A Goal-Directed and Policy-Based Approach to System Management

A thesis submitted in accordance with the requirements of
the University of Stirling for the degree of Doctor of Philosophy

by

Gavin Andrew Campbell

Department of Computing Science & Mathematics
University of Stirling, Scotland, UK

December 2008

Declaration

I hereby declare that the work presented in this thesis is my own original work unless otherwise indicated in the text, and that it has not been submitted for any other degree or award. Where work presented in this thesis appears in publications of which I am a named author, references are indicated in the text.

In particular, I note the following contributions from others:

- The ACCENT policy system and APPEL policy language discussed in Chapter 3 was used as a basis for the extensions and tools presented in this thesis. The original authors were Stephan Reiff-Marganiec, Lynne Blair, Kenneth J. Turner and Jianxiong Pang.
- The approach to policy conflict filtering presented in Chapter 5 was research proposed by my supervisor Kenneth J. Turner. My contribution was to develop the ontology and tool support to implement and apply the approach.
- The goal-directed approach presented in Chapter 6 evolved through collaboration with my supervisor Kenneth J. Turner.

Gavin A. Campbell

December 2008

Abstract

This thesis presents a domain-independent approach to dynamic system management using goals and policies. A goal is a general, high-level aim a system must continually work toward achieving. A policy is a statement of how a system should behave for a given set of detectable events and conditions. Combined, goals may be realised through the selection and execution of policies that contribute to their aims. In this manner, a system may be managed using a goal-directed, policy-based approach.

The approach is a collection of related techniques and tools: a policy language and policy system, goal definition and refinement via policy selection, and conflict filtering among policies. Central to these themes, ontologies are used to model application domains, and incorporate domain knowledge within the system. The Accent policy system (Advanced Component Control Enhancing Network Technologies, `http://www.cs.stir.ac.uk/accent`) is used as a base for the approach, while goals and policies are defined using an extension of Appel (Adaptable and Programmable Policy Environment and Language, `http://www.cs.stir.ac.uk/appel`).

The approach differs from existing work in that it reduces system state, goals and policies to a numerical rather than logical form. This is more user-friendly as the goal domain may be expressed without any knowledge of formal methods. All developed techniques and tools are entirely domain-independent, allowing for reuse with other event-driven systems. The ability to express a system aim as a goal provides more powerful and proactive high-level management than was previously possible using policies alone. The approach is demonstrated and evaluated within this thesis for the domains of Internet telephony and sensor network/wind turbine management.

Acknowledgements

I gratefully acknowledge the support of the UK Engineering and Physical Sciences Research Council for funding this PhD work (under grant C014804), carried out as part of the Prosen project (Proactive Condition Monitoring of Sensor Networks – `http://www.prosen.org.uk`). My thanks also to fellow Prosen researchers at the Universities of Lancaster and Strathclyde, whose collaboration contributed to the development and application of this thesis work within the sensor network and wind turbine domain.

I would like to thank all the academic, secretarial and technical staff within the Department of Computing Science & Mathematics who have aided me throughout the course of my studies at Stirling.

Thank you to all the past and present PhD students within the department who have been an invaluable support network and willingly had their (technical) ears bent from time to time – in particular, Liam Docherty, Paul Godley, Chris McCaig and Lloyd Oteniya.

Finally, many thanks to my supervisors Prof. Ken Turner and Prof. Evan Magill. This work could not have been completed without their technical help and general guidance. In the case of Prof. Turner, this gratitude extends to the marathon proof reading and red pen assaults on drafts of this thesis, and the multitude of lengthy discussions that helped shape the work presented here.

To my parents, their parents, and Jordan J.

List of Publications

Based on the work presented in this thesis, the following technical reports and papers have been published, listed in chronological order.

1. G. A. Campbell. An overview of ontology application for policy-based management using Poppet. Technical Report CSM-168, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

2. G. A. Campbell. Ontology stack for a policy wizard. Technical Report CSM-169, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

3. G. A. Campbell. Ontology for call control. Technical Report CSM-170, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

4. G. A. Campbell. Ontologies for resolution policy definition and policy conflict detection. Technical Report CSM-172, Department of Computing Science and Mathematics, University of Stirling, UK, February 2007.

5. G. A. Campbell and K. J. Turner. Ontologies to support Call Control Policies. In N. Meghanathan, D. Collange and Y. Takasaki, editors, *Proc. 3rd Advanced International Conference on Telecommunications* (AICT'07), pages 5.1-5.6, IEEE Computer Society, New York, May 2007

6. K. J. Turner, G. A. Campbell and F. Wang. Policies for Sensor Networks and Home Care Networks, in Mohammed Erradi (ed.), *Proc. 7th International Conference on New Technologies for Distributed Systems*, pages 273-284, ISBN 9981-9704-7-6, June 2007.

7. G. A. Campbell and K. J. Turner. Policy Conflict Filtering for Call Control. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th International Conference on Feature Interactions in Software and Communications Systems*, pages 93108, France, September 2007. IMAG Laboratory, University of Grenoble

8. G. A. Campbell. Sensor Network Policy Conflicts. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th International Conference on Feature Interactions in Software and Communications Systems*, France, September 2007. IMAG Laboratory, University of Grenoble

9. G. A. Campbell and K. J. Turner. Goals and Policies for Sensor Network Management. In M. Benveniste, B. Braem, C. Dini, G. Fortino, R. Karnapke, J. L. Mauri and M. S. H. Monsi, editors, *Proc. 2nd International Conference on Sensor Technologies and Applications* (SENSORCOMM'08), pages 354-359, IEEE Computer Society, Los Alamitos, California, August 2008.

Contents

List of Figures

List of Abbreviations

ACCENT	Advanced Component Control Enhancing Network Technologies
AI	Artificial Intelligence
APPEL	Adaptable and Programmable Policy Environment and Language
API	Application Programming Interface
CPU	Central Processing Unit
EBNF	Extended Backus–Naur Form
FI	Feature Interaction
GenPol	Generic Policy Language Ontology
GUI	Graphical User Interface
GW	Gigawatts
HTML	Hyper Text Markup Language
JVM	Java Virtual Machine
kV	Kilovolts
kW	Kilowatts
MW	Megawatts
OWL	Web Ontology Language
POPPET	Policy Ontology Parser Program – Extensible Translation
PROSEN	Proactive Condition Monitoring of Sensor Networks
PSTN	Public Switched Telephone Network
RECAP	Rigorously Evaluated Conflicts Among Policies
RMI	Remote Method Invocation
SCADA	Supervisory Control And Data Acquisition

SIP	Session Initiation Protocol
URI	Uniform Resource Indicator
URL	Uniform Resource Locator
V	Volts
VoIP	Voice over Internet Protocol
W3C	World Wide Web Consortium
W	Watts
WizPol	Wizard Display Ontology
WWW	World Wide Web
XML	eXtensible Markup Language

non sine pulvere palma

Chapter 1

Introduction

1.1 Thesis Statement

The claim of this thesis is that goals and policies provide an effective approach to managing systems. This thesis aims to show that a goal-directed policy-based approach offers high-level control of several kinds of systems, and contributes toward proactive management of both the system and the environment it monitors.

The domains in which the thesis ideas are evaluated are Internet telephony and sensor network/wind turbine management. Policies have previously been applied to telephony, but their use in the context of sensor networks and wind turbine management has received limited prior attention. Goal-directed configuration for telephony and sensor network/wind turbine management is an entirely new application area.

Within this thesis, each chapter deals with a different aspect of the approach to goal- and policy-based management. Although this thesis investigates and applies goals and policies to Internet telephony and sensor network/wind turbine management, the overall approach is applicable to any domain specialisation of the policy language.

1.2 Context Overview

In a changing system environment, the handling of an event depends on the circumstances of its occurrence. The most appropriate response at any given point varies with time, system state and other run-time factors. A management system should offer customised event handling based on such conditions. Furthermore, this must be achieved in real time without interrupting normal system service. This thesis presents an approach to dynamic system management using goals and policies.

A goal is a general, high-level aim a system must continually work towards achieving. Goals can be achieved through appropriate sets of actions that accomplish them. A policy is a statement of how a system should behave for a given set of detectable events and conditions. A goal is distinct from a policy in several ways. Goals are abstract aims and objectives of a system, which are not immediately achievable through any single low-level system action. In contrast, a policy is a structured description of how a particular (detectable) system event can be handled using actions that dynamically modify system behaviour. Goals are expressed as general aims that do not consider the technical capabilities of the underlying system, whereas policies relate directly to specific system behaviour. As a general example, consider a goal and a policy from a vehicle management system:

Goal: Minimise vehicle breakdown

Policy: Run a full diagnostic checking program on the engine
when an oil warning light is activated

The goal is to avoid a vehicle breakdown. The goal itself does not specify any action(s) to take to in order to achieve it. The policy is more detailed and defines a course of action (run a diagnostic check) to take when a specific event occurs (a warning light is activated). Combined, goals may be realised through the selection and execution of policies that contribute to their aims. In the example above, the given policy might be selected (along with other policies) to aid the given goal. In this manner, a system may be managed using a goal-directed, policy-based approach.

Goals have been used extensively in the field of Artificial Intelligence for applications such as robotics and agent-based programming. While the concept of goal-directed behaviour is not new, the use of goals to direct policies is still an ongoing research topic, and no goal-directed policy-based technique has been fully implemented or widely adopted. Existing and related goal approaches in this field are described in detail in

Chapter 6, including approaches that use formal methods to define and refine goals and policies (e.g. [35] and [93]) and policy hierarchies that view goals as more abstract policies (e.g. [82] and [55]).

Policies originally gained popularity in the realms of system security and access control, but their use has subsequently widened. The Accent project (Advanced Component Control Enhancing Network Technologies [13]) supported policies for Internet telephony, allowing end users to customise their call handling preferences. The policy system framework developed as part of Accent was used as a base for the approach presented here.

Each aspect of the approach described in this thesis is applied to the domains of Internet telephony and sensor network/wind turbine management. The broader context of these application areas is described in Chapter 2.

1.3 Objectives

The purpose of this work was to design a generic approach to high-level system management using goals and policies. For testing and demonstration purposes, this approach is applied within the separate domains of Internet telephony and sensor network management.

Building on the existing Accent policy system, the main objectives of this work were:

- To define a language through which goals may be represented
- To design and implement a process of refining a set of high-level goals into a set of policies that achieve them
- To define a policy language for the domain of sensor network management
- To enhance existing policy conflict handling using automated filtering of policy actions
- To generalise the whole approach by modelling domain-specific knowledge in ontologies

- To develop supporting software tools to integrate and utilise ontology-defined knowledge within the generic policy and goal systems and related components.

1.4 Approach

The approach is a collection of related techniques and tools: the policy language and policy system, goal definition and refinement via policy selection, and conflict filtering among policies. Central to these themes, ontologies were used to model the application domain and to incorporate domain knowledge within the system.

Policies form the basic unit of event handling. A policy is defined in a particular policy language. The language used here is an extension of Appel (Adaptable and Programmable Policy Environment and Language [91]) which was developed previously for Internet telephony using the Accent policy system.

Conflicts may arise between the actions of policies at run-time. Manual methods for identifying such conflicts were previously implemented in the Accent system. This approach has been enhanced by the construction of an automated filtering tool (named Recap – Rigorously Evaluated Conflicts Among Policies) which analyses policy actions and their parameters for potential conflicts. The approach is generic and applicable to any domain specialisation of the policy language – no domain-specific information is hard-coded in the tool. The tool also automatically generates resolution policy templates for detected conflicts. Resolution policies are similar in structure to regular policies but are triggered by pairs of conflicting regular policy actions instead of specific system events. The action of a resolution policy selects one of the conflicting regular policies for execution.

Goals are defined as an extension of Appel. The process of refining goals into policies is achieved in two stages: static filtering and dynamic optimisation. The system is initialised for a given set of goals and prototypes (policy templates). Statically, the prototype set is filtered: prototypes contributing toward goals are selected and instantiated as policies. Dynamically, as the system is notified of events, applicable policies are optimised against a goal evaluation function and are optionally parameterised based on the current state of the system.

To ensure the generality of the system, all domain-specific knowledge is defined within ontologies. An ontology defines the terms used to describe and represent an area of knowledge, together with the logical relationships among these terms. A framework of interrelated ontologies has been developed to model the generic and domain-specific elements of the Appel policy language. Additional domain information for goal analysis and policy conflict detection is also modelled. Ontologies are made accessible to goal and policy system components via a stand-alone query engine and parser named Poppet (Policy Ontology Parser Program Extensible Translation).

1.5 Achievements

A goal-directed, policy-based approach has been developed and implemented. The approach has been demonstrated for Internet telephony and sensor network/wind turbine management. A goal language (based on an extension of the Appel policy language) has been developed, along with a system to refine goals into policies that achieve them. A policy language for sensor networks has been defined, also using Appel. To aid policy conflict handling, the Recap tool provides automated conflict filtering and resolution policy generation. Finally, a framework of ontologies has been developed to model generic and domain-specific aspects of Appel, along with domain knowledge pertaining to policy conflict analysis and system goals. Ontologies are utilised by system components via the generic interface of the Poppet tool.

Success of the whole thesis approach is measured in the combined functionality of the developed techniques and tools. The approach is entirely domain-independent in that no components of the goal and policy system and supporting tools have hard-coded details of any particular application area. This allows for reuse in new domains – through customisation of an ontology.

Reuse of the approach is constrained to event-driven systems that require a degree of autonomy in their control. Specifically, viable systems must be capable of generating and reporting events (changes in the system environment that act as policy triggers) and must also permit dynamic configuration of its components (services, resources or variables in the system environment alterable via policy actions). In turn, a suitable interface must be devised to enable the policy system to communicate with the managed system.

1.6 Thesis Structure

The following chapters each describe a different aspect of the goal-directed and policy-based approach. Each main chapter (with the exception of Chapter 2) opens with an introduction and background to its topic, and concludes with an evaluation of the work presented. The overall structure of this thesis is as follows:

Chapter 1: The introduction to this thesis: a summary of the context, objectives, approach and achievements.

Chapter 2: This provides an overview of the application domains in which the thesis work is applied. These domains are Internet telephony and sensor network/wind turbine management.

Chapter 3: This introduces policy-based management and describes the policy system and policy language used as a base for the approaches in this thesis. Policy language specialisations for Internet telephony and sensor networks/wind turbine management are described and example policies are given.

Chapter 4: This gives an introduction to ontologies and describes how they have been used to model policies and to generalise an existing policy system user interface. Ontologies developed to model policy language specialisations for Internet telephony and sensor networks/wind turbine management are also presented.

Chapter 5: This introduces the field of policy conflict and describes a new approach and tool for automated conflict filtering among policy actions. The tool is demonstrated for Internet telephony and sensor network/wind turbine management policies.

Chapter 6: This chapter introduces and summarises previous goal-related approaches, and presents new work on goal-directed policy-based management of systems. It describes how goals are defined and refined into policies which execute them.

The goal-directed approach is demonstrated for Internet telephony and for sensor network/wind turbine management.

Chapter 7: Concludes the work presented in this thesis.

Appendix A: This provides a technical explanation as to the values used to configure certain wind turbine-related parameters discussed in Chapter 3 and used in policy examples throughout the thesis.

Appendix B: This provides further technical properties relating to the ontologies described in Chapter 4.

Appendix C: This describes how the RECAP conflict filtering tool described in Chapter 5 is used and provides a screenshot of the application interface.

Chapter 2

Application Background and Context

This thesis describes generic policy-related approaches and tools which have been applied and demonstrated in two different application areas: Internet telephony and sensor network management within a wind farm environment. Two contrasting domains were selected in order to demonstrate the generality of the approach. This chapter provides a background to these domains and describes how they are used within this thesis. Section 2.1 introduces Internet telephony and explains how its characteristics are modelled in this thesis. Section 2.2 describes the domain of sensor networks/wind turbine management. A summary is given of wind farms and wind turbine operation, including existing wind farm condition monitoring and control systems.

2.1 Internet Telephony

2.1.1 VoIP: Internet Telephony

Traditional telephony can be described as the exchange of voice messages over a circuit-switched telephone network (commonly referred to as the Public Switched Telephone Network (PSTN)), using fixed-line analogue or digital transmission. In contrast, Internet telephony (or Voice over Internet Protocol (VoIP)) refers to the exchange of voice data over the Internet using a packet-switched protocol that encodes/decodes speech as streams of digital audio. Calls are initiated and received using a computer terminal or VoIP telephone handset connected to the Internet. As Internet connectivity has widened and bandwidth has increased (e.g. broadband network provision), VoIP has proved a popular method of communication that is more cost effective than the fixed-line PSTN.

Internet telephony offers similar features to a PSTN network, but for little cost and with greater user control over how features are configured. For example, services such as call forwarding, conference call set-up, and the ability to transmit/receive multiple calls using the same connection are standard features of a VoIP system. In addition, VoIP utilises an Internet connection. Calls may therefore be combined with multimedia features such as video streaming, instant messaging, status tracking, and media file transfer.

2.1.2 The Accent Project

Accent (Advanced Component Control Enhancing Network Technologies [13]) was a project funded by the EPSRC (Engineering and Physical Sciences Research Council) between September 2001 and March 2005. Research was carried out by the University of Stirling and supported by Mitel Networks Corporation (`http://www.mitel.com/`). The project aimed to develop an advanced language and system to allow individuals and enterprises to specify policies and preferences for call processing – particularly Internet telephony. The project developed a call control language to define user policies, to analyse conflicts in these, and to support policies in an operational environment.

Accent is also the name given to the generic policy-based framework resulting from the Accent project. The framework was originally applied within the domain of Internet telephony, to enable users to manage and configure call preferences.

Specific objectives of the project were:

- To define a user-friendly policy language for call control
- To perform rigorous offline analysis to detect conflicts among policies within this language

- To translate policies into XML scripts that are executed in a SIP (Session Initiation Protocol) server
- To detect and resolve policy conflicts at run-time
- To undertake trials to evaluate the project approach.

Notable achievements of the Accent project included Appel (Adaptable and Programmable Policy Environment and Language [91]) and its associated tool support for execution and conflict analysis. The Accent policy system was implemented in a three-layer architecture: a user interface layer (consisting of a policy wizard for non-technical users to create and edit policies), a policy server layer (to store and deploy policies), and a communications layer (to connect with the monitored system environment). For telephony, the Accent system was implemented to support three specific communication protocols: SIP, H.323 (an audio-visual protocol used in videoconferencing) and a proprietary PBX (Private Branch Exchange). The Accent support of telephony therefore deals with common features of these particular communications mechanisms.

Both Accent and Appel were adapted and extended for the techniques and tools described throughout this thesis. In particular, the Appel language has been specialised for the new domain of sensor network/wind turbine management. The Accent policy system and the Appel policy language are explained in detail in Chapter 3, along with a review of other main policy-based approaches.

The Accent system was chosen for use over other established systems as it offers a generic, readily reusable framework with strong support for policy creation and management through its policy wizard user interface. The Appel language is simpler and less formal than other established languages that offer more complex types of policies, and was deemed more appropriate for this thesis work. Additionally, other established policy systems and languages have evolved from the initial policy application area of resource management and promote features more geared towards policies for system security and user access control, rather than more general event-based policies. Appel is designed for more general management policies and has a core, generic set of language constructs that can be easily extended and specialised for new domains.

2.1.3 Call Control Using Policies

Accent policies include handling the set-up of calls and taking customised actions in the event of an incoming call request. The features supported in policies can be summarised as follows:

- Policies may detect a call connection, a call disconnection or a call that goes unanswered, and take appropriate actions.
- The caller may request an amount of bandwidth for use during a call, and either have this accepted or rejected by the recipient.
- The presence and availability of a particular user can be used to trigger actions.
- Call handling features include adding/removing callers in an existing call and forking/forwarding calls to other users.
- Multimedia handling includes adding/removing various call media (specifically, the ability to include audio, video or a digital whiteboard in a call) and playing recorded audio clips to callers.
- Policies may have conditions based on parameters associated with the call environment, such as the address of the caller/callee, the call content, type, cost, priority, quality, and the media in use.

2.1.4 Internet Telephony Domain

The Appel policy language, designed as part of the Accent policy system, was originally specialised for the domain of telephony. In particular, the language defines the types of triggering events, conditions and actions within a VoIP environment. These language components are described in full in section 3.3 which describes the policy language for this domain. The policy language for telephony was developed previously by researchers within the Accent project, and its application within this domain unchanged.

2.2 Sensor Networks/Wind Turbine Management

2.2.1 Wireless Sensor Networks

A wireless sensor network consists of sensor nodes which monitor, collect and communicate data between themselves (known as a peer-to-peer (P2P), ‘ad hoc’ or ‘mesh’ network) and/or with a central processing point. Individual sensor nodes are robust enough to be positioned in any internal or external environment and are designed to operate without human intervention for extended periods of time (often several years). Each sensor node may contain one or more individual sensors. Such sensors may measure aspects such as light, sound, motion, or environmental conditions (e.g. temperature). To operate wirelessly over long time periods, sensor nodes sustain themselves using battery power and have restricted processing capability, memory and use of bandwidth. To optimise power usage, sensor platforms have been designed that give software applications direct access to and control of hardware modules, to conserve energy by turning off particular sensors or subsystems within a node when not in use.

Wireless sensor networks have been applied in a wide range of applications, for example to monitor conditions in extreme climates, to aid home security systems, to track animals in their natural habitat, and to support military defence systems. The advantages of such networks lie in their ability to gather data within environments where a human presence is impractical, impossible or dangerous.

2.2.2 The Prosen Project

The Prosen project (Proactive Condition Monitoring of Sensor Networks – `http://www.prosen.org.uk`) was funded by the EPSRC (Engineering and Physical Sciences Research Council) between 1st October 2005 and 30th November 2008. The project combined research at the Universities of Essex, Lancaster, Strathclyde and Stirling, as well as collaboration with industrial partners. The work presented in this thesis formed part of the contribution from the University of Stirling.

The broad aim of Prosen was to investigate and develop new techniques for proactive condition monitoring in large-scale wireless sensor networks. In the context of the Prosen project, such techniques were developed and applied in the domain of wind farm operation. In summary, Prosen had the following overall aims (the last three being of particular relevance to this thesis work):

- the design of systems with a rich self-observation capability
- the construction of machines that can continuously monitor parallel streams of input
- the development of efficient proactive control techniques
- automatic configuration and maintenance of large-scale sensor networks
- the specification and implementation of goal-driven configuration management
- the federation of software services in such a distributed and changing environment
- the integration of novel research into a practical large-scale demonstration that encourages exploitation.

These aims were addressed through an investigation of techniques that enable the automated control and management of sensor arrays to be proactive. To achieve these aims, proactive control and management software was to be assisted through integration with a "policy-driven management infrastructure that uses high-level user goals to constrain the instrument-level proactive behaviours" [27]. The work presented in this thesis delivers a solution to this requirement.

2.2.3 Wind Power Introduction

The basic structure of a wind turbine is shown in Figure 2.1. A wind turbine produces electricity using natural wind power to drive a generator. Typically, turbines have three blades which rotate around a horizontal hub called the nacelle, at the top of a steel or concrete tower. A turbine generally starts to generate electricity at wind speeds of 8mph and can generate maximum output in speeds of 30mph. Risk of damage is liable when wind speed reach levels of 50mph. In such circumstances, turbines must be shut down to prevent malfunction. The height of a turbine ranges from 100-120 meters (m). Of this, towers are generally 60-80m and the blades can be 30-40m long [26].

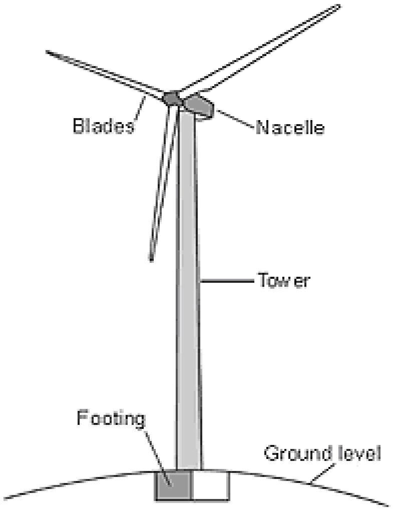


Figure 2.1: Wind Turbine Components (Source: `http://www.seda.nsw.gov.au`)

The internal components of a typical nacelle are shown in Figure 2.2. When in active operation, the turbine blades face into the direction of the wind. As the wind passes over the blades, it exerts a turning force which rotates the blades. The blades turn a shaft inside the nacelle which is normally connected to a gearbox. The gearbox works to increase the rotation speed for the generator. As in conventional forms of power generation, the generator uses magnetic fields to convert the rotational energy into electrical energy. Output power is then fed into a transformer to convert the generated electricity (from around 700 Volts (V)) to a level compatible with regional distribution networks or the National Grid (usually between 11 and 132 kilovolts (kV)) [26]. Within the nacelle, an electronic controller continuously monitors the condition of the wind turbine and collects statistics on its operation. The controller communicates bi-directionally with external systems and the operator via some communications link (e.g. telephone, radio or fibre optics) to send alarms or to report on the status of turbine components.

Wind speed and direction are detected using instruments on top of the nacelle. An anemometer measures the wind speed and a wind vane measures the wind direction. Both measurements are fed into the turbine and used to alter its operation. As the wind direction changes, the motors within the nacelle turn or “yaw” the rotor head so that the blades remain constantly in the face of the wind – known as yaw control. In

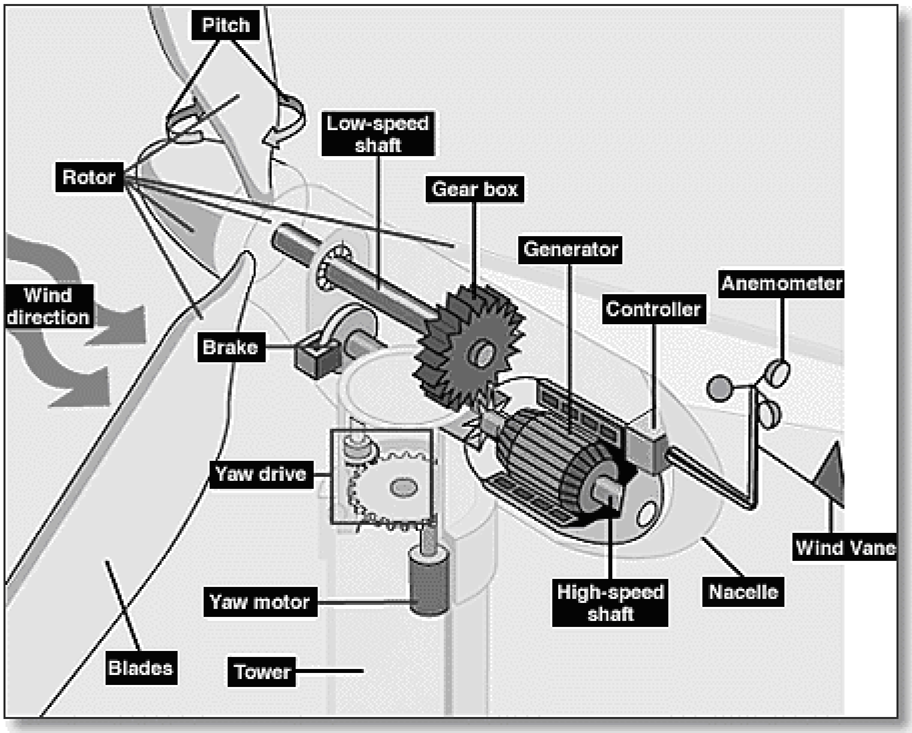


Figure 2.2: Wind Turbine Nacelle (Source: `http://www1.eere.energy.gov`)

addition, the blades are also angled or "pitched" to gain optimal power from the wind.

Wind turbines are monitored remotely, usually from a centralised control centre which may be off-site. The site may house security staff or maintenance engineers, but in general the turbines can operate effectively without direct human control.

2.2.4 Wind Farm Operation Issues

A major drawback of current wind farms is the percentage of time they can operate effectively. A typical turbine produces power only 75-80% of operational time, generating different levels of output depending on the speed of the wind. As a result a turbine produces only 30% of its theoretical maximum output. The lack of output from a turbine is increased by the downtime associated with scheduled turbine maintenance and repair time following any fault. This compares to a figure of around 50% for conventional power stations [25]. Specifically, wind turbines must be turned off when wind speed reaches gale levels of 50mph, to prevent malfunction and to reduce degradation caused by friction within internal components. Predicting trends in the external turbine environment, such as weather conditions and the impact on internal components is not easy. The key to improving operational efficiency lies in more effective monitoring of turbine components to detect the signs of typical problems and plan to rectify them before component failure and turbine downtime occurs.

2.2.5 Proactive Management

Proactive management of a system involves monitoring system components in an attempt to spot tell-tale signs of faults or degradation during normal operation, and take appropriate action to prevent more serious damage or failure within the system. This type of approach is in contrast to reactive management, which aims to take action (or react) to system events once damage has already occurred. Proactive management is a more preferable approach in the context of wind farms and mechanical systems in general. For proactive approaches to work, however, system conditions must be carefully monitored and the early signs of faults known in advance.

2.2.6 Existing Wind Farm Control Systems

Mechanical systems (including wind farms) operate using Supervisory Control and Data Acquisition (SCADA) systems [58]. SCADA systems were originally developed in the 1960s. Since then they have evolved rapidly, and are now utilised widely throughout industry for a variety of large-scale, industrial processes. Example systems include traditional power stations, natural gas utilities, water and sewage utilities, and telecommunications [28]. SCADA networks are implemented as software packages which are interfaced with the hardware components of a system. This provides a form of monitoring at a supervisory level. Operators cannot use a SCADA system to physically control hardware. Instead, they may gain insight into operational behaviour and make decisions based on this [58]. While SCADA provides such monitoring capabilities, it does not offer the means to proactively control a system. SCADA alone cannot support self-adaptation or dynamic component configuration based on changes in external and internal conditions. For this reason, dedicated condition monitoring can provide more effective support in system control.

2.2.7 Wind Farm Condition Monitoring

Condition monitoring is the act of continually observing and analysing a system to detect tell-tale signs of failure before fully-fledged faults develop. Previously, the main method of identifying wind turbine faults was for engineers to physically carry out checks on-site, using hand-held equipment and visual spot checks. Condition monitoring techniques can offer proactive management of wind farms, allowing operators to monitor the internal components of a turbine as well as its external environment. The main difference between condition monitoring techniques and existing monitoring systems, such as SCADA, is in their proactive capabilities.

Using built-in knowledge of the wind farm environment, condition monitoring can predict failures and output warnings, recommend actions to operators, or even issue control commands directly. In particular, an operator is given the chance to stop a turbine before excessive damage to components occurs [32]. Identifying potential faults early gives the opportunity to schedule maintenance and order replacement parts in advance. Such coordination of maintenance and repair greatly reduces the cost of manpower, specialist equipment sourcing, and replacement of parts following excessive damage. Furthermore, it can reduce the length of time a turbine is out of service and hence lessen the energy lost during downtime.

Mechanical faults in wind turbines develop gradually and can be identified through abnormalities such as “hot spots”, unusual vibration and debris in lubricants [32]. Sensors placed within internal components can be used to monitor temperature. When components become hotter than usual, it can signal frictional issues due to distortions, component degradation or lubrication failures. Lubricant monitoring, in oil for example, can indicate contamination by metal particles from gears and bearings. Vibration monitoring is effective for rotating machinery. A rise in vibration frequency level over time can indicate mechanical stress.

In March 2004, the German-based company SKF [11] installed the world’s first large-scale, online condition monitoring system in 126 turbines throughout Germany and France. The system, called WindCon, uses a range of vibration sensors mounted on the main-shaft bearings, the drive-train gearbox and the generator of each turbine, to continuously monitor vibration signals from rotating components [24]. The gathered sensor data is used in conjunction with data from the standard SCADA turbine control system to detect changes in turbine performance and to output warnings to engineers through the system user interface.

Distress in rotor blades is also detectable by analysing changes in strain, acoustic and vibration signals [32]. Interpreting faults using these techniques is generally assumed difficult due to the contribution of the wind itself in generating random frequency variations across a blade. Current commercially adopted techniques use fibre-optic strain gauges embedded on the blade surface. Smart Fibres Ltd. [12] and LM Blade Monitoring [5] are in active production of such devices.

Ultimately, a complete condition monitoring system should offer a means of collating monitored data, analysing it against a predefined knowledge base including trends of component operation. The system should output warnings, recommendations and other results to an operator.

The most common cause of turbine downtime results from mechanical malfunctions and component failure within the nacelle – typically, the gearbox, bearings and rotor blades. Such faults are expensive, both in terms of manpower and part replacement. Turbines have no means to monitor mechanical components proactively to detect potential failures.

Condition monitoring has been used successfully within traditional power generation and aviation applications [32], but it is not until recently that any widespread attempt has been made to develop similar technology for wind farms.

2.2.8 Sensor Network/Wind Turbine Domain

The results in this thesis are demonstrated for sensor network and wind turbine management. The thesis examples use both free-standing environmental sensors (referred to as 'sensor nodes') and sensors that monitor turbine mechanics.

Sensor nodes are based on portable weather stations developed as part of the Prosen project, and are physically deployed on a wind farm site. For the purpose of this thesis, these sensor nodes are managed separately from individual wind turbines. The reporting and sampling frequencies of the sensors on each node may be configured, and individual sensors may be switched on or off.

It was not feasible to trial the systems and approaches presented in this thesis with operational wind turbines due to issues of access and cost. Therefore, for demonstration purposes it is assumed that the policy system is interfaced with the turbine control system and policies may use common turbine parameters. Common parameters are reported to the policy system and may be configured through policy actions. Direct actions from the policy system for simulation purposes include the modification of certain parameters to manipulate mechanical operation. For example, the policy system may set parameters like rotor blade pitch or rotor yaw angle, or may configure components such as the rotor brakes, yaw brakes, generator or gearbox. More detail on the nature of these parameters is given with a description of the policy language for this domain within Chapter 3.

Chapter 3

Policy-Based System Management

This chapter explains how policies are employed in systems to provide customised event handling based on the preferences of users. In prior work not part of this thesis, a policy system and policy language were developed to support policies for Internet telephony. This work has been used as a basis for the design of a new policy language for sensor network management. In this chapter, an overview of this existing policy work is given, and new work that has adapted the existing policy language for the new application domain of sensor network and wind turbine management is described.

Section 3.1 introduces the concept of a policy and summarises the history of where policies, policy-based languages and systems have been applied. Section 3.2 outlines the generic features of the policy language adopted for this thesis work. Section 3.3 describes how this language was originally specialised for Internet telephony, while section 3.4 reports on a new specialisation of the language for the domain of sensor network management in the context of a wind farm. Section 3.5 evaluates the work presented here.

3.1 Introduction and Background

Traditionally, the features offered by a system have been centrally controlled, deployed and accessed. This approach is inflexible as such features cannot be easily customised to the preference of a user and are owned solely by the service provider or administrators of the system. Policies have emerged as a method of controlling decentralised services in networks, providing users with more control over how the services they use are configured. Using policies, a user may customise a service and define high-level goals or low-level actions a system should take depending on the context in which an event occurs. A policy contains information that can be used to dynamically modify the behaviour of a system, depending on whether defined conditions apply (e.g. time, system state or user context). Traditional features do not provide this level of user control.

The following subsections introduce the concept of a policy, provide a background to their use in computing, and describe a variety of existing policy languages and systems.

3.1.1 Policy Definition

A policy, in general, is described as *"a set of ideas or a plan of what to do in particular situations that has been agreed officially by a group of people, a business organization, a government or a political party"* (Cambridge English Dictionary [88]). Policies are typically formed as documents that outline the purpose of the policy, who or what the policy affects, the date or timescale to which it applies and a set of policy statements that describe the specific intentions, regulations and actions that the policy is creating. Common examples include insurance policies for vehicles or properties, and political or economic policies issued by the Government. Note that a policy is different from a law in that it aims to guide actions toward those most likely to achieve a desired outcome, rather than require or prohibit actions.

Stemming from this general notion, a policy in the context of computing is a high-level statement of how the behaviour of a system should be controlled for a given set of detectable events and conditions. Similar to general policies, these are expressed as structured documents and contain details of the purpose, applicability, scope, timescale and intentions of a policy. Simple high-level policy examples might be: "To ensure personal safety, when the fire alarm sounds all persons must vacate the building using the nearest exit", or "If the printer is out of paper during office hours, alert a technician".

Computing policies are specified in a high-level language, possibly using near-natural language phrases. Policies are defined in some structured or formal description language for storage and analysis. This may be automatic, using a wizard or similar user-friendly graphical interface. Collections of policies are stored and managed within a specialist policy system. A policy system typically provides a facility to create, store and deploy policies, and a mechanism to retrieve, process and execute eligible policies upon receipt of state information and events that occur within the system environment.

3.1.2 History of Policies

Policies in computing were originally applied to handle security and user authentication issues. During the past decade their use has greatly expanded, as policies have been recognised as a practical solution to dynamic modification of a system during its operation based on events it encounters. In addition, policies are aimed at end users and can be defined in non-technical terms, providing domain experts and system operators with the ability to manage a system without the need for technical programming knowledge.

The earliest policies were based on security models for systems and networks. Security policies define high-level rules to regulate user and system access to data and hardware (or software) resources [39]. Typically, security policies define whether requests from a particular entity (e.g. individual or group of users) should be granted or denied. Examples of such policy systems and models include access control matrices, such as the Lampson model, otherwise known as ACL (Access Control List [76]). Later security policy models enhanced the notion of access control, including the Bell-LaPadula Model [36] which concerns data confidentiality and an integrity policy model proposed by Biba [37], which describes how the validity of system data should be maintained as a system changes state. In 1992, the concept of Role-Based Access Control (RBAC) was introduced as a security model by Ferraiolo and Kuhn [63]. Although the RBAC model is not used directly for policy specification, it has gained wide acceptance and use as a general model through which to specify and enforce organisational access control policies. A role-based policy regulates user access to information in a system based on the activities or responsibilities that user is defined to have [39]. A refined version of the RBAC model was standardised by the National Institute of Standards and Technology (NIST) in 2004 (INCITS 359-2004). Some policy-based frameworks (discussed in the following sections) have implemented the RBAC security model, such as Ponder [57] and XACML [67].

In addition to security, networking policies have been a dominant area in policy-based management. As local area and wide area (Internet) network access has evolved since the late 1980s, so too have policies to govern quality of service (QoS), the routing of data, and prioritised user access to networked resources [39]. Early routing policy frameworks formed the basis for network protocols in use currently. For example, the COPS (Common Open Policy Service) protocol [40], published in 2000, relates to the general administration, configuration and enforcement of policies in IP (Internet Protocol) networks.

3.1.3 Existing Policy Systems and Languages

A policy is defined in a particular high-level policy description language which specifies the syntax and semantics of policy components. This language typically specifies aspects such as the structure of a policy, and the format its rules may take. Crucially, a policy language will define generic policy components and those particular to an application domain. A policy system is a framework through which policies (in a particular description language) may be stored, retrieved and executed. A policy system interacts with both the human policy creators and the underlying system being managed.

Different policy-based approaches have been developed for different application areas, resulting in a number of policy systems and description languages in active use. In addition, languages have evolved to utilise developments in mark-up languages such as XML [15]. Early languages were geared entirely toward network policy specification, such as SRL ([42]) and PPL (Path-based Policy Language [96]). Later languages took a less domain-specific view, and began to focus on managing systems in general. PDL (Policy Description Language [80]) was one of the first languages to formulate policies using the Event-Condition-Action (ECA) rule paradigm, which maps a series of system events to sets of actions, based on sets of conditions.

In the subsections that follow, the main established policy-based approaches are summarised, including Accent, KAoS, Rei, Ponder, Ponder2, XACML and PMAC. Of these, the Accent policy-based environment is the subject of the approaches presented in this thesis, and the reasons why it was chosen over other policy-based frameworks is explained in more detail in the following subsection.

Accent **Overview and Motivation for Use**

The Accent project (Advanced Component Control Enhancing Network Technologies [13]) developed a policy-based management system together with a policy language called Appel (Adaptable and Programmable Policy Environment and Language [91]). Both this system and language were used as a basis for the work of this thesis. The Accent policy system comprises a web-based user interface (a policy wizard) to create and manage policies, a policy server and policy store, and a customisable network interface to the system being controlled. The policy system is domain-independent and may be reused across multiple domains. The policy system implements the Appel policy language. Appel is a generic policy language that can be specialised for any domain. Both the system and language are designed to support rule-based policies of ECA (Event-Condition-Action) format.

The primary needs of a policy system to support this thesis work were: a readily available, generic framework to create, store and deploy policies, and an extensible policy language that could be customised for the sensor network/wind turbine domain. A suitable user-interface to support policy creation by non-technical users was also required. The Accent framework was chosen over other policy-based approaches as it meets these requirements most strongly. The Accent system is generic and readily reusable in new domains. In addition, no other framework offers a web-based user-interface like the Accent policy wizard, and the ability to distribute major Accent system components (i.e. the policy server, wizard and policy store) more flexible than other approaches. The Appel language is extensible to new domains and its non-formal method of language specification more attractive and less complex than ontology-based approaches (such as those of KAos and Rei). The Accent framework was designed to be generic and not tailored to or evolved from the domain of security or access control (like the Ponder framework for example). Consequently, the Appel language is less complex in that it offers fewer types of policies (two types versus three in Ponder for example) and is deemed easier for non-technical users to create policies. The ability to explicitly permit and forbid actions as promoted in KAoS, Rei and Ponder, and support for policy groupings and user roles offered by Ponder, are features more suitable for security policies and less relevant for the work presented in this thesis.

A more detailed overview of Accent and Appel is described in section 3.1.4. Other main policy-based approaches are now outlined with reasons why each was less suited to this thesis work.

KAoS

The KAoS framework (separate from the Kaos framework for Requirements Engineering (RE)) is a platform-independent set of services which use ontology[1] concepts to represent policies using OWL (the Web Ontology Language [19]). In KAoS, there are generic ontologies which describe the basics for any environment. This core framework is then extended through additional ontologies which specialise the generic concepts for a particular domain.

The KAoS policy service offers four basic policy types which are defined as core ontologies. These are positive and negative authorisation policies (policies that permit or forbid some action), and positive and negative obligation policies (policies that require some action when a state/trigger occurs, or define what action not to perform). A domain policy is an instance of one of these basic policies, specialised for the domain by extending the appropriate ontology with data specific to the execution environment.

