.org/2003/pellet/index.shtml, June 2006.

[13] Protégé home page: http://protege.stanford.edu/, June 2006.

[14] Racer Systems GmbH & Co. KG. Home Page and download links: http://www.racer-systems.com/index.phtml. May 2006.

[15] RDF: The Resource Description Framework. http://www.w3.org/RDF/, May 2006.

[16] RDF Schema (RDFS): http://www.w3.org/TR/rdf-schema/, June 2006.

[17] Reiff-Marganiec, S., Turner, K.J., *APPEL: The ACCENT Project Policy Environment/Language*. Technical Report CSM-161, June 2005.

[18] Reiff-Marganiec, S, Turner, K.J, *The ACCENT Policy Server*. Technical Report CSM-164, May 2005.

[19] SPARQL: Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/, June 2006.

[20] Turner, K.J., *The ACCENT Policy Wizard*. Technical Report CSM-166, May 2005.

[21] Wizard Policy Language Ontology document (`wizpol.owl`). Located online at URL: http://www.cs.stir.ac.uk/schemas/wizpol.owl, June 2006.