Policies in KAoS are defined purely in OWL, and can therefore be read, edited and analysed using any third-party tool supporting OWL – independent of any KAoS tools. Unlike the Accent framework, KAoS has no readily available, distributed architecture through which to interface policies with the system being managed. The Accent approach is also more flexible in that the policy language is specified separately from the

[1] An ontology defines the main concepts within a domain, the relationships that exist among these concepts, and the properties (or attributes) they have. Ontologies are discussed in detail in Chapter 4.

mechanism through which policies are actually stored. KPAT (KAoS Policy Administration Tool) was a tool previously in use to define/manage KAoS policies, although this is no longer maintained or easily obtainable.

Rei

Rei [70] is a policy environment that supports policy specification, analysis and reasoning. The Rei policy language was designed to be flexible and application-independent. The language allows users to express rules to represent the concepts of *rights* (positive authorisation), *prohibitions* (negative authorisation), *obligations* (positive obligation) and *dispensations* (negative obligation). Rei policies may define rules to associate an entity (e.g. an individual or organisation) with a set of these such concepts. Rights, prohibitions, obligations, dispensations and policy rules for a particular domain are specified in an ontology. The ontology also describes policy actions, including target objects on which the action may be performed, pre-conditions to be satisfied prior to action execution, and the effects that will result following execution. Like KAoS (in the previous subsection), Rei uses a generic ontology which can be extended through further ontologies for a particular application domain. The ontology uses first-order logic and RDF as its specification language. Again, Rei, like KAoS, does not provide any graphical user interface to define policies, which is a drawback compared with Accent. Similarly, Rei focuses on the language support for policies and does not provide the flexible, distributed policy system framework offered by Accent.

The Ponder Framework

Ponder [57] is an object-oriented, declarative programming language which can be used to specify policies. In addition to the core policy language, the complete Ponder release comprises a policy deployment framework and a tool-kit to support the policy life-cycle [57]. Although Ponder is readily applicable to specifying security policies (it implements the RBAC model discussed in section 3.1.2) and more general-purpose management policies, it is intended to be extensible to support other types of policy. Developed by the policy research group at Imperial College, UK, the principal application areas of

Ponder have been security policy creation and management, although the framework is designed to support system management in general.

Ponder supports three core “types” of policy:

Authorisation policies: Used for security and access control purposes, these define the services or resources a software agent or human user may, or may not, access. Specifically, they express whether a subject is allowed (positive authorisation policy) or forbidden (negative authorisation policy) to perform a particular action on a target.

Obligation and Refrain policies: Used to define specific duties of a policy subject. An Obligation policy defines what a subject must do, while a Refrain policy defines what must not be done.

Delegation policies: Specify an action (or authorisation) a subject may delegate to others.

Each type provides a base for a policy template that can be instantiated to express particular high-level aims via customised parameter values. Using this technique, Ponder allows policies to be formulated in a generic manner.

Ponder also provides a means of managing policies across an application in addition to defining them. Policies relating to the same domain or department may be grouped together for more effective organisation and to encourage policy reuse. Policies may also be linked semantically using a role function – such as by common subject or level of organisational hierarchy. Such grouping is advantageous for managing security policies. Ponder is therefore a useful solution to managing a large number of policies across an enterprise-wide network.

Ponder offers a more complex range of policy types (e.g. authorisation, obligation and delegation policies, versus a just user and resolution polices in APPEL) and is deemed more appropriate for managing security policies based on both the range of policies and it’s strong support for policy grouping. The grouping and user role support offered by Ponder policies is less relevant to the needs of this thesis work. Additionally, a weakness of the Ponder language is the inability to define user preference data in a policy (some indication from the policy designer as to how strongly policy execution should be adhered to). Appel supports policy preferences which can aid in the process of conflict negotiation to resolve clashes in policy execution paths by prioritising policy execution to achieve the most desirable outcome.

Ponder2

Recently, Ponder has influenced the development *Ponder2*. Ponder2 is an extensible policy framework intended for use on different levels of scale – from small embedded devices to large systems. The Ponder framework is not this scalable. Ponder2 includes a general-purpose object management system housing a domain service (hierarchy structure for managing objects), a policy interpreter (that handles rules of Event-Condition-Action (ECA) format) and a command interpreter (accepts XML-based commands from communications interfaces to perform invocations on managed objects via the domain service). Ponder2 is used in [73] to achieve policy-based management in the context of body-sensor networking. However, Ponder2 is only just emerging at the time of writing, and is not as yet a stable, publicly available framework. Indeed, at the outset of this thesis work, Ponder2 was still in the design stages and obviously not a contending approach for this work.

XACML

XACML (eXtensible Access Control Markup Language [67]) was developed by the OASIS interoperability consortium (`http://www.oasis-open.org/`) for the purpose of standardising security access control using XML. The standard (first issued in 2003 and revised in 2005) defines an XML-based general-purpose access control policy language and a request/response language. Policies in XACML are modelled in XML and specify Subjects, Targets, Resources, and Actions. The Target is a constraint that must be met by the Subject and Resource in order for the policy Action to be applicable. The request/response language is used to form queries on policy actions which can be used to grant or deny access requests (from an individual, organisation or service) to system resources. While XACML is intended for general use, its focus is on access control rather than general system management, making it less suitable than Accent for the work of this thesis.

PMAC

PMAC (Policy Management for Autonomic Computing [29]) was developed by IBM in 2005. PMAC is a generic middleware platform that allows software applications to receive input from a policy-based management system using embedded software components. The platform is Java-based and provides two different policy languages: ACPL (Autonomic Computing Policy Language) which is XML-based and SPL (Simplified Policy Language) which uses more user-friendly and concise constructs in *if-condition-then-action* format. The PMAC platform model provides a selection of Java-based interfaces that allow the framework to be embedded within Java-based applications. The advantage of this approach is that an existing Java application (or web service) may be extended to utilise policies without the need for a separate system. For the purpose of this thesis however, the complete, stand-alone Accent framework for policy creation and deployment was favoured as there was no pre-existing application through which PMAC could be used. Therefore, although PMAC provides an excellent user-friendly language for policy definition, it is more suitable in cases where an existing system requires adaptation or refactoring to utilise policies.

3.1.4 The Accent Policy System

Accent (Advanced Component Control Enhancing Network Technologies, [13]) is a policy-based management system developed at the University of Stirling, originally for use in Internet call control. Accent incorporates a generic policy language and a system for deploying and enforcing policies defined using the language. Although initially applied for call control, the core architecture of the Accent system is sufficiently generic to support policy handling in any application area. The major architectural components of the system are arranged in a three-layer structure as shown in Figure 3.1 and explained below:

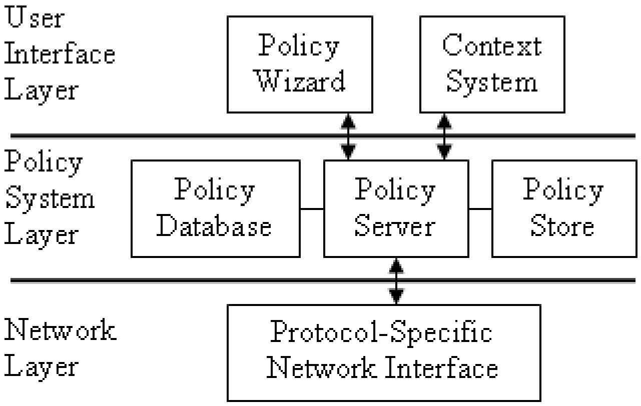


Figure 3.1: Accent Policy System Architecture

User Interface Layer: The top layer includes the Policy Wizard user interface, which offers a user-friendly environment for defining and creating policies, and a context system to hold information, such as user role, location, and similar characteristics.

Policy System Layer: This layer comprises a policy database, policy store and a policy server. Here, policy documents are uploaded, stored, searched and processed on request. Context information and policies are passed to the policy server and held in the policy store.

Network Layer: This is the base layer which connects the policy system to the external environment. This layer can be customised for a specific network protocol.

This architecture maintains a level of abstraction and independence of the user interface from the underlying communications network, allowing for the creation and use of policies across multiple domains. Further technical details surrounding the implementation of the system can be found in [90]. The Accent 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 expressions used to determine whether the triggers justify policy execution)

- an action set (resulting actions taken by the system upon policy execution).

A policy is eligible for execution only on the occurrence of certain triggers it defines. When the policy server is informed of an event, applicable policies are retrieved and executed. The policy server interprets events passed to it via the communications network. Each event is mapped and translated to a relevant trigger term as defined within the policy language. For example, in the context of telephony, an "incoming call" triggers policies relating to the recipient. Policy execution depends on context information such as who the caller is, and perhaps the current location and status of the recipient. Trigger information is therefore derived from the protocol. Using another example from telephony, an INVITE request in the SIP communication protocol would be mapped to the 'connect' trigger in policy language terms. Such mapping details are stored in the policy database. Similarly, outgoing policy actions must be reverse-mapped to the communications protocol in use. The triggers, conditions and actions are all specified within the policy description language used.

3.1.5 The Appel Policy Language

A comprehensive policy description language called Appel (An Adaptable and Programmable Policy Environment and Language, [91]) was designed to facilitate the creation of policies within the Accent policy system. Appel comprises a core language schema which can be extended to support policy management for any given domain. Originally the language was defined for (Internet) telephony and call conflict resolution, but has since been applied to new domains.

Appel is described using an XML-based grammar – its syntax is defined by means of an XML Schema. Each policy is stored within the Accent system as an individual XML document. However, as policies are aimed at non-technical users of a system, the Accent framework incorporates a policy wizard which provides a web-based interface through which to create, edit and manage policies expressed using Appel, thus avoiding user exposure to XML.

Appel describes the make-up of a policy, including the components it may contain and specific triggers, conditions and actions that may be used. Specifically, a policy document may contain zero or more policies. A policy consists of one or more policy rules. Each policy rule may contain an optional trigger (a detectable event), an optional condition (constraint on execution), and a compulsory action (the resulting system behaviour following policy execution). Further to these main components, the language outlines various policy attributes and variable structures, together with a range of operators and rules governing how they may be applied to combine policy rules. In addition, policies can be designated as active or inactive (on or off), or set to be active within a specific time period. Each policy also has a unique ID and applies to a particular user or domain group.

Appel distinguishes between two main types of policy document: regular policies that let users customise system event handling, and resolution policies that allow administrators to customise how conflicts between executable policies are resolved. The outcome of a regular policy is to modify the behaviour of the underlying system. The outcome of a resolution policy may be to select one of the conflicting actions for execution. Since policy conflict detection and resolution are actually out-with the policy system, a specialisation of Appel is convenient to define resolution policies. Resolution policies are similar in syntax to regular policies, but have a modified structure. In particular, the triggers of a resolution policy must be the actions of conflicting regular policies.

A major feature of Appel is the incorporation of preference information. An optional preference (namely “must”, “should”, “prefer”, “prefer not”, “should not”, “must not”) allow a user to define how strongly they feel a regular policy should be considered when selected for execution. Thus, the policy system is given some insight into the desired outcome or goal of a user. The actions of a resolution policy in Appel may be either a generic resolution action based on the properties of the policies in conflict (such as to apply the newer policy, or the policy with the stronger or weaker preference), or any permitted regular policy action. The topic of policy conflict and resolution is covered in depth in Chapter 5.

The remainder of this chapter concentrates on the Appel policy language, describing the core and domain-specific constructs of the language. The previous language specialisation for Internet telephony is outlined, and new work to specialise the core language for the domain of sensor networks and wind turbine management is described.

3.2 Appel Policy Language Syntax

The complete Appel language syntax is described in a published technical report [91]. The language is defined by a collection of XML schema documents, including the core constructs, additional definitions for different types of policy, and domain-specific specialisations. The full collection of XML schemas may be accessed from [21]. Appel policies are split into two types: regular and resolution policies. The following subsections describe the core constructs of Appel used to define these types of policy and explain the differences between them. Example policies in XML are provided to highlight the language syntax.

3.2.1 Core Language Outline

Appel has a number of core features and triggers, conditions and actions which are applicable to any domain specialisation of the language. Policy features are summarised below, with examples in the following sections to make this more concrete.

Document Structure

Every XML document in Appel begins by defining a policy document and the associated XML Schema through which it must be validated. A policy document may contain regular policies, resolution policies or policy variables. Each type has associated attributes. Common attributes include a unique document ID, the policy owner (creator), the individual or group (domain) the policy applies to, and the date last modified. Regular and resolution policies may contain one or more policy rule blocks. Multiple rules may be joined using operators that determine which of the rules are selected and the order in which they are executed when the policy is triggered. These operators are:

guarded: if the first policy rule is valid, it is applied, otherwise the second rule is applied.

unguarded: if one of the two rules is applicable, it is chosen and applied. If both rules are applicable, one rule is selected at random and applied.

sequential: rules are checked in the order they are defined. If the first rule is not valid, the second rule is checked.

parallel: the order of execution is unimportant. If both rules are valid, the policy system chooses the order of execution.

A policy rule may optionally contain a trigger and/or condition, but must contain at least one action.

Triggers

A trigger is an event which causes a policy to be selected for potential execution within the policy system. Multiple triggers are combined using the operators *and* and *or*.

Conditions

Conditions are optional tests which must be satisfied before the policy may be executed. If conditions are not satisfied, the policy is discarded despite being triggered. Multiple conditions are combined using *and*, *or* and *not*. Conditions must contain a parameter (an operand) an operator and a value (a second operand), to form a conditional statement. The parameter may be an argument of a trigger, and the value may be the value established by that trigger argument. Condition operators include "eq" or "ne" to test for equality ($=$) or inequality($\neq$) between values, or "gt", "ge", "lt", "le" meaning $>$, $\geq$, $<$ and $\leq$ respectively. In addition, the operators "in" and "out" may be used to test a set of values with the meaning "among" or "not among". Comparisons may be between numerical or string values depending on the context. Each condition evaluates to either true or false.

Actions

A policy action states how the policy system environment should be modified on execution of the policy. Multiple actions can be combined in complex ways, using the operators *and*, *or*, *else*, *andthen* and *orelse*. Further explanation and examples of the use of these operators is provided in the language technical report [91].

Variables

A variable is a name for a value. Variables are defined in a similar structure to policies. A variable name must be prefixed by ":" when used in a defined policy. For example, ":engineer_email" might represent the email address of an engineer that is defined separately. Variables may be used when defining trigger arguments, condition parameters or values, or action arguments.

Expressions

Variables can be set to an expression by calling the "set variable" action in a policy. A variable binding is removed using the "unset variable" action. An expression may be a literal value, the name of a variable, or a general expression. A number of string-based functions may be used within an expression, including *indexOf(given string,search string)* returning -1 for not found or 0 for found, *join(separator,string,...)* returning the strings joined by the given separator (possibly empty), *length(string)* returning 0 for empty, and *substr(string,start,count)* for 0 as the start position. Expressions are parsed left to right, with operators having equal precedence in relation to one another.

Timers

Interval (count-down) timers may be used within a policy. Each timer has a unique identifier, and only one instance of a timer may be running at any time. A timer becomes active when it is started, and ceases to exist when it counts down to zero. At this point, an expiry event occurs for the timer identifier. Generic policy actions may be used to restart or stop active timers.

State History

The policy system logs and timestamps all triggers it receives and all actions it issues. A trigger history function (*trigger_count(trigger,period)*) and an action history function (*action_ count(action,period)*) may be used when forming a condition. The trigger history function is a condition formed as a query that counts the number of times a particular trigger was received within a specified time period. Similarly, the action history function is a condition formed using a count of the number of occurrences of a particular action issued during a specified time period. The *trigger* and *action* arguments may be any generic or domain-specific trigger or action respectively. The *period* argument is a non-negative integer n (during the last n minutes) or a time in HH:MM:SS format (since this time, spanning a maximum of 24 hours).

3.2.2 Defining Generic Policies

This section summarises the syntax and structure of policies in general. The constructs described here form a generic framework which can be extended to specify policies for a particular domain. Standard (or regular) policies are described first, followed by resolution policies.

Generic Standard Policies

A standard policy is defined to handle detectable events within the system environment. A standard policy may contain one or more policy rules. Each policy rule may have zero or more triggers, zero or more conditions, and at least one action.

Generic triggers, conditions and actions for use in standard policies are shown in Figure 3.2. These constructs are supported by functions internal to the policy system, and are therefore usable in any domain-specific standard policy.

Trigger	timer_expiry(identifier)	The timer with the given *identifier* reaches zero
Condition	date	A date of the format YYYY-MM-DD
	day	A day of the week represented by the values 1 to 7. (Monday = 1, Sunday = 7)
	time	A time of the format HH:MM:SS
Action	log_event(message)	Logs the date, time and *message* specified to a log file
	restart_timer(identifier)	Immediately restarts the timer identified by its *identifier* for the original period specified
	send_message(URL, message)	Sends the given *message* to the specified *URL* which also includes the protocol (i.e. mailto, sms, etc.)
	set_variable(identifier, expression)	Sets a variable with the specified *identifier* to the specified *expression*
	start_timer(identifier, period)	Immediately starts a timer with the specified *identifier* for the specified *period* (of the format HH:MM:SS)
	stop_timer(identifier)	Immediately stops the timer with the specified *identifier*
	unset_variable(identifier)	Removes the variable with the specified *identifier*

Figure 3.2: Generic Standard Policy Triggers, Conditions and Actions

An example standard policy in XML with a single policy rule is shown below, using only generic triggers, conditions, and actions:

```
 0 <policy_document
 1   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 2   xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel_policy.xsd">
 3   <policy
 4     owner="gca@cs.stir.ac.uk"
 5     applies_to="gca@cs.stir.ac.uk"
 6     id="wakeup_alert"
 7     changed="2008-04-21T10:20:59"
 8     enabled="true"
 9     profile="personal"
10     valid_from="2008-10-01T12:00:00"
11     ailed_to="2008-10-31T11:59:59" >
12     <preference>should</preference>
13     <policy_rule>
14       <trigger arg1="wakeup">timer_expiry(arg1)</trigger>
15       <condition>
16         <parameter>day</parameter>
17         <operator>in</operator>
18         <value>1..5</value>
19       </condition>
20       <actions>
21         <and/>
22         <action arg1="Timer expired">log_event(arg1)</action>
23         <action arg1=":gavin" arg2="Time to wake up">send_message(arg1,arg2)</action>
24       </actions>
25     </policy_rule>
26   </policy>
27 </policy_document>
```

Based on this example policy, the constructs of standard policies can be explained as follows.

policy_document: This is the highest-level construct in APPEL. It specifies the XML Schema against which the document can be validated (lines 1-2).

policy: A standard policy has a number of associated attributes:

owner: the address of the person or entity the policy belongs to

applies_to: the address of the person or entity the policy applies to

id: a unique identifier for the policy

changed: the date the policy was last modified

enabled: whether or not the policy is selectable

profile: (optional) an identifier that groups the policy with others

valid_from: (optional) date from which the policy becomes valid

valid_to: (optional) date on which the policy ceases to be valid.

preference: A policy may have an optional preference which states how strongly the policy writer feels it should be considered for execution. Preferences are used in the resolution of policy conflicts and are ordered from strongly positive to strongly negative. Valid preferences are *must*, *should*, *prefer*, *prefer not*, *should not* and *must not*.

policy_rule: A policy may contain one or more policy rules. Each rule has an optional set of triggers and/or conditions, and at least one action.

trigger: An event which triggers the policy. The example has a single trigger (line 14) which is the expiry of a timer with the identifier “wakeup”.

condition: Conditions must be satisfied before the policy may be executed. The example uses one condition (lines 15-19), which states that the day must be in the range 1 (Monday) to 5 (Friday).

action: There are two actions (lines 20-24). As they are combined with an “and” operator, both actions should be carried out. The first action logs an event and the second action sends a message to an address represented by the policy variable “:gavin”. At the point of policy execution, the current variable value is substituted into the action.

Generic Resolution Policies

A resolution policy is similar in structure to a standard policy, except that there are some additional restrictions in the way the constructs may be used. Language-wise, a resolution policy differs from a standard policy in that it may not specify a profile attribute (which is irrelevant) and cannot have a preference associated with it (recursive resolution is not permitted). The triggers of a resolution policy are the generic and domain-specific actions used in standard policies.

Generic triggers, conditions and actions for use in resolution policies are shown in Figure 3.3. These constructs are supported by functions internal to the Accent policy system, and are therefore usable in any domain-specific resolution policy.

Trigger		Any generic standard policy action (refer to Figure 3.2 for details)
Condition	preference[0-9]	Converted to a numerical value, bound by resolution triggers
	variable[0-9]	Converted to a numerical value, bound by resolution triggers
Action	apply_default	Internal use only, selects an action at random
	apply_inferior	Chooses action that applies to the inferior domain
	apply_negative	Chooses action with the more negative preference
	apply_newer	Chooses action of policy defined most recently
	apply_older	Chooses action of policy defined first
	apply_one	Chooses one action arbitrarily, e.g. the first one
	apply_positive	Chooses action with the more positive preference
	apply_stronger	Chooses action with the stronger preference, ignoring positive or negative implications
	apply_superior	Chooses action that applies to the superior domain
	apply_weaker	Chooses action with the weaker preference, ignoring positive or negative implications

Figure 3.3: Generic Resolution Policy Triggers, Conditions and Actions

An example generic resolution policy in XML with a single policy rule is shown below:

```
 0 <policy_document
 1   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 2   xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel_resolution.xsd">
 3   <resolution
 4     owner="admin@cs.stir.ac.uk"
 5     applies_to="@cs.stir.ac.uk"
 6     id="Stop timer conflict"
 7     changed="2008-04-21T10:20:59"
 8     enabled="true" >
 9     <policy_rule>
10       <triggers>
11         <and/>
12         <trigger arg1="variable0">stop_timer(arg1)</trigger>
13         <trigger arg1="variable1">stop_timer(arg1)</trigger>
14       </triggers>
15       <condition>
16         <parameter>variable0</parameter>
17         <operator>eq</operator>
18         <value>variable1</value>
19       </condition>
20       <action>apply_newer</action>
21     </policy_rule>
22   </resolution >
23 </policy_document>
```

This example aims to convey the syntax and structure of a resolution policy. The nature of the constructs in a resolution policy can be explained as follows:

policy_document: This is the highest level construct in Appel. It specifies the XML Schema against which the document can be validated (lines 1-2).

resolution: A resolution policy has a number of associated attributes as for standard policies except the “profile” is not used.

trigger: Resolution policy triggers are the actions of regular policy actions. Resolution policies have restricted trigger arguments: *variable0* to *variable9* are bound explicitly to the arguments of resolution triggers. In the example, the arguments for each trigger are shown bound to these variables (lines 12-13). Note also that *preference0* to *preference9* are implicitly bound to the preferences of the policies which triggered the resolution – that is, the preference associated with each resolution trigger.

condition: Resolution policy condition values are bound by resolution triggers. Conditions can be expressed on the values of variable[0-9] and preference[0-9]. Preferences are converted to numeric values for comparison (e.g. “must” = +3, "must_not" = −3). The example resolution policy has one condition (lines 15-19) that tests if the parameters for each trigger are the same. Within resolution policy conditions, operators "eq", "ne", "lt", "le", "gt" and "ge" perform a numerical comparison between two parameters or a parameter and literal value. The "in" and "out" operators are typically used to compare preferences and should be read as "in keeping with" and "out of keeping with" – i.e. similar and opposite respectively. (This is different from "among" or "not among" as for standard policy conditions). For example, the condition *:preference0 out :preference1* should be read as "if :preference0 is the opposite of :preference1". So, if *preference0* is "should" and *preference1* is "must_not" the condition would be true. If *preference1* was "must", the condition would be false (preferences are similar).

action: Actions of a resolution policy are either generic resolution actions or any other generic or domain-specific actions. Generic resolution actions choose one of the conflicting policy actions based on their properties (e.g. date created) or preferences (associated with the policy). In the example resolution policy, the action (line 20) is to apply the action of the policy which was created most recently (the newer of the two).

3.2.3 Defining Domain-Specific Policies

The generic language constructs described in the previous subsection may be extended to specialise the Appel policy language for a particular domain. In this section the general extensions required for standard and resolution policies are described. Specific domain specialisations are given in the next section.

Standard Policy Extensions

A domain specialisation of Appel for standard policies must specify the triggers, conditions and actions relevant to the underlying system, together with any associated parameters. Both triggers and actions must be closely related to the system and its monitored environment. Incoming event notifications received by the policy system must be mapped to policy triggers, and outgoing policy actions must be similarly translated into specific actions that modify the system.

Resolution Policy Extensions

The triggers for resolution policies are the full range of permitted generic and domain-specific policy actions defined. The only domain customisation for resolution policies is in the types of actions applicable to resolve conflicts. The core resolution action list may be extended to include actions specific to the domain based on the established environment. This is illustrated for the application of Internet telephony in the next section.

3.3 Application 1: Policies for Internet Telephony

This section describes how the Appel policy language is specialised for the domain of Internet telephony. The implementation of the language for this domain is not new work, but is included here for reference as the domain is later extended for the topic of goal direction (Chapter 6). (The policy wizard supporting Internet telephony has also been completely redesigned as described later in Chapter 4). The telephony policy language was developed by the original authors of the Accent policy system. The summary of the language and the examples used in this section are adapted from the Appel language technical report [91]. Section 3.3.1 provides an overview of the language requirements for the domain. Section 3.3.2 describes the language for standard telephony policies, and section 3.3.3 describes the extensions for resolution policies.

3.3.1 Language Requirements

Prior to the Accent project, policies had never been applied in the domain of telephony. The requirements for the language were derived largely from the underlying communications protocols the policy system was required to support. The protocols trialled with the Accent system were SIP, Mitel 7000 ICS (a softswitch) and H.323. Each of these protocols has a particular range of triggers, conditions and actions in the policy language.

In a telephony environment, policies are aimed at ordinary non-technical users of a telephone network for the purpose of customising how their calls are handled. Each user has different preferences over when and how they may be reached depending on conditions such as their current status, the time of day, the identity of the caller or even the topic of call. Users may wish to take different actions depending on these conditions, such as forward calls to a more convenient number, reject calls, or play customised voice messages. The following subsections outline the language for standard and resolution policies respectively.

3.3.2 Standard Policy Language Specification

The Appel specification for Internet telephony is described in full in the language technical report [91]. The language specialisation for standard telephony policies defines new trigger, condition and action components in addition to the core set. It also defines a set of common environment variables usable within triggers, conditions and actions.

There are many environment variables defined for Internet telephony (e.g. bandwidth, caller, callee, call_type, device, medium, priority, etc.) which may be used as condition parameters (refer to the description of telephony conditions that follow).

The language for Internet telephony defines triggers, conditions and actions specific to this domain. These domain-specific constructs and their relationships to generic language constructs are summarised in Figure 3.4. Further details about each telephony construct can be found in the language technical report [91].

Example standard policies for Internet telephony that demonstrate the language now follow. Each example gives a high-level description of the policy together with the XML definition. For each telephony policy, an XML wrapper is required to specify the XML schema to validate the policy against. This takes the form:

```
<?xml version="1.0" encoding="UTF-8"?>
<policy_document
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
  "http://www.cs.stir.ac.uk/schemas/appel_regular_call.xsd">
  ... policy body ...
</policy_document>
```

Trigger	**Parameters**	**Actions Permitted****
*internal**		note_availability, note_presence
absent*		note_presence
available*	topic (empty if *availability* variable is ‘true’)	connect_to, note_availability
bandwidth_request‡	bandwidth‡, callee, caller, medium, network_type	confirm_bandwidth‡, reject_bandwidth‡
connect, connect_incoming, connect_outgoing, no_answer, no_answer_incoming, no_answer_outgoing	active_content‡, bandwidth‡, call_content, call_type, callee, caller, capability, capability_set‡, cost, destination_address‡, device, location, medium, network_type, priority, quality, role, signalling_address‡, source_address‡, topic, traffic_load‡	add_caller‡, add_medium, add_party, fork_to, forward_to, note_availability, note_presence, play_clip, reject_call, remove_medium, remove_party
disconnect, disconnect_incoming, disconnect_outgoing	call-regularee, caller, medium, network_type	note_availability, note_presence, play_clip
event	call-regularer, network_type, topic	note_availability, note_presence
present*	location (empty if *presence* variable is ‘true’)	connect_to, note_presence
register, register_incoming, register_outgoing	call-regularer, network_type	note_presence, reject_call
timer_expiry*		note_presence
unavailable*		note_availability

* internal trigger that may be combined with any other

** all internal actions are also permitted:
log_event, restart_timer, send_message, set_variable,
start_timer, stop_timer, unset_variable

‡ available with only certain communications systems

Figure 3.4: Internet Telephony Policy Language

Forward Incoming Call To A Mobile

This forwards an incoming call to a mobile when out of normal weekly office hours (before 8:30am and after 5:30pm, Monday (1) to Friday (5)). The policy is valid for one month (during November 2008). The policy preference is “should”. The policy is owned by and applies to “gca@cs.stir.ac.uk”.

```
<policy owner="gca@cs.stir.ac.uk" applies_to="gca@cs.stir.ac.uk"
id="Forward Incoming Call To A Mobile" enabled="true"
valid_from="2008-11-01T00:00:00" valid_to="2008-11-30T23:59:00"
changed="2008-10-27T09:23:05">
  <preference>should</preference>
  <policy_rule>
    <trigger>connect_incoming</trigger>
    <conditions>
      <and/>
      <condition>
        <parameter>hour</parameter>
        <operator>out</operator>
        <value>08:30:00..17:30:00</value>
      </condition>
      <condition>
        <parameter>day</parameter>
        <operator>in</operator>
        <value>1..5</value>
      </condition>
    </conditions>
    <action arg1="00447977123456">forward_to(arg1)</action>
  </policy_rule>
</policy>
```

Add Whiteboard to Department Calls

This policy adds a whiteboard to calls within the same domain as the policy owner (i.e. *cs.stir.ac.uk*) or with the defined location of “department”. It also logs this event.

```
<policy owner="gca@cs.stir.ac.uk" applies_to="gca@cs.stir.ac.uk"
id="Add Whiteboard To Department Calls" enabled="true"
changed="2008-10-27T09:40:00">
  <policy_rule>
    <trigger>connect</trigger>
    <conditions>
      <or/>
      <condition>
        <parameter>location</parameter>
        <operator>eq</operator>
        <value>department</value>
      </condition>
      <condition>
        <parameter>caller</parameter>
        <operator>in</operator>
        <value>@cs.stir.ac.uk</value>
      </condition>
    </conditions>
    <actions>
      <and/>
      <action arg1="whiteboard">add_medium(arg1)</action>
      <action arg1="Department call received">log_event(arg1)</action>
    </actions>
  </policy_rule>
</policy>
```

3.3.3 Resolution Policy Language Specification

This section describes the domain-specific triggers, conditions and actions used in resolution policies for Internet telephony. For resolution policies, any action may be associated with a trigger. The actual parameters of resolution actions must be literal values, or the values of *variable0* to *variable9* (if a trigger has bound them).

Since resolution policies are triggered by policy actions, the resolution policy triggers for Internet telephony are identical to the set of standard policy actions for this domain. Resolution policy conditions are the same as for standard policies for this domain. Domain-specific resolution actions extend the core set outlined in Figure 3.3 and include the set of domain-specific standard actions for the domain. For Internet telephony, there are two specific resolution actions: *apply_caller* and *apply_callee*. These actions choose the call action associated with the caller or callee respectively.

Example resolution policies for Internet telephony that demonstrate the language now follow. Each example gives a high-level description of the policy together with the XML definition. For each policy, an XML wrapper is required to specify the XML schema to validate the policy against. This takes the form:

```
<?xml version="1.0" encoding="UTF-8"?>
<policy_document
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
  "http://www.cs.stir.ac.uk/schemas/appel_resolution_call.xsd">
  ... policy body ...
</policy_document>
```

Add Medium Conflict

This detects the conflict where one party wishes to add media to the call (e.g. video) and another party prefers not to add the same media to the call. In this scenario, the conflicting action is *add_medium* with identical parameter arguments. This conflict is resolved by applying the action with the stronger associated preference. For example, say *preference0* is *should* and *preference1* is *must_not*. The stronger preference in this example is *must_not* so the medium will not be added. Note the "out" operator (line 17) should be read as "out of keeping with" or "similar". An explanation of preferences and the "in"/"out" operator in resolution policies is given in the resolution condition description in section 3.2.2.

```
 0 <resolution id="Add Medium Conflict" owner="admin@cs.stir.ac.uk"
 1   applies_to="@cs.stir.ac.uk" enabled="true" changed="2008-10-01T12:33:00">
 2   <policy_rule>
 3     <triggers>
 4       <and/>
 5       <trigger arg1="variable0">add_medium(arg1)</trigger>
 6       <trigger arg1="variable1">add_medium(arg1)</trigger>
 7     </triggers>
 8     <conditions>
 9       <and/>
10       <condition>
11         <parameter>variable0</parameter>
12         <operator>eq</operator>
13         <parameter>variable1</parameter>
14       </condition>
15       <condition>
16         <parameter>preference0</parameter>
17         <operator>out</operator>
18         <value>preference1</value>
19       </condition>
20     </conditions>
21     <action>apply_stronger</action>
22   </policy_rule>
23 </resolution>
```

Call Forward/Reject Conflict

This detects a domain-specific conflict when one party wishes to forward a call and another wishes to reject it. In this scenario, only the preferences of the call parties are relevant: the conflict is in the action alone and it does not matter whether the parameter arguments are similar or not. If the preferences associated with each action are inkeeping/similar (e.g. *preference0* is *must* and *preference1* is *should*), the resolution is to apply the action associated with the caller.

```
<resolution id=″Call Forward-Reject Conflict″ owner=″admin@cs.stir.ac.uk″
  applies_to=″@cs.stir.ac.uk″ enabled=″true″ changed=″2008-10-01T12:45:00″>
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1=″variable0″>forward_to(arg1)</trigger>
      <trigger arg1=″variable1″>reject_call(arg1)</trigger>
    </triggers>
    <condition>
      <parameter>preference0</parameter>
      <operator>in</operator>
      <value>preference1</value>
    </condition>
    <action>apply_caller</action>
  </policy_rule>
</resolution>
```

3.4 Application 2: Policies for Sensor Networks

This section describes new work of the Appel policy language specialisation for the domain of sensor network management, which is intended to support policies for wind farm condition monitoring as well. Section 3.4.1 outlines the initial requirements for the language and summarises the domain it is intended to support. Section 3.4.2 describes the language for regular sensor network policies, while section 3.4.3 describes resolution policies. Example policies are also provided together with high-level explanations of their function.

3.4.1 Language Requirements

Initial investigation into the features and capabilities required for a sensor network policy language formed a variety of policy examples based on general sensor network knowledge and the type of data the network was likely to yield. Language features are based on the measurements and network configurations implemented for the sensor network developed on Prosen (see section 2.2.2 for details of the project). This includes stand-alone sensor nodes for environmental condition monitoring (e.g. ground level wind speed, temperature, humidity, or rainfall) and wind turbine measurements and status information (e.g. current wind speed/direction and blade pitch angle, yaw angle, generator speed, gearbox oil temperature).

Appel support for timers were found to be of value for user-interface purposes, alerts and response monitoring. Monitoring state history was also useful in allowing policies to query previous values and recorded measurements in order to take action based on the state of the monitored environment over time. Sensor readings are discrete in that a single value is often not enough to detect and/or confirm an anomaly. Often, appropriate decisions can only be made using multiple measurements over a period of time. In addition, measurements across different sensors are useful in detecting/confirming anomalies in the system. State history functions are a method for dealing with this.

The sensor network domain is very different to that of Internet telephony. In the telephony environment, events and actions are more concrete, bound by the particular communications protocol in use. For sensor networks, configuration details are more open. To ensure a flexible approach to sensor management, the particular sensor nodes, types, and their capabilities may vary. Consequently, the language is not tied to specific events and actions. This flexibility allows the underlying network structure to change with limited impact on the language design. Changes to the specific event types and message content are made by simply changing particular strings defined in policies. Provided the network message format (i.e. the sequence of parameters) stays constant, the interface can easily cope with changes in the strings representing message types, entity names, entity instances and the values reported. Similarly, the actions of policies can be modified in the same way.

The language was required to support potential interaction with the physical system and network environment such as sensor nodes, turbine controllers, and other devices, as well as software agents and operator consoles. Consequently, the language needed to be flexible enough to easily communicate with these entities without becoming too complex for ordinary users to construct policies.

On analysis of potential triggers and actions, the range of entities, message types and their parameters was wide. Ultimately, the content of a particular trigger or action varies depending on the entity sending or receiving the message. For example, sensor nodes may supply a lot of information in a single message, including the node identifier, time stamps of measurements collected, and perhaps multiple measurement values. Triggers from other entities may be less detailed, such as a command from an operator console. While it is possible to construct an alternative trigger and action for each entity, this constrains the policy language, rendering its reuse with new devices or interfaces impossible. The use of different trigger, action and parameter combinations also makes the language more complex when constructing policies.

With system extensibility in mind, a simpler, more generic design was adopted. Instead of a wide range of complex triggers and actions, the language uses a single trigger and action which may be parameterised accordingly for each entity. This approach defines a single common message format for use across all interfacing entities. The components of the language are described in the following sections.

3.4.2 Standard Policy Language Specification

The Appel specification for sensor networks is described in full in the language technical report [91]. The approach to the language design is also discussed in a published paper [99]. The language specialisation for standard sensor network policies defines new trigger, condition and action components in addition to the core set. It also defines a set of common environment variables usable within triggers, conditions and actions.

Environment variables can be used within actions, for example to output variable values to a log file or within an operator message. There are five environment variables:

message_type: Identifies the type of trigger or action. This value will vary depending on the particular *entity_name*. Example trigger message types might be "generator_temperature", "rotor_speed" or "turbine_anemometer". Example action types might be "set_parameter" or "set_rule" in the case of turbine configuration and sensor nodes.

entity_name: Identifies an entity external to the policy system that has associated policies. Examples include physical entities like "sensor_node", "turbine" or "operator_console" and software entities like "policy_agent".

entity_instance: Identifies a particular instance of an *entity_name*. Not all entities have instances, but common examples might include the number or ID of a sensor node or turbine like “C1” or “alarm_console”.

message_period: A time period to which the trigger or action refers. The value can be either a non-negative integer n (during the last n minutes for a trigger, in n minutes for an action), or a time in HH:MM:SS format (since this time for a trigger, at this time for an action). The explicit time spans a maximum of 24 hours from the point of its processing. For a trigger, if the time is currently 12:00:00, the time 11:00:00 means “since 11AM today”, and 14:00:00 means “since 2PM yesterday”. For an action at the same point in time, 11:00:00 means “at 11AM tomorrow”, and 14:00:00 means “at 2PM today”.

parameter_values: An open-ended string that may consist of a single value (e.g. “56.5” or “device active”) or a comma-separated list of values (e.g. “10,34,56.5”).

These parameters are used within the the triggers and actions defined for this domain. The language for sensor networks has one external trigger:

device_in(message_type,entity_name,entity_instance,message_period,parameter_values)

Each parameter argument corresponds to the parameters just outlined. The *message_type* is mandatory, but the rest are optional for a trigger message.
Similarly, the language defines one external action:

device_out(message_type,entity_name,entity_instance,message_period,parameter_values)

Like the *device_in* trigger, the first parameter *message_type* is mandatory, and the rest are optional. The only difference between the semantics of trigger/action parameters is that for *device_in* the *message_period* refers to the time when the event or measurement occurred, whereas in *device_out* this time period refers when the action should occur.

The parameters used within *device_in* and *device_out* are uninterpreted strings. If a parameter is omitted, the default value is an empty string. The content of each trigger and action is deliberately unconstrained so as not to restrict the types of devices or message formats in the system. This open message format also simplifies the language as it reduces all triggers and actions to the same format for policy designers. The only requirement is that the strings used by the underlying network be appropriately matched (or mapped via the policy server) with the strings defined within individual policies.

Policy conditions may be constructed based on established trigger parameters (which are also environment variables). Condition parameters for the sensor network language therefore include *message_type*, *entity_name*, *entity_instance*, *message_period* and *parameter_values*. The parameters are restricted to use different subsets of operators when forming conditions. The *message_type*, *entity_name*, *entity_instance*, may be used with the operators =, ≠, *in* and *out*. The operators *in* and *out* check for presence or absence of a parameter within a list of values. The remaining parameters *message_period* and *parameter_values*, may be used with the full range of operators (=, ≠, >, ≥, <, ≤, *in*, *out*). The *parameter_values* parameter may be considered as items in an array list. For comma-separated lists of values, the first value (or only value in the case of no list) is *parameter_values[0]*. Subsequent list values are referred to by the relevant index (e.g. *parameter_values[2]* refers to the third item in the list). In the case of generic parameters such as day, date and time, the *in* and *out* operators also allow for range specification, such as “in 12:00:00..14:00:00” to mean a time between 12-2pm.

A summary of the relationship between these components for sensor network policies is given in Figure 3.5.

For the purposes of this thesis work, the language is used to configure and monitor sensor nodes and interface with a turbine controller. The policy system communicates with free standing environmental sensor (referred to as ‘sensor nodes’) and turbine controllers.

This sensor network consists of multiple sensor nodes, but no limits on the precise number are defined. Each sensor node houses a number of individual sensors that

Trigger	**Parameters**	**Actions Permitted**
day* date* time*		device_out, log_event*, restart_timer*, send_message*, set_variable*, start_timer*, stop_timer*, unset_variable*
device_in	entity_name, entity_instance message_period message_type parameter_values	device_out, log_event*, restart_timer*, send_message*, set_variable*, start_timer*, stop_timer*, unset_variable*
timer_expiry*	timer_instance	device_out, log_event*, restart_timer*, send_message*, set_variable*, start_timer*, stop_timer*, unset_variable*

* internal (generic) policy trigger or action

Figure 3.5: Sensor Network Policy Language

measure environment conditions such as temperature, wind speed, rainfall, humidity and soil moisture. Each node contains the same number/type of sensors and may be configured similarly. Each node communicates wirelessly and is powered by a battery. The battery level and sensor measurements are reported from each node. Sensors may be configured as follows:

- The reporting frequency of each sensor (the frequency with which measurements are sent from the sensor node to the policy system) may be increased or reduced. The value is a whole number representing the time interval between each report on a sensor measurement (typically between 1 and 60 minutes). The reporting frequency for a sensor can either be the same or lower than its sampling frequency.

- The sampling frequency of each sensor (the frequency with which measurements are taken) may be increased or reduced. The value is a whole number representing the time interval between each measurement (typically between 1 and 15 minutes) with the exception of the rainfall sensor which samples continually and so may be set only to 1 or 0.

- Each sensor may be switched on or off individually. Setting a sensor status to “1” indicates the sensor should be operational. Setting the status to “0” indicates the node should switch off the sensor.

Each sensor is configured by altering or setting "rules" on each sensor node. This rule-based approach was developed within the PROSEN project and is not part of the approach described in this thesis. A rule defines the name of a configurable parameter, a condition or value to set this to, and (if a condition is used) an optional action the node should take if the condition is satisfied. Rules are set up using the command "set_rule(parameter,value,action)". Parameter values are modified using the command "set_parameter(parameter,value)".

As explained in section 2.2.3, a wind turbine is continually monitored using a computerised controller within the nacelle. It was not feasible to trial the systems and approaches presented in this thesis with operational wind turbines. For demonstration (and simulation) purposes it is therefore assumed that the policy system is interfaced with the turbine controller and that policies may use common turbine variables. According to the Danish Wind Industry Association [14], there are between 100 and 500 parameter values maintained by a turbine controller depending on the particular model of turbine. These parameters may be monitored either by the controller itself or an operator. It is assumed that the control system reports the values of these variables at regular defined time intervals. These reports are interpreted as policy triggers. While discrete mechanical adjustments are made in real time under the command of the turbine controller, the policy scenarios described in this thesis are higher-level commands based on monitored events. The purpose of simulating the turbine controller interaction in this way is to show that the presented approaches can feasibly provide effective high-level management of sensor-based systems.

The selection of the most common parameter values used for the examples in this thesis are:

Turbine Rotor and Rotor Blades: The rotor speed, thickness of the rotor brake lining, the status of the rotor and yaw brake (on or off), the current yaw and rotor blade pitch, and the size and frequency of measured vibration in the rotor blades.

Turbine Gearbox and Generator: The oil temperature in the gearbox and the temperature of the gearbox bearings, the temperature, voltage and cooling fan speed of the generator. The reporting frequency of these parameters may also be configured using policies.

Nacelle Monitoring and Wind Detection: The size and frequency of vibration in the turbine nacelle, the temperature within the nacelle, the number of power cable twists in the turbine tower, and the power drain from fans/heaters within the nacelle. The current wind speed and direction are also reported.

In terms of policy-related actions, some assumptions are made about each turbine and the range of control that may be administered. It is assumed that the simulated wind farm has multiple turbines and that all turbines are mechanically identical (typical vertical, tri-blade design with pitch-controlled blades and yaw-controlled rotor). Individual turbines are independently controllable and identified using a unique address (e.g. T10). The physical layout and positioning of turbines within the wind farm, and the topography of the surrounding landscape, are deemed irrelevant for this simulation. Configuration of rotor blade pitch and yaw angle is explained in Appendix A.

Example regular policies for sensor networks that demonstrate the language now follow. Each example gives a high-level description of the policy together with the XML definition. The examples span a number of scenarios and include interaction with sensor network nodes, wind turbine controllers and an operator console.

For each sensor network policy, an XML wrapper is required to specify the XML schema to validate the policy against. This takes the form:

```
<?xml version=″1.0″ encoding=″UTF-8″?>
<policy_document
xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
xsi:noNamespaceSchemaLocation=
  ″http://www.cs.stir.ac.uk/schemas/appel_regular_sensor.xsd″>
  ... policy body ...
</policy_document>
```

Low Sensor Battery Alert

This policy handles the event of a sensor node battery failing. In this example, the trigger is a battery level report from a sensor node. The level is below 9 volts (the standard level). To conserve battery power, the actions are to alter the sampling and reporting rates of the power-hungry anemometer at this node, and to switch off the rain gauge which is also power hungry. The policy applies to all sensor nodes in the wind farm site.

```
<policy
    owner=″sensor-operator@site0113.windorg.com″
    applies_to=″@site0113.windorg.com″
    id=″Low Sensor Battery Alert″
    enabled=″true″
    changed=″2008-04-21T10:20:59″>
  <policy_rule>
    <trigger arg1=″battery_voltage″ arg2=″sensor″>
      device_in(arg1,arg2)
    </trigger>
    <condition>
      <parameter>parameter_values</parameter>
      <operator>lt</operator>
      <value>9</value>
    </condition>
    <actions>
      <and/>
      <action arg1=″set_rule″ arg2=″sensor″ arg3=″:entity_instance″
      arg5=″[anemometer_samp_freq,10.0]″>
        device_out(arg1,arg2,arg3,,arg5)
      </action>
      <actions>
        <and/>
        <action arg1=″set_rule″ arg2=″sensor″ arg3=″:entity_instance″
        arg5=″[anemometer_rep_freq,10.0]″>
          device_out(arg1,arg2,arg3,,arg5)
        </action>
        <action arg1=″set_rule″ arg2=″sensor″ arg3=″:entity_instance″
        arg5=″[rain_gauge,off]″>
          device_out(arg1,arg2,arg3,,arg5)
        </action>
      </actions>
    </actions>
  </policy_rule>
</policy>
```

Turbine Shutdown in High Wind

Suppose a turbine anemometer reports the wind speed is in a high range (here, over 25 m/s). This policy takes action to shut the turbine down to prevent damage to mechanical components. Specific actions include pitching the rotor blades to 0 degrees and applying the rotor brakes. It also outputs a message to an operator to report that maintenance is not possible based on predefined levels of wind-maintenance relationships.

```
<policy
    owner=″sensor-operator@site0113.windorg.com″
    applies_to=″@site0113.windorg.com″
    id=″Turbine Shutdown In High Wind″
    enabled=″true″
    changed=″2008-04-21T13:25:07″>
  <policy_rule>
    <trigger arg1=″anemometer″ arg2=″turbine″>
      device_in(arg1,arg2)
    </trigger>
    <condition>
      <parameter>parameter_values</parameter>
      <operator>gt</operator>
      <value>25</value>
    </condition>
    <actions>
      <and/>
      <action arg1=″set_parameter″ arg2=″turbine″ arg3=″:entity_instance″
          arg5=″[blade_pitch,0]″>
        device_out(arg1,arg2,arg3,,arg5)
      </action>
      <actions>
        <and/>
        <action arg1=″set_parameter″ arg2=″turbine″ arg3=″:entity_instance″
        arg5=″[rotor_brake,on]″>
          device_out(arg1,arg2,arg3,,arg5)
        </action>
        <action arg1=″message″ arg2=″console″
        arg5=″[Warning - maintenance prohibited due to high winds]″>
          device_out(arg1,arg2,,,arg5)
        </action>
      </actions>
    </actions>
  </policy_rule>
</policy>
```

Gearbox Oil Temperature Warning

This policy is triggered by a report of high gearbox oil temperature (over 60 degrees) in turbine B12. The second policy condition (lines 17–23) checks the state history for other oil temperature reports in the past 24 hours (assuming this type of report is only received when the temperature is high). If the number of previous reports in the last 24 hours is at least 3, then anomalies in measurement are ruled out (a high temperature is consistently detected) and the turbine is put into low-power mode (the blades are pitched out of the wind to minimise power output). The state history function *trigger_count* is explained in the State History description in section 3.2.

```
 0 <policy
 1     owner=″sensor-operator@site0113.windorg.com″
 2     applies_to=″@site0113.windorg.com″
 3     id=″Gearbox Oil Temperature Warning″
 4     enabled=″true″
 5     changed=″2008-04-21T10:34:12″>
 6   <policy_rule>
 7     <trigger arg1=″gearbox_oil_temperature″ arg2=″turbine″ arg3=″B12″>
 8       device_in(arg1,arg2,arg3)
 9     </trigger>
10     <conditions>
11       <and/>
12       <condition>
13         <parameter>parameter_values</parameter>
14         <operator>gt</operator>
15         <value>60</value>
16       </condition>
17       <condition>
18         <parameter>
19           trigger_count(device_in(gearbox_oil_temperature,turbine,B12),24:00:00)
20         </parameter>
21         <operator>ge</operator>
22         <value>3</value>
23       </condition>
24     </conditions>
25     <action arg1=″set_parameter″ arg2=″turbine″ arg3=″:entity_instance″
26         arg5=″[blade_pitch,0]″>
27       device_out(arg1,arg2,arg3,,arg5)
28     </action>
29   </policy_rule>
30 </policy>
```

Operator Response Timeout

Two policies are given in this example. The first receives notification of high temperature (over 28 degrees) in the nacelle. The actions are to email the operator and to start an *alert* timer for 10 minutes to check for a response. The second policy triggers when this timer expires; the action is to send a further alert to the operator console.

```
<policy
    owner=″sensor-operator@site0113.windorg.com″
    applies_to=″@site0113.windorg.com″
    id=″Nacelle Temperature Alert″
    enabled=″true″
    changed=″2008-04-21T16:45:40″>
  <policy_rule>
    <trigger arg1=″nacelle_temperature″ arg2=″turbine″>
      device_in(arg1,arg2)
    </trigger>
    <condition>
      <parameter>parameter_values</parameter>
      <operator>gt</operator>
      <value>28</value>
    </condition>
    <actions>
      <and/>
      <action arg1=″alert″ arg2=″00:10:00″>start_timer(arg1,arg2)</action>
      <action arg1=″op1@site0113.windorg.com″
          arg2=″Nacelle temperature above 28, please respond″>
        send_message(arg1,arg2)
      </action>
    </actions>
  </policy_rule>
</policy>
```

```
<policy
    owner=″sensor-operator@site0113.windorg.com″
    applies_to=″@site0113.windorg.com″
    id=″Alert Expiry″
    enabled=″true″
    changed=″2008-04-21T16:59:02″>
  <policy_rule>
    <trigger arg1=″alert″>timer_expiry(arg1)</trigger>
    <action arg1=″message″ arg2=″console″
        arg5=″[Check email logs and respond to alert message]″>
      device_out(arg1,arg2,,,arg5)
    </action>
  </policy_rule>
</policy>
```

3.4.3 Resolution Policy Language Specification

This section describes the domain-specific triggers, conditions and actions used in resolution policies for sensor networks/wind turbine management. For resolution policies, any action may be associated with a trigger. The actual parameters of resolution actions must be literal values, or the values of *variable0* to *variable9* (if a trigger has bound them).

Since resolution policies are triggered by policy actions, the resolution policy triggers for sensor networks/wind turbine management are identical to the set of standard policy actions for this domain. Resolution policy conditions are the same as for standard policies for this domain. Domain-specific resolution actions extend the core set outlined in Figure 3.3 and include the set of domain-specific standard actions for the domain (i.e. *device_out*).

Example resolution policies for this domain that demonstrate the language now follow. Each example gives a high-level description of the policy together with the XML definition. For each sensor network/wind turbine resolution policy, an XML wrapper is required to specify the XML schema to validate the policy against. This takes the form:

```
<?xml version="1.0" encoding="UTF-8"?>
<policy_document
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
  "http://www.cs.stir.ac.uk/schemas/appel_resolution_sensor.xsd">
  ... policy body ...
</policy_document>
```

Sensor Node Configuration Conflict

This detects when two actions attempt to configure the same parameter on a sensor node at the same time (note the *message_type* argument must be *set_parameter*). The resolution is to apply the action of the newer policy (i.e. defined most recently). Note the use of the *join* function within the first condition (lines 16–24) – this is an alternative and neater way of comparing variable pairs for equality rather than using four separate conditions (i.e. comparing *:variable0* against *:variable1*, *:variable2* against *:variable3*, etc.). The *join* function is interpreted by the policy server at run-time (refer to the description of Expressions in section 3.2.1 for further explanation).

```
0  <resolution id="Parameter Config Conflict" owner="op1@site0113.windorg.com"
1    applies_to="@site0113.windorg.com" enabled="true" changed="2008-10-01T12:33:00">
2    <policy_rule>
3      <triggers>
4        <and/>
5        <trigger arg1="variable0" arg2="variable2" arg3="variable4"
6              arg4="variable6" >
7          device_out(arg1,arg2,arg3,arg4)
8        </trigger>
9        <trigger arg1="variable1" arg2="variable3" arg3="variable5"
10             arg4="variable7" >
11         device_out(arg1,arg2,arg3,arg4)
12       </trigger>
13     </triggers>
14     <conditions>
15     <and/>
16       <condition>
17         <parameter>
18             join(&,:variable0,:variable2,:variable4,:variable6)
19         </parameter>
20         <operator>eq</operator>
21         <parameter>
22             join(&,:variable1,:variable3,:variable5,:variable7)
23         </parameter>
24       </condition>
25       <condition>
26         <parameter>:variable0</parameter>
27         <operator>eq</operator>
28         <value>set_parameter</value>
29       </condition>
30     </conditions>
31     <action>apply_newer</action>
32   </policy_rule>
33 </resolution>
```

Turbine Blade Adjustment Conflict

This detects a conflict between two actions that attempt to set the blade angle of the same turbine to a different angle simultaneously. This time the resolution is domain-specific (as opposed to a generic resolution action) as the conflict affects turbine operation and it is safer to alert a human operator for further instruction rather than choose between actions. In this case, the resolution alerts an operator to the conflict and sets the blade pitch angle to 0 degrees (pitching the blades out of the wind and effectively stopping the turbine for safety reasons). Note the comparison of individual list values within the *parameter_values* argument using array notation – "[0]" meaning the first element in the list of values. The final condition (lines 25–29) checks that the value of *:variable8[0]* is the literal string “blade_pitch_angle”. Without this check, the resolution would be applied to any pair of *device_out* actions with the same parameter arguments, and not just those setting the blade pitch. A similar check for *:variable9[0]* is not necessary as it is compared against *:variable8[0]* in the previous condition. Note once again the use of the *join* function in the first condition (refer to the description of Expressions in section 3.2.1 for further explanation).

```
<resolution id="Blade Pitch Config Conflict" owner="op1@site0113.windorg.com"
  applies_to="@site0113.windorg.com" enabled="true" changed="2008-10-01T11:03:47">
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1="variable0" arg2="variable2" arg3="variable4"
          arg4="variable6" arg5="variable8">
        device_out(arg1,arg2,arg3,arg4,arg5)
      </trigger>
      <trigger arg1="variable1" arg2="variable3" arg3="variable5"
          arg4="variable7" arg5="variable9">
        device_out(arg1,arg2,arg3,arg4,arg5)
      </trigger>
    </triggers>
    <conditions>
    <and/>
      <condition>
        <parameter>
           join(&,:variable0,:variable2,:variable4,:variable6,:variable8[0])
        </parameter>
        <operator>eq</operator>
        <parameter>
           join(&,:variable1,:variable3,:variable5,:variable7,:variable9[0])
        </parameter>
      </condition>
      <condition>
        <parameter>:variable8[0]</parameter>
        <operator>eq</operator>
        <value>blade_pitch_angle</value>
      </condition>
    </conditions>
    <actions>
    <and/>
      <action arg1="op1@site0113.windorg.com"
          arg2="Blade pitch angle setting conflict - take action">
        send_message(arg1,arg2)
      </action>
      <action arg1="set_parameter" arg2="turbine" arg3=":entity_instance"
          arg5="[blade_pitch_angle,0]">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
  </policy_rule>
</resolution>
```

3.5 Conclusion

3.5.1 Chapter Summary

This chapter introduced the concept of a policy and described how policies can provide high-level, autonomous management of a system. Policy-based management was originally applied in computing for security purposes and low-level network management. More recently, policy languages and frameworks have evolved, and policies are now recognised as an effective means through which to manage systems in general. Although a variety of policy languages exist, this chapter focused on the Appel policy language, developed in conjunction with the Accent policy-based management system – the result of previous research at the University of Stirling. Appel comprises a core, generic policy language based on the notion of ECA (Event-Condition-Action), which may be extended to specialise the language for a particular domain area. Policies in Appel typically have a series of triggers, conditions and actions. A policy action dynamically affects the managed system (e.g. altering environment parameters or sending user alerts). A policy is triggered by an event in the underlying system, and its action(s) are executed provided its conditions are satisfied. Policies in Appel are one of two types: standard (or regular) policies and resolution policies. Standard policies detect and respond to events within the underlying system. Resolution policies detect and resolve conflicts among the actions of standard policies. The core Appel policy language was previously specialised for the domain of Internet telephony. In addition to outlining telephony policies, this chapter described and explained a new specialisation of the language for the domain of sensor networks/wind turbine management. Sample standard and resolution policies were given to demonstrate the language for each domain.

3.5.2 Evaluation

The core Appel policy language and its specialisation for Internet telephony were designed and implemented by the authors of the original Accent policy system. The language specialisation for sensor networks/wind turbine management is new research and is the focus of evaluation.

Policy language extensions for sensor networks include a single external domain-specific trigger (*device_in*) and action (*device_out*). These elements are generic and may be customised using particular parameter arguments. In comparison to the language for Internet telephony, this may seem very restrictive, but in fact reflects the flexibility of the language to support a wide range of sensor event notifications, and to configure devices not specific to any one type of sensor or to the wind power domain. For instance, the parameter arguments *message_type*, *entity_name* and *entity_instance* allow for any type of measurement from any entity – from sensors and portable devices to systems, software agents and graphical interfaces. This readily permits the language to be extended to other application areas. For example, the sensor network policy language specialisation has been adapted for the domain of home care networks under the auspices of the MATCH project [99]. In this area, the same trigger and action format is used to manage different types of sensors within a home environment (e.g. indoor temperature, door, window and bed occupancy sensors), to configure devices (e.g. switch off a TV or radio, adjust volume levels, and set an alarm clock) and to send alert messages (e.g. user messages to alert/remind a person a door is open or unlocked, or alert a carer when no movement is detected in the home during a day). The language is therefore flexible and applicable to multiple sensor-based domains.

One issue of the language is the use of the *message_period* parameter argument – optionally specifiable in *device_in* and *device_out.* The value for this parameter states when data was collected or sent (for a trigger) and when an action should be performed (for an action). Due to delays in processing and transmission over a network, it is unlikely that the times specified within this parameter will always be accurately respected. For example, if an action is to be performed at 1pm, network delays might defer this until 1:02pm. This factor is external to the underlying network implementation and out of the scope of the policy language and system. However, the inclusion of the parameter does give useful information even if the times are slightly skewed. How significant this might be would depend on the domain and the measurement or configuration in question. As the parameter is optional, it may be omitted in such circumstances. This issue would benefit from further research and pragmatic experimentation.

Chapter 4

Policy Ontology Modelling

This chapter introduces the concept of ontology in a computing context and explains how ontologies have provided an innovative solution to the issue of a domain-dependant policy user interface. The ACCENT policy-based management system and the APPEL policy description language were introduced in Chapter 3. ACCENT is a generic policy framework originally implemented for Internet telephony. APPEL is a policy language which may be specialised for a particular application. Motivated by a desire to reuse the ACCENT system across multiple domains, this chapter describes how ontologies have been employed to radically generalise the policy system user interface known as the "policy wizard" – which was previously specific to telephony. The presented approach re-engineered the policy wizard to replace hard-coded domain information with appropriate queries on a specially structured domain ontology. Domain knowledge is defined within an extensible hierarchy of ontologies that model the generic elements of the APPEL policy language together with user interface information and details particular to a domain specialisation of the policy language. The result of this work is a domain-independent policy wizard which can manage policies for any specialisation of the APPEL policy language.

Section 4.1 introduces ontologies and summarises available ontology languages and tools. This section also distinguishes the work presented in this chapter from related work where ontologies and policies are used, and outlines the motivation for undertaking the work. Section 4.2 outlines an approach to creating an ontology in general. Section

4.3 describes the overall approach to policy modelling using ontologies, while section 4.4 describes the ontologies that were created to model the policy language. Sections 4.5 and 4.6 describe domain-specific policy language ontologies for the applications of Internet telephony and sensor network/wind turbine management respectively. Section 4.7 explains how ontologies are integrated with the Accent policy wizard using an ontology parser system known as Poppet. Section 4.8 describes and evaluates the process of re-engineering the original policy wizard so it may utilise the ontologies. Finally, section 4.9 summarises and evaluates this work.

4.1 Introduction and Background

This section introduces the nature of an ontology and its application in computer science, together with available ontology languages and tools. The motivation for modelling the Appel policy language and associated domain information in an ontology is outlined, and related work combining the use of ontology and policies is summarised.

4.1.1 Ontology in Computing

The concept of ontology has been of great significance in the field of philosophy for a number of decades. Since the mid 1990s however, ontology 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 or the Semantic Web. The Semantic Web aims to revolutionise the Web by assigning explicit meaning to textual information to enable machines to automatically process and integrate information in a way not currently possible using plain text.

An ontology can be defined as the set of 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 rules of inference. Advantages of 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.

An ontology is distinct from existing data storage and modelling technologies such as expert systems, databases and object-oriented modelling [78]. Expert systems, knowledge-based systems and other AI techniques, present fundamentally useful concepts but lack the means to collectively combine knowledge on a global distributed scale. Ontologies can be combined and shared over the Internet.

In comparison with a relational database, an ontology differs in two major ways. Firstly, an ontology represents a inferable data model whereas a database models have a fixed structure and act as a data repository. Ontologies may be used to describe the interface through which an application may access data, while actual data instances are managed within a database. Secondly, an ontology can be queried in a more powerful fashion than a database can. Databases may only retrieve data as it was stored, whereas an ontology may be processed by an inference engine to reason about asserted facts and retrieve additional implied statements.

An ontology also differs from object-oriented modelling mechanisms such as the Unified Modelling Language (UML [20]), in that an ontology is built on logic which permits automated reasoning. Unlike an object-oriented model, an ontology can process the semantic associations of objects and properties.

In summary, ontologies are not a replacement for existing data storage and modelling concepts. The aim is to use ontologies in conjunction with other such approaches.

4.1.2 Ontology Languages

An ontology is defined using a particular ontology language. A variety of specialised languages exist. In general terms, ontology languages provide a mark-up to represent machine-readable content. The earliest ontology languages such as DAML (DARPA Agent Markup Language [3]) and OIL (Ontology Inference Layer [6]) were designed to develop tools and ontologies for specific communities – namely medical research and proprietary commercial applications [78]. However, following the semantic web initiative led by the W3C, several languages emerged to provide ontology compatibility with the World Wide Web. Specifically, a number of semantic mark-up languages were developed based on XML (eXtensible Mark-up Language [15]). XML provides syntax for structured documents and is closely coupled with the XML Schema language

[15], which restricts the content and structure of an XML document. While this is sufficient for the purpose of exchanging data, neither XML nor XML Schema supports the definition of semantic constraints. Consequently, a number of new languages have evolved for ontology use – based on XML syntax.

The Resource Description Framework (RDF [17]) was the first ontology language to be standardised by the W3C. RDF is based on XML notation and can be used to create data models of objects and to express the relationships between them. However, as use of RDF widened, it was found to be too limited in its expressiveness. The result was the extension of RDF to form DAML+OIL [2] (combining aspects of DAML and OIL language components respectively). Following early application, DAML+OIL grew and was revised to form OWL (the Web Ontology Language [19]).

Ontologies 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. The language was officially standardised by the W3C in February 2004. It was designed to combine and extend the customisable tagging of XML with the flexible data representation ability of RDF, with a view to formally describing the semantics of terminology in a domain. Consequently, OWL provides a larger function range than any other ontology language to date. The formal semantics of OWL specify how to derive facts not literally defined in the ontology, but entailed by the semantic expressions associated with classes and individuals.

OWL was the language of choice for the ontologies later described in this thesis. It was chosen primarily due to its standardisation and the benefits this brings in terms of available software tool support.

4.1.3 The OWL Ontology Language

An OWL 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 simple ontology example is shown in Figure

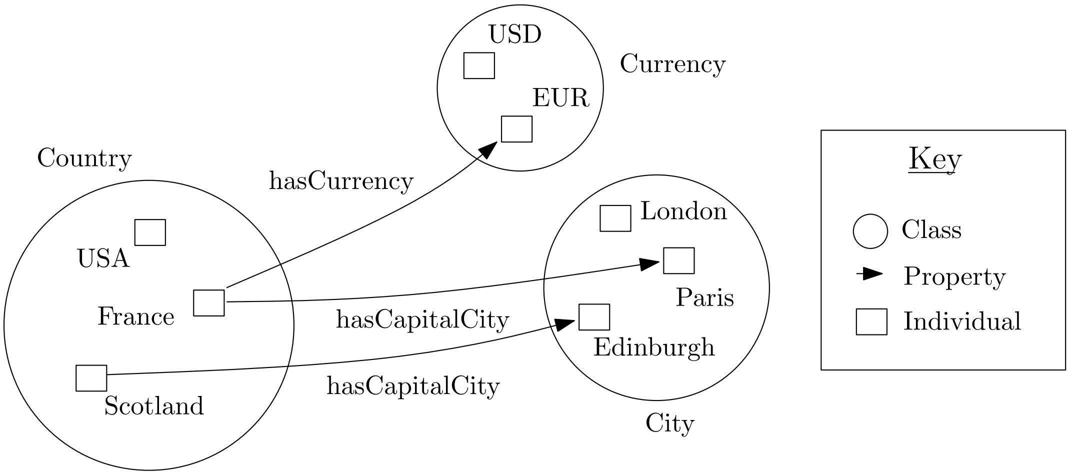


Figure 4.1: OWL Ontology Example

4.1, describing countries, capital cities and currencies.

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 [19] provides a complete description of the language and its constructs.

OWL supports the sharing and reuse of ontologies by means of ontology importation. Using this mechanism, definitions of classes, properties and individuals within an imported ontology, may be used by the importing ontology. Semantic entailments can also thus be made based not just on a single document, but multiple distributed ontologies that have been combined using the import mechanism. This is a key aspect of a Web-based ontology language such as OWL.

An short example OWL document based on the "countries" example above is shown in Figure 4.2 and continued in Figure 4.3.

The key elements to note are:

- The document begins with the usual XML version header (line 0). The first tag declares an RDF document (as OWL is built on top of and extends previous RDF constructs). A list of XML namespaces is given (*xmlns* declarations) which

```
<?xml version=″1.0″?>
<rdf:RDF
    xmlns:rdf=″http://www.w3.org/1999/02/22-rdf-syntax-ns#″
    xmlns:rdfs=″http://www.w3.org/2000/01/rdf-schema#″
    xmlns:owl=″http://www.w3.org/2002/07/owl#″
    xmlns:pm=″http://www.example.com/owl/planet-map.owl#″
  xml:base=″http://www.example.com/owl/world-map.owl″>

  <owl:Ontology rdf:about=″″>
    <owl:versionInfo>World Map Ontology v.1.2. GAC 2008</owl:versionInfo>
    <rdfs:comment>Describes countries, cities and currencies</rdfs:comment>
    <owl:imports rdf:resource=″http://www.example.com/owl/planet-map.owl″/>
  </owl:Ontology>

  <!-- Classes -->

  <owl:Class rdf:ID=″Country″>
    <rdfs:subClassOf rdf:resource=″pm:PlanetLandArea″/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:someValuesFrom rdf:resource=″Currency″/>
        <owl:onProperty rdf:resource=″hasCurrency″/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:someValuesFrom rdf:resource=″City″/>
        <owl:onProperty rdf:resource=″hasCapitalCity″/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality>1</owl:cardinality>
        <owl:onProperty rdf:resource=″hasCapitalCity″/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:ID=″City″>
    <rdfs:subClassOf rdf:resource=″pm:PopulatedArea″/>
  </owl:Class>

  <owl:Class rdf:ID=″Currency″
    <rdfs:subClassOf rdf:resource=″owl:Thing″/>
  </owl:Class>
```

Figure 4.2: Example OWL Document (Part 1 of 2)

```
45
46    <!-- Individuals -->
47
48    <Country rdf:ID=″Scotland″>
49      <hasCapitalCity rdf:resource=″#Edinburgh″/>
50    </Country>
51
52    <Country rdf:ID=″France″>
53      <hasCapitalCity rdf:resource=″#Paris″/>
54      <hasCurrency rdf:resource=″#EUR″/>
55    </Country>
56
57    <Country rdf:ID=″USA″/>
58
59    <City rdf:ID=″London″/>
60    <City rdf:ID=″Paris″/>
61    <City rdf:ID=″Edinburgh″/>
62
63    <Currency rdf:ID=″USD″/>
64    <Currency rdf:ID=″EUR″/>
65
66    <!-- Properties -->
67
68    <owl:ObjectProperty rdf:ID=″hasCurrency″/>
69    <owl:ObjectProperty rdf:ID=″hasCapitalCity″/>
70
71  </rdf:RDF>
```

Figure 4.3: Example OWL Document (Part 2 of 2)

specifies URI locations of existing documents whose elements are referenced or extended in this ontology, and defines the abbreviations used throughout this document (e.g. "owl" is the abbreviation for *http://www.w3.org/2002/07/owl#* shown in line 4). To separate the URI location address from an identifier reference (e.g. named ontology construct such as a class or property name) a *fragment identifier* in the form of a "hash" symbol ("#") is appended to the URI (shown in lines 2-5).

- The base namespace of this ontology document is defined (line 6). Note this is just a URI and does not require a fragment identifier ("#") like the ontology import references above it.

- The *owl:Ontology* block declares optional metadata including the document version (line 9) and a comment (line 10). Imported ontologies are listed here as well. This ontology imports an ontology called "planet-map.owl" (specified on line 11). Note the namespace for this ontology is defined as *pm* previously on line 5.

- Lines 16 to 44 define three OWL classes.

  - The *Country* class (lines 16-36) is a subclass of *pm:PlanetLandArea* (extending a class in the imported ontology) (line 17). Property restrictions in OWL are also defined as subclasses (using *rdfs:SubClassOf*) using embedded OWL tags. The *Country* class has three property restrictions. The first is a *someValuesFrom* property restriction associating this class with the class *Currency* using the property *hasCurrency* (lines 18-23). This specifies abstractly that some (none, one or more, but not all) individuals or subclasses of the class *Country* are related to an individual or subclass of the class *Currency*. Similarly, the *Country* class is related to the *City* class using the property *hasCapitalCity* (lines 24-29). The final property restriction (lines 30-35) defines a cardinality – the class *Country* must have exactly one relationship with another class/individual using the *hasCapitalCity* property. In other words, a country must have one capital city.
  - The *City* class (lines 38-40) is a subclass of *pm:PopulatedArea* and has no defined property restrictions.
  - The *Currency* class (lines 42-44) does not extend any previously defined class and is therefore a subclass of the document root: *owl:Thing* (line 43). Note that all OWL classes are inferred subclasses of this root OWL element.

- Lines 48 to 64 define individuals for each of the three classes. Referring to the example ontology diagram of Figure 4.1, the *Country* class has three individuals (instances of the class which inherit the restrictions on the class as well as those defined by their particular restrictions). The format of an individual definition is to begin the tag of the associated class, then specify an attribute declaring the ID of the individual (e.g. *rdf:ID="Scotland"* on line 48). Properties (links to other individuals or classes) define the property name (the ID) and the particular resource to link to (e.g. individual "Scotland" *hasCapitalCity* "Edinburgh" on line 49). Note the use of a fragment identifier ("#") (lines 49, 53-54) which indicates the referenced individual is defined within the "base" ontology.

- Lines 68 and 69 define the properties *hasCurrency* and *hasCapitalCity*. These are object properties that link individuals and classes with other individuals and classes. OWL offers many other types of property, including data-type properties that are intended to link an individual or class with a specific value (int, String, boolean, etc.) rather than a specific ontology construct.

In addition to the OWL constructs described in the example code, classes may include a variety of RDF or RDFS attributes such as labels, comments, version information, or other user-defined meta-data. For example, *rdfs:label* can specify a human-readable text label for a class that might be used in queries rather than the unique class name or URI, and *rdfs:comment* specifies a human-readable class description useful for applications that process the ontology and when browsing/editing the ontology in an editor.

The OWL language is broken into three language levels that provide mounting strengths of expressiveness to meet the needs of different users and implementers. In descending order, these 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 Description Logic (DL) reasoning.

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 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 reasoning tools.

The OWL ontologies later described in this thesis conform to the OWL DL sublanguage. A further, formal definition of the differences between OWL dialects can be found in the OWL Semantics and Abstract Syntax guide [18].

4.1.4 Ontology Tools

The choice of tools for ontology development depends on the language used, the level of sharing required, and how the ontology will be used and accessed by an application or system. A number of ontologies and an ontology parser system have been developed and are presented later in this chapter. These ontologies were written using OWL and the tools used during development are outlined below.

OWL ontologies can be checked for consistency using syntax and semantic validation tools. OWL syntax checking confirms the dialect of language the document conforms to (e.g. Full, DL or Lite) based on the constructs found. OWL consistency checking detects semantic errors including errors in referenced ontologies (i.e. any imports to the document) and reports any inconsistencies found. The web-based WonderWeb [22] OWL ontology validator can be used to carry out both types of checking.

An ontology editor provides a visual representation of a conceptual view of classes, properties and individuals, while generating the raw document mark-up on the fly. Protege [9] is a free, open-source ontology editor and knowledge-base framework. Protege offers a plug-in environment through which to create and manage ontologies, and has built-in support for consistency checking and also automated reasoning via an external Description Logic (DL) compatible reasoning engine. OWL ontologies are supported through the Protege-OWL plug-in. An extensible framework, Protege is enhanced using various third party graphical plug-ins such as OWLViz [7] and Jambalaya [4].

Jena [16] is an open-source Java-based framework, designed for use in building semantic web applications. Given an ontology, it offers methods to access, parse and build a scalable model of the document. Additional methods can search the model for specific classes and properties, and can query the restrictions and constraints placed upon them.

An inference engine (or automated reasoner) derives additional statements that an

OWL document does not explicitly express. Pellet [8] is an open-source Java-based reasoner, specifically geared toward OWL DL compliant ontologies. RacerPro [10] is a standalone commercial reasoner and inference server available under free educational license. Both reasoners were used to develop and utilise the ontologies presented later in this thesis.

4.1.5 Existing Policy and Ontology Work

Ontologies were originally used to represent web page content and services in the context of the Semantic Web, with the aim of improving the accuracy and relevancy of online Web searches. More recently, ontology languages have been used to represent and reason with policies – in particular, the KAoS framework discussed in section 3.1.3. However, these approaches use ontologies as a mechanism to *define* policies and store policy data.

The approach presented in this chapter does not use ontologies as a policy language or policy system. Here, ontologies describe the structure of domain-specific policies that may be defined using the Appel language. The domain ontologies presented here are not used to create and store policies. Instead, they provide a structured knowledge base through which drive the policy wizard user-interface and other policy system tools and components.

The closest related work to the approach in this chapter lies in techniques that use ontologies to customise software components within applications or interfaces, such as [41] and [97].

4.1.6 Motivation For Policy Ontology Modelling

The Accent policy system, discussed in Chapter 3, was originally developed for the domain of Internet telephony. While the policy server and network interface components were implemented as a generic framework, the web-based policy wizard was hard-coded to this domain. In particular, the policy triggers, conditions and actions selectable via this interface were specific to a telephony environment. This presented a fundamental flaw in fully utilising the Accent system for policy-based control within a different application area.

The solution to this problem lay in extracting hard-coded telephony details from the policy wizard and housing them externally in a way that could be dynamically accessed. An extensible structure was required through which the policy language (and the policy wizard) might easily be customised for new domains. Ontologies were chosen as the means to model the Appel policy language and domain-specific language aspects.

Ontologies were deemed more appropriate over a flat file structure or database as they define information at different levels of abstraction. This ability to build hierarchical specifications of an area of knowledge is useful to model the generic aspects of Appel and extensions for different domain specialisations of the language. Ontology documents are also processable by software applications, therefore making them suitable for integration with the policy wizard.

4.2 Ontology Building

This section describes a generic approach to building an ontology. While the procedure may vary depending on the chosen ontology language, the tools adopted, and the domain being modelled, there are general steps applicable when developing any ontology:

1. **Gather domain knowledge:** Compile information resources and expertise to accurately and formally describe the domain in question. This includes key terms and their associations which can be modelled using an ontology.

2. **Design the ontology structure:** Construct a conceptual structure of the domain. This should involve mapping gathered terms and associations into appropriate classes and properties, and defining the nature of the relationships between them.

3. **Expand the details:** Following creation of a basic structure, more detail may be added. Iteratively, concepts (classes), relationships (properties), data restrictions, comments and individuals may be added to the level of detail necessary. However, further details may always be included at a later date.

4. **Perform logical checking:** To verify correct form, the document syntax can be checked against the chosen ontology language – usually automatically. Logical tests can also be carried out to determine semantic inconsistencies among defined ontology elements. Such consistency checking may also involve automatic classification that defines new concepts based on individual properties and class relationships.

5. **Publish the ontology:** Once the designer and domain experts are satisfied with the ontology it can be published at a publicly accessible location or used in a private application.

The approach above is generic. Tools-wise, software modelling and software requirements elicitation techniques might commonly be used for step 1. Steps 2 and 3 would be tackled using an ontology editor. Graphical visualisation is useful for classes, properties and relationships as they are constructed. Validators should be used to check syntax and semantic consistency in step 4.

4.3 Policy Ontology Approach

With the aim of re-engineering the Accent policy wizard to incorporate ontological data in place of hard-coded domain knowledge, developments were undertaken in three stages:

1. The Appel policy language was modelled using ontology constructs. The ontology language used was OWL (the Web Ontology Language [19]), discussed previously in section 4.1.2. Separate OWL ontologies were formed to model the generic policy language constructs, additional policy wizard interface features, and information specific to a domain specialisation of the language. Combined, these three ontologies form a single, structured OWL document.

2. A system named Poppet (Policy Ontology Parser Program – Extensible Translation) was developed to parse an ontology and build an interpretable model of its structure. In addition, an API was designed to enable an application (such

as the policy wizard) to query the stored ontology model and retrieve relevant information.

3. The Accent policy wizard and associated user-interface layer components were re-engineered to remove hard-coded telephony knowledge, and replace it with appropriate calls to an ontology via the Poppet system interface.

Section 4.4 outlines the ontology framework (step 1), while section 4.7 describes the Poppet system and how ontologies are parsed and processed (step 2). Section 4.8 describes how the Accent policy wizard was adapted to use these ontologies (step 3).

4.4 Policy Ontology Stack

The Appel policy language was modelled as a set of separate but interrelated OWL ontologies. From this, domain-specific extensions to the core language were defined through the creation of further ontology documents. Developing ontologies to describe the core and specialist elements of Appel provides a method of separating generic language knowledge from application-specific knowledge. This provides a generalised ontology structure that is easily adaptable to defining policy information for different domains. The work presented here has been published in [52], for telephony.

Domain-specific knowledge applicable to the policy system is captured within a single ontology that imports ontologies describing the core policy language and common policy wizard interface features. This ensures domain knowledge is defined within a structure which relates it to policy language terms. A domain-specific ontology is therefore constructed using a three-tier "ontology stack" or hierarchy of separate ontology documents, as shown in Figure 4.4.

A description of each of these ontologies shortly follow, using diagrams where appropriate to show the implemented ontology classes and properties. These diagrams were created using the OWLViz [7] and Jambalaya [4] plug-ins for the Protege ontology editor. In these diagrams, a square or oval represents an OWL class. The connecting lines between classes either depict class inheritance (where a hollow triangular arrow appears at the end of a line) or an OWL property restriction (where a hollow triangu-

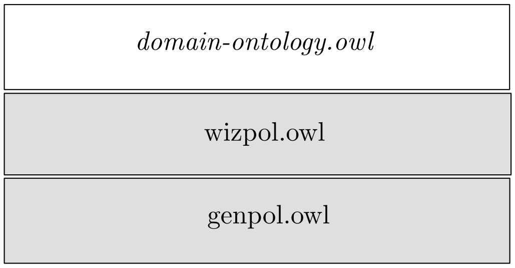


Figure 4.4: Policy Ontology Stack

lar arrow appears at the middle of a line). A class that appears with darker shading indicates some or all of its subclasses are inferred as a result of logical reasoning. Solid black arrows to the left or right of a class (within an oval) indicate presence of further superclasses or subclasses respectively, which are hidden for clarity. Note that the class *owl:Thing* is the top-level class of any OWL ontology and all defined ontology objects (i.e. classes, data types, properties, etc.) are subclasses of this.

The following subsections outline the *GenPol* and *WizPol* ontology structure used to model the core constructs of the Appel policy language only. These ontologies also define domain information for use with the goal-directed approach (discussed in Chapter 6) and policy conflict filtering (discussed in Chapter 5), but this content is described later in this thesis.

4.4.1 Generic Policy Ontology

The generic policy language ontology, referred to by the acronym *GenPol*, describes the core constructs of the Appel policy language. A detailed description of *GenPol* is given in [49] and in [50] for resolution policies. The OWL ontology document may be accessed from [21]. Contained within this ontology is a definition of key language terms and how they relate to one another. This includes the concept of a “policy document” and its various constituent parts – including policy rules, trigger events, conditions, actions and additional attributes, variables and operators. Relationships between such concepts are defined by way of a specified property or traditional inheritance, and describe named associations, inheritance properties and cardinality restrictions. *GenPol* ultimately describes a skeleton structure of ontology classes and properties which can be imported and extended within a domain-specific policy language ontology. The following subsections describe the basic class structure. A summary of the ontology properties defined for *GenPol* is included in Appendix B.

GenPol **Standard Policy**

Standard policies react to domain events. *GenPol* describes the make-up of a *StandardPolicy* as shown in Figure 4.5.

GenPol **Standard Policy Triggers, Conditions and Actions**

A *PolTriggerEvent* is defined as a subclass of *TriggerEvent* as shown in Figure 4.5. Core policy triggers are defined in *GenPol.* There is one generic trigger *TimerExpiry* as shown in Figure 4.6.

A *PolCondition* is defined to have a single association with each of *PolConditionParameter*, *ConditionOperator* and *PolConditionValue.* The parameter and value of a *PolCondition* are customisable to the domain, whereas the operator must be selected from the subclasses of *NamedConditionOp* as shown in Figure 4.5. *GenPol* defines generic condition parameters for use in domain-specific policy language specialisation. These include date, day and time parameters, and state history functions for counting triggers and actions. These parameters are defined as subclasses of *PolConditionParameter* as shown in Figure 4.7.

A *PolAction* is defined as a subclass of *Action* as shown in Figure 4.8. A standard policy action can also be a trigger within a resolution policy. Therefore, *PolAction* is also a subclass of *ResTriggerEvent.* Conversely, *ResTriggerEvent* is a subclass of *PolAction.* Generic policy actions are internal actions, including actions to start/restart/stop a timer, set/unset a variable, log events and send messages. These actions are defined as subclasses of *PolAction* as shown in Figure 4.8.

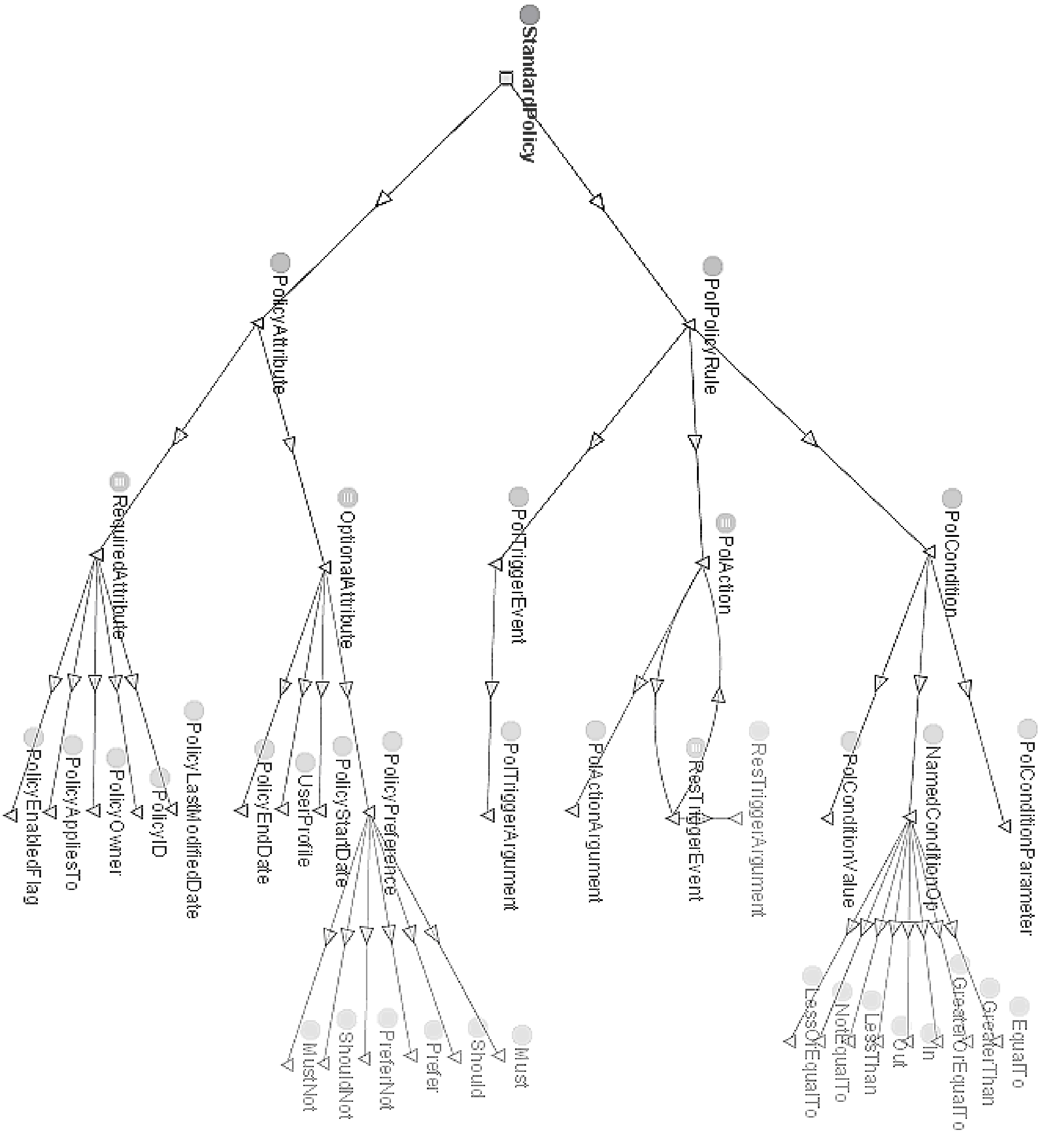


Figure 4.5: *GenPol* Standard Policy

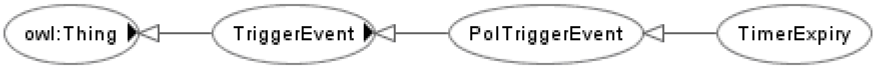


Figure 4.6: *GenPol* Standard Policy Triggers

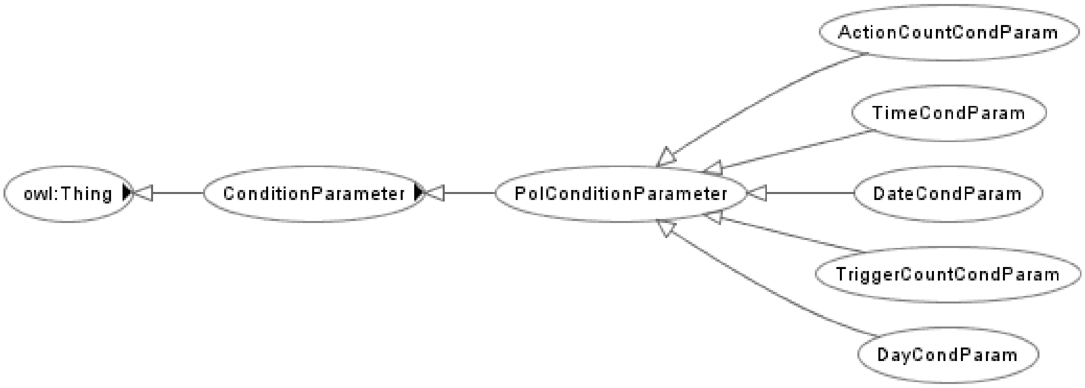


Figure 4.7: *GenPol* Standard Policy Condition Parameters

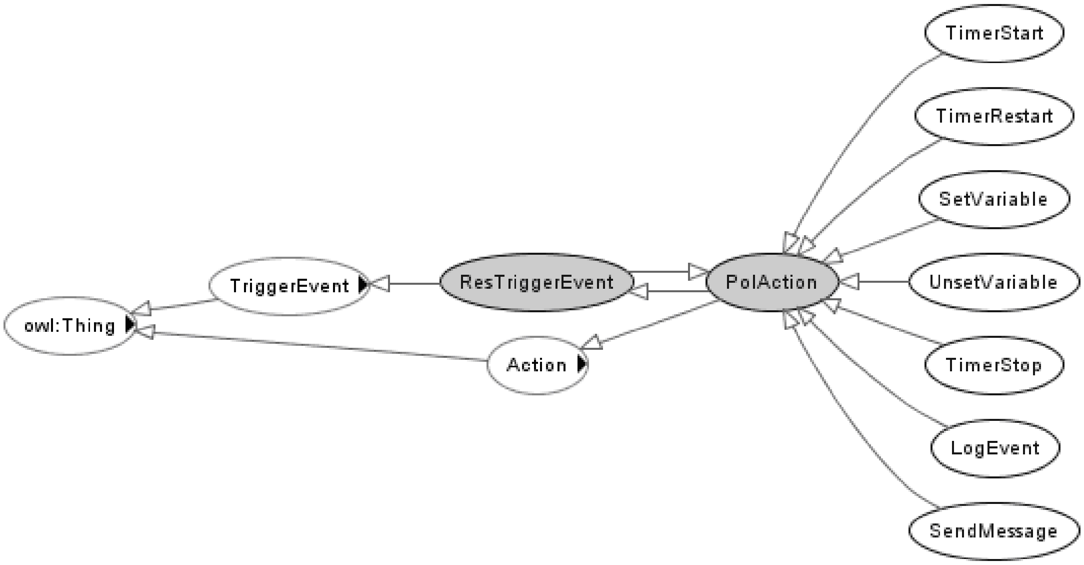


Figure 4.8: *GenPol* Standard Policy Actions

***GenPol* Resolution Policy**

A resolution policy is used to express how conflicts between actions of standard policies may be resolved. Resolution policies follow the same structural format as standard policies, with differences occurring in the types of arguments and parameters of triggers, conditions and actions. *GenPol* defines the make-up of a *ResolutionPolicy* as shown in Figure 4.9.

4.4.2 Wizard Display Ontology

The wizard display ontology, referred to by the acronym *WizPol* incorporates ACCENT policy wizard format and layout information used in the presentation of domain policy information. This information is purely user-interface related and does not define actual policy language constructs. The OWL ontology document may be accessed from [21]. Although these features are not part of the policy language itself, they are common and useful to any domain-specific ontology that is geared towards use with the policy system. *WizPol* imports the *GenPol* ontology, and associates generic display and user interaction details with policy constructs. For example, *WizPol* defines classes that group domain-specific triggers, conditions and actions into categories. These categories are used to group policy options into appropriate on-screen menus and restrict the range of policy options displayed depending on the designated user level (explained in the following sections) of those accessing the wizard. In this way, *WizPol* acts as a generic description of user interface options that can specialise the APPEL language for use with the ACCENT policy wizard. *WizPol* also defines additional class structures used to specify wizard-related information in the form of status variables, data typing and unit typing. The key aspects of this ontology are now outlined. A summary of the ontology properties defined for *WizPol* is included in Appendix B.

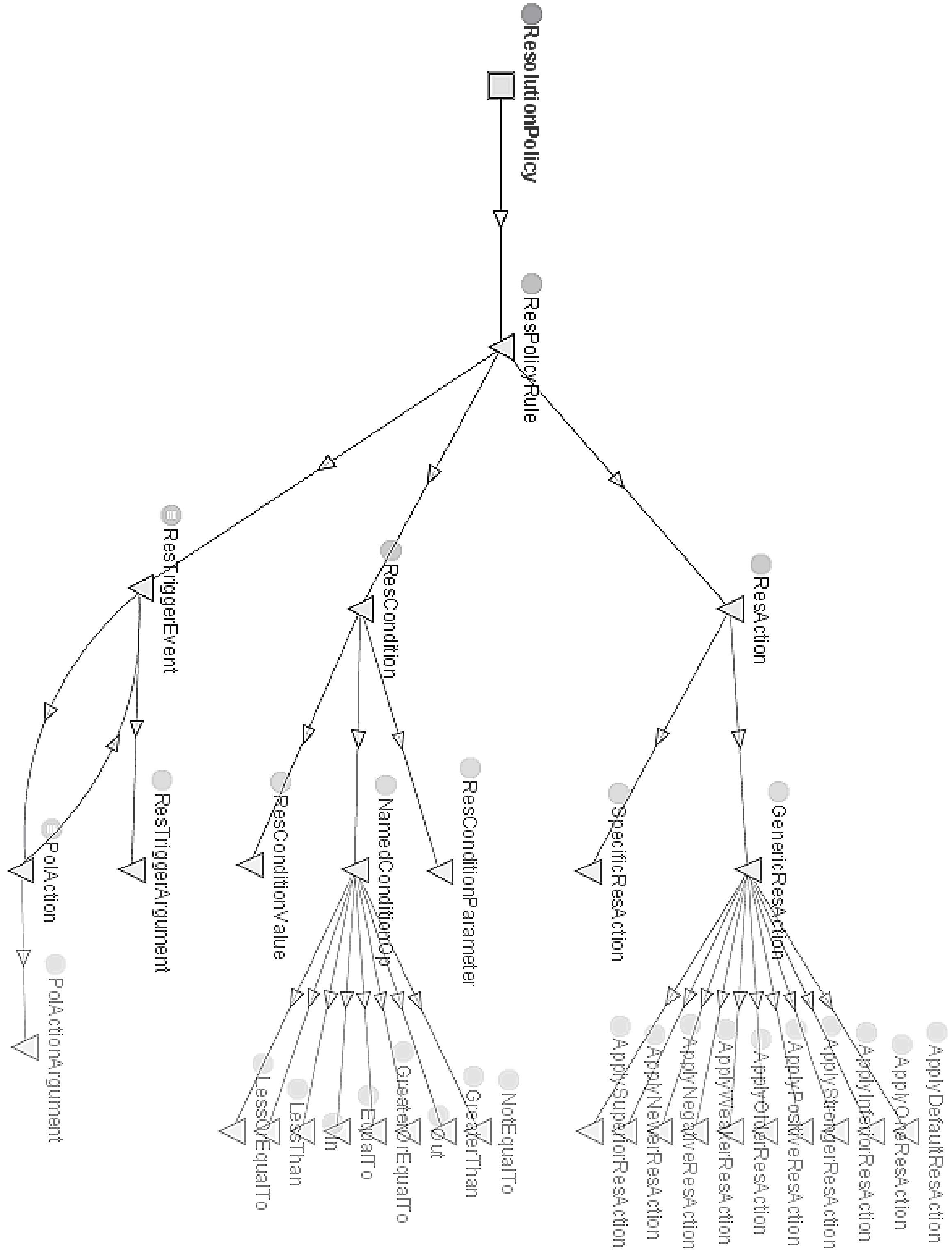


Figure 4.9: *GenPol* Resolution Policy

WizPol Class Categorisation

The policy wizard categorises (or groups) related triggers, conditions, actions and operators in a domain for processing and display purposes. Each category represents a subset of common policy options. There are five categories defined in *WizPol* as shown in Figure 4.10.

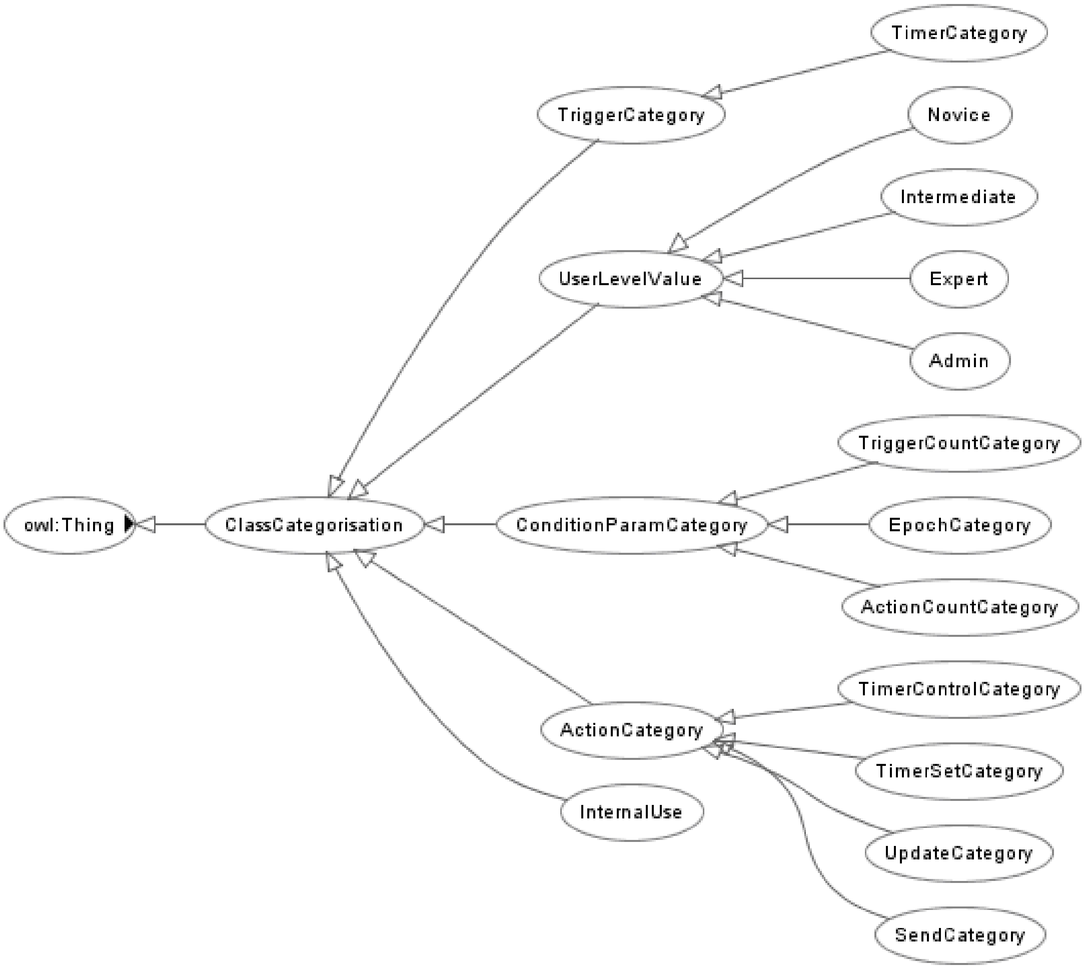


Figure 4.10: *WizPol* Class Categorisation Top Level

The top-level class in this structure is *ClassCategorisation*. Defined subclasses of this class represent the *UserLevelValue*, *InternalUse*, and categories through which domain-specific trigger, condition parameter and action categories can be specified.

WizPol **User-Level Categorisation**

The policy system interface supports a four-level classification of its users, offering varying degrees of functionality depending on the expertise of a user. In particular, each user level corresponds to a specific subset of triggers, condition parameters and actions permitted for display and selection. Each user level is defined as a subclass of *UserLevelValue* as shown in Figure 4.10.

WizPol **Internal Use Categorisation**

Within a domain, a trigger, condition or action may be defined which accesses or modifies a variable stored locally in the policy system. Such instances can be classified as having "internal usage". *WizPol* provides an *InternalUse* class and the property *hasInternalUse*. These can be used together to infer classes deemed as internal.

WizPol **Trigger, Condition and Action Categorisation**

Rather than displaying triggers, conditions and actions as single continuous lists, the wizard assembles them into categories, which are presented to the user as shorter sub-lists. This categorisation is useful not only for display purposes, but also for grouping options with similar properties, such as by number of parameter arguments or by related parameter data types. Such grouping makes it easier to validate user input when defining policies.

WizPol defines top-level categories named *ActionCategory*, *TriggerCategory* and *ConditionParamCategory* as shown in Figure 4.10. In a domain-specific ontology, particular named categories are defined as subclasses of these, which categorise domain-specific policy elements.

4.4.3 Domain-Specific Policy Ontology

GenPol and *WizPol* are intended to be entirely reusable in defining an ontology that represents a particular application domain for the policy system. Due to the nested 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.

Although a domain-specific ontology structure is 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, tools or agents.

A worked example of creating a domain-specific ontology is outlined in a technical report describing the policy ontology approach [47]. In summary, the process involves the following steps (which correspond with steps 2 and 3 in the ontology building steps in section 4.2):

1. Create a new OWL ontology, define its namespace URI, and import the *WizPol* ontology (which in turn imports *GenPol*)
2. Create named triggers, conditions and actions for the domain
3. Define categories of trigger, condition parameters and actions used for display and grouping purposes within the policy wizard
4. Place property restrictions on triggers, conditions, actions and other policy components to associate them with relevant categories
5. Optionally define additional non-policy language related knowledge about the domain.

For compatibility with existing formal reasoning tools, both *GenPol* and *WizPol* were designed to conform to the OWL DL sub-language. These base ontologies, and the domain-specific ontologies which extend them, should 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 and 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, a domain-specific ontology is 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 instances of classes are created, modified and removed in real time to reflect the changing nature or state of the domain.

Domain-specific policy ontologies have been developed that model the Appel policy language specialisations for Internet telephony and sensor network/wind farm monitoring. These ontologies have been used within the Accent policy system and related software tools described in this thesis. A description of each domain ontology is given in the following sections.

4.5 Application 1: Ontology for Internet Telephony

This section describes the ontology classes and structure defined to model the Appel policy language specialisation for Internet telephony. This ontology extends the class structures defined in the core language ontologies (*GenPol* and *WizPol*) to define the triggering events, condition parameters, actions, arguments, variables and data types specific to this domain. Additionally, the ontology describes general, non-policy related knowledge. The ontology for Internet telephony is explained in full in a technical report [48]. The OWL ontology document may be accessed from [21].

Section 4.5.1 summarises the main domain extensions for standard telephony policies. Section 4.5.2 summarises the main domain extensions for resolution policies.

4.5.1 Standard Policy Extensions

Telephony Class Categorisation

The telephony ontology extends *wizpol:ClassCategorisation* to define domain-specific categories of triggers, condition parameters and actions. Classes defined under this structure are used to achieve category inference throughout the ontology. Each category (or grouping) usually represents a set of constructs that are logically similar or have the same number and form of parameter arguments.

Telephony Triggers

Specific triggers for telephony are defined as subclasses of *wizpol:NamedTriggerEvent* as shown in Figure 4.11.

The use of each trigger can be inferred as a subclass of a particular category (for policy wizard display purposes) or user-level (to associate different sets of triggers with each "admin", "expert", "intermediate" and "novice" designation). Association between triggers and categories is achieved by defining properties on each named trigger. For example, the trigger *BandwidthRequestEvent* can be defined to have an association with the *CallTriggerEvent* category using the property "*wizpol:hasCategory* CallEventCategory". Logical inference (using an inference engine), can then determine that *BandwidthRequestEvent* is a subclass of both *wizpol:NamedTriggerEvent* and *CallTriggerEvent*. So a class may be defined once and associated with others simply through the properties it is given. When the class is deleted, so too are the links to other classes. Similarly, each trigger may be associated with user levels. For telephony, the trigger categories are "Availability" and "Call". Inferred subclasses of these categories are shown in Figure 4.12 and Figure 4.13 respectively. A trigger may belong to just one of these categories – no trigger may be associated with both.

Telephony Conditions

The full set of defined telephony condition parameters is shown in Figure 4.14. Inferred sets of these condition parameters for each condition category ("Address", "Amount" and "Description") are shown in Figure 4.15, Figure 4.16, and Figure 4.17 respectively.

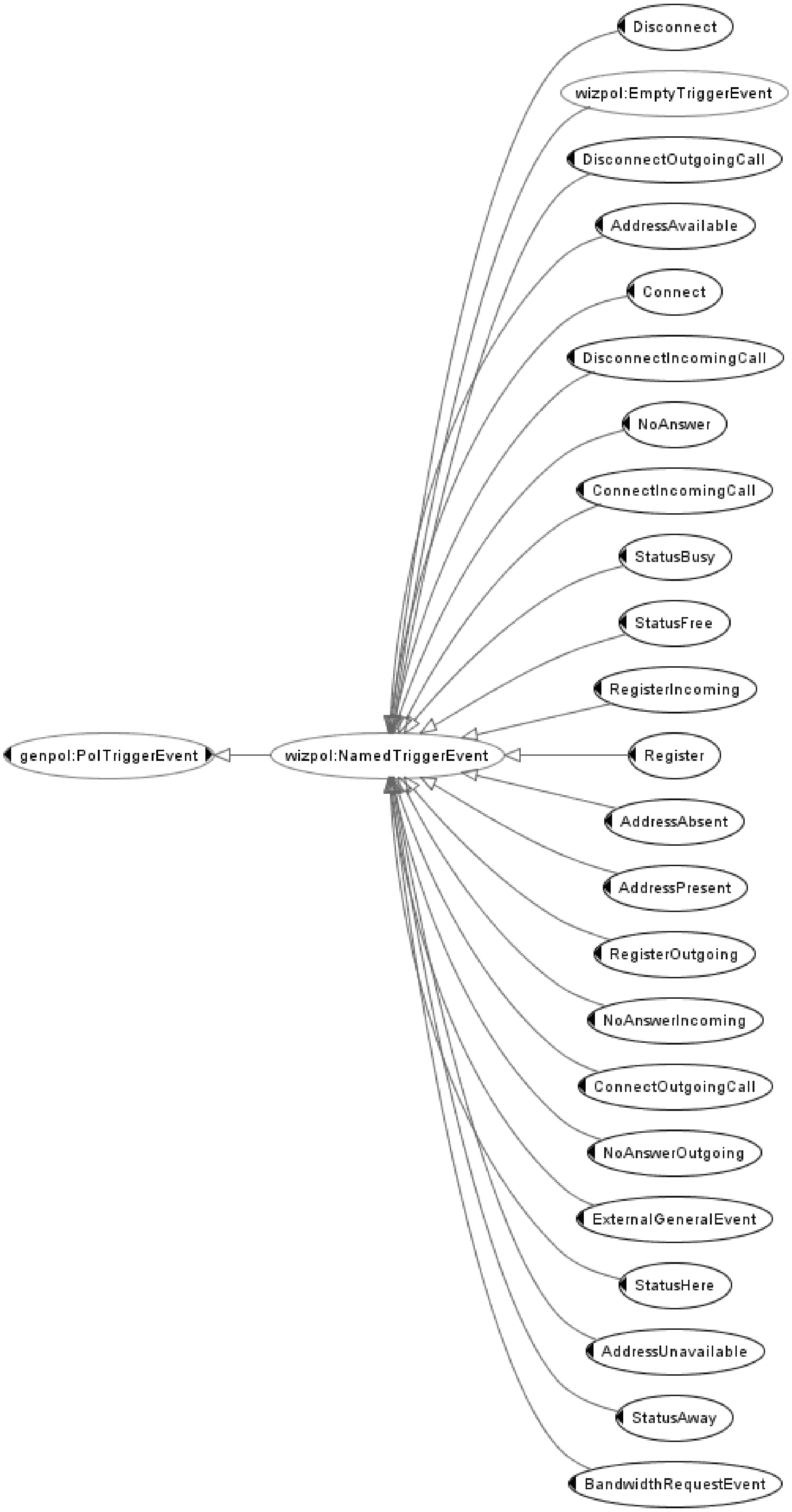


Figure 4.11: Telephony Named Triggers

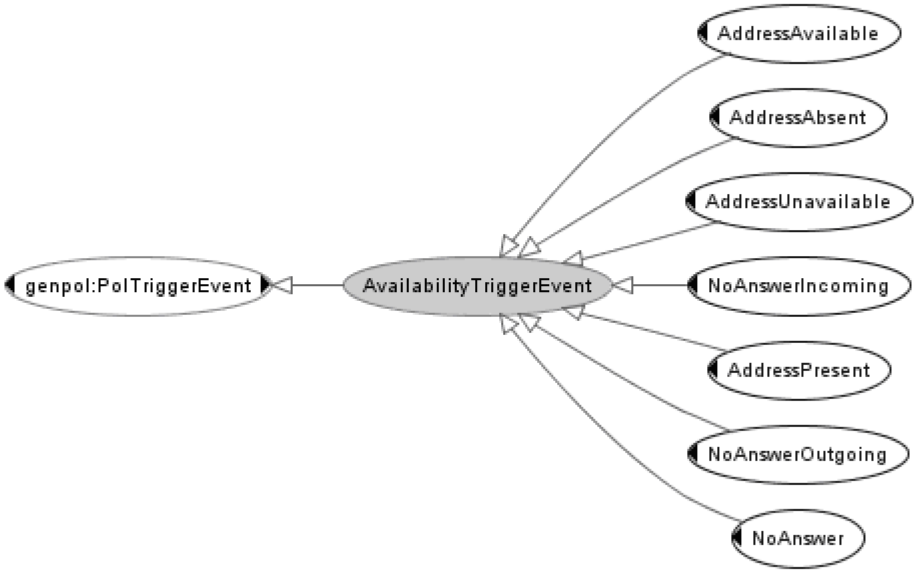


Figure 4.12: Telephony Inferred Triggers for the Availability Category

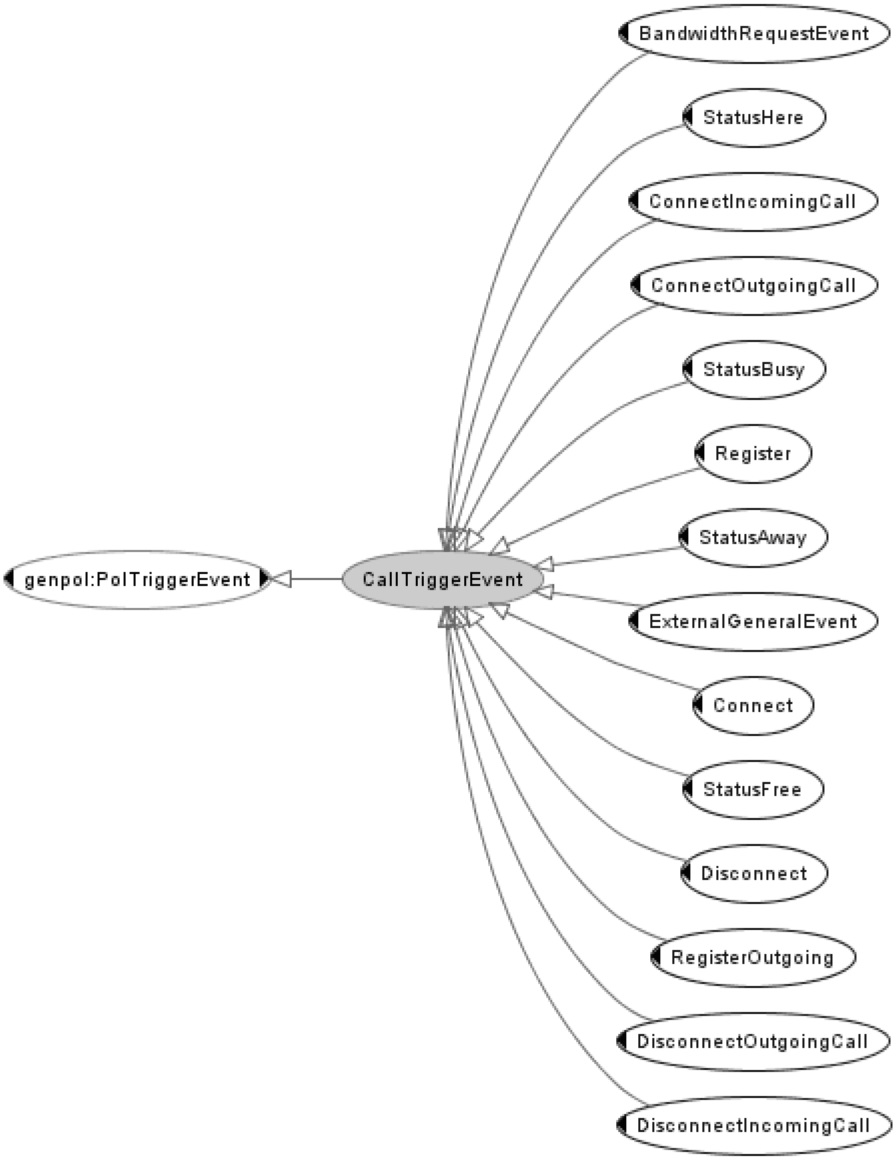


Figure 4.13: Telephony Inferred Triggers for the Call Category

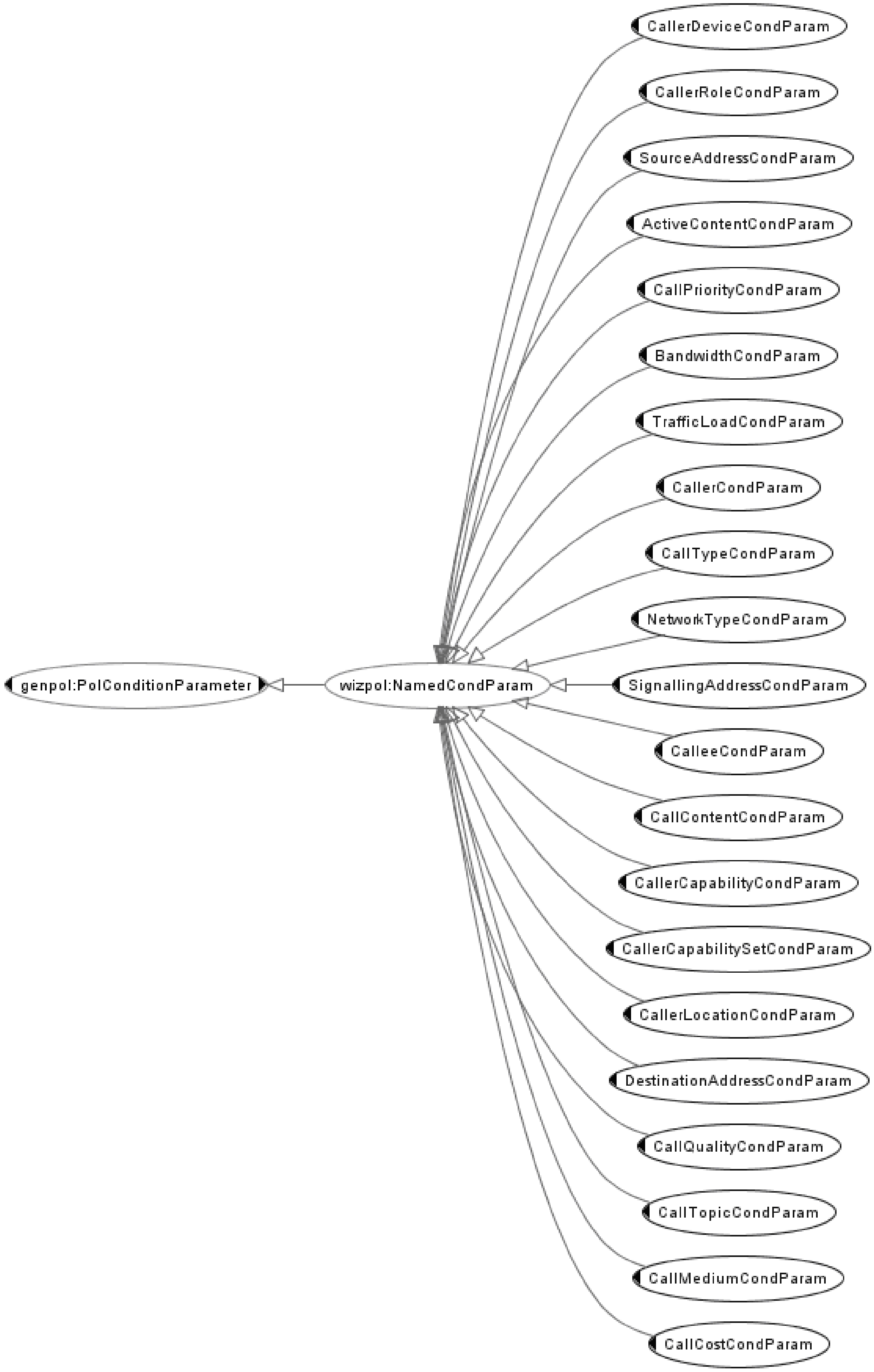


Figure 4.14: Telephony Named Condition Parameters

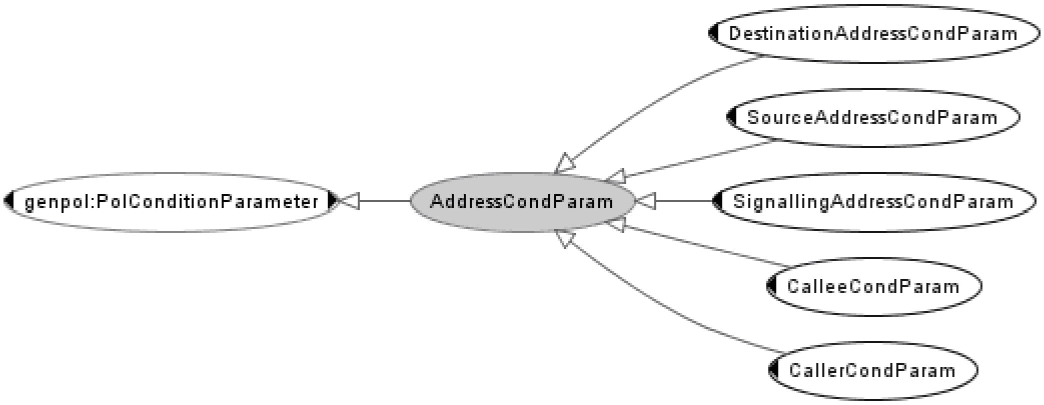

DestinationAddressCondParam
SourceAddressCondParam
genpol:PolConditionParameter
AddressCondParam
SignallingAddressCondParam
CalleeCondParam
CallerCondParam

Figure 4.15: Telephony Inferred Condition Parameters for the Address Category

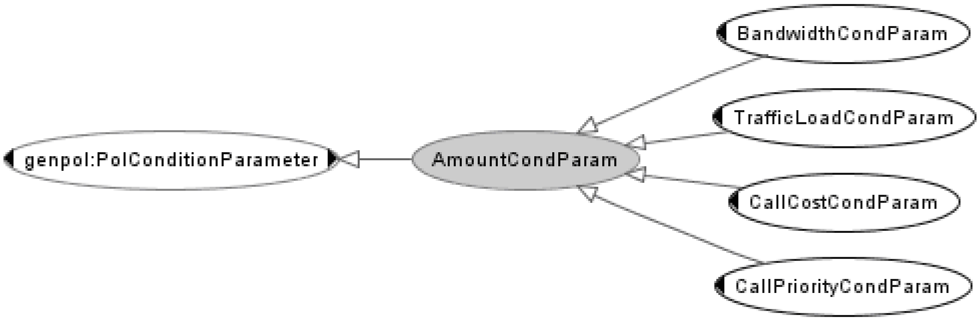


Figure 4.16: Telephony Inferred Condition Parameters for the Amount Category

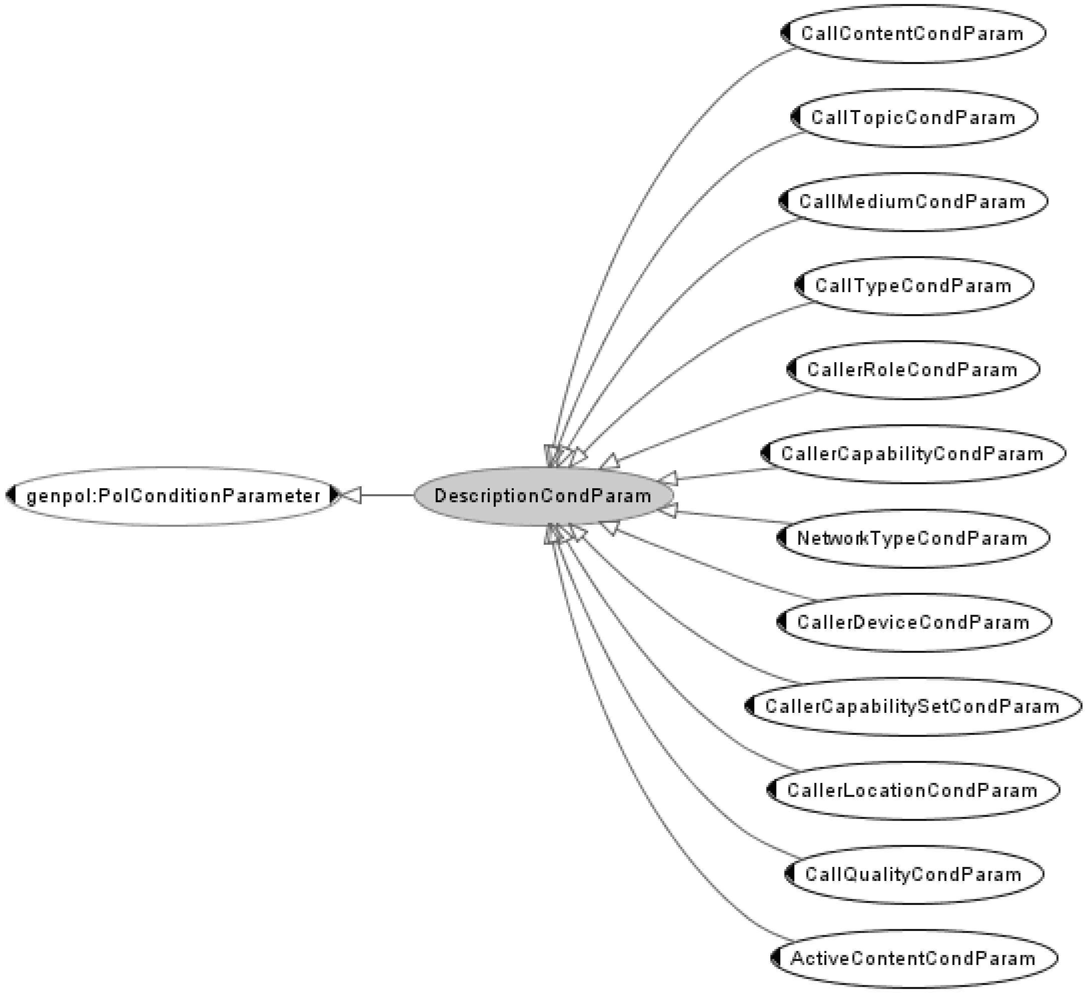


Figure 4.17: Telephony Inferred Condition Parameters for the Description Category

Telephony Actions

Specific actions for the telephony domain are defined as subclasses of *wizpol:NamedAction*, as shown in Figure 4.18. Domain-specific action categories include "Call" and "Update". Inferred subclasses of these categories are shown in Figure 4.19 and Figure 4.20 respectively. An action is defined to be a member of one category only – no action may be associated with multiple categories. Note that the "Call" trigger category (*CallTriggerEvent* class) is different to the "Call" action category (*CallAction* class). Both classes are defined under separate class hierarchies and are therefore distinctly identified.

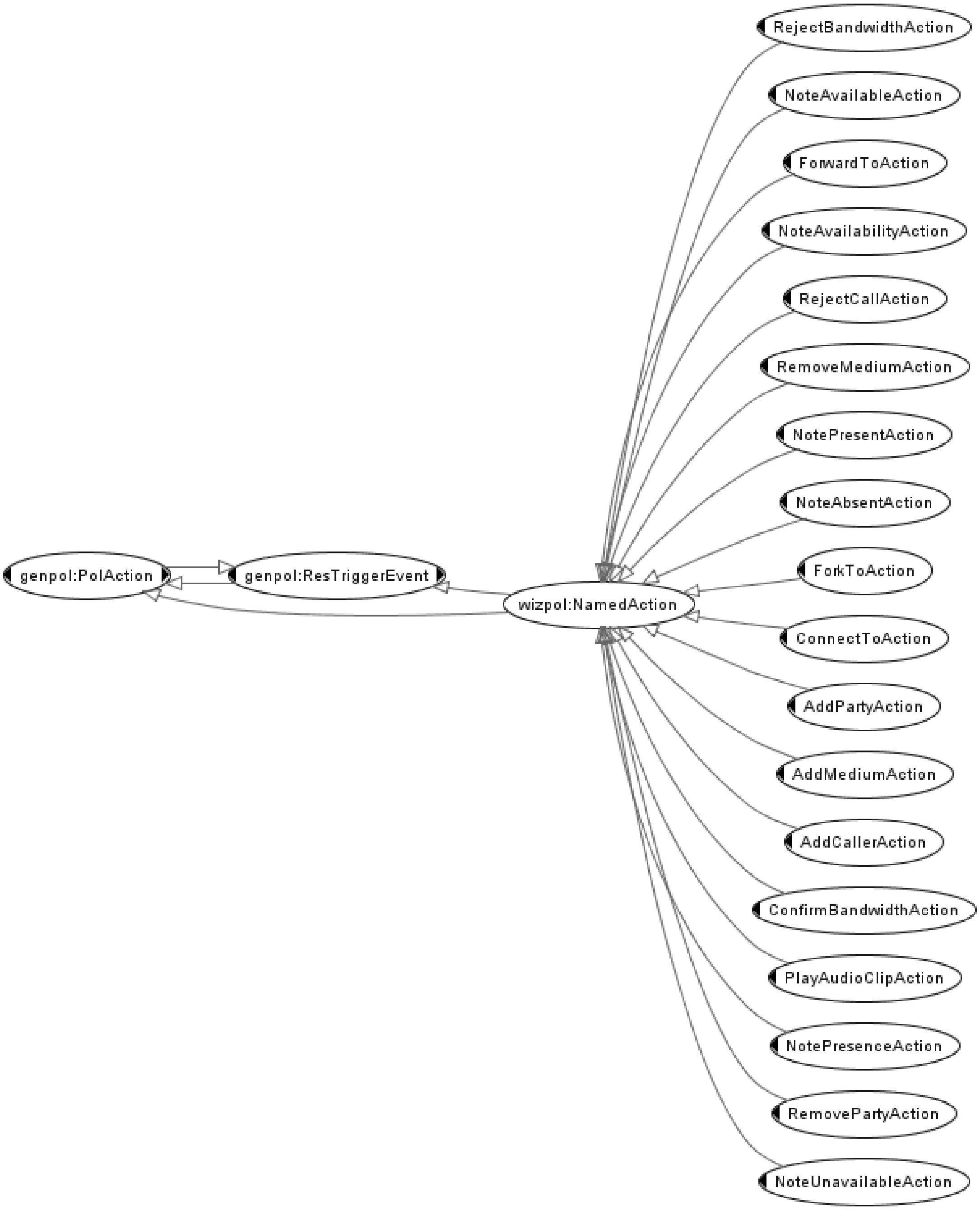


Figure 4.18: Telephony Named Actions

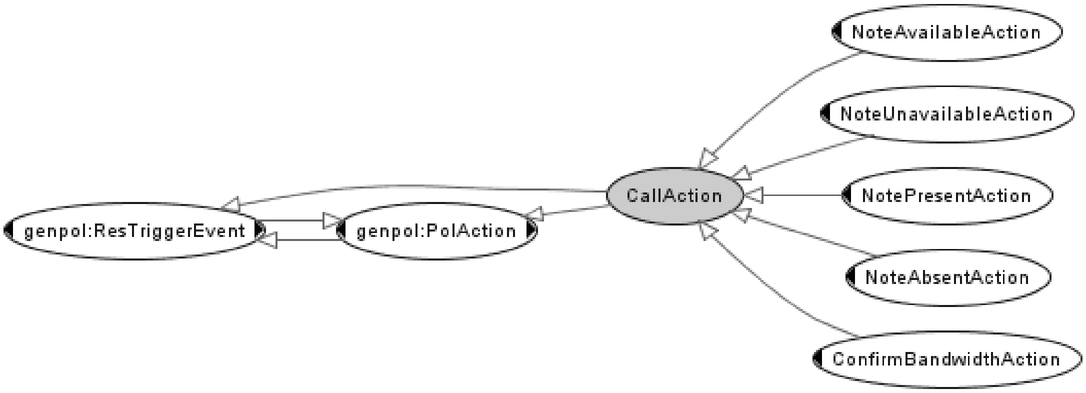


Figure 4.19: Telephony Inferred Actions for the Call Category

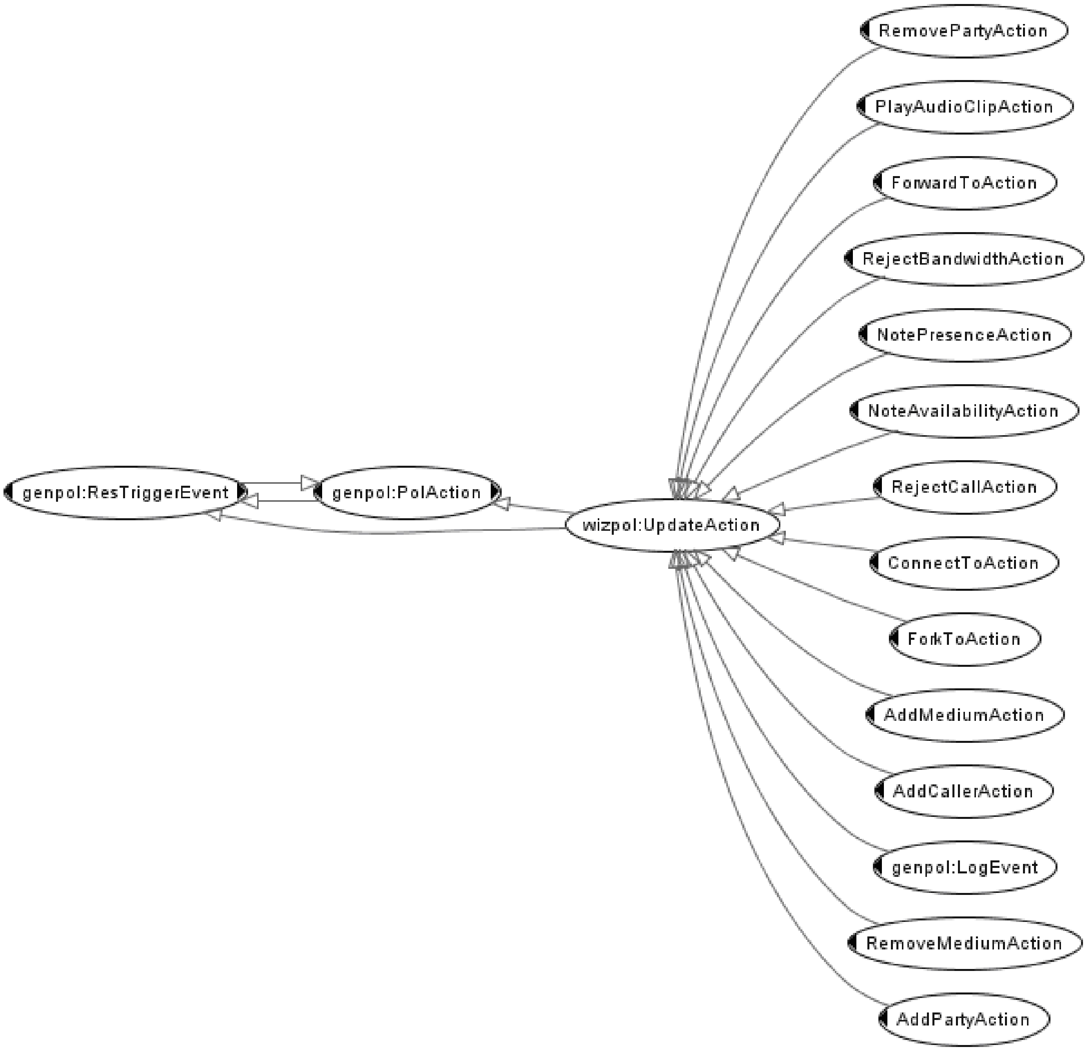


Figure 4.20: Telephony Inferred Actions for the Update Category

Telephony Unit Types

Within the telephony domain, there are three unit types typically linked with trigger, condition and action values as shown in Figure 4.21. This is an example of additional domain-specific knowledge defined in the ontology which does not model particular aspects of the policy language. The “Kbps”, “general” and “seconds” unit types are used to associate appropriate textual labels with displayed argument values within in the policy wizard. For example, the trigger *BandwidthConditionParam* has a restriction along the *hasUnitType* property linking it with the *KbpsUnitType* class.

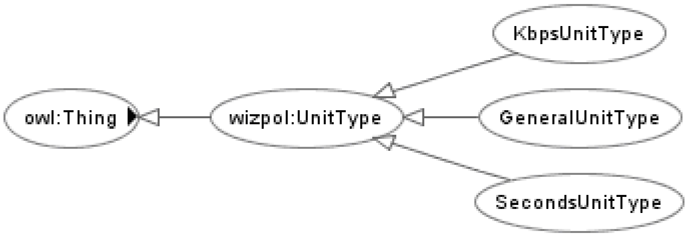


Figure 4.21: Telephony Unit Types

Telephony Domain-Specific Information

While the telephony ontology primarily defines domain-specific class and property extensions to the base ontologies of *GenPol* and *WizPol*, additional knowledge is included not directly for policy system usage. This knowledge relates to a higher level conceptual view of the form and operation of Internet telephony, as shown in Figure 4.22. Classes are defined to describe general aspects of a *Call* and *CallParty* that are not explicitly obvious from the policy language alone. A *Call* is associated with a *Caller* and *Callee*, and has various attributes including the type of call (such as "international", "emergency", "conference" or "standard"), the quality, priority, topic and cost. A *CallParty* can be a *Caller* or *Callee* and also has associated attributes including the role, device, location, and capabilities of the caller/callee.

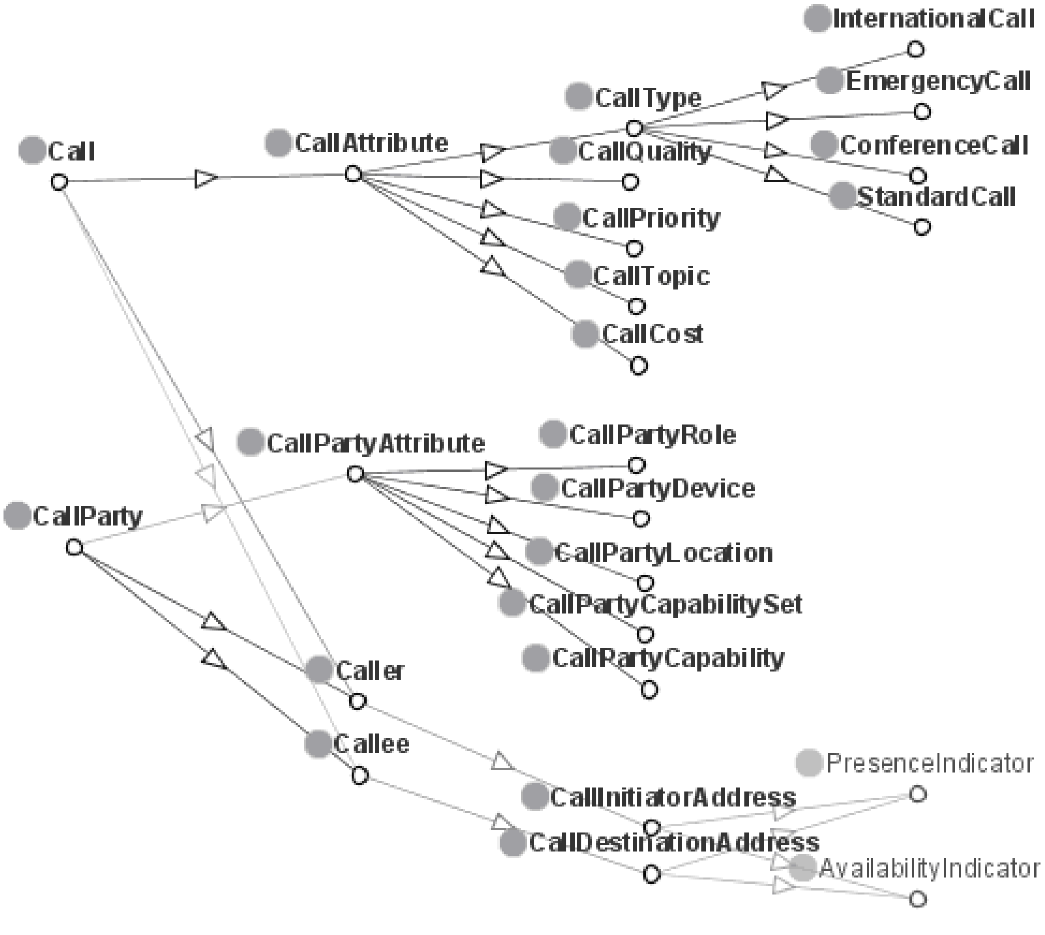


Figure 4.22: Telephony Additional Domain Information

4.5.2 Resolution Policy Extensions

Resolution policy extensions for the domain of Internet telephony are specific standard actions available in this environment. The ontology defines the resolution actions *ApplyCalleeResAction* and *ApplyCallerResAction* as subclasses of *genpol:SpecificResAction* as shown in Figure 4.23. These actions are available in addition to the set of generic resolution actions. *ApplyCalleeResAction* resolves policy conflict by applying the policy action associated with the callee, whereas *ApplyCallerResAction* applies the action associated with the caller.

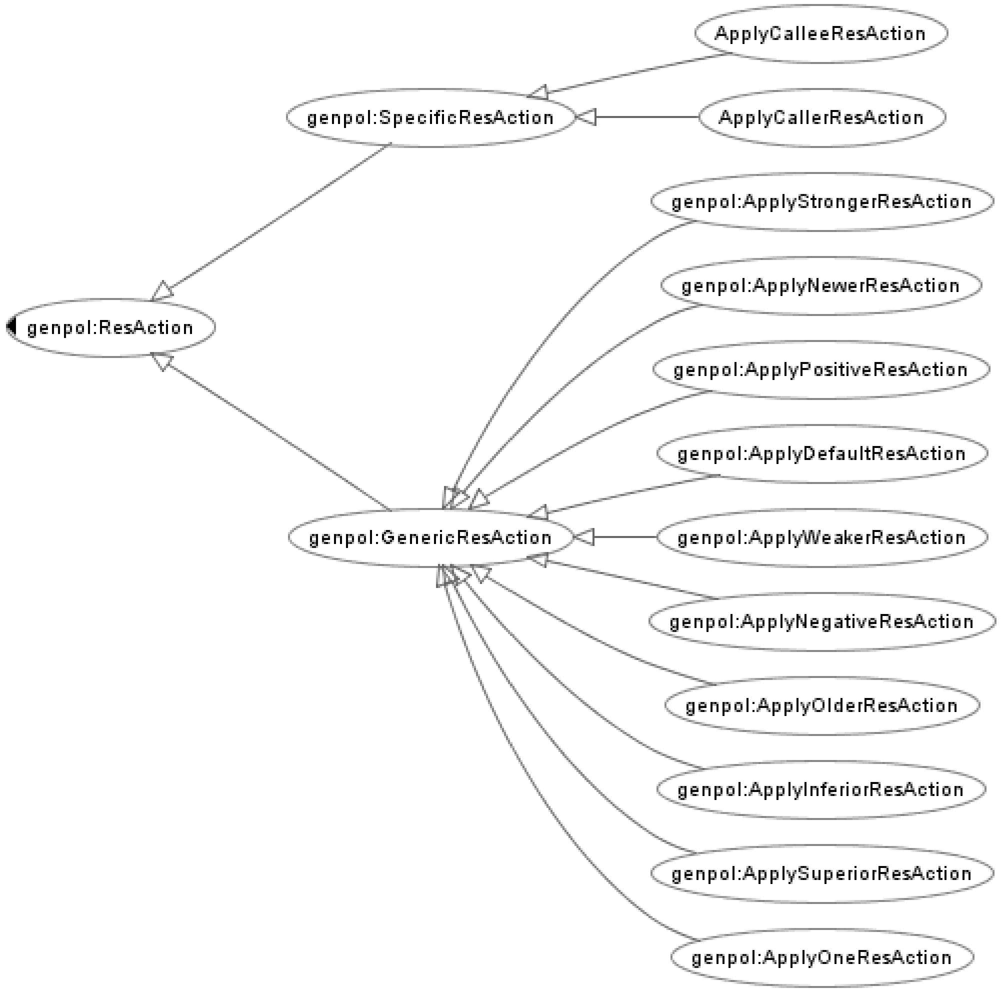

ApplyCalleeResAction
genpol:SpecificResAction
ApplyCallerResAction
genpol:ApplyStrongerResAction
genpol:ApplyNewerResAction
genpol:ResAction
genpol:ApplyPositiveResAction
genpol:ApplyDefaultResAction
genpol:GenericResAction
genpol:ApplyWeakerResAction
genpol:ApplyNegativeResAction
genpol:ApplyOlderResAction
genpol:ApplyInferiorResAction
genpol:ApplySuperiorResAction
genpol:ApplyOneResAction

Figure 4.23: Telephony Specific Resolution Policy Actions

4.6 Application 2: Ontology for Sensor Networks

This section describes the ontology classes and structure defined to model the Appel policy language specialisation for sensor networks/wind turbine management. This ontology extends the class structures defined in the core language ontologies (*GenPol* and *WizPol*) to define the triggering events, condition parameters, actions, arguments, variables and data types specific to this domain. Additionally, the ontology describes general, non-policy related knowledge. For convenience, the section and figure headings in this section are shortened to "sensor network", meaning "sensor network/wind turbine domain". The OWL ontology document may be accessed from [21].

Section 4.6.1 summarises the main domain extensions for standard sensor network policies. Section 4.6.2 summarises the main domain extensions for resolution policies.

4.6.1 Standard Policy Extensions

Sensor Network Class Categorisation

The sensor network ontology extends *wizpol:ClassCategorisation* to define domain-specific categories of triggers, condition parameters and actions. Classes defined under this structure are used to achieve category inference throughout the ontology. Each category (or grouping) usually represents a set of constructs that are logically similar or have the same number and form of parameter arguments. Trigger, condition and action categories are discussed in more detail within the following subsections.

Sensor Network Triggers

Specific triggers for sensor networks are defined as subclasses of *wizpol:NamedTriggerEvent* as shown in Figure 4.24.

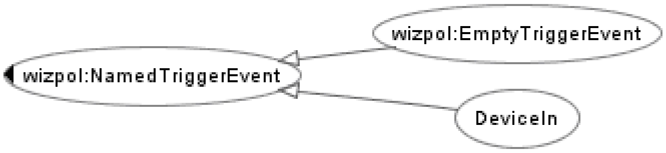


Figure 4.24: Sensor Network Named Triggers

For sensor network policies, there is a single external trigger called *DeviceIn*. The use of this trigger can be inferred as a subclass of a particular category (for policy wizard display purposes) or user level (to associate different sets of triggers with each “admin”, “expert”, “intermediate” and “novice” designation). Association between triggers and categories is achieved by defining properties on the *DeviceIn* trigger. For sensor networks, there is only one additional domain trigger category (as there is only one domain trigger) called *ExternalTrigger*. The *DeviceIn* trigger an inferred subclass of this category.

Sensor Network Conditions

The sensor network ontology directly extends the top-level *genpol:PolConditionParameter* class structure, defining subclasses representing the condition categories "Period" and "Qualifier". The inferred subclasses for each category are subsets of the full range of condition parameters defined under *wizpol:NamedConditionParameter*. Each condition parameter is related to exactly one category. The full list of defined sensor network condition parameters is shown in Figure 4.25. Inferred sets of condition parameters for each condition category are shown in Figure 4.26 and Figure 4.27 respectively.

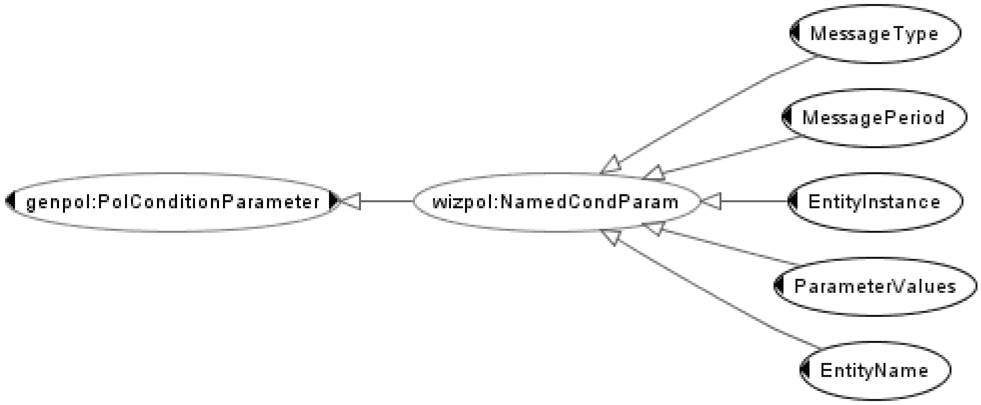


Figure 4.25: Sensor Network Named Condition Parameters

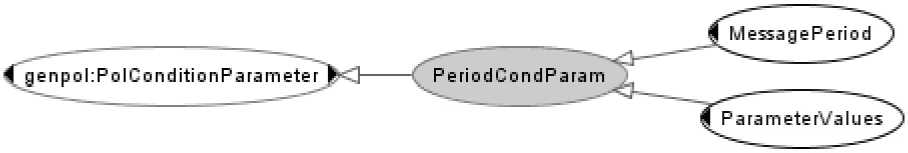


Figure 4.26: Sensor Network Inferred Condition Parameters for the Period Category

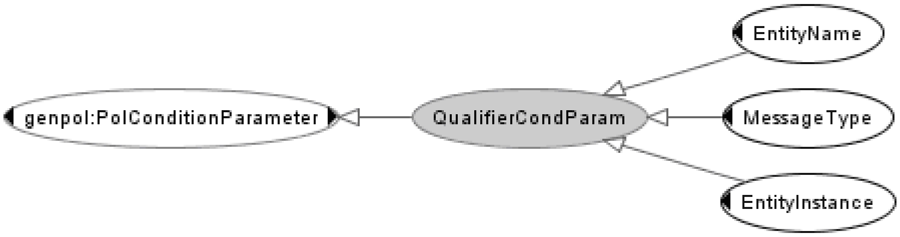


Figure 4.27: Sensor Network Inferred Condition Parameters for the Qualifier Category

Sensor Network Actions

Specific actions for the sensor network/wind farm domain are defined as subclasses of *wizpol:NamedAction*, as shown in Figure 4.28.

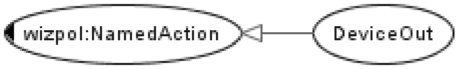


Figure 4.28: Sensor Network Named Actions

For sensor network policies, there is a single external action called *DeviceOut*. The use of this action can be inferred as a subclass of a particular category (for policy wizard display purposes) or user level (to associate different sets of actions with each “admin”, “expert”, “intermediate” and “novice” designation). Association between actions and categories is achieved by defining properties on the *DeviceOut* action. For sensor networks, there is only one domain action category (as there is only one domain action) called *ConfigureAction*. The *DeviceOut* action is an inferred subclass of this category.

Sensor Network Status Variables, Unit Types and Data Types

Unlike the previous application for Internet telephony, the sensor network ontology does not define domain-specific status variables, unit types or data types. For sensor networks, the domain does not use specific status variables. As the policy language for this domain was designed to be deliberately generic, the parameter arguments for the *DeviceIn* trigger and the *DeviceOut* action as treated as Strings with no assumptions made about their value in terms of data type (e.g. String, int, float, Date) or how to interpret the values and allocate them unit type labels.

Sensor Network Domain-Specific Information

The ontology models the Appel policy language for sensor networks which is intended for use in monitoring and managing a sensor network in general, and is not specialised for any particular deployment application or environment. As such it contains no specific knowledge of sensor network operation, such as node addressing (e.g. sensor IDs or network addresses), locations of individual sensors (co-ordinates, altitude or position in physical environment) or the nature of the system or environment the sensors are monitoring. This information is not related directly to the policy language or the policy wizard, but would be helpful in defining the application area. Class hierarchies can describe the generic and specific nature of components and how they are related. For example, for a particular wind farm site, concepts of "sensor node", "wind turbine" and "sensor instance" can be defined along with other physical components and their attributes. A sensor node might have attributes including a unique ID and address, a location, and sensing capabilities (e.g. supported rates of sampling and reporting, or available network interfaces). Wind turbines might describe their capabilities in terms of number of blades, maximum power output, generator and gearbox type, manufacturer, and operational status. This additional detail surrounding a domain implementation is useful for non-policy system tools such as an operator interface.

4.6.2 Resolution Policy Extensions

Resolution policy extensions for the domain of sensor network/wind turbine management are the specific standard policy actions for this domain – namely, *DeviceOut* (as there is the only standard policy action). There are no domain-specific resolution policy extensions for the domain of sensor network/wind turbine management.

4.7 Ontology Parsing and Integration

This section describes how OWL ontologies can be parsed and queried by an application using a generic custom-built tool called Poppet (Policy Ontology Parser Program – Extensible Translation). The Poppet approach will be explained, together with an example of its use.

4.7.1 Poppet Ontology Parser

Poppet (Policy Ontology Parser Program – Extensible Translation) is a generic framework designed to integrate an OWL ontology with an application. The framework is responsible for building a model of a given OWL ontology, and offers an API through which an external program may query the model to extract required information. An ontology is a structured document containing a description of a domain. The contents of an ontology become meaningful only on appropriate processing by an application. There are two aspects to this: parsing the ontology to build a logical semantic model, and querying this model to extract desired facts about the domain.

The Poppet system bridges the gap between an ontology and systems that need to use its information. Designed with maximum abstraction in mind, Poppet can be used to parse and model any given OWL ontology. The domain or data content is entirely irrelevant to Poppet, as is the particular dialect used (e.g. OWL Full, DL or Lite). This gives maximum scope for its reuse and extension within multiple contexts and by a wide range of applications. Access to the model is via an extensible API.

In summary, Poppet has the following key functions:

- accessing an OWL ontology document defined in a given location
- parsing a given OWL ontology and performing semantic inference via an external reasoner
- constructing and storing a model of a parsed OWL ontology
- providing an interface and methods (an API) to enable external applications to query the model.

The architecture of the Poppet system and an explanation of its operation are given in the following subsections.

4.7.2 Poppet Architecture

The complete Poppet system architecture is shown in Figure 4.29. An entirely Java-based system, Poppet utilises a number of pre-existing tools in its design – the Jena Parser and Pellet Reasoner are shown in the figure below. For a recap of ontology tools refer to section 4.1.4.

Poppet bases document parsing on the Jena [16] package. Jena is an established Java-based framework which offers extensive support for OWL ontology parsing and

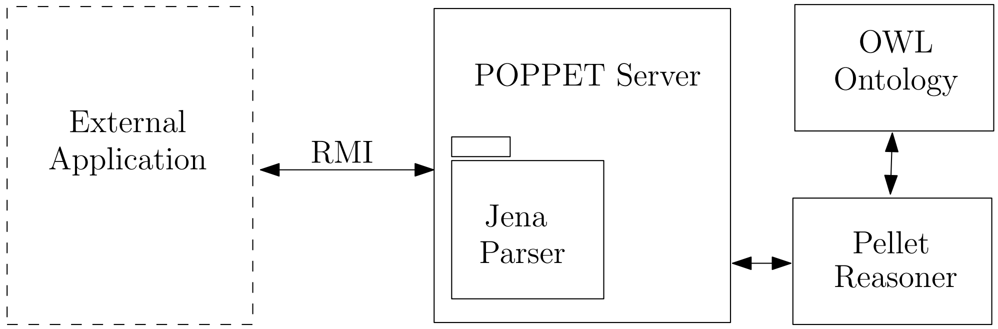


Figure 4.29: Poppet System Architecture

model building. Due to the strong OWL support, it was the most suitable choice of parser over other generic ontology parsing techniques and saved considerable development time. Poppet hides the implementation of the ontology parsing, allowing applications to interact using a generic interface without any knowledge of ontology model objects. Additionally, the use of Poppet combines multiple ontology-model queries into one method call, allowing applications to interact with the ontology model in a much simpler and more abstract fashion.

Jena 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 is Pellet [8]. Although Jena includes its own rule-based inference engine, it is not specifically designed for use in conjunction with OWL. Pellet was preferred as it is guaranteed to recognise and correctly interpret all OWL constructs. Pellet has also 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.

Poppet is run as a stand-alone server application. When invoked, it is passed the URL of an ontology to be read and modelled. Following successful ontology parsing, the constructed model is held in memory and may be queried by an external application.

Applications connect to Poppet using Java Remote Method Invocation (RMI). 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. RMI provides a secure method of object access using an established technique. It also allows the Poppet server to be run on a different host machine from the application accessing it, enabling greater flexibility. Such distribution in architecture ensures an application remains independent of the system used to access an ontology.

4.7.3 Poppet Usage Example

API methods offered by Poppet are called from a client application via RMI and return Strings, booleans or TreeSets (ordered sets of objects); the use of Jena-specific ontology objects is transparent. Parameter arguments are in the form of Strings representing either the name, label or URI of an ontology element (i.e. class or property). Client applications must know the format and details of the ontology they are querying, but are not concerned with how the parsing process is implemented.

As a small example, suppose there is an ontology called *Drink.owl*, located at the URI: *http://www.example.com/owl/Drink.owl*. The ontology defines the following four classes (the *rdfs:label* attribute text for each class is shown in parenthesis):

http://www.example.com/owl/Drink#HotDrink (hot_drink)
http://www.example.com/owl/Drink#Coffee (coffee)
http://www.example.com/owl/Drink#Tea (tea)
http://www.example.com/owl/Drink#Chocolate (chocolate)

The classes *Coffee*, *Tea* and *Chocolate* are subclasses of the class *HotDrink*. Client applications are aware the ontology contains a class called *HotDrink* but want to know the types of hot drink available. Using Poppet, the client application may request the subclasses of *HotDrink*. The client only knows the names of classes and not their full ontology URIs. The client also wishes the returned results to be the labels of classes (i.e. textual descriptions of the types of hot drink available) as opposed to ontology URIs.

The following steps describe the process of this request:

1. The client queries the ontology model using a method in the POPPET API that returns subclasses of a given class - in this example the class passed as a parameter to the method is "HotDrink".

2. POPPET then identifies the URI of the class *HotDrink*), and searches the ontology model to produce a list of its subclasses (either direct or inferred). The properties of each subclass are then considered: if a subclass has a label attribute, the text value is parsed. For example, the first subclass found might be *http://www.example.com/owl/Drink#Tea.* The label of this class is "tea" and this is returned to the client application – rather than the long class URI string.

3. The complete list of drink types (found subclass labels) are then passed back to the client. The client application may then process the returned value set (e.g. display results within a user interface form or menu, populate a database, or perform further queries on the ontology).

The next section shows how the POPPET system was used to parse and process domain-specific policy ontologies, and use this ontology knowledge within the ACCENT Policy Wizard.

4.8 An Ontology-Driven Policy Wizard

As policies are aimed at non-technical users of a system, the success of the policy-based approach relies on the presence of an efficient and user-friendly method of policy creation and manipulation. The ACCENT policy system includes a web-based policy wizard GUI to define and manage policies. However, in its original implementation the policy wizard contained a large proportion of hard-coded knowledge of the telephony domain. In order to generalise the wizard so it could generate a user interface specific to any custom domain, all details related to telephony were replaced by relevant method calls to retrieve data from an ontology via the POPPET system interface. Section 4.8.1 summarises technical extensions made to the original policy wizard. Section 4.8.2 evaluates the ontology-driven interface.

4.8.1 Policy Wizard Re-engineering

The core of the web-based policy wizard is implemented as a set of Java Server Pages (JSP), using additional Java code to process requests via the user interface. This code also handles communication with the policy server (to retrieve stored policies and policy-related data) and constructs the HTML page output tailored to the policy domain. Previously, the domain knowledge embedded in the wizard made use of hard-coded strings. For example, the content of drop-down menus in HTML form fields was hard-coded with telephony options, and text examples and tool-tips (text that appears when the mouse is held over certain screen elements) were directly embedded in HTML. The process of re-engineering the wizard involved removing all hard-coded domain-specific text and page formatting, and replacing this with appropriate method calls to retrieve data from an external source – in this case a domain ontology via Poppet. In addition, the wizard was extended to support the creation and uploading of resolution policies in addition to standard policies. Previously, resolution policies had to be defined in XML and uploaded manually to the policy store. Note that 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.

4.8.2 Policy Wizard 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 have also been preserved. Therefore, in terms of functionality, the new system does not sacrifice any previous features. This was tested by comparing the results of attempting to create and upload policies using the old and new wizard interfaces. The pages displayed and the domain content provided by each are identical as intended.

The new wizard has also been applied to sensor network/wind turbine management (using the ontology described in section 4.6), and successfully integrates policy constructs for this new domain. This has demonstrated that the new generalised policy wizard is indeed reusable for new domains.

With regard to system design and performance evaluation, the new wizard is sufficiently distributed and generic in that individual components may be altered in isolation. For example, it is entirely feasible to modify the Poppet ontology parser or reasoning engine, without affecting the policy wizard or the developed ontologies. Similarly, alterations to the policy wizard, or the web technologies it incorporates, can be made independant of Poppet or the ontologies. 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. Typically, the time taken to display a requested page is between one and three seconds, varying on the number of ontology queries required to populate each particular page. This delay is, however, implementation specific and dependent on the speed and state of network connectivity. Although greater than for the hard-coded version of the wizard, the delay is acceptable compared to the length of time it would take to generate a new hard-coded wizard for each new domain. In this way, speed is sacrificed in order to promote its reuse. However, as the process of creating policies is not time critical (only execution of policies is handled in real-time), the overhead does not compromise overall functionality of the policy system.

Potential solutions to performance issues include locating the Poppet system on the same physical (local) machine as the policy wizard (to avoid routing messages over a busy network) or using an alternative method of communication to RMI. However, these options might reduce the benefits of distributing the system. Additionally, future work might implement local caching of common ontology queries within the policy wizard to optimise the number of remote method calls made.

4.9 Conclusion

4.9.1 Chapter Summary

This chapter introduced the concept of an ontology and described how ontologies have been used to generalise the Accent policy wizard user interface so it may be used to create and edit policies for any domain specialisation of the Appel policy language.

Using a series of interrelated ontologies, the core and domain-specific elements of the Appel policy language, discussed in Chapter 3, were modelled using OWL (Web Ontology Language). Three types of ontologies were developed: *GenPol*, *WizPol*, and various domain ontologies. The *GenPol* ontology defines the core aspects of a policy in Appel, the *WizPol* ontology defines additional user interface aspects (including formatting and display information), and a domain-specific ontology extends the constructs of both *GenPol* and *WizPol* to define the policy language for a specific application area. The approach has been demonstrated through the creation of domain ontologies for Internet telephony and sensor network/wind turbine management.

The Accent policy wizard user interface was previously developed for the domain of telephony. Using the designed ontologies, the policy wizard was re-engineered to utilise ontology-defined language constructs as opposed to hard-coded domain information. Integration of ontology-defined knowledge with the policy wizard was achieved using a custom-built parser known as Poppet (Policy Ontology Parser Program – Extensible Translation). Poppet is a Java-based system which parses a given ontology. It provides a generic API through which an application can query the ontology and extract useful information. The policy wizard was altered to replace hard-coded telephony knowledge with ontology queries via the Poppet system interface.

The result of this work is a domain-independent user interface to manage policies within the Accent system, and a flexible and extensible ontology model of the policy language. The presented ontology framework has been extended for other policy-related system components and tools. A policy conflict filtering tool (discussed in Chapter 5) and a goal-based system (discussed in Chapter 6) both utilise and extend these core and domain ontologies.

4.9.2 Evaluation

With regard to the implemented ontology stack, there is scope for the reuse of both *GenPol* and *WizPol*, either as extensions of one another or independently. As the core policy language details and wizard extensions have been defined within separate ontologies, the policy language could potentially be specialised through direct extension of *GenPol* alone. This would be useful if the language were intended for use in another application or with a different user interface. For example, if the language were to be applied within another policy system, *GenPol* could be extended directly to specify a new language. Also, should the policy wizard be altered, *WizPol* could either be adjusted accordingly or a new ontology could be created (which imported *GenPol*) to describe the new interface. The implementation of domain ontologies for Internet telephony and sensor network/wind turbine management, and their integration with the policy wizard, have shown that ontologies are a flexible and successful method of defining domain policy knowledge in a structured way.

The ontology-driven approach was evaluated by comparing the appearance and functionality of the new policy wizard with that of the original hard-coded version. Use of the new wizard is identical to the original implementation, with the only noticeable difference to its users being a small overhead in the time taken to display each page.

Chapter 5

Policy Conflict Detection and Resolution

This chapter explains the concept of policy conflict and describes a new ontology-driven method for automated identification of potential conflicts between policies. A tool named Recap (Rigorously Evaluated Conflicts Among Policies) has been developed to support this approach. Its use is demonstrated for Internet telephony and sensor network/wind turbine management. The approach presented in this chapter enhances the existing conflict detection and resolution capabilities of the Accent policy system and associated Appel policy language, both described in Chapter 3. Some of the work reported in this chapter has been published in [51] and [53].

Section 5.1 introduces the concept of policy conflict, discusses related work in this field, and outlines the motivation for this work. Section 5.2 describes the new approach to automated policy conflict filtering, and section 5.3 outlines the Recap tool which implements the approach. Sections 5.5 and 5.6 demonstrate conflict filtering and the use of Recap for the domains of Internet telephony and sensor network/wind turbine management respectively. Section 5.7 evaluates this chapter.

5.1 Introduction and Background

The following subsections provide a background and overview of Feature Interaction (FI) and policy conflict. In particular, policy conflict handling within the Accent policy system is outlined, as this framework forms the basis for the approach and tool reported in this chapter. Existing work on policy conflict filtering (the focus of this chapter) is given together with an explanation of how the presented work relates.

5.1.1 Feature Interaction (FI) Overview

Feature Interaction (FI) is the term used to describe the effects of interacting services or components of a system. A “feature” is a component of a system that offers an additional function or service over core system functionality. For example, in telephony, ‘call forward’ and ‘call waiting’ are features introduced to complement the basic functions of making and receiving calls. Individual features may operate correctly in isolation, but can lead to undesirable system state when combined and executed with other features. FI is a potential issue in any system where multiple services may run concurrently.

A background to general FI issues and approaches can be found in Cameron *et al.* in [45]. Feature interaction in telephony has been explored extensively in previous work. This is mainly due to the fact telephony features are well defined and understood, making it easier to test, demonstrate and compare FI approaches as interactions are generally known. An early survey of FI issues in the telecommunications domain is given by Keck and Kuehn in [72]. A more recent review is given by Calder *et al.* in [44]. This summarises and compares major research trends in FI analysis including software engineering approaches, formal methods and online techniques. These techniques form a basis for FI work in general.

Many techniques have been developed to automate feature interaction detection at the system specification stage when features are programmed in software. Software engineering approaches to FI analysis involves defining a model of the complete development process, which aims to detect feature interactions early on during software service creation. This can, for example, identify when shared system variables or interfaces are used by a service and take necessary action in code to ensure shared components are managed without conflict. Examples approaches include use case driven analysis [84] [31], filtering-based analysis [74], requirements elicitation [68] and state transition analysis [75] [87].

Formal methods of FI analysis encompass a range of formal description, modelling and reasoning techniques (e.g. process algebras, finite and infinite state automata, (temporal) logic, and languages such as SDL, Promela, Z, and LOTOS). While software engineering approaches are geared towards the service design stage, formal methods are used to detect feature interactions offline when services are ready for integration with a system. Formal methods can be used to define abstract properties of a basic service and feature in logic, with interactions identifiable through inconsistencies within the logic [38], [65]. Alternatively, formal methods can be used to define the behaviour (or operation) of a feature including state or temporal aspects, with interactions determined using combinations of generic methods such as state simulation [33], [102] and deadlock detection [86]. Formal methods can validate expected interactions and also detect unpredicted interactions. Detecting new interactions is the aim of FI analysis, but the majority of FI results using formal approaches have only shown to verify interactions that are already known [44].

Online techniques are applied when a service is running on a network and consequently aim to detect and resolve interactions. Online approaches include feature manager based techniques (where a central or distributed entity is introduced solely to observe and control call processes) such as [43] and [66], and negotiation based approaches (where individual features communicate directly to detect and resolve their actions), such as the AI-based technique in [100].

Despite having no direct relation to policies, FI techniques have strongly influenced approaches designed to combat *policy conflict*.

5.1.2 Policy Conflict Overview

Policy conflict is the equivalent of the Feature Interaction (FI) problem in a policy-based environment – an executable policy may be thought of as a feature. A comparison between features and policies is provided by Dini *et al.* in [61]. Policies offer a wide range of customisable options for users to select. While this allows flexibility in high-level system management, policy conflict is a possible side-effect. Conflicts among policies occur between the actions of individual policies at run time when they are eligible for simultaneous execution. The result of such conflict is that one action may negatively affect another. For example, actions may attempt to set the value of the same variable, or may perform contradictory functions (e.g. connecting/disconnecting, adding/removing a system aspect, etc.). Such circumstances must be detected and resolved appropriately.

In general, domain polices may provide large sets of actions and, in turn, complex conflicts between them. Unless policies are reduced to a small collection of options, it is non-trivial to predict all possible behavioural outcomes which may arise from their simultaneous execution. In addition, policies may be created, managed and stored across multiple domains, by different service providers, and deployed over a range of networks. This makes it difficult to check the compatibility of policies and to maintain control over their generation.

Conflict handling within a policy-based environment is the process of detecting a potential conflict between two actions (prior to their execution) and attempting to resolve it. Policy conflict handling comprises three different aspects: analysis to identify and filter conflict-prone policies at the language level, definition of conflict detection mechanisms at the policy level, and definition of a conflict resolution strategy. Conflict detection can happen at three stages:

- When the policy language is being designed, conflicts between individual policy actions can be identified and filtered during the specification stage (offline).
- When policies are defined and introduced to the policy system (offline – after the language design stage but before policy is triggered).
- When deployed policies are triggered at run time in the policy system (online).

In the approach of this chapter, potential policy conflicts are detected and filtered offline at the policy language design stage (the initial stage of the three above).

Conflict detection and resolution techniques for policies in general are discussed in [56] and [81]. Closer to the work of this chapter, feature interaction for policies is discussed by Reiff-Marganiec and Turner in [89], which outlines a generic taxonomy for detecting and resolving conflicts between policies. This approach was implemented in the Accent policy system. The purpose of the work in this chapter was to enhance the existing conflict handling approach within the Accent system to compliment other improvements to the policy framework covered in this thesis. Existing conflict handling in Accent is now described.

5.1.3 Accent Policy Conflict Handling

This section describes how policy conflicts are handled by the Accent policy system (Advanced Component Control Enhancing Network Technologies [13]), with respect to the Appel policy language (Adaptable and Programmable Policy Environment and Language [91]). To achieve maximum extensibility within Accent, the Appel language was designed separately from the framework used for policy conflict detection and resolution. This approach was implemented in the original Accent system.

Policy Conflict Detection

Conflict handling within Accent is described in [98]. Accent allows for both static and dynamic conflict detection. Static detection can (in principle) be performed when a policy is defined and uploaded to the policy system, while dynamic detection occurs at run-time when policies are selected for execution. Although both methods are permitted, only dynamic detection is currently implemented. This focus was intentional since conflict detection and resolution must work in real-time.

On the detection of an event, the Accent policy server determines the subset of policies that are triggered by it. If there is more than one action within the triggered set, conflict detection and possible resolution are required. Conflict handling is specific to an application domain, therefore conflicts between actions are defined external to the policy server. Unless two actions within a domain are explicitly defined as conflicting, they are assumed to be compatible. While this allows the policy language to remain flexible, further mechanisms (either manual or automatic) must be used to identify and define conflicts.

The identification of conflicts involves the consideration of all pair-wise combinations of actions within a domain specialisation of the Appel policy language. A pair of actions that perform contradictory functions (e.g. adding and removing the same aspect, or setting and unsetting the same variable), may logically be identified as conflicting. However, this depends on the domain. Some conflicts are only identifiable through knowledge of the domain in operation. Conflict detection is defined to be commutative (e.g. if action A conflicts with action B, action B also conflicts with action A) and associative (e.g. the outcome of a conflict is the same regardless of the way in which actions are combined).

Policy Conflict Resolution

Conflicts in Accent are resolved by way of resolution policies which express when and how the system should respond to conflicts. Resolution policies are defined as an extension of the core Appel language, and therefore use the same syntax as standard policies, but with a different vocabulary. In particular, the triggers of a resolution policy are conflicting standard policy actions. The actions of a resolution policy provide a solution to the conflict. Such actions may be generic (select one of the conflicting actions based on some conditions) or specific (to apply domain-specific actions). Appel has a built-in notion of policy preference which allows a user to indicate how strongly they wish a policy to be applied. This allocates priorities to policies as one means of resolving conflicts. Chapter 3 describes the language for resolution policies and provides examples of their use. The outcome of the resolution process is a set of non-conflicting policy actions, which may then be executed.

Resolution policies in Accent are applied in a manner similar to the process for general domain-specific policies. However, the system does not support recursive conflict resolution due to its impact on execution time and to avoid infinite regress. Resolution policies provide considerable flexibility in that conflict handling is not embedded within the policy system – it is defined externally and can be domain-specific.

5.1.4 Motivation for Automated Conflict Filtering

In the original implementation of the Accent policy system, domain-specific policy actions were analysed manually by a system expert. This included reading XML schema descriptions of policy actions and applying domain knowledge in an attempt to identify actions which may conflict. Following this analysis, resolution policies to handle each conflicting action pair were generated manually as individual XML documents, and uploaded to the policy server. Not surprisingly, this was a time-consuming and potentially error-prone process, as conflicting action pairs may be missed during this analysis. In addition, there was no obvious way to view identified conflicts or to keep track of the resolution policies created to handle them.

This chapter presents a technique to automate the previously manual phase of analysing, identifying and filtering potential conflicts between policy actions at the domain specification stage (when the policy language is being designed). Existing online policy conflict detection and resolution techniques (previously implemented within the Accent system) are not affected. The conflict filtering approach is intended to operate offline, considering combinations of actions at the policy language design stage rather than combinations of policies at run time.

5.1.5 Existing Conflict Filtering Approaches and Tools

This section summarises existing approaches and tools to identify and filter conflicts between services or features of a system. In summary, the approach presented in this chapter filters pairs of policy actions using non-formal methods, at the point when a domain policy language is designed. Conflict is assumed when two actions affect or utilise a common aspect or resource of the managed system environment. The approach is implemented as a software tool which may be applied to any domain policy language in Appel.

Interaction Filtering Approaches

The notion of “interaction filtering” was initially presented by Kimbler in [74], to improve the efficiency of detecting interactions between services before they are deployed and integrated with a system. In the proposed methodology, filtering is intended to be the first step in a FI detection process, providing a rough and fast evaluation of conflicts. The filtered results serve to indicate clashing between service or feature types, which may be used to detect specific interactions at a later stage. The approach uses non-formal methods to analyse services including service descriptions and pre-/post-condition analysis. The approach is proposed and discussed with regard to FI in telecommunications systems only.

Although not directly related to policies, Kimbler's approach influenced the work of this chapter. The attraction of the approach is that it achieves significant results using simple (non-formal) methods. Although formal approaches to FI can produce a thorough analysis of all possible service combinations, the approach is extremely time consuming and is not proven to detect interactions on a large scale (i.e. large sets of services or actions across multiple networks and systems). One formal verification approach reporting this issue is [33]. Work presented here uses a form of analysis based on Kimblers static service descriptions.

In other related work using filtering-based approaches, Nakamura *et al.* [84] present a filtering technique based on Use Case Maps which is applied to telephony features. The filtering also takes place at the feature specification stage and first identifies FI-prone services, then analyses these to find common scenarios (patterns) of where FIs occur. Heuristics are proposed to solve such patterns, but the approach is of limited transferability as it is focused on solving only telephony FI issues. In [101], Wu and Schulzrinne use preconditions and postconditions to identify inconsistencies in features for a telephony-based scripting language called LESS (Language for End Systems Services). The FI detection approach uses action conflict tables in LESS combined with a tree-merging technique to detect and resolve potential conflicts. Although the rule-based language of LESS shares similarities with rule-based policies, this language is entirely telecommunications-specific.

Interaction Filtering Tools

There are several existing tools to support automated filtering of feature interactions. One is a prototype outlined by Keck [71] which is based on the Kimbler approach. The prototype detects interactions specific to a call environment and filters interactions among Intelligent Network (IN) services, using simple descriptions of the static structure of each service. Interactions are detected for groups of services used in particular call scenarios. The tool presented in this chapter is generic rather than geared toward IN services, but uses similar static descriptions to detect common effects of policy actions.

Formal approaches to interaction filtering have also been adopted, although previously noted issues surrounding scalability in domains other than telecommunications remain. FIX (Feature Interaction eXtractor [62]) is an example of a domain-independent approach, although only application to telephony is reported. FIX uses the model checker COSPAN to run consistency tests on specifications of system features or services. In a further stage, the tool allows the user to investigate the generated scenarios and decide on their accuracy. The tool in this chapter also offers a stage for users to inspect and alter filtered conflicts.

Policy Conflict Filtering

The approaches to interaction filtering discussed so far have been devised for FI analysis in general communications systems rather than a policy-based environment.

In [60], Davy *et al.* present a two-phase conflict analysis algorithm to detect and filter potential conflicts between policies in a policy continuum. The algorithm considers the relationships between pairs of policies (rather than actions as in the approach here) using a domain information model. It applies application-specific conflict patterns to determine if each pairing is potentially conflicting. Potential conflicts are flagged to system users who decide whether to ignore or resolve them. The approach is application-independent, but has not yet been implemented or demonstrated for real policy deployments.

Work in this chapter applies the concept of interaction filtering to policy actions, using the Appel policy language. In related work on the Appel policy language, Montangero *et al.* [83] use temporal logic to formalise the semantics of the language, leading to a formal basis for automated detection of conflicts. This approach is technically more complex and aims to conduct more elaborate conflict detection than the filtering methods presented here.

In other related work on Appel, Layouni *et al.* [77] present a method for discovering conflicts based on the pre/post-conditions of actions. This work is closest to the approach of this chapter as it also builds on Kimblers approach but using formal pre/post-condition analysis rather than using static descriptions. In their approach, pre/post-conditions of policy actions are defined, and then conflicts among these pre/post-conditions are defined. Using a first-order logic model-checking tool, semantically-based inferences are then drawn about the compatibility of policy actions based on the defined conflicts. The approach is said to be domain-independent although call control is the only application so far. As the approach uses formal methods of analysis, it is computationally expensive compared to the approach presented here.

Summary

In summary, the approach in this chapter differs from existing conflict and interaction filtering work in the following ways:

- The presented approach and tool are generic, rather than specific to telecommunications or any other particular domain.
- Formal specification of a managed system and its policies is not required. This makes the approach simpler to set up and more intuitive, i.e. relying only on domain knowledge rather than formal techniques. Domain experts, rather than formalists, can therefore define the information needed for conflict filtering.

5.2 Automatic Conflict Filtering Approach

The main limitation of policy conflict handling in the original Accent system implementation was that conflicts and their resolutions had to be defined manually. This section describes a new approach to automatically identifying policy conflicts, with the aim of enhancing the existing conflict handling capabilities within Accent, rather than replacing or modifying these. The foundational work presented here has been published in [53]. For clarity, the approach is outlined using examples from the domain of Internet telephony. Conflicts within this domain are described later in section 5.5.

5.2.1 Action Effects Ontology Support

Conflicts can occur between actions themselves or between the parameters of actions. For example, actions that add and remove the same aspect are potentially in conflict, such as the telephony-related actions *add_party* and *remove_party*. Other conflicts are far more subtle, and cannot easily be identified by naming alone.

Action parameters may use enumerated types (e.g. the telephony parameter *medium* has possible values *audio*, *video* and *whiteboard*). Where an action has an enumerated parameter type, conflicts between instances of the same action (at the language definition stage) are likely only if the parameters are the same. For example, *add_medium(audio)* could be considered to conflict with a second *add_medium(audio)*. However, if the second action wished to add *video* this would not be an obvious conflict. For this reason, actions with distinct values in an enumerated parameter should be treated as distinct actions.

Policy actions have specific effects on the execution environment. These effects might be technical or physical (e.g. bandwidth) or more abstract (e.g. privacy). Conflicts are likely to occur where two actions share a common effect. In addition, any action may potentially conflict with itself. For completeness in analysis, all (two-way) action pairs must be considered. Effect categories will differ depending on the language domain, and are defined by domain experts.

As discussed in section 4.4, a series of interrelated ontologies have been used to model the generic and domain-specific aspects of Appel: *GenPol*, *WizPol*, and a domain-specific ontology. It is convenient to define action effects in these ontologies to reuse and extend descriptions of existing domain knowledge. The ontologies are used to specify the effects of actions only and play no role in conflict detection or resolution. As conflict detection is not an integral part of Appel, the concept of an action effect is defined in the *WizPol* ontology (rather than *GenPol*). This allows conflict information to be specified outside the core language, while maintaining the advantage of further specialisation in domain-specific ontologies. Effect information is defined in *WizPol* through the *ActionEffect* class and the *hasActionEffect* property. The *ActionEffect* class is a superclass of all effect categories for both internal and domain-specific policy actions. Generic action effects are defined as subclasses of this class in *WizPol*, as shown in Figure 5.1. Domain-specific action effects are defined as subclasses within a separate domain ontology that imports *WizPol*. Each policy action is linked to the appropriate effect category class using the *hasActionEffect* property. This relates actions and effects, allowing a tool to infer overlapping actions.

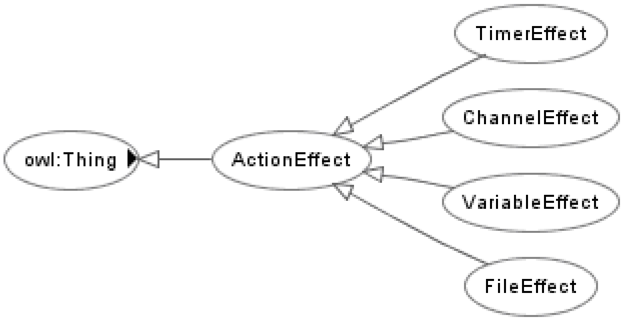


Figure 5.1: *WizPol* Generic Action Effect Categories

5.2.2 Conflict Detection Algorithm

Only pairs of actions need to be considered in the analysis of policy conflicts. Potential conflicts between actions are inferred from the ontology-defined effect categories through a two-stage algorithm:

1. Any two actions sharing at least one common effect are identified as potentially conflicting.

2. Actions with enumerated parameter types are analysed. Where two actions share the same parameter value then they potentially conflict, otherwise it is assumed that no conflict exists.

The total number of action pairs, including self-conflicts, is $\frac{n(n+1)}{2}$ where n is the number of possible policy actions. For example, the policy language for Internet telephony has 21 possible actions and therefore a total of 231 action pairs.

The ontologies allow a list of actions to be inferred for each effect category. If two actions are present in some category, they can be marked as potentially conflicting. All action pairs deemed to conflict in this way are then automatically reviewed with respect to their parameters. As explained earlier, actions with enumerated parameter types are considered in more detail. This increases the total number of action pairings as an action may be instantiated multiple times with different parameter values. For example, the telephony action *add_medium* with its parameter (one of *audio*, *video* or *whiteboard*) is equivalent to three distinct actions. This allows more accurate analysis of potential conflicts. Where actions might be treated as potentially conflicting based on a shared effect, this might not be the case when particular parameters are considered.

To explain this more concretely, some examples for *medium* are shown in Figure 5.2. An action may conflict with itself if there is a common parameter (e.g. both instances wish to add video), and may not conflict if the parameters are different (e.g. adding video and whiteboard respectively). Different actions with a common effect and the same parameter indicate potential conflict (e.g. attempting to add and remove audio simultaneously). Actions with a common effect and dissimilar parameters are assumed not to conflict (e.g. altering the medium by adding video and removing whiteboard).

Action1	**Action2**	**Conflict**
add_medium(audio)	*remove_medium(audio)*	✓
add_medium(audio)	*add_medium(video)*	×
add_medium(video)	*add_medium(video)*	✓
add_medium(video)	*remove_medium(whiteboard)*	×

Figure 5.2: Sample Telephony Conflicts with Action Parameters

5.3 The Recap Conflict Filtering Tool

This section describes the tool which implements the algorithm and approach to automated conflict analysis discussed in the previous section.

5.3.1 Automated Support for Conflict Filtering

The Recap tool (Rigorously Evaluated Conflicts Among Policies) has been developed to automate the algorithm outlined in the previous subsection. The tool is described in more detail in Appendix C.

Recap shows pairs of actions, why they conflict (their shared effects), and when the conflict was identified (either automatically or manually). Depending on the domain, the conflicts identified by Recap may or may not be complete and correct. Conversely, subtle conflicts that are not automatically flagged can be added manually by the user. To ensure no such subtle domain-specific conflicts are missed, the tool allows for human judgement in the detection process. Recap lists automatically inferred conflicts but will produce skeleton resolution policies to handle each identified conflict only following human confirmation that the conflicts are real.

Recap is mainly intended to analyse conflicts when a domain policy language is initially defined, using an ontology as the source of action effects. This initial analysis is saved to file and can subsequently be reloaded into the tool. This avoids the user and the tool from having to repeat a prior analysis, particularly if the user has manually modified the conflict list.

5.3.2 Recap Architecture

The Recap tool is Java-based and designed for stand-alone use. The architecture of the tool and interfaces with the previously described Accent policy system and ontologies are shown in Figure 5.3.

Recap communicates with a domain ontology using the Poppet system (Policy Ontology Parser Program – Extensible Translation) described in section 4.7.1. Generated resolution policies are uploaded to the Accent policy system.

5.3.3 Automated Support for Resolution

Recap can turn a list of conflicts into a set of outline Appel resolution policies that define the *detection* part of conflict handling. Generated policies define the conflicting triggers and parameter conditions, but resolution actions must be completed manually. The policies are automatically uploaded to the policy system, and the policy wizard is used to define the resolutions. The policy wizard is a graphical user interface to define and edit Appel policies and was discussed in section 4.8. In the original system implementation, resolution polices were created manually in XML and uploaded to the policy server. As part of this work to enhance conflict handling in the Accent system,

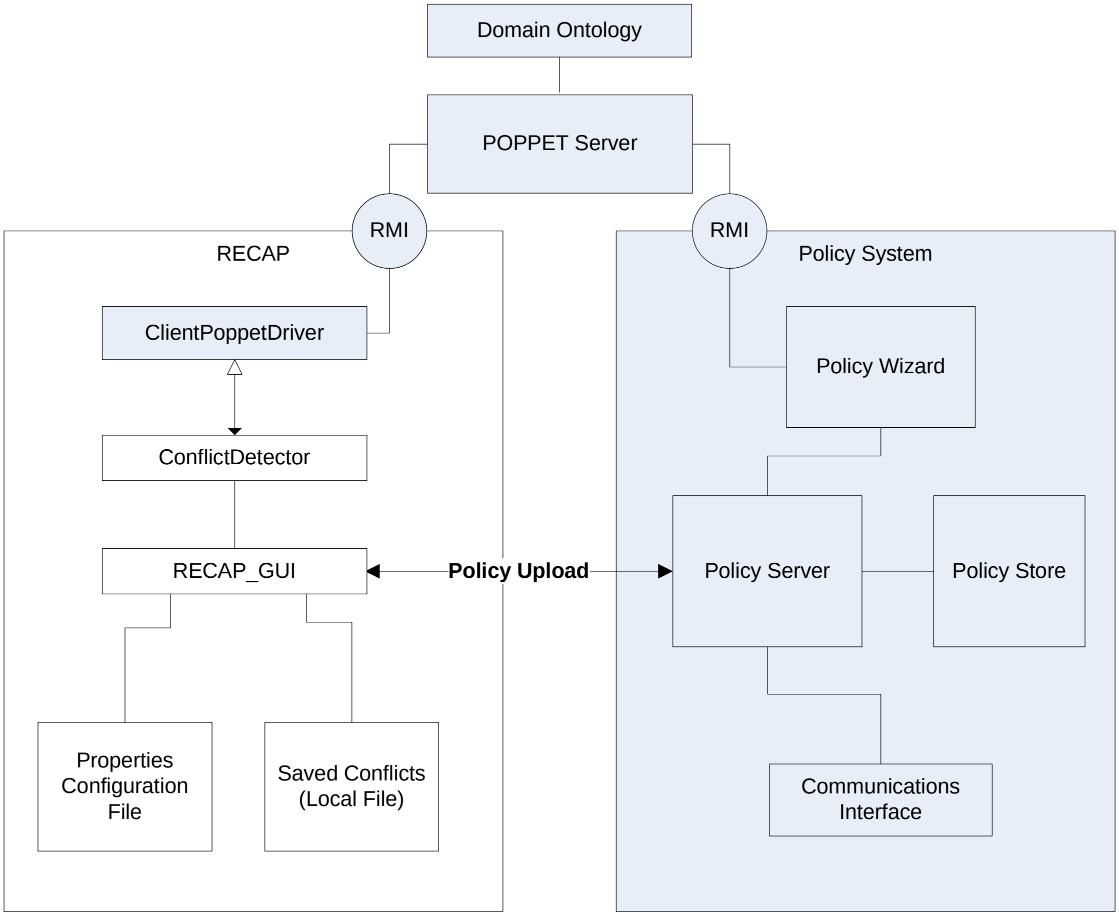


Figure 5.3: Recap Tool Architecture

the wizard was extended to support resolution policies in a similar manner to standard domain policies.

Resolution policies can be simple or complex, specific or generic, and dependent on many factors including the conflicting policies and their parameters. One or more actions may be required of a resolution. Taking another telephony example, suppose one party wishes to add video to the call with *add_medium(video)*, while the other party wishes to conference in a third person with *add_party(person)*. This might be considered a conflict since the third party would be able to view the call parties and their workplaces (affecting *privacy*). Using human judgement, it might be decided to allow video and the third party. However, someone (e.g. a manager) should be included in the call to oversee it.

In view of this complexity, Recap generates only outline resolution policies that specify default policy attributes, triggers corresponding to the conflicting actions, and default actions to resolve the conflict (e.g. choose the action with the higher priority). The outline resolutions are then uploaded and customised using the policy wizard. Resolution policy editing is handled outside Recap so that the tool may remain domain-independent and not be constrained to a particular resolution technique or policy language. In addition, resolution policies are edited through the same interface as regular domain policies.

On upload to the policy server, resolution policies are normally disabled (i.e. the "enabled" attribute is set to "false"). This ensures they are ignored by the policy server until they have been edited to include a specific resolution – avoiding accidental use of incomplete or inconsistent resolutions. Policies in general are distinguished by unique identifiers typically chosen by the user. Resolution policies automatically created by Recap have machine-generated (but human-readable) identifiers.

5.4 Generic Action Conflict Filtering Results

Internal (generic) policy actions affect the policy system itself, such as setting system properties or accessing system resources. Figure 5.4 shows the effects of internal (generic) policy actions applicable to any domain-specific specialisation of the Appel policy language (see Chapter 3 for an explanation of these). Effects for internal policy actions are distinct from those of domain actions, as internal and external actions should not normally conflict.

Action	**Effect**
log_event(arg1)	file
restart_timer(arg1)	timer
send_message(arg1,arg2)	channel
set_variable(arg1,arg2)	variable
start_timer(arg1,arg2)	timer
stop_timer(arg1)	timer
unset_variable(arg1)	variable

Figure 5.4: Generic Appel Policy Action Effects

The result of filtering internal conflicts for Appel using Recap is shown in Figure 5.5. Conflicts are numbered in the figure according to the underlying effect. As an example, actions *start_timer* and *stop_timer* are in conflict because they both have a *timer* effect as indicated at their intersection. The results of this analysis by Recap was deemed correct (following examination by a domain expert) and no alterations were necessary.

log_event	**restart_timer**	**send_message**	**set_variable**	**start_timer**	**stop_timer**	**unset_variable**	**Action1/Action2**
2							**log_event**
	3			3	3		**restart_timer**
		1					**send_message**
			4			4	**set_variable**
				3	3		**start_timer**
					3		**stop_timer**
						4	**unset_variable**

Conflict: 1 channel, 2 file, 3 timer, 4 variable

Figure 5.5: Internal Conflicts Identified by Recap for Appel

5.5 Application 1: Policy Conflicts for Internet Telephony

This section provides an overview of typical conflicts within the domain of Internet telephony and discusses their resolution. The results of analysing the policy language using Recap are also presented.

5.5.1 Telephony Conflicts Overview

The Appel policy language specialisation for the domain of Internet telephony was previously described in section 3.3. The language defines a large set of distinct policy actions, few of which have enumerated parameter values. Conflicts in this domain therefore largely result from clashes between different actions, and less so between action and parameter instances.

Example telephony policies were given in section 3.3.2. In general, policies are triggered when a call is initiated, received, modified, disconnected or goes unanswered. Policy actions may include forwarding, forking or rejecting a call, adding a caller to an existing call (e.g. conference calling), confirming/rejecting a bandwidth request, playing a customised sound clip, and adding/removing media such as video or a whiteboard. There are two enumerated parameters: *medium* and *method*. The *add_medium* and *remove_medium* actions both specify the *medium* as a sole parameter, values of which may be either *audio*, *video* or *whiteboard*. The *add_caller* action takes a *method* as its only parameter, the values of which may be either *release*, *monitor*, *wait*, *conference* or *hold*.

Virtually any telephony action may conflict with itself if its arguments are the same and the preferences of the caller and callee are opposite. Resolution policy examples for telephony conflicts were given in section 3.3.3. Generic resolutions include selecting the policy action based on time (date modified), priority (preference allocated) or domain (the hierarchic level to which the policy applies). In addition, telephony policies may choose to select and apply the action associated with the caller or callee. Any standard telephony policy action may also be used (e.g. disconnecting, forwarding, forking or rejecting a call, or playing a customised sound clip).

5.5.2 Telephony Action Effects

Figure 5.6 shows the effects of telephony actions. These effect categories were derived from analysis of the language by a domain expert.

Action	**Effect**
add_caller(conference)	party, privacy, bandwidth
add_caller(hold)	party, privacy, bandwidth
add_caller(monitor)	party, privacy, bandwidth
add_caller(release)	party, privacy
add_caller(wait)	party, privacy
add_medium(audio)	medium, privacy, bandwidth
add_medium(video)	medium, privacy, bandwidth
add_medium(whiteboard)	medium, privacy, bandwidth
add_party	party, privacy, bandwidth
confirm_bandwidth	bandwidth
connect_to	route
fork_to	route
forward_to	route
note_availability	availability
note_presence	presence
play_clip	medium
reject_call	call
reject_bandwidth	bandwidth
remove_medium(audio)	medium, privacy, bandwidth
remove_medium(video)	medium, privacy, bandwidth
remove_medium(whiteboard)	medium, privacy, bandwidth
remove_party	party, privacy, bandwidth

Figure 5.6: Telephony Action Effects

Each category is defined within the ontology for Internet telephony and linked with the associated actions. An example of this is shown in Figure 5.7. The action *AddMediumAction* is linked with *MediumEffect*, *PrivacyEffect* and *BandwidthEffect*. This is because adding media to a call may affect both *medium* (as audio or video may be added to the call), *privacy* (as the introduction of, say, video may conflict with the callee who does not wish to have an image broadcast) and also *bandwidth* (as adding media such as video or a whiteboard, will increase the bandwidth required for the call). Together with *AddMediumAction*, the actions *PlayAudioClipAction* and *RemoveMediumAction* also affect the *medium* category. From this it is inferred that any pairing of these three actions could potentially result in conflict.

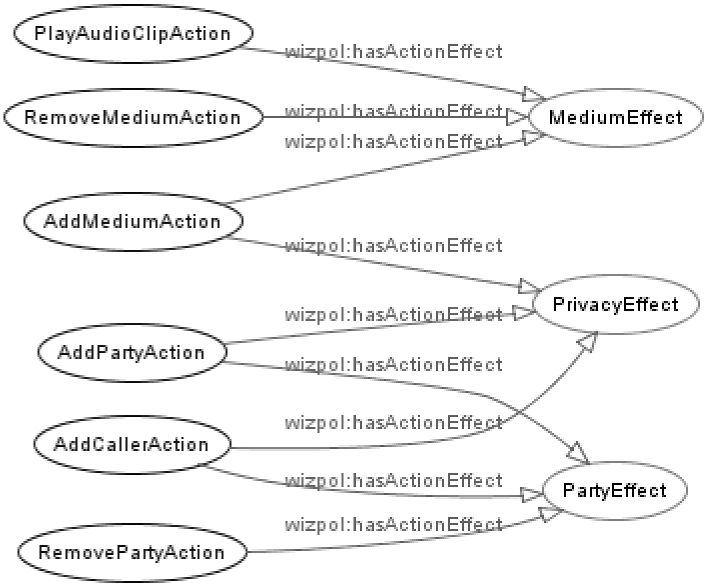


Figure 5.7: Example Telephony Actions and Effect Categories

5.5.3 Telephony Conflict Filtering Results

For convenience, domain-specific telephony actions are described here separate from internal actions, though in practice they are combined by Recap.

The total number of action pairs, including self-conflicts, is $\frac{n(n+1)}{2}$ where n is the number of possible policy actions. The policy language for call control has 21 possible actions and therefore a total of 231 action pairs. Telephony actions deemed conflicting by Recap are shown in Figure 5.8. Conflicts are numbered in the figure according to the underlying effect. As an example, the actions *fork_to* and *forward_to* potentially conflict as they both affect the *route*.

In the tool, actions with enumerated parameter types are displayed and compared as separate actions. The full pair-wise list of action/parameter combinations is too large to include in table form here, but will be summarised. For telephony, all the enumerated parameters for the actions *add/remove_medium* and *add_caller* have identical effects. Recap therefore identifies each action/parameter pairing as being in conflict with itself, and every other enumeration for the same action. For example, *add_caller(hold)* conflicts with itself, *add_caller(conference)*, *add_caller(monitor)*, *add_caller(release)* and *add_caller(wait))*. Examples of enumerated parameter conflicts for *add_medium* are shown previously in Figure 5.2. On human inspection, some conflicts may not be a real problem depending on the underlying implementation (e.g. adding or removing the same media twice may not be a problem).

Detailed study by a domain expert confirmed that all detected conflicts but one are real, and that no conflicts have been missed. Some conflicts are non-obvious (e.g. *add_caller* and *add_medium*). There is a possible problem in that *confirm_bandwidth* is indicated to conflict with itself due to a shared *bandwidth* effect. This could be a potential error in that it might lead to bandwidth being allocated twice. However, in the Accent system it is harmless to confirm bandwidth more than once. Without human guidance, this action pair would be flagged as a conflict. It should be noted that the *bandwidth* effect is still required as it correctly identifies the conflict between *confirm_bandwidth* and *reject_bandwidth*.

add_caller	**add_medium**	**add_party**	**confirm_bandwidth**	**connect_to**	**fork_to**	**forward_to**	**note_availability**	**note_presence**	**play_clip**	**reject_bandwidth**	**reject_call**	**remove_medium**	**remove_party**	**Action1/Action2**
2,5,7	2,7	2,5,7										2,7	2,5,7	**add_caller**
	2,4,7	2,7							4			2,4,7	2,7	**add_medium**
		2,5,7										2,7	2,5,7	**add_party**
			2							2				**confirm_bandwidth**
				8	8	8								**connect_to**
					8	8								**connect_to**
						8								**forward_to**
							1							**note_availability**
								6						**note_presence**
									4			4		**play_clip**
										2				**reject_bandwidth**
											3			**reject_call**
												2,4,7	2,7	**remove_medium**
													2,5,7	**remove_party**

Conflict: 1 availability, 2 bandwidth, 3 call, 4 medium,
5 party, 6 presence, 7 privacy, 8 route

Figure 5.8: Telephony Conflicts Identified by Recap

5.6 Application 2: Policy Conflicts for Sensor Networks

This section provides an overview of typical conflicts within the domain of sensor network/wind turbine management and discusses their resolution. The results of analysing the policy language using Recap are also presented.

5.6.1 Sensor Network Conflicts Overview

The Appel policy language specialisation for sensor networks was previously described in section 3.4. In general, sensor network policies have a single external trigger (*device_in*) or action (*device_out*) for communicating with an external entity (e.g. a sensor node or operator console). This trigger and action carries five parameters: the *message_type* (defines the nature of the trigger or action), the *entity_name* (identifies the external component), an *entity_instance* (a unique identifier for the entity), the *message_period* (defines duration reported in a trigger or the time at which an action should be performed), and the *parameter_values* (a string of values that qualify the trigger or action). The *message_type* is mandatory, while the others are optional. Example sensor network policies that use this language were given in section 3.4.2.

As there is only one external action, conflicts can only occur between of a pair of *device_out* actions. For example, one policy may wish to set the anemometer reporting interval on sensor node instance S32 to 10 minutes, while another policy wishes to set this to 20 minutes. Similar conflicts can arise when setting sampling frequencies or other sensor parameters. For sensor node parameter conflicts, the *parameter_values* string within a *device_out* action changes format depending on the *message_type* string. This allows detection of conflicting values. The values (or sub-values) in *parameter_values* can be compared in a resolution policy. Example resolution policies to handle this type of conflict were given previously in section 3.4.3.

Generic resolutions select one of the conflicting actions based on some general attribute. For example, choosing the earlier defined policy or the action that applies to a higher domain (e.g. all anemometers rather than a particular one). Specific resolutions are actions that a sensor network can perform. For example, an operator might be alerted to the conflict and asked to take action. If the conflict is deemed to be non-critical (say, backing up logs), it might be resolved by delaying one of the conflicting actions.

Further to conflicts detectable at the language level, the nature of sensor networks brings additional complexities which might cause conflict. For example, a policy action might conflict with a prior action and not just a concurrent action (e.g. a policy executed at 12:01pm switches off a sensor, while an action two minutes later requests upload of the sensor log). A solution to this problem exists in the language. Appel supports state history functions which can be used in policies as a way of resolving potential conflicts without an explicit resolution (e.g. the policy requesting a sensor log checks the action history to determine if the sensor was switched off recently). This can allow conflicts to be detected between actions and states.

A further kind of conflict may arise within the sensor network itself. Sensor nodes have limited memory, bandwidth, electrical power and processing capacity. Such constraints affect their ability to perform certain actions simultaneously. Examples include:

- Limitation on shared bandwidth from a sensor node to the policy system (e.g. a request for the upload of log data may conflict with a request for the most recent camera image data on cloud cover).

- Limitation on a shared sensor node processor (e.g. a request for log data to be compressed may conflict with a request to reset all sensor sampling frequencies to their default values).

- Limitation on sensor node battery power (e.g. setting the sampling interval for each sensor to 60 seconds while simultaneously requesting threshold values be checked every 90 seconds).

Superficially, actions in the scenarios above are conflict-free, but due to underlying resource limitations (which may or may not be known to policy creators), they cannot be carried out concurrently. While this type of indirect conflict is out of the scope of the policy system, it is desirable to account for external resource restrictions where possible when defining policies. One possible solution (not yet implemented) would be to model resource information in the domain ontology. Each resource could be described along with its constraints. For example, limitations could include data (bandwidth, processing), parameterisation (memory, processing), and computation (processing, memory, power). This information could be used by the policy system to link with possible action/parameter combinations and to detect situations which may cause resource conflicts.

5.6.2 Sensor Network Action Effects

As the only action to conflict is *device_ out*, from the perspective of the policy language the only possible conflict is this action conflicting with itself. The action will have different effects depending on the specific values given to its parameters. As this action is parameterised, it has no defined set of enumerated values at the language specification level. Therefore, analysis is limited to detecting conflicts between the combinations of parameters.

For this analysis, each individual parameter of the *device_ out* action is noted to have an effect on the sensor network environment, with the exception of the *message_ type*. As this parameter is mandatory in all *device_ out* actions, if it were assigned an effect, the result would be that all combinations of the action conflicted. This is of little help in the analysis. For this reason, the *message_ type* is omitted from effects. This will become clearer on explanation of the results. The remaining parameters have the following effects:

entity_ name = group_configuration, single_configuration
entity_ instance = single_configuration
message_ period = scheduling
parameter_ values = configuration

Specifying an *entity_ name* indicates a potential group of entities (e.g. all anemometers), but this could be a single entity (e.g. the single operator console), so this has two potential effects. The use of an *entity_ instance* identifies a single object. Specifying the optional *message_ period* will affect *scheduling*. The *parameter_ values* could affect a number of aspects depending on the number of sub-values and how they are interpreted. However, at the language specification stage, the only known effect is *configuration*.

To assign these effects, each specific combination of parameters of the *device_ out* action is considered as a separate action. These combinations and their effects are shown in Figure 5.9.

Action	**Effect**
device_ out(arg1,arg2,arg3,arg4,arg5)	group/single_ configuration, scheduling, configuration
device_ out(arg1,arg2,arg3,arg4)	group/single_ configuration, scheduling
device_ out(arg1,arg2,arg3)	group/single_ configuration
device_ out(arg1,arg2)	group/single_ configuration
device_ out(arg1)	
device_ out(arg1,,arg3,arg4,arg5)	single_ configuration, scheduling, configuration
device_ out(arg1,,arg3,,arg5)	single_ configuration, configuration
device_ out(arg1,,arg3,arg4)	single_ configuration, scheduling
device_ out(arg1,,arg3)	single_ configuration
device_ out(arg1,,,arg4,arg5)	scheduling, configuration
device_ out(arg1,,,arg4)	scheduling
device_ out(arg1,,,,arg5)	configuration
device_ out(arg1,arg2,,arg4,arg5)	group/single_ configuration, scheduling, configuration
device_ out(arg1,arg2,,arg4)	group/single_ configuration, scheduling
device_ out(arg1,arg2,,,arg5)	group/single_ configuration, configuration
device_ out(arg1,arg2,arg3,,arg5)	group/single_ configuration, configuration

Arguments: *arg1* message_ type, *arg2* entity_ name, *arg3* entity_ instance, *arg4* message_ period, *arg5* parameter_ values

Figure 5.9: Sensor Network Action Effects

5.6.3 Sensor Network Conflict Filtering Results

For convenience, domain-specific actions are described here separate from Internal actions, though in practice they are combined by Recap. The total number of action/parameter pairs, including self-conflicts, is $\frac{n(n+1)}{2}$ where n is the number of possible policy actions. The policy language for sensor networks has 16 action/parameter combinations and therefore a total of 136 action pairs. The results for sensor network policy conflict analysis are shown in Figure 5.10. Note that the action parameters are abbreviated within the table for clarity (e.g. *arg1* is just 1).

From the figure, all parameter combinations conflict with themselves (i.e. all actions that specify an *entity_ instance* (no. 3) potentially conflict with any other action specifying an *entity_ instance*). As the first parameter argument (1, *message_ type*) was not

Action1/Action2	**device_out(1,2,3,4,5)**	**device_out(1,2,3,4)**	**device_out(1,2,3)**	**device_out(1,2)**	**device_out(1)**	**device_out(1,,3,4,5)**	**device_out(1,,3,,5)**	**device_out(1,,3,4)**	**device_out(1,,3)**	**device_out(1,,,4,5)**	**device_out(1,,,4)**	**device_out(1,,,,5)**	**device_out(1,2,,4,5)**	**device_out(1,2,,4)**	**device_out(1,2,,,5)**	**device_out(1,2,3,,5)**
device_out(1,2,3,4,5)	2345															
device_out(1,2,3,4)	234	234														
device_out(1,2,3)	23	23	23													
device_out(1,2)	23	23	23	23												
device_out(1)																
device_out(1,,3,4,5)	345	34	3	3		345										
device_out(1,,3,,5)	35	3	3	3		35	35									
device_out(1,,3,4)	34	34	3	3		34	3	34								
device_out(1,,3)	3	3	3	3		3	3	3	3							
device_out(1,,,4,5)	45	4				45	5	4		45						
device_out(1,,,4)	4	4				4		4		4	4					
device_out(1,,,,5)	5					5	5			5		5				
device_out(1,2,,4,5)	2345	234	23	23		345	35	34	3	45	4	5	2345			
device_out(1,2,,4)	234	234	23	23		34	3	34	3	4	4		234	234		
device_out(1,2,,,5)	235	23	23	23		35	35	3	3	5		5	235	23	235	
device_out(1,2,3,,5)	235	23	23	23		35	35	3	3	5		5	235	23	235	235

Conflict: 2 group_configuration, 3 single_configuration, 4 scheduling, 5 configuration

Figure 5.10: Sensor Network Conflicts Identified by RECAP

defined to have an explicit effect, the action *device_ out(1)* conflicts with no other action. Depending on the types of messages that may be sent in a particular sensor network implementation of the language, this action may be in conflict. For example, conflicting situations might arise between contrasting commands such as starting/stopping an alarm. On inspection in the tool, the pairing *device_ out(1)/device_ out(1)* would therefore be marked as a conflict.

The detected conflicts do however provide a helpful series of generic resolution templates which can then be customised. The resolution can distinguish clashes between similar and different parameter values. The nature of this will depend on the particular combinations of values used in each action, and could be interdependent such as the interpretation of the *parameter_ values* depending on the *message_ type* and/or the particular *entity_ name*.

To conclude, the flexibility of Appel is beneficial towards its reuse, but renders conflict detection difficult without explicit knowledge of the parameter values in use.

5.7 Conclusion

5.7.1 Chapter Summary

This chapter presented a new automated approach to identifying and filtering potential conflicts between policy actions at the language specification stage, in relation to the Accent policy system and Appel policy language discussed in Chapter 3. Policy conflict occurs at run-time when multiple policy actions become eligible for execution simultaneously. Unless conflicting actions are detected and resolved by the policy system, their execution may leave the managed system in an unpredictable or undesirable state. Previous conflict analysis and resolution generation mechanisms within the Accent system were manual, which was time-consuming and error prone.

In the presented approach, the domain ontologies described in Chapter 4 are extended to define the effects of each action on the managed system environment. Actions with common effects are potentially conflicting. The Recap tool has been implemented to support this approach. The tool provides a graphical user interface that displays all pair-wise combinations of policy actions for a given domain, and marks those potentially in conflict. Due to the complexity of policy interaction, the tool allows for manual filtering of this list, so a human user may view and confirm inferred conflicts, and flag any additional subtle domain-specific conflicts. As a final step, RECAP automatically generates skeleton resolution policies to handle these conflicts. Resolution policies may in turn be edited using the policy wizard – also altered as part of this work to support resolution policies in a similar manner to standard policies. Conflicts in the domains of Internet telephony and sensor network policies have been outlined, together with the results of analysing these domains using RECAP.

5.7.2 Evaluation

The presented work can be evaluated in terms of the adopted approach and of the tool which implements it.

Conflict Filtering Approach

Policy conflict is the equivalent of Feature Interaction (FI) – the issue of undesirable or irregular system behaviour caused by interfering services in a system. Interaction filtering was originally proposed as a first step in detecting FIs in telecommunications services. A variety of approaches and tools have used the filtering notion, but existing work is either specific to solving FI in telecoms and/or uses formal methods of analysis. The approach and tool presented in this chapter analyses policy action combinations as opposed to services, is domain-independent (not just applicable to telephony) and uses non-formal methods of analysis.

Filtering is used here as an initial stage to detect potential policy conflicts in a domain-specific specialisation of the APPEL policy language. Existing conflict detection handling in the ACCENT policy system takes place at the policy level (after the policy language has been defined). The approach here filters conflicts between pairs of actions and their parameters much earlier at the language specification stage. This language level analysis is more abstract, providing a rough overview of potential conflicts based on the effects of each action upon the system environment. At this earlier stage, filtering allows for refactoring of the policy language in light of the types of conflicts identified

– a task that is not easy once the language is finalised and policies are created and deployed.

The filtering approach is non-formal and is more scalable, pragmatic and simpler to use than formal methods of FI and policy conflict filtering. Associating actions with their effects is very simple compared to formal methods, but has been shown here to give effective results. As the approach is straightforward and domain-oriented, it is much less expensive to use than one that requires a complete formal model, and is more likely to scale effectively for domains with larger sets of action and parameter combinations than those documented within telephony.

The use of non-formal methods may however account for potential weaknesses in the approach, namely omissions in allocating effects to actions and in the verification of conflicts (the issue of false-negatives and false-positives). As the approach relies on a definition of action effects in an ontology beforehand, incorrect linking between categories or omissions of an action/category association may result in failure to identify potential conflicts. This issue is partially helped by an inspection stage of the approach which allows the user to confirm conflicts or identify conflicting combinations that may have been missed.

False-negatives are action combinations the approach wrongly deems to be conflicting, while false-positives are combinations that the approach finds non-conflicting but are actually conflicts. In the presented approach, analysis of the telephony domain detected all previously known conflicts, but also identified a false-negative (confirm bandwidth conflicting with another confirm bandwidth action). This combination is not a conflict in terms of the underlying system implementation but was flagged as a potential conflict. In the Accent system, confirming bandwidth allocation more than once is perfectly acceptable, but in other telephony systems this action combination might be disallowed or looked upon as inefficient. This particular false-negative appears due to a lack of information about how the underlying system handles bandwidth confirmations.

In general, false-negatives might be reduced or avoided if more detailed action descriptions were introduced. Other methods of tackling the problem of false-positives and false-negatives may be to use formal analysis in combination with the presented approach. A more rigorous method of filtering could be to combine formal and non-formal methods. For example, conflict detection techniques for APPEL discussed in section 5.1.5 (such as the formal semantic analysis by Montangero *et al.* [83], or the formal approach to filtering presented by Layouni *et al.* [77]) might be investigated further as possible extensions to the current approach. Additionally, in [46], Cameron *et al.* present a tool called FIMAMP which groups features with similar properties. This allows FI analysis to focus on groups of feature types rather than individual features. This narrows down the list of potential interactions that should be considered in more detail at a later stage. The reduction in combinatorial complexity could improve the time and scalability issues associated with using formal methods, making the introduction of formal analysis more appealing for this work.

In conclusion, the issue of how to eradicate false positives and negatives is still a matter of debate. Until FI analysis is applied in a wider range of system domains (rather than predominately telephony) it cannot be said for sure whether subtle domain conflicts can be automatically identified without the act of human checking.

The Recap Tool

As a common theme of the approaches presented in this thesis, the RECAP tool enables the ACCENT system (and APPEL language) to be more easily reused across new domains and to make conflict identification more friendly. RECAP automates the previously manual process of analysing pair-wise actions in a domain policy language. It allows the user to explore inferred conflicts and to confirm and refine these conflicts prior to generating resolution policies. This has improved the scalability of APPEL, and has substantially reduced the time and complexity of dealing with conflicts.

The RECAP tool is domain-independent. Unlike many existing approaches and tools, policies in any domain may be analysed. The tool is also useful for policy applications where action parameters play a bigger role. RECAP can also be used both initially and in later revisions of a particular policy language, to refine conflicts and generate resolutions. RECAP has also been designed for stand-alone use. Although conflict data is expected to derive from an ontology, conflict information may be input from a local file. This allows data generated by other tools or systems to feed into RECAP for conflict filtering. The only requirement for this is knowledge of the conflict data format used by RECAP.

RECAP has been used to detect conflicts in two very different policy language specialisations. The tool has been shown to accurately aid conflict analysis for Internet telephony. However, this did confirm that human guidance is still required in a very small number of cases. Detecting conflicts in the sensor network domain is less easy as conflicts arise between the combinations of parameters within a single action – the specific values of which are dependent on the sensor domain. Effects attributed to each parameter have yielded a core set of resolutions which can be customised to suit the domain.

In terms of improvements, RECAP could be given a more user-friendly interface to change the default resolution policy structure and parameters. Currently this is achieved by manually editing the properties file. Resolution policies generated by RECAP are assigned an identifier. If the identifier of such a policy is changed manually (e.g. directly within the associated XML or via the policy wizard) there is a possibility this could lead to duplication of a resolution policy. The tool could potentially detect this situation by looking for overlap of resolution triggers and conditions. Another useful feature of the tool would be to communicate two-way with the policy server. Further to uploading resolutions to the policy server, it could read existing resolution policies (defined manually or through the policy wizard) and update the conflict matrix with conflicts derived from these. This would allow conflicts defined manually via the policy wizard to be used in conjunction with conflicts identified by RECAP.

Chapter 6

Goal-Directed System Management

This chapter presents an approach to managing a system using high-level goals. In the methodology developed, a goal is a user-defined aim for system operation, which is realised through the selection, parameterisation and execution of suitable policies. Whereas policies are only applicable when triggered, goals persist throughout the life of the system. In this chapter, section 6.1 introduces the concept of a goal and summarises existing goal-based systems and approaches. Section 6.2 presents a goal-directed approach and section 6.3 explains how this has been implemented as a working system. Sections 6.4 and 6.5 demonstrate the goal-directed approach for the applications of Internet telephony and sensor network/wind turbine management respectively. The final section 6.6 summarises and evaluates the chapter.

6.1 Introduction and Background

This section outlines the differences between goals and policies, the motivation for using goals in a policy-based environment, and summarises general goal-based approaches in computing, and discusses prior work using goal-based policy approaches. Explanation is given as to how the approach presented in this chapter relates to previous work in this field.

6.1.1 Goals vs. Policies

A goal is distinct from a policy in several ways. Goals are abstract aims and objectives of a system. These are not immediately achievable through any single low-level system action (e.g. “minimise vehicle breakdown” or “increase sales of blue Smarties”). In contrast, a policy is a structured description of how a particular (detectable) system event can be handled using actions that dynamically modify system behaviour (e.g. “run a full diagnostic checking program on the engine when an oil warning light flashes”). Goals are expressed as general aims that do not consider the technical capabilities of the underlying system, whereas policies relate directly to specific system behaviour.

6.1.2 Motivation for Goal-Direction

The need for goal direction in a policy-based environment stems from the need to control and manage a system from a higher-level, more abstract viewpoint. While policies allow users to customise the way a system is managed, this flexibility is something of a double-edged sword. For almost any application domain, there may be restrictions imposed from a higher level. These may be administrative guidelines, health or safety procedures, legal requirements, technical restrictions on the system concerned, or more general business objectives. Such objectives must somehow be adhered to through the policies executed. Conflicts must be handled between the actions of both levels. Policies detect and handle specific system events at the moment they occur, with limited consideration as to the state of the system as a whole. In the event of several policies applying under the same conditions, the policy system will simply execute them all should no conflict be determined. Using policies alone, it cannot be determined which policies are “better” to choose. The use of high-level objectives (goals) to direct system behaviour offers the following benefits:

- high-level management aims can be specified above technical actions
- greater autonomy is offered in system management and control
- the most appropriate triggered policy or policies may be selected for execution depending on their contribution to the high-level goals

- goals exist continuously throughout the system lifetime, constantly managing and directing low-level system behaviour.

6.1.3 Goal Identification and Definition

Goals must be identified and then defined in a structured and processable manner. Similar to software system requirements elicitation and analysis, basic information gathering techniques can produce sample goals from the human, software or hardware actors that interact with the system. Sources of high-level goals may be wide-ranging, including human operators (e.g. technical and non-technical users of the system), commercial organisations and government organisations (for security or legal requirements). User requirements based on use cases and scenarios can be used to extract goal descriptions. These descriptions may then be elaborated in formal languages or natural language, expanding and breaking down individual descriptions into subgoals or actions. Once identified, goals are defined as individual statements using a particular formal or structured language.

6.1.4 Goal Refinement

Goal refinement, or goal elaboration, is the process of incrementally breaking a goal down into one or more subgoals (actions or policies) to achieve it. A number of areas have taken advantage of the notion of goal refinement, including software and systems development techniques. A goal-based approach to policy refinement views goals as high-level abstract policies which can be refined into low-level concrete policies to realise them. Two types of refinement can be discussed in relation to the work of this chapter: policy-to-policy refinement and goal-to-policy refinement.

Policy to Policy Refinement

Policy-to-policy refinement (or “policy refinement”) is the act of deriving low-level enforceable policies from higher-level more general policies. Policy refinement implies a policy model consisting of multiple levels (a hierarchy) of policies, the top levels being used to influence policies at lower levels. The concept of a policy hierarchy was first addressed by Moffet and Sloman in [82], identifying the need for definition of broader business objectives that could be translated into low-level system actions to achieve them. For example, a high-level policy might be: “If the current network is overloaded, choose another channel to route the message”. A low-level policy that could eventually act on this might be: “If network channel 3 is free, route a message through this channel”. The main objectives of a policy refinement process are:

- Determine the resources that are needed to satisfy the requirements of the policy. (This involves mapping abstract high-level policy entities to specific objects/devices in the underlying network.)
- Translate high-level policies into operational policies that the system can enforce. (This ensures policies derived from the refinement process are supported by the underlying system.)
- Verify that the lower-level policies actually meet the requirements specified by the higher-level policy. (This ensures the incremental refinement process can be checked for consistency and correctness at all stages.)

These objectives are aimed solely at policy-to-policy refinement, differing from the notion of higher-level goals which are not necessarily structured in the same form as an executable policy. The initial application described by Moffet and Sloman concentrated on the configuration of a network. Refinement therefore focused on the physical network resources and sets of operations supported by the physical hardware – which are relatively concrete to determine.

Moffet and Sloman identified two distinct phases of policy refinement. The first phase is the translation of abstract policies into operational subpolicies. The second phase is to map these subpolicies to specific operations or actions that configure the system to achieve the (initial abstract) policy. Subpolicies are mapped to one or more policy actions. Of these phases, the first is generic (a technique applied to any domain policy sub-system), whereas the second is highly domain-specific.

Goal to Policy Refinement

Refinement of goals into executable policies requires plan formation and a mechanism to either follow this plan or refine/regenerate it in response to changes in system behaviour. Although policy-based management techniques have expanded, standard methods of deriving executable policies from higher-level system objectives is the focus of research. Goal direction using goals and policies requires:

- a method of representing goal-related information for a domain
- a technique for refining high-level goals into concrete policies
- a means of inferring the combinations of actions that will achieve these concrete policies.

6.1.5 General Goal-Based Approaches in Computing

Goals are not a new concept in the context of system design and operation. Goal-driven systems have been utilised in a variety of approaches since the early 1960s. Many goal-based techniques and approaches exist in the form of systems, models, frameworks and formal languages which govern how goals may be identified, defined and refined into actions that achieve them. This section outlines the main goal-based approaches in computing which have been influential in the design of existing goal and policy-based systems (described in the next section) and also the approach to goal directed configuration described in this chapter.

AI Planning

Automated planning and scheduling is a topic related to that of goal-directed configuration. It was formed as a general branch of Artificial Intelligence (AI) and applied initially in the field of robotics. Planning solves problems by considering desired future states, and then deriving a suitable action plan to describe what actions to take to change a system from its current state to its goal state. In known environments with available models, planning can be done offline. Dynamic environments require more complex online approaches, where strategies need constantly revised and models of the domain area need to be adapted. Such dynamic situations are largely addressed using trial and error approaches, and ultimately the results are unpredictable. A typical planning algorithm takes three inputs encoded in a formal language: a description of the initial state of the world, a description of the desired goal(s), and set of possible actions.

Agent-Based Systems and Intelligent Agents

Agent-based systems work toward achieving a given goal in a modelled environment. There are two types of entity to model in such a system: actors (electronic or human entities who seek to achieve objectives or goals) and resources (materials, devices, systems, engineers, etc.). AI agent-based systems program agents based on the concepts of belief, goal and plan. An agent tries to fulfill its goals by selecting appropriate plans, depending on its beliefs about the environment. Through the execution of a plan, the world and the beliefs about the world change – resulting in successful goal achievement.

Intelligent Agents provide systems with a form of autonomous control, allowing software to reason about the environment in which it operates. To do this, agents must have access to information about the history and operation of the system concerned. An agent perceives the environment through sensors and acts on the environment through actuators (output action signals) [69]. Although various types of agents exist, goal-based, and knowledge-based agents are particularly relevant here.

Goal-based agents act to try to achieve set goals. They monitor the current state of the system via sensor input, consider what the world would be like if each possible action were applied, and choose the action that is most likely to eventually lead to achievement of its goals. Specialist languages exist to support the development of goal-directed agents, such as 3APL [1] (pronounced *triple-a-p-l*).

Knowledge-based agents employ first-order logic to deductively reason about an environment – a process otherwise known as monotonic reasoning. In contrast, non-monotonic reasoning deals with situations where conclusions may require revision in the light of new knowledge being received [95]. Knowledge-based agents use a Knowledge Base (KB) which defines facts about the world (i.e. states, goals and system actions), expressed formally in a Knowledge Representation Language. STRIPS (Stanford Research Institute Problem Solver [64]) and it's successor ADL (Action Description Language [85]) are two well-known representation languages.

Requirements Engineering (RE) Methods

Goal orientation has been used in Requirements Engineering (RE) to derive software specifications in system design. A good background covering this topic can be found in [30] and [79]. Requirements Engineering (RE) is the process of gathering, defining and analysing the needs, conditions and functions of a system or software application. A high-level system goal can be refined into formal specifications of software services or operational functions. The sources of goals for such RE purposes are based on human and organisational behaviour, with less focus on the technical requirements for a system. Many RE systems have utilised the KAOS framework [23] to model and refine goal-based scenarios.

KAOS (Knowledge Acquisition in autOmated Specification) is a framework for goal-oriented requirements engineering based on temporal logic and AI refinement techniques. Goals and states are rigorously defined in order to generate a formal proof that derived requirements match the goals defined for the system. Goals are collected through analysis of system functionality, technical documentation, and input from future system users. Gathered goals are structured so that:

- each goal (except the top-most goals) is typically justified by at least one other goal that explains why the goal was introduced
- each goal (except the bottom-most goals) is refined as a collection of subgoals describing how the refined goal can be achieved.

Goals can conflict when the system reaches a state in which it cannot satisfy goals simultaneously. Agents in KAOS are human or automated components. Each goal is continually refined into subgoals that are assigned to and achieved by multiple agents. The refinement process stops when the top-most goal is placed under the responsibility of a single agent.

Summary and Analysis

Of the goal-related approaches just described (AI planning, agent-based systems and RE methods), none are particularly suitable for adaptation in a policy-based domain.

AI planning methods are more effective when used in an offline environment, and not so helpful when trying to derive policies from goals dynamically. Pure agent-based modelling techniques are inappropriate as they concentrate on the prediction and modelling of future outcomes as opposed to transforming current system behaviour. The policy system environment is non-monotonic (the best plan of action is subject to change depending on feedback from the underlying system) making knowledge-based agents unsuitable. However, goal-based agent techniques are useful as a base for the dynamic goal refinement approach described later in this chapter.

The use of RE methods of goal definition appear best suited to the development of a system from scratch. The goal-directed approach presented in this chapter was developed in an almost bottom-up fashion in that, through the Appel policy language, the types and structure of actions into which goals can be refined were already known. As discussed in the next section, other existing goal-based policy approaches have utilised Kaos. While Kaos is useful for accurately modelling goals and actions and ensuring all goals are implementable by the underlying system, the approach lacks any mechanism to refine or implement the specification generated – extra work is required to interpret the specification and generate appropriate actions to achieve it, making it less suitable for the approach in this thesis.

6.1.6 Existing Goal-Based Policy Approaches

This section summarises and critically analyses existing work incorporating goals and policies that is most relevant to the goal-directed approach presented later in this chapter. This work includes frameworks and mechanisms for both goal-to-policy and policy-to-policy refinement. Existing work is first outlined, followed by a summary explaining how the work of this thesis differs.

Existing Approaches

In [60], Davy *et al.* present a formal policy continuum model and authoring process, together with an approach to policy conflict detection in goal-derived policies. In this approach, high-level policies are designated as goals to be achieved by refinement into groups of lower level policies. The policy continuum is the concept of separating sets of policies (possibly by different authors) into different levels, but in a flat structure rather than a traditional policy hierarchy. For example, policies specifying business objectives written by administrative users appear at one end, whereas policies that define more complex system aspects written by technical experts are at the opposite end. However, the framework outlines a generic methodology rather than describing a particular implementation.

A concept termed "policy-based reconfiguration management" is presented in [55]. This approach presents a framework for creating policies dynamically at run time to suit changing system requirements. This utilises a policy hierarchy model to refine high-level user goals into low-level objectives.

A goal-based approach to policy refinement is presented by Bandara *et al.* in [35]. This approach uses the Ponder policy language and environment [57] to represent policies and goals. The system environment and goals are defined formally using Event Calculus. Using abductive reasoning techniques, a method is shown to derive functional sequences of operations to achieve goals.

An implementation of this approach described in [34] uses KAOS to specify goals both informally (in natural language) and formally (using temporal logic rules). Refinement patterns are then used to decompose these goals into subgoals that logically entail them. Goal refinement is based on a technique defined by Darimont and Lamsweerde [59]. The system also uses obstacles (negated goals) to state what the system should not achieve. Obstacles are defined in the same way as regular high-level goals and require similar refinement into subgoals. Obstacles are avoided by introducing new goals. However, refined goals require conversion into policy operations (actions) before execution.

Rubio-Loyola *et al.* discusses an approach using linear temporal model checking for goal-oriented policy refinement frameworks [94]. The approach is based on Requirements Engineering (RE) and model checking techniques, using goal elaboration mechanisms proposed by Bandara *et al.* in [35]. The approach suggests formal verification techniques to analyse policies in an attempt to aid policy refinement. An application of the approach is outlined in [92], demonstrated using a scenario for Quality of Service (QoS) Management, with Ponder as the policy language and environment.

This work was extended further by Rubio-Loyola to define a complete policy refinement methodology [93]. This approach is deemed more applicable to realistic refinement scenarios than previous work. The methodology provides a formal procedure for defining refineable policy hierarchies and a framework to support the refinement process. The framework allows goals and the policy hierarchy to be input, and supports a two-stage refinement process. The first stage refines goals into other goals, effectively placing the input goals in a hierarchy. The second stage refines each goal sub-tree into sets of policies within the policy hierarchy. These stages both occur offline. During system operation, administrators may select which goals to achieve. Dynamically, the lowest-level executable policies are selected to best achieve the selected goals. The approach aims to define a complete framework to capture all processes involved in policy refinement and management, and is not currently a working system.

Summary and Analysis

Of the existing work discussed, most approaches rely on the notion of policy and goal hierarchies, and concentrate on goal refinement as a process of logical entailment by analysing policies and goals at different levels of abstraction.

The approach taken in this thesis differs in the following ways:

- Common to existing approaches is the use of formal methods to define and refine goals, and also to verify policy solutions. In this thesis approach, numerical analysis is used in the form of goal measures, together with optimisation techniques to govern policy selection. This is instead of inferring actions from the goals themselves using formal methods of statement analysis as in the approach by Bandara *et al* [35]. The lack of formal definition makes the task of specifying a goal domain more user-friendly as no experience of formal methods is required.

- Goals are defined using similar language syntax as for ordinary policies, reducing complexities involved in translating a goal language into policy constructs. Note that although the same syntax is used for both goals and policies in this thesis approach, goals are still viewed semantically as separate entities from policies – a major difference to existing work which views goals as more abstract policies.
- The approach does not utilise goal or policy hierarchies, thus avoiding documented issues where modifying policies at lower levels affects the aims of higher-level policies. Goal-related policies are not intended to be modified. In addition, avoiding multiple layers of abstraction in goal and policy definition is simpler to implement and design than the complex hierarchies developed by Rubio-Loyola [93].
- Similar to existing policy-based approaches, many goal-based systems have evolved and have been applied within the domains of security and system management. Existing approaches have not been applied to domains other than network service management, thus failing to demonstrate the benefits that high-level goals can bring to system management over the use of policies alone. The approach presented in this thesis demonstrates a system capable of goal direction using policies in more general domains (specifically, Internet telephony and sensor network/wind turbine management).

6.1.7 Optimisation Overview

In section 6.1.5, problem solving was discussed with a view to forming a plan of actions that ultimately allow a system to move between an initial state and a goal state. Associated with this is the issue of “optimisation”. When attempting to form a plan there may be several possible solutions to a problem, and some may be more favourable than others. Determining the optimal solution depends on a number of factors including the state of the system or environment at a given time and the constraints upon the system imposed by its goals.

The goal-directed, policy-based approach presented in this chapter uses a numerical form of representing the system state and optimisation to select the best set of policies to dynamically achieve goals. The optimisation approach uses a goal function which is similar in nature to the so-called "fitness function" associated with common optimisation algorithms. As a background, the chief characteristics of optimisation and common approaches are given here.

Where there are multiple options that may or may not help achieve a goal, possible action sequences must be examined to determine their effect on the system. The optimal solution is the sequence of actions that moves the system closest to its goal state. In optimisation terms, the process of determining the best sequence of actions to select is known as a "search" [95]. The set of all possible solutions to a problem is known as the search space. As the number and type of search algorithms available is vast, only commonly used methods are mentioned here.

The most efficient search method to use depends on the problem domain, including the size of search space, whether it is fixed or continuous, and the distribution of good and bad solutions within it. Search algorithms can be broadly split into two types: generic uninformed searches (where no problem-specific information is considered) and informed or *heuristic* searches (which use problem-specific knowledge to search more efficiently) [95]. Optimisation algorithms use an evaluation function or fitness function to numerically determine the value of a particular solution. Additionally, algorithms may perform a local search (where only the optimal value found is important) or a global search (where the path or steps taken to get to the optimal value are important). Local search algorithms, such as hill-climbing and simulated annealing, search continuously in the direction of increasing value until no neighbouring value yields a higher value. These algorithms suffer from the problem of "local maxima" whereby higher peaks exist elsewhere in the search space but are overlooked, and they can encounter a "plateau" where the search space is flat and optimisation cannot progress.

Global heuristic search methods for optimisation problems include biologically-inspired evolutionary algorithms. Two common approaches include Estimation of Distribution Algorithms (EDAs) and Genetic Algorithms (GAs), which are based on a population of randomly generated points (or individuals) within the search space. Each individual is typically represented as a binary string that is repeatedly tested against the fitness function to iteratively select (or breed) further individuals that yield higher fitness values. After each generation, an individual may mutate to another point in the search space. The algorithm terminates when it exceeds a designated maximum number of generations (regardless of whether a good solution has been found), or when a satisfactory fitness level is reached for the population. This approach relies upon input parameters such as the initial size of population to choose, the rate of mutation, the maximum number of generations and the time allowed before terminating the search. Consequently, these algorithms are highly problem-specific, and their parameters are difficult to accurately identify [95].

6.2 Goal-Directed Approach

A policy exists to handle a specific event under specified conditions, and is invoked and executed only when those circumstances occur. A goal is intended to have an extended lifespan. Goals define high-level abstract aims of the system, while policies provide a means of responding to specific events with actions that modify system behaviour. This section presents a new approach to goal-directed system management which takes a set of goals and continually tries to achieve them by selecting and executing policies that contribute toward their measures. This work has been published in [54].

6.2.1 Overview

Goals are abstract system objectives from the viewpoint of human users, and are not directly implementable by a single low-level system action. Goals within a policy-based system environment specify the desired optimal states (or classes of states) of the system. Traditional goal-based approaches form a plan to get the system into a goal state from some initial state. On arrival at this goal state, the aim is achieved and the system ceases in its quest. This approach is not effective for goal direction in a policy-based environment for two reasons. The first is that the dynamic nature of a policy system renders it unlikely the system will remain in a single state permanently. Thus, when a goal state is reached, external factors may well force it out of this state.

The second reason concerns the lifetime of the system. A policy system is intended to operate continuously, and should not stop once a goal state is reached. In the approach presented here, the aim is for the system to continuously adapt its behaviour to ensure it remains indefinitely within the scope of its goals. The system does not discard its goals or terminate when its goals are initially achieved. This approach also differs from other known refinement approaches in that it reduces system state, goals and policies to a numerical rather than logical form. Basing refinement on purely numerical analysis allows the use of optimisation to select the most appropriate policies and their parameters at run-time depending on the current values of system variables and other state information. This technique is also more user-friendly towards non-technical domain experts as the goal domain may be expressed without any knowledge of formal methods.

The approach defines the system domain in terms of variables that describe its state. Goals are associated with numerical measures which rely on state variables. Policies have actions which modify the system state in some way. Policies are selected to modify state variables so as to achieve goal measures. The process of selecting policies to achieve goals is implemented in two stages. The first stage is performed statically, where an initial set of prototypes (template policies) is matched against goals and is instantiated as actual policies that contribute to them. This is achieved by matching the effects of prototype actions against each goal measure. The first stage may be involved offline, but also online when goals and policies are defined (or removed). The second stage is performed at run time, while the policy system is operational. When goal-related policies are triggered, goal optimisation attempts to derive the policy combination that, when executed, will contribute most effectively to system goals.

The following subsections outline the goal-directed approach and the algorithms used to select and execute policies to achieve goals. Section 6.2.2 describes the goal domain, including a definition of state variables, goals, goal measures and the overall evaluation function. Section 6.2.3 describes the form of a prototype policy, used as a policy template. Section 6.2.4 explains the static process of analysing and instantiating prototypes to serve goals. Section 6.2.5 explains the run-time process of selecting optimal policy sets that contribute to system goals based on the current state of the system.

6.2.2 Goal Definition and System State

The goal domain is defined in terms of variables that describe the state of the system at any given time. These so-called "state variables" represent aspects of the environment subject to change while the system is operational. Such variables may include configurable system parameters and external values established while the system is running. State variables are of two types:

Controlled: These variables have values that can be set dynamically by a policy action (e.g. allocated bandwidth, sensor reporting frequency or turbine blade angle).

Uncontrolled: These variables have externally modified values that may not be known until run-time. Commonly, these variables are established by a policy trigger reporting on current system events (e.g. current network traffic load or current wind speed). The value of an uncontrolled variable cannot be modified through the actions of a policy. It is altered only as a consequence of changes in the system environment.

Each state variable has an initial default value (used until the actual value is known) and maximum and minimum values which establish its range within the context of the system. The value of each variable may alter as the system changes state.

A goal is an abstract aim defined by a human operator. Every goal has a measure which defines how a numerical value for the goal may be calculated. A goal measure is a function comprising any mixture of controlled variables, uncontrolled variables and optional constants. In addition to a measure, a goal has a direction in which it should be achieved, namely "maximise" or "minimise". Maximising a goal measure aims to find the highest achievable value of its measure, while minimising aims to find the lowest value.

Goals may have optional conditions which act as constraints. A goal condition is effectively a pre-condition on the policies selected to support it, defining additional requirements that must be satisfied for the goal to be achieved. Conditions may use controlled or uncontrolled state variables, or any other parameter known to the system at run time. The format of goal conditions must be compatible with the language used to specify policies.

A system will likely have multiple goals as opposed to a single goal. A set of goals should be achieved concurrently, and may contain measures which tend in different directions – maximising and minimising different functions simultaneously. When defining multiple goals, the same controlled or uncontrolled variable may be specified in more than one goal measure. When defining multiple goals, a "super" measure is required to calculate a numerical value across all goals. This combined measure is termed the goal evaluation function and is a combination of the individual goal measures. Where the goal is to maximise a function, the measure is preceded by a plus sign ($+$). Where the goal is to minimise a function, the measure is preceded by a minus sign ($-$).

When combining multiple goal measures, weightings can be applied to define their relative importance. A domain expert may determine the most appropriate weightings to allocate, based on experience, knowledge of the calculations used in goal measures, or a general preference as to which goals should be ranked over others. The use of weightings also overcomes certain conflicts between goals. Conflicts can occur when multiple goal measures share a common state variable, and at least two of these are tending in opposite directions. That is, one goal is attempting to maximise a function, while the other is attempting to minimise a function. Goal conflicts are different to policy conflicts (discussed in Chapter 5) and are detected and resolved independently of one another.

The elements of the goal approach are explained below using an abstract example. Assuming the system domain is described using the state variables w, x, y and z, and Alfa and Bravo are some measurable aspects of the system we wish to control, the following two goals may be defined:

Goal 1: Maximise Alfa = $x * y$

Goal 2: Minimise Bravo = $x + z + (3.14 * w)/100$

Goal 1 is expressed in terms of the variables x and y. Goal 2 is more complex and is calculated using a combination of constants and state variables. At run time, the values of the variables w, x, y and z will vary in response to environmental events and the actions of policies. The aim of the goal-directed system is to attempt, where possible, to select policies whose actions result in the highest value for Goal 1 and the lowest value for Goal 2. Suppose it is decided that it is more important to minimise Bravo whenever possible at the expense of maximising Alfa. The two goals can be weighted accordingly. The evaluation function is then determined as follows:

Evaluation function:

= Give Alfa weight 0.3 and Bravo weight 0.7

= + (0.3 * Alfa) − (0.7 * Bravo)

= $(0.3 * (x * y)) - (0.7 * (x + z + (3.14 * w)/100))$

Defining the goal domain, including the identification and specification of state variables, goals, measures and weightings, is a task for domain experts. However, the state variables chosen must accurately link with available actions within the chosen policy language. In this approach, policy actions directly or indirectly influence the modification of state variables, therefore their effects must realistically reflect the range of actions supported by the underlying policy system. It is acceptable that a declared state variable might not be affected by any defined prototype policy. However, it is a static error should a prototype policy effect a variable other than those in the declared state variable set.

6.2.3 Prototype Policies

Goals are refined into sets of policies whose actions may contribute to goal achievement. As an initial stage in this refinement process, it is necessary to use a template policy known as a "prototype". A prototype is instantiated as an actual executable policy with additional information which specifies the goal or goals it potentially supports. A set of prototypes must initially be defined as input to the goal system. Prototypes may also contain parameterised actions. The values of a parameter remain open until policy selection time.

All prototypes must contain at least one "effect" statement. Effects are linked to the actions of the prototype and specify how the prototype influences or modifies the state of the system. Both controlled and uncontrolled variables may be referenced within prototype effects. Although uncontrolled values cannot be modified through the actions of a policy, it is possible consequential actions might influence changes in their value. For example, an uncontrolled variable such as a message counter might be incremented by the act of sending an email message.

Figure 6.1 specifies the format of an effect statement using EBNF notation. Note that a prototype may have one or more effects, in a comma-separated list. In summary, a valid effect statement must consist of a variable name (a defined controlled or uncontrolled state variable), followed by an operator (increment, decrement or equals), followed by a value (a parameter or float value). The increment and decrement operators (+=,−=) increase or reduce the current value of the controlled/uncontrolled variable by the specified value. The equals operator indicates the controlled/uncontrolled variable is assigned the specified value. Referring to Figure 6.1, supposing there exists a controlled variable "bar_height", a valid effect statement is "bar_height = 25.5". This indicates one action of the prototype is to assign this state variable the value of 25.5 units. Similarly, the effect could indicate a parameterised action using the notation "bar_height = $h". Based on these effects, a prototype can be evaluated against each goal measure. Where a prototype modifies a state variable contained within a goal measure, it potentially contributes to the achievement of that goal. At this point, the magnitude of the change made by the prototype on each variable is irrelevant.

```
<effects>            ::= <effect_statement>(","<effect_statement>)*
<effect_statement>   ::= <variable_name> <operator> <value>
<variable_name>      ::= controlled_variable|uncontrolled_variable
<operator>           ::= "+="|"-="|"="
<value>              ::= <parameter>|"-"?<digit>+"."<digit>+
<parameter>          ::= "$"<symbol>+
<symbol>             ::= (a..z)|(A..Z)|"_"
<digit>              ::= 0|1|2|3|4|5|6|7|8|9
```

Figure 6.1: Prototype Effect Statement Format

6.2.4 Offline Prototype Analysis and Instantiation

Statically, it is possible to determine whether the execution of a prototype modifies a variable used within a goal measure. Whether this modification actually helps or hinders the system achieve its goals depends on the context and state of the system at run time when a policy is triggered. Consequently, the static stage of goal refinement is limited to identifying prototypes that potentially contribute to goals.

Inputs to this initial processing are a set of goals, their measures and conditions, and a set of prototypes. The output is a set of policies, one policy for each prototype deemed potentially helpful to achieving at least one goal. The process is performed in two steps. The first step is to scan the set of prototypes, matching each one to goals it might contribute to. The second step scans this set again, instantiating prototypes that contribute to goals and disregarding those which do not.

The first step is shown in Figure 6.2. The algorithm considers each prototype in turn, comparing its effects against the measure for each goal. If a prototype modifies any variable within a goal measure, it potentially contributes to that goal. For each prototype:

- If at least one effect modifies a variable in the goal measure, the prototype is marked as contributing to that goal, and the next goal measure is considered.
- If no effect modifies a variable in the goal measure, no contribution is made, and the next goal measure is considered.

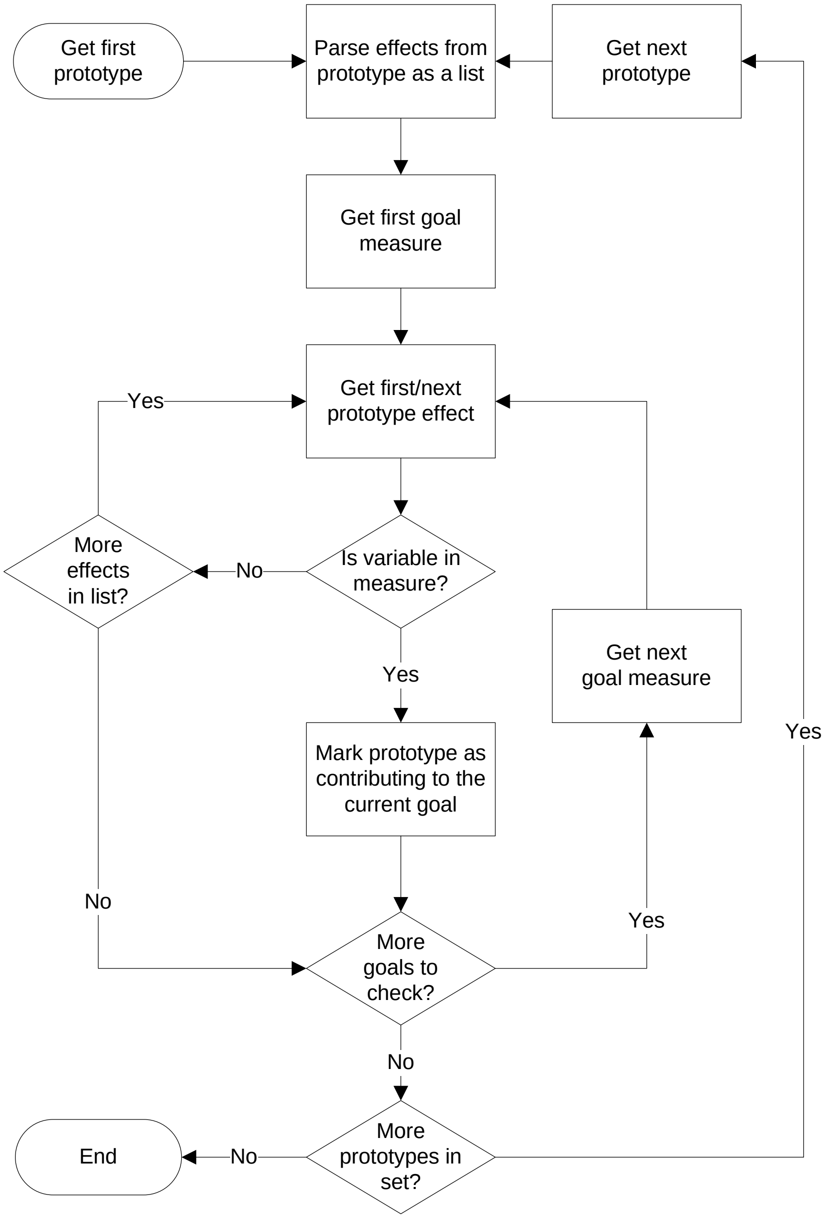


Figure 6.2: Prototype and Goal Matching Process

The second step instantiates prototypes that contribute to goals, and filters out currently irrelevant prototypes. The instantiation process clones each prototype and alters the copy to be a valid, executable policy. In particular, this involves inserting any conditions associated with goals the policy supports. Goal conditions are appended to any existing conditions inherited from the prototype. The approach requires that to make a contribution toward at least one goal at run time, a policy must satisfy any prototype-inherited conditions, and at least one set of conditions associated with a goal. For the goal set a policy contributes to, if all goals have conditions, the conditions from each goal are appended to the new policy. If none or a subset of the goals have conditions, the policy remains unchanged in its conditions. To demonstrate this procedure, suppose a prototype has a set of conditions and is instantiated to support three goals, the third of which has no conditions. The resulting policy conditions are combined as follows:

<Prototype conditions> AND
  (<Goal 1 conditions> OR <Goal 2 conditions> OR <Goal 3 *no conditions*>)

As the third goal has no associated conditions (i.e. there is an implicit "true" condition), the policy will always be eligible for execution if just its original policy conditions hold. To avoid redundant coding and unnecessary runtime processing, the algorithm ignores the conditions of Goals 1 and 2 during the instantiation process – inserting no goal-related conditions. The full prototype instantiation process is shown in Figure 6.3, and is summarised as follows:

- If a prototype is not marked as contributing to any goal, it is disregarded.
- If a prototype is marked as contributing to a single goal, the prototype is instantiated as a policy and any conditions on that goal are inserted.
- If a prototype is marked as contributing to more than one goal, and none of these goals have conditions, the prototype is instantiated as a policy.
- If a prototype is marked as contributing to more than one goal, and all of these

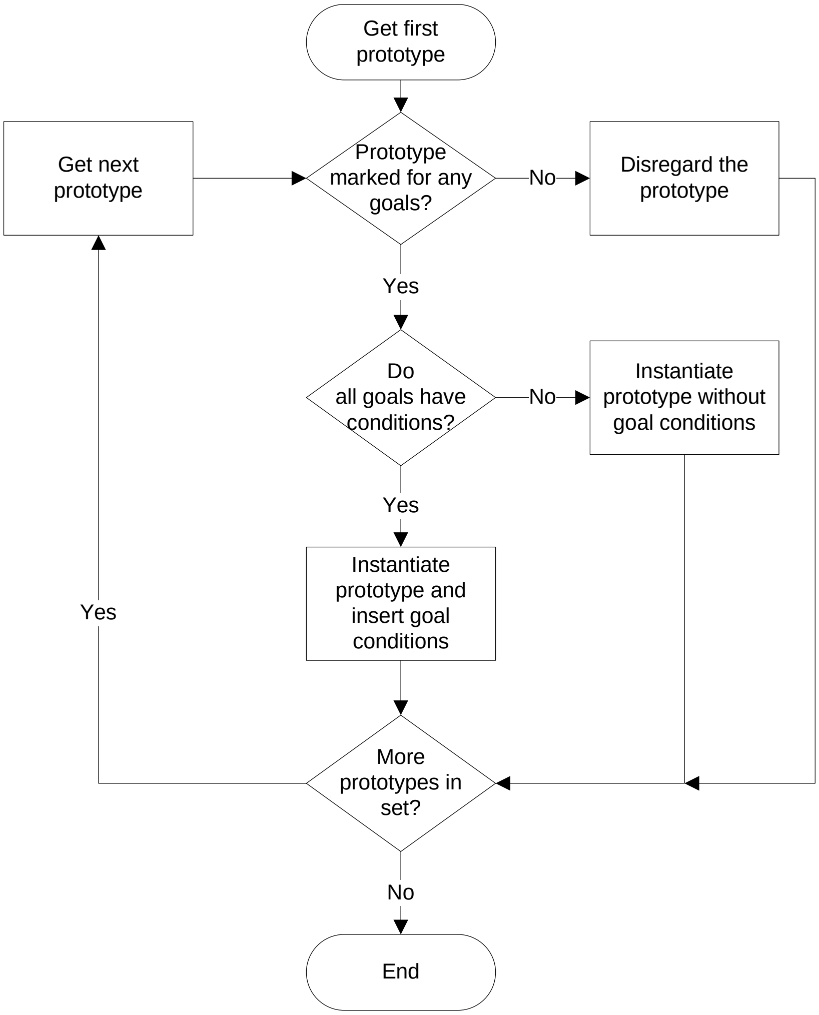

Get first prototype
Get next prototype
Prototype marked for any goals?
No
Disregard the prototype
Yes
Do all goals have conditions?
No
Instantiate prototype without goal conditions
Yes
Instantiate prototype and insert goal conditions
Yes
More prototypes in set?
No
End

Figure 6.3: Prototype Instantiation Process

goals have conditions, the prototype is instantiated as a policy and all goal conditions are inserted.

- If a prototype is marked as contributing to more than one goal, but only a strict subset of these goals have conditions, the prototype is instantiated as a policy and **no** goal conditions are inserted.

On completion of this process, prototypes are no longer required for execution (though they remain for future use when goals change). All run-time references are made to the newly generated policies.

This two-step process of prototype analysis and instantiation must be performed each time the goal set is altered. Alterations include the modification of goals, their conditions or measures, state variables or prototypes. Such changes in the goal domain should be infrequent, but it is plausible that over time the system goals will change to reflect fresh objectives or new policy actions.

A concrete demonstration of the described process is given in the following example. There are three goals and three prototypes, Goal 1 and Goal 2 have associated conditions and the defined state variables are v, w, x, y and z:

Goals

Goal 1: Maximise Alfa (Conditions: $x > 0$ and $w > 0$)

Goal 2: Minimise Bravo (Conditions: $w = 1$)

Goal 3: Minimise Charlie

Measures

Goal 1: $x * y$

Goal 2: $x + z + (3.14 * w)/100$

Goal 3: $x * z$

Prototypes

Prototype 1: $y \mathrel{+}= 2$

Prototype 2: $x = 1$, $z \mathrel{+}= 1$

Prototype 3: $v \mathrel{-}= 5$

Static analysis of the three prototypes gives the following results:

- Prototype 1 modifies the value of y which is present in the measure of Goal 1. Prototype 1 is instantiated in support of Goal 1, and the conditions of this goal are inserted into the new policy.
- Prototype 2 modifies the value of x and z. The effect on x is compared against each goal measure. As x is present in all three goal measures, the second effect statement referring to z need not be considered at this stage (since the prototype is definitely needed). Prototype 2 is instantiated in support of Goals 1, 2 and 3. As some but not all of these goals have associated conditions, no conditions are inserted into the new policy.
- Prototype 3 modifies the value of v which does not feature in any of the goal measures. Prototype 3 is therefore disregarded.

The result is two new policies which, when triggered, potentially contribute to the defined goals.

6.2.5 Run-time Policy Selection and Parameterisation

The previous subsection outlined an initial stage in the approach to goal refinement, which instantiates prototypes as executable policies to support system goals. This section explains an approach to achieving goal-directed behaviour at run-time, by dynamically selecting appropriate sets of policies that move the system closer toward its goals. This run-time process is performed when goal-related policies are triggered, and consists of two main activities: policy selection and parameter optimisation.

The selection process applies only to goal-related (or prototype derived) policies. When a policy is triggered, the values of state variables used within the goal evaluation function are known. Depending on the current values of these variables, the goal evaluation function might be better or worse if a triggered policy is executed. If more than one goal-related policy is triggered, every combination of these policies must be considered in order to determine the one that yields the highest value for the function. As part of this process, combinations that include parameterised policies must be optimised to select the most appropriate value for each parameter. The outcome is therefore a subset of triggered policies along with optimal values for any parameters contained within them. Possible scenarios are:

- No policies are selected as every combination yields a lower value when compared to the current state of the system. In other words, executing any of the policies triggered will move the system further away from its goals.
- All or some of the triggered policies are selected for execution, as the combination of their actions pushes the system closer to its goals.

The run-time selection procedure applies only to goal-related policies whose conditions are satisfied. Triggered goal-related policies whose conditions are not satisfied are automatically filtered out by the policy server. An overview of the process is described in Figure 6.4.

The approach is summarised as follows:

1. On receipt of a trigger, the policy system retrieves all policies which are applicable and whose conditions are satisfied.
2. All possible combinations of prototype-derived policies are considered.
3. Each combination is considered in turn:
   (a) The effects of each policy are compared.
   (b) If at least two policies in the combination seek to modify a common state variable, the combination is deemed conflicting and discarded.
   (c) If there are parameterised policies, the policy set is optimised against the goal evaluation function for possible values of each parameter. The highest score obtained for the function is noted along with the parameter values associated with it.
   (d) If no conflicts exist, and no parameterised policies are present, the combination of policies is evaluated against the goal evaluation function and the result stored.

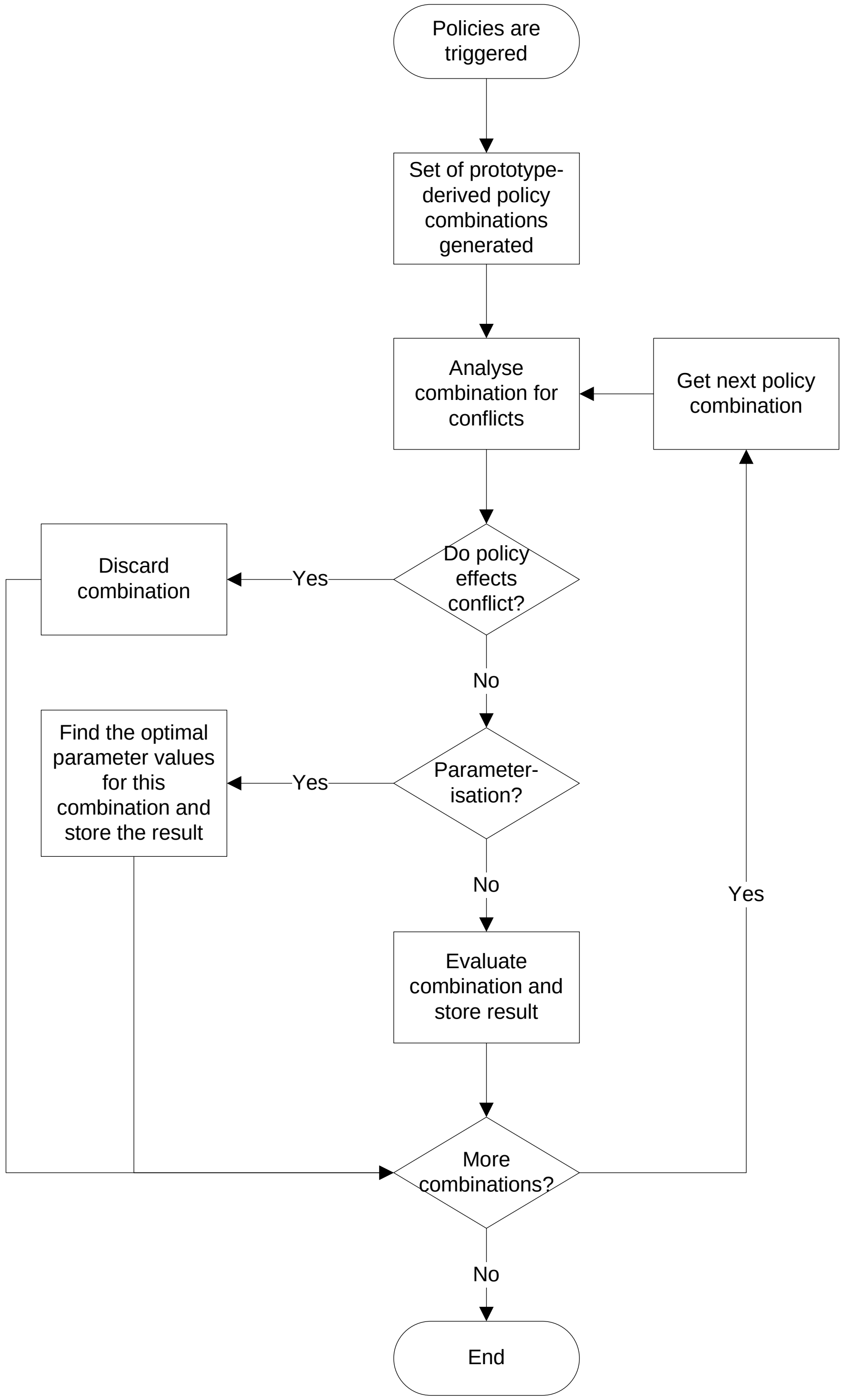

Policies are triggered
Set of prototype-derived policy combinations generated
Analyse combination for conflicts
Get next policy combination
Do policy effects conflict?
Yes
Discard combination
No
Parameter-isation?
Yes
Find the optimal parameter values for this combination and store the result
No
Evaluate combination and store result
Yes
More combinations?
No
End

Figure 6.4: Runtime Policy Selection and Optimisation

4. The combination of policies (and parameter values if applicable) resulting in the highest score for the goal evaluation function is selected for execution. Any parameterised policies are instantiated with the chosen parameter values.

Optimisation algorithms are largely domain-dependent. While powerful optimisation techniques, such as those outlined in section 6.1.7, are suitable for specific problems, simpler techniques may be sufficient for others. The nature of the problem space varies across applications, with different topology and greater or fewer local maxima or minima. The most efficient and effective optimisation algorithm to apply will therefore be different for each goal domain. The approach therefore defines the inputs and outputs of runtime optimisation without defining the exact algorithm that should be used.

The inputs to the optimisation algorithm are a set of goal-related policies, the goal evaluation function (an expression), and a list of state variables and their current values. Additionally, the range of values for each state variable must be known, based on the maximum and minimum value for each. The output of the optimisation is a set of policies, including values for any parameters contained within them.

The process of evaluating policy combinations against a goal function is demonstrated in the following example. This example assumes three goal-related policies have been triggered (labelled P1, P2 and P3) and that these have the effects shown below. State variables are x, y and z, and their current values are as below. The evaluation function is deliberately simplified in this example to easily calculate a value for each policy combination.

Goal Evaluation Function:

$(x * y) - (x * z)$

Triggered Goal-related Policies:

P1: $y \mathrel{+}= 2$

P2: $x = 1$, $z \mathrel{+}= 1$

P3: $x \mathrel{-}= 5$

Current State Variable Values:

x: 10

y: 10

z: 1

The result of optimising the policy set is shown in the table below.

P1	**P2**	**P3**	x	y	z	**Result**
0	0	0	10	10	1	90
0	0	1	5	10	1	45
0	1	0	1	10	2	8
0	1	1	*	10	2	-
1	**0**	**0**	**10**	**12**	**1**	**110**
1	0	1	5	12	1	55
1	1	0	1	12	2	10
1	1	1	*	12	2	-

The best solution is the combination of policies that result in the highest value for the goal evaluation function. A "*" denotes a variable that is altered by more than one policy in the selected combination. This constitutes a conflict and the combination is automatically discarded. In this example, P2 and P3 both seek to modify the variable x and therefore cannot be executed together. The first row in the table represents the state of the system before executing any of the triggered policies, using the variable values as they are currently known. Should this value yield the highest result, no goal-related policies would be selected for execution. In this case, the optimal solution is to select P1, as this yields the highest value of 110. At this moment, executing P1 is therefore the most helpful in trying to achieve the defined goals.

During the policy selection process, values for variables in any parameterised policies (within the triggered policy set) are chosen. The following example assumes the same goal evaluation function, current state variable values and triggered policy set as just described for policy selection, except this time P2 is a parameterised policy which sets the variable z to some optimal value (noted as \$T). For this example, the value range for z is small – between 1 and 3 inclusive. The run-time parameterisation process evaluates each value of z for every policy combination that includes P2. The results of this are given in the table below.

Triggered Goal-related Policies:

P1: $y += 2$

P2: $x = 1$, $z = \$T$ (\$T = 1 or \$T = 2 or \$T = 3)

P3: $x -= 5$

P1	**P2**	**P3**	x	y	z	**Result**
0	0	0	10	10	1	90
0	0	1	5	10	1	45
0	1	0	1	10	\$T=1	9
			1	10	\$T=2	8
			1	10	\$T=3	7
0	1	1	*	10	\$T=1	-
			*	10	\$T=2	-
			*	10	\$T=3	-
1	**0**	**0**	**10**	**12**	**1**	**110**
1	0	1	5	12	1	55
1	1	0	1	12	\$T=1	11
			1	12	\$T=2	10
			1	12	\$T=3	9
1	1	1	*	12	\$T=1	-
			*	12	\$T=2	-
			*	12	\$T=3	-

The results show that of the possible values for z, choosing the value 1 for $T for policy combinations that include P3, yields a higher result than choosing 2 or 1. In this example however, the highest value overall still comes from selecting P1 alone. If say the combination of P1 and P2 had resulted in the highest result, these policies would have been selected and P2 parameterised with the value 1 for $T.

6.3 Goal System Implementation

The previous section outlined an approach to managing systems using high-level goals and policies. An implementation of this approach is now described. A goal system has been designed and interfaced with the existing Accent policy system. Goals, prototypes and policies refined from goals are defined using an extended form of the Appel policy language. Ontologies provide a means of defining goal-related information that can be retrieved by the goal system at run-time. The goal system and goal-directed approach have been evaluated for the applications of Internet telephony and sensor network/wind turbine management. These applications are outlined in the following sections.

Section 6.3.1 describes the use of an ontology to define the goal domain. Section 6.3.2 outlines the syntax of goals, prototypes and goal-related policies. Section 6.3.3 provides an overview of the goal system architecture. Section 6.3.4 explains how the implemented system operates.

6.3.1 Goal Domain Ontology Definition

Information associated with the goal domain is specified and stored in an ontology. Ontologies were previously introduced and discussed in Chapter 4. Chapter 3 described how a collection of related ontologies was created to define the generic and domain-specific aspects of the Appel policy language. The core, generic aspects of Appel and other policy environment details are defined in the OWL ontology named *GenPol*. This ontology encompasses a definition of high-level concepts that describe components of the goal domain and how they relate to one another.

The goal-related concepts include definitions of controlled and uncontrolled state variables and the general definition of a goal measure. Domain-specific ontologies import the contents of *GenPol* and may then extend these high-level concepts to define actual named variables, measures and their properties, including a maximum, minimum and default value for each. The relationship between goal measures and state variables is also expressed in a domain-specific ontology.

Information stored within an ontology is accessed by the goal system using the Poppet server interface (discussed in section 4.7). A domain-specific ontology may be queried for a particular goal measure to obtain a list of its associated controlled and uncontrolled state variables. Similarly, the maximum, minimum and default value for any state variable may be retrieved.

6.3.2 Goal and Prototype Syntax

Goals, prototypes and goal-related policies are defined as individual XML documents. Their syntax uses minor extensions to the Appel policy language in the form of two additional policy attributes:

supports_goal Applicable to a policy only. A string listing the goals a policy supports. The format is a comma-separated list of goal document IDs. This attribute and its contents are generated during the process of static prototype instantiation.

effects Applicable to prototypes and policies. A string listing the effects of a prototype or policy on the goal domain. The format is a comma-separated list of effect statements.

Only policies derived from prototypes require the “supports_goal” attribute above, which acts as a link at run time to distinguish between policies defined by domain experts and policies generated to support goals. Policies that do not contain a goal-related attribute are not included in the optimisation process.

Defining goals and prototypes using Appel maintains consistency between the goal- and policy-based approaches, and has several advantages when instantiating prototypes as executable policies. Firstly, the process of creating goal-related policies is simplified as prototype documents can readily be cloned and altered. Secondly, goal conditions are already in a policy-friendly format (i.e. composed of a parameter, operator and value) and so may be readily inserted into new policies. In addition, existing policy system components may be reused or shared. The policy store can be used to hold goals and prototypes, and the policy wizard can be used to create and edit goals and prototypes in a similar way to that of user-defined policies.

A description of a goal, prototype and goal-related policy document is now given. The XML Schema for goal and prototype documents can be accessed at [21].

Goal Document

A goal is similar to a regular Appel policy, except it cannot have a trigger. The element type is "goal" rather than "policy" to distinguish a goal document from a policy or prototype. A goal may have unlimited conditions but is restricted to a single action. Two possible actions exist: `maximise(arg1)` and `minimise(arg1)`. For each action, the parameter `arg1` is a String representing a goal measure which is previously defined. The "owner" and "applies_to" attributes must be a unique entity for the goal system and not an existing policy owner. An example goal is shown below.

```
<policy_document
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
    "http://www.cs.stir.ac.uk/schemas/appel_goal.xsd">
  <goal
      owner="goalkeeper@cs.stir.ac.uk"
      applies_to="goalkeeper@cs.stir.ac.uk"
      id="Maximise Multimedia Use"
      enabled="true"
      changed="2008-04-21T10:20:59">
    <policy_rule>
      <condition>
        <parameter>day</parameter>
        <operator>in</operator>
        <value>1..5</value>
      </condition>
      <action arg1="multimedia_use">maximise(arg1)</action>
    </policy_rule>
  </goal>
</policy_document>
```

Prototype Document

A prototype is identical in structure and format to a policy, except the element type is defined as “prototype” instead of “policy”. This ensures only executable policies may be triggered within the policy server at run-time, and only prototypes are retrieved by the goal system during the static prototype analysis stage. An example prototype is shown below.

```
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
    "http://www.cs.stir.ac.uk/schemas/appel_prototype.xsd">
  <prototype
      owner="goalkeeper@cs.stir.ac.uk"
      applies_to="@cs.stir.ac.uk"
      effects="bandwidth += 256"
      id="Add Video To Call"
      enabled="true"
      changed="2008-04-21T10:20:59">
    <policy_rule>
      <trigger>call_incoming</trigger>
      <condition>
        <parameter>bandwidth</parameter>
        <operator>lt</operator>
        <value>1024</value>
      </condition>
      <action arg1="video">add_medium(arg1)</action>
    </policy_rule>
  </prototype>
</policy_document>
```

Goal-Related Policy Document

Goal-related policies (i.e. those generated through prototype instantiation and not normally written by a policy developer) are defined using almost the same syntax as regular Appel policies, but must contain values for the two additional policy attributes “supports_goal” and “effects”. The presence of both attributes distinguishes goal-related policies from user-defined policies at runtime. Only policies with both these attributes can be processed by the goal system. Should a regular user-defined policy include these attributes, it would be treated as goal-derived. Regular policies should therefore omit these attributes. An example of a goal-related policy is shown below.

```
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
    "http://www.cs.stir.ac.uk/schemas/appel_policy.xsd">
  <policy
      owner="goalkeeper@cs.stir.ac.uk"
      applies_to="@cs.stir.ac.uk"
      effects="bandwidth += 256"
      supports_goal="Maximise Multimedia Use"
      id="Add Video To Call Policy"
      enabled="true"
      changed="2008-04-21T10:20:59">
    <policy_rule>
      <trigger>call_incoming</trigger>
      <conditions>
        <and/>
        <condition>
          <parameter>bandwidth</parameter>
          <operator>lt</operator>
          <value>1024</value>
        </condition>
        <condition>
          <parameter>day</parameter>
          <operator>in</operator>
          <value>1..5</value>
        </condition>
      </conditions>
      <action arg1="video">add_medium(arg1)</action>
    </policy_rule>
  </policy>
</policy_document>
```

6.3.3 Goal System Architecture

A stand-alone goal system has been designed to implement the goal-directed approach. The goal system is responsible for all aspects of goal-related processing. Entirely Java-based, it is interfaced with the Accent policy system and the Poppet ontology server via a collection of sockets. Consequently, the location of the goal system is not restricted to the same physical machine as the policy server or Poppet. The flexibility offered by such system distribution is helpful as it allows dedicated resources to be used for applications where greater power is sought. For example, a processor-intensive goal optimisation algorithm necessitates a large amount of memory and CPU time. The goal system might require a stand-alone machine dedicated for this purpose.

The architecture of the goal system is shown in Figure 6.5. In this figure, the previously described policy system and ontology components appear shaded. Customisable goal system components appear with a dashed border.

The Poppet client provides the goal system with an RMI interface to query and

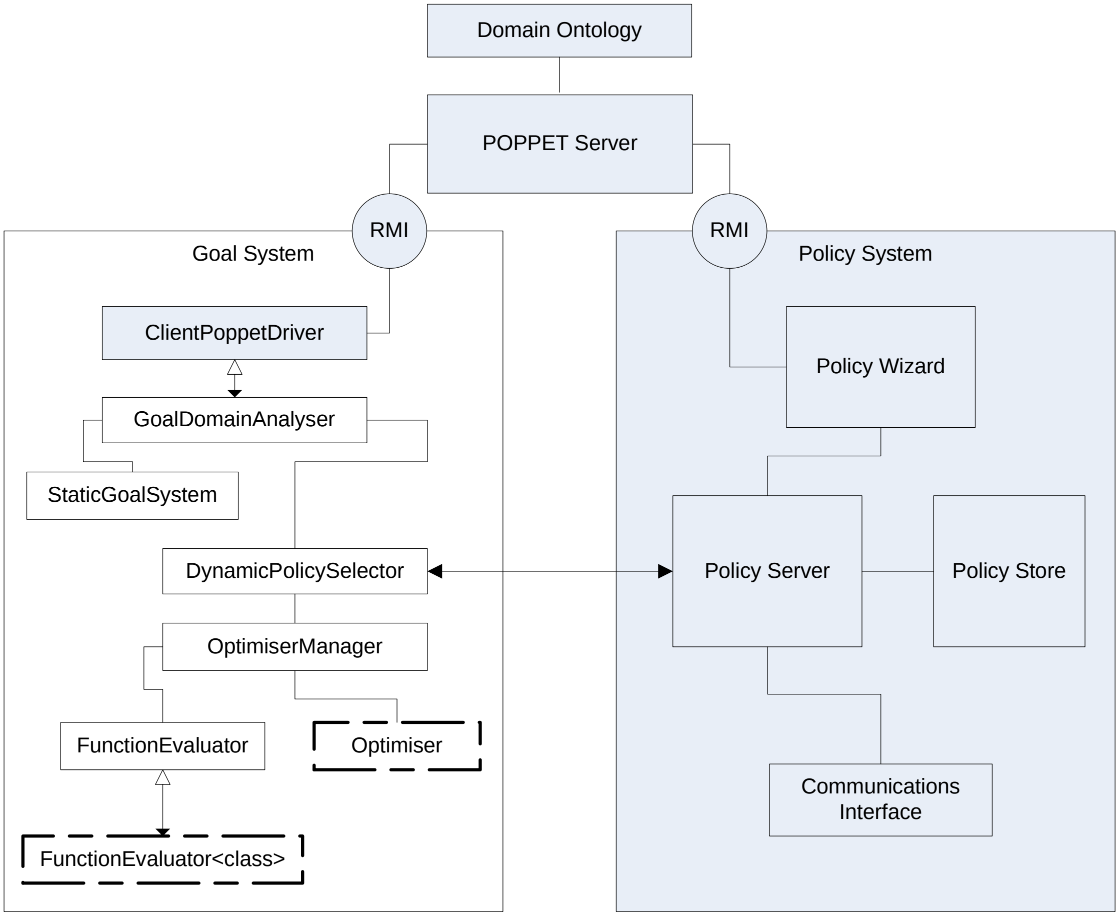


Figure 6.5: Goal System Architecture

retrieve goal-related information from a domain ontology. The goal system is integrated with the existing Accent system via socket connections, allowing both entities to remain independent. The policy server was altered to include additional hooks to the goal system, but is otherwise unaware of goal-related activity. Effectively, the goal system operates as an optional "plug-in" for the Accent system, so that the policy system may operate with or without goal direction as desired. The goal system also utilises the policy store to hold goals and prototypes. However, this is achieved transparently as the goal system communicates only via the policy server. The mechanism used to store and retrieve policies may be altered without affecting goal-related components.

The optimisation algorithm is customisable. The most appropriate algorithm should be implemented to suit the application domain. An algorithm is specified by replacing or altering the class `Optimiser`.

The goal evaluation function for a domain is also customisable, and is defined as a subclass of `FunctionEvaluator`. The subclass must implement the inherited abstract method "evaluate" with suitable code to process the goal evaluation function, including an expression detailing each goal measure complete with any weightings. This method is called to evaluate each combination of policies at run-time. The class must be pre-compiled as it is dynamically loaded at runtime. Dynamic class loading allows the goal evaluation function to be altered without the need to recompile the entire goal system.

Configurable environment variables are defined in a properties file. This includes the policy system host machine and upload port, POPPET connection details, the symbol prefix to identify parameters within goal-related policies (normally "$"), and the "owner" of goal, prototype and goal-related policy documents (e.g. `goalkeeper@cs.stir.ac.uk`). The name and location of the goal evaluation function class must also be defined, for dynamic loading at runtime.

6.3.4 Static and Runtime Procedures

This section provides a step-by-step explanation of how the goal system works. The initial phase of prototype instantiation is described and the complete run-time phase is explained. This also shows where the goal-directed process fits into the context of regular policy server behaviour, describing the process from when a trigger is received to when actions are executed. Concrete examples of this process are detailed in the applications demonstrated in the next two sections.

Offline Prototype Instantiation

The static prototype analysis stage takes a set of goals and a set of prototypes as its primary input. Goals and prototypes are defined as individual XML documents stored as separate files within the policy store. The approach to static prototype analysis and instantiation was discussed in section 6.2.4.

The goal system is invoked independently for the initial stage. Goals and prototypes are uploaded to the policy store via the policy server. The goal system then retrieves and checks each goal, extracts its goal measure and action (maximise or minimise), and retrieves a list of state variables associated with its measure from the domain ontology. Next, each prototype is retrieved from the policy store, and its effects are compared against the list of state variables for each goal measure. Prototypes contributing to at least one goal are instantiated as executable policies. The technical steps involved in prototype instantiation are summarised as follows:

1. A prototype element is cloned, and the element type of the clone is altered to be "policy" rather than "prototype". The prototype is no longer used at this point by the goal system, but persists in the policy store for any future time that goals need to be re-evaluated.
2. The attributes of the `policy` node are modified:
   (a) the phrase "_pol" is appended to the `policy_id` to distinguish this policy from the prototype
   (b) the `changed` attribute is modified to reflect the current date and time
   (c) a `supports_goal` attribute is added containing a comma-separated list of goals this policy supports
3. If all the goals this policy supports have conditions, their conditions are appended to the policy condition block.
4. The new policy is uploaded to the policy store, where it may be triggered and retrieved by the policy server at runtime.

The final step of this process is to create policy variables to store the default value of each state variable using information sourced from the ontology. Each policy variable is uploaded to the policy store and shares the same "owner" as goals, prototypes and goal-related policy documents. At run-time, these policy variables are updated to represent the current state of the system based on event notifications to the policy server.

Run-time Policy Selection and Parameter Optimisation

The goal system comes into play whenever policies are triggered. The run-time algorithm considers all combinations of triggered goal-related policies. Where parameterised policies are encountered, the algorithm considers only the maximum and minimum value for the variable. A policy parameter in this context is a variable that appears at least once in the goal evaluation function.

The range of possible values for a goal-related policy parameter may be extremely large (for example, a variable representing the duration of a call might range from 1 minute to 300 minutes). Given the real-time constraints of the policy system, the consideration of every unique permutation of parameter value for every policy combination is impracticable without the aid of an optimisation algorithm.

For the applications reported at the end of this chapter, it was acceptable to consider only the "extreme" values for a parameter, that is, just the maximum and minimum value. At run time, the maximum and minimum value for the parameter is evaluated and the one that yields the higher result for the goal function is the optimal choice. A simple parameterisation example was given in section 6.2.5. This example considered a parameter with only 3 possible values – the minimum value yielding the highest results for the evaluation function. Where the parameter value range is much wider and the goal evaluation function is far more complex (such as the application scenarios presented later in this chapter), it has been found sufficient to consider just the maximum and minimum values. In general, the nature of the evaluation functions tested so far means either the highest or lowest parameter value yields the best result. Depending on which extreme value is tested, the evaluation result either increases or decreases. Effectively, considering the extreme values gives the best and worst result for the goal evaluation. Choosing a value somewhere in between yields a result somewhere in between these two points. Therefore, the optimal value is one of the extreme values.

There is a clear benefit in processing overhead when considering only the extreme values of a parameter. Suppose the number of parameterised variables within a combination of policies is `vNum`. If there are two parameter values to select from (the maximum value and the minimum value) there exists 2^{vNum} combinations of parameter values for a combination of policies. For example, for a combination of three policies, the number of possible solutions is as follows:

No parameters:

P1 P2 P3

One parameterisable value:

P1 P2 P3 p(max)

P1 P2 P3 p(min)

Two parameterisable values:

P1 P2 P3 p(max) q(max)

P1 P2 P3 p(min) q(max)

P1 P2 P3 p(max) q(min)

P1 P2 P3 p(min) q(min)

The number of solutions using extreme value parameterisation is acceptable for a small number of parameters. However, when this number grows to five parameters, the number of combinations to be evaluated rises to 32. With a relatively small policy set, the time taken to process the set and find the optimal solution outweighs the benefit of run-time goal direction. For the applications tested with this approach, it was deemed unlikely that the number of policies triggered would be large enough to merit a full optimisation algorithm (e.g. a GA), and the number of likely parameter values is usually no more than two at any one time.

The average time to run the policy selection and parameter optimisation has been found in practice to be about five seconds. This is an acceptable overhead for processing goals in real time (since the policy system is triggered in the order of minutes not seconds). There are further steps which might improve performance. It is possible that the same set of policies might be triggered frequently, so caching previous policy selections might save time and increase efficiency. This would involve storing the state of the system, the goal-related policies triggered, and the results of policy selection and parameterisation. Previous selections would only be applicable when the system is in an identical state beforehand.

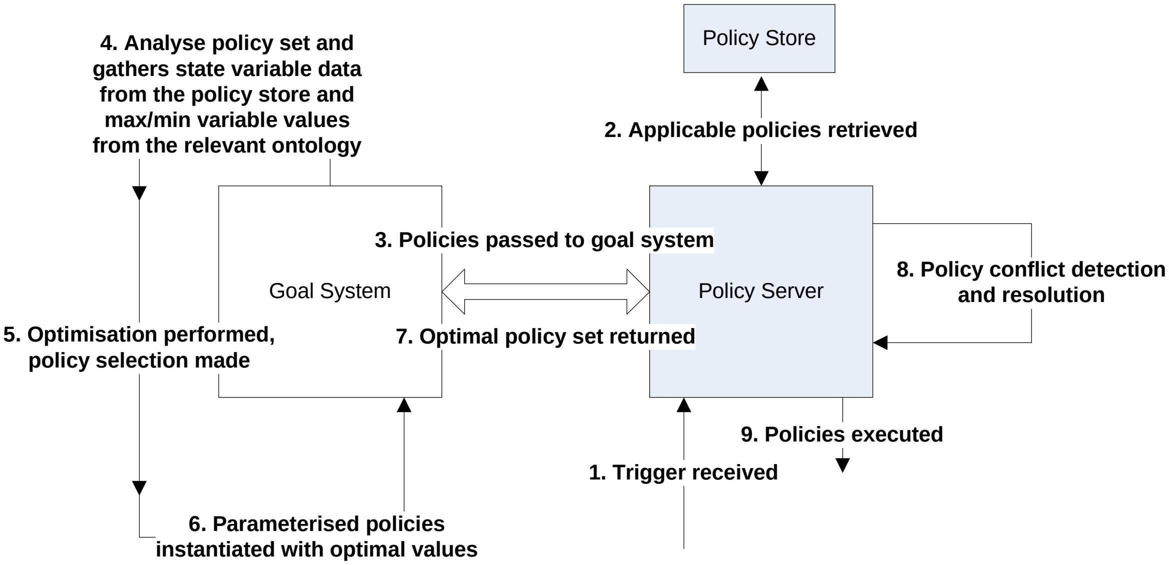

4. Analyse policy set and gathers state variable data from the policy store and max/min variable values from the relevant ontology
Policy Store
2. Applicable policies retrieved
3. Policies passed to goal system
Goal System
Policy Server
8. Policy conflict detection and resolution
5. Optimisation performed, policy selection made
7. Optimal policy set returned
9. Policies executed
1. Trigger received
6. Parameterised policies instantiated with optimal values

Figure 6.6: Runtime Process of Goal Refinement

With reference to Figure 6.6, the key steps of the runtime algorithm are as follows:

1. The policy server receives a trigger.

2. Policies triggered by this information are retrieved. The policy server checks the eligibility of each triggered policy and discards those whose conditions are not met.

3. The policy server passes applicable policies to the goal system. If the goal system is not connected (e.g. networking error or goals are not needed for the domain in question), the policy server continues its normal execution process, skipping to step 8.

4. The goal system separates goal-related policies from user-defined policies. If no goal-related policies are found, the goal system performs no further action and returns the list of policies to the policy server unchanged. Current values of state variables are obtained from the policy store, and their maximum and minimum values obtained from an ontology. The evaluation function class is dynamically loaded.

5. The optimisation algorithm evaluates the set of goal-related policies, and selects the optimal policy combination including any parameter values.

6. Parameterised policies are instantiated with their chosen parameter values if applicable.

7. The goal system returns the optimal set of goal-related policies, together with any user-defined policies originally passed to it.

8. The policy server checks for conflicts between policy actions, resolving these as appropriate using resolution policies.

9. The policy server executes the final set of actions. Following this, the policy server notifies the goal system that policy execution has taken place. The goal system then removes any dynamically instantiated policies created as a result of parameterisation (the goal system keeps a note of the instantiated policy ID's in memory). Original unparameterised prototypes persist as they may be triggered and used again.

6.4 Application 1: Goals for Internet Telephony

The first application of the goal-directed approach is Internet telephony. Section 6.4.1 describes the domain as it is implemented for goal-directed management, while section 6.4.2 lists example goals and prototypes. The process of static and dynamic policy selection is demonstrated using a number of test cases in sections 6.4.3 and 6.4.4 respectively.

6.4.1 Goal Domain

The telephony domain for goal-directed management is based on the existing Appel policy language for call control, with some additional aspects not previously implemented. Specifically, goals and prototypes rely on policy variables to represent state, such as the call duration and the number of calls received. These aspects are implemented using internal timers and policy variables that were included in the revised core Appel language described in section 3.2.

State Variables

The goal domain is an extension of the policy domain for telephony. Prototypes are based on the existing definition of the Appel policy language for call control, including the triggers, conditions and actions described in section 3.2. Goal measures and their relationships with controlled and uncontrolled variables are defined within the ontology for telephony (see section 4.5). The state variables used for the domain are shown in the table below. Each variable has an associated maximum, minimum and default value.

Variable Type	Variable Name	Description	Max Value	Min Value	Default Value
Controlled	bandwidth	The amount of bandwidth allocated to a call at a given time in Kbps.	512	32	64
	call_duration	The length of a call in minutes.	300	1	0
	qos_rate	Quality of Service (QOS) indicator. Numerical measurement representing network quality (transit delay, error rates, etc.). Optionally specified by the caller (10 best, 1 worst).	10	1	10
Uncontrolled	call_cost_rate	Cost of a call per minute in some monetary unit.	50.0	0.0	1.0
	calls_received	The number of calls received by a callee in some defined period (e.g. a day).	500	0	0

Goals, Measures and the Evaluation Function

Based on general knowledge and the capabilities of the policy system implementation, the following four goals are defined for telephony:

G01 Minimise Call Cost

G02 Minimise Interruption Time

G03 Maximise Multimedia Use

G04 Maximise Bandwidth Use

These goals support telephony aspects from a high-level management view. The choice of goals aims to strike a balance between the financial impact of network use and call quality in terms of available call features and bandwidth. These goals served as an initial test-bed for developing and testing the goal-directed approach. G01 aims to keep the cost of calls as low as possible, and is measured by the cost of calls and their duration. G02 aims to reduce the time a callee is interrupted, perhaps during certain periods in the day, or when the callee is in scheduled meetings or at lunch. This goal is measured by the number of calls received in a certain period and their duration. G03 and G04 aim to maximise the use of multimedia and bandwidth. G03 is measured by the bandwidth used and the quality rating expected by call parties. G04 is measured by bandwidth and the duration of a call.

The goal evaluation function is the sum of these four goals, though other combinations such as a weighted sum could easily have been used. Minimising a goal measure means use of a minus ("−") sign and maximising means use of a plus sign ("+"). The goals and measures are summarised in the goal evaluation function as follows:

Evaluation Function = G01 + G02 + G03 + G04

=− Call Cost − Interruption Time + Multimedia Use + Bandwidth Use

=− (call_cost_rate * call_duration) − (calls_received * call_duration)
+ (bandwidth * qos_rate) + (bandwidth * call_duration)

6.4.2 Goals and Prototypes

This section outlines goals and prototype policies for telephony, using pseudocode for readability. Applicable goal conditions are preceded with "Provided". Effects are preceded by "For". The triggers, conditions, actions and effects are given for each prototype. A selection of goals and prototypes include XML notation to clarify their syntax (the format of goals and prototypes was explained previously in section 6.3.2). A high-level description of each goal/prototype is also provided to outline the purpose of their use.

Goals

G01: Minimise Call Cost

The goal is to minimise call cost. The goal conditions state that it must be a weekday (Monday (1) to Friday (5)) and the bandwidth must be greater than or equal to 256 Kbps.

Provided the day is in 1..5 and

Provided the bandwidth ≥ 256

Minimise Call Cost

```
<?xml version=″1.0″ encoding=″UTF-8″?>
<policy_document
    xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
    xsi:noNamespaceSchemaLocation=
    ″http://www.cs.stir.ac.uk/schemas/appel_goal.xsd″>
  <goal
      owner=″goalkeeper@cs.stir.ac.uk″
      applies_to=″goalkeeper@cs.stir.ac.uk″
      id=″Minimise Call Cost″
      enabled=″true″
      changed=″2008-04-21T10:20:59″>
    <policy_rule>
      <conditions>
        <and/>
        <condition>
          <parameter>day</parameter>
          <operator>in</operator>
          <value>1..5</value>
        </condition>
        <condition>
          <parameter>bandwidth</parameter>
          <operator>ge</operator>
          <value>256</value>
        </condition>
      </conditions>
      <action arg1=″call_cost″>minimise(arg1)</action>
    </policy_rule>
  </goal>
</policy_document>
```

G02: Minimise Interruption Time

The goal is to minimise interruption time to the callee. The goal condition is that it must be lunchtime (between 12:30 and 2pm).

Provided the hour is in 12:30:00..14:00:00

Minimise Interruption Time

G03: Maximise Multimedia Use

The goal is to maximise multimedia use. The goal condition is that it must be a weekday (Monday (1) to Friday (5)).

Provided the day is in 1..5

Maximise Multimedia Use

G04: Maximise Bandwidth Use

The goal is to maximise bandwidth use. There are no conditions on this goal.

Maximise Bandwidth Use

```
<?xml version=″1.0″ encoding=″UTF-8″?>
<policy_document
    xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
    xsi:noNamespaceSchemaLocation=″http://www.cs.stir.ac.uk/schemas/appel_goal.xsd″>
  <goal
      owner=″goalkeeper@cs.stir.ac.uk″
      applies_to=″goalkeeper@cs.stir.ac.uk″
      id=″Maximise Bandwidth Use″
      enabled=″true″
      changed=″2008-04-21T10:20:59″>
    <policy_rule>
      <action arg1=″bandwidth_use″>maximise(arg1)</action>
    </policy_rule>
  </goal>
</policy_document>
```

Prototypes

P01: Add Video On Bandwidth Request

For requests for bandwidth less than 700 Kbps, and when the current medium is audio, the bandwidth request is confirmed and video is added to the call. This prototype has the effect of incrementing the bandwidth by a fixed amount of 256 Kbps.

When there is a bandwidth request

If the current bandwidth is < 700 and the current medium is audio

Do confirm the bandwidth request and add video

For bandwidth += 256

P02: Add Video To Incoming Call

Adds video capability to the call providing the bandwidth is less than 1024 Kbps and the medium is initially set to audio. This prototype has the effect of incrementing the bandwidth by a fixed amount of 256 Kbps to support video capability.

When there is an incoming call
If the current bandwidth $<$ 1024 and the current medium is audio
Do add video
For bandwidth += 256

P03: Add Whiteboard To Department Calls

Adds a whiteboard to incoming calls where the caller is in a particular (local) domain and the medium being used is not already whiteboard. This prototype has the effect of incrementing bandwidth for the call by a fixed amount of 500 Kbps – an assumed amount reserved for a whiteboard.

When there is an incoming call
If the caller is @*cs.stir.ac.uk* and
  the current media do not include whiteboard
Do add whiteboard
For bandwidth += 500

P04: Disconnect After 60 Minutes

Using an internal call timer function, the call is disconnected once a call has been active for 1 hour. This prototype has the effect of limiting the duration of a call to 60 minutes.

When the call timer expires
If the call time is 60 minutes
Do Disconnect the call
For call_duration = 60

P05: Extend Call On Bandwidth Request

Confirms bandwidth requests for 512 Kbps and sets a call timer to expire after 10 minutes. This prototype has the effect of setting the bandwidth to the requested amount of 512 Kbps and extending the call duration by ten minutes. The effect is to increase the current call duration (+=) (which may have been set by a previous policy), rather than set the call duration to ten minutes (+).

When there is a bandwidth request

If the bandwidth request is 512

Do confirm the bandwidth request and set the call timer for 10 minutes

For bandwidth = 512 and call_duration += 10

```
<?xml version=″1.0″ encoding=″UTF-8″?>
<policy_document
      xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
      xsi:noNamespaceSchemaLocation=
      ″http://www.cs.stir.ac.uk/schemas/appel_prototype.xsd″>
  <prototype
        owner=″goalkeeper@cs.stir.ac.uk″
        applies_to=″@cs.stir.ac.uk″
        effects=″call_duration += 10,bandwidth = 512″
        id=″Extend Call On Bandwidth Request″
        enabled=″true″
        changed=″2008-04-21T10:20:59″>
    <policy_rule>
      <trigger>bandwidth_request</trigger>
      <condition>
        <parameter>bandwidth</parameter>
        <operator>eq</operator>
        <value>512</value>
      </condition>
      <actions>
        <and/>
        <action arg1=″call_cut″ arg2=″10″>set_timer(arg1,arg2)</action>
        <action>confirm_bandwidth</action>
      </actions>
    </policy_rule>
  </prototype>
</policy_document>
```

P06: Forward Call If Busy

Forwards an incoming call to the address stored for the variable "secretary" when the callee status is set to busy. This prototype has the effect of reducing the number of calls that might have been answered by the callee.

> **When** there is an incoming call
>
> **If** my status is set to busy
>
> **Do** forward the call to :secretary
>
> **For** calls_received $-=$ 1

P07: Limit Expensive Call Time

Limits the duration of calls to 15 minutes if the cost rate is greater than 40 pence. This prototype has the effect of setting the call duration to 15.

> **When** there is an incoming call
>
> **If** the cost rate $\geq$ 0.40
>
> **Do** set the call timer for 15 minutes
>
> **For** call_duration = 15

P08: Limit International Call Time

Limits an incoming international call to 20 minutes. This prototype has the effect of limiting the call duration to 20 minutes after the call starts.

> **When** there is an incoming call
>
> **If** the call type is "international"
>
> **Do** set the call timer for 20 minutes
>
> **For** call_duration = 20

P09: Parameterised Call Duration

Limits the duration of an incoming call when the bandwidth is greater than or equal to 512 Kbps. The duration to which the call timer should be set is a parameter (denoted by the "$" symbol). This prototype has the effect of setting the call duration to a fixed value decided at run time.

When there is an incoming call

If the bandwidth ≥ 512

Do set the call timer for (\$T)

For call_duration = \$T

```
<?xml version=″1.0″ encoding=″UTF-8″?>
<policy_document
     xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
     xsi:noNamespaceSchemaLocation=
     ″http://www.cs.stir.ac.uk/schemas/appel_prototype.xsd″>
  <prototype
       owner=″goalkeeper@cs.stir.ac.uk″
       applies_to=″@cs.stir.ac.uk″
       effects=″call_duration = $T″
       id=″Parameterised Call Duration″
       enabled=″true″
       changed=″2008-04-21T10:20:59″>
    <policy_rule>
      <trigger>connect_incoming</trigger>
      <condition>
        <parameter>bandwidth</parameter>
        <operator>ge</operator>
        <value>512</value>
      </condition>
      <action arg1=″call_cut″ arg2=″$T″>set_timer(arg1,arg2)</action>
    </policy_rule>
  </prototype>
</policy_document>
```

P10: Reject High Bandwidth

Rejects bandwidth requests greater than or equal to 512 Kbps. This prototype has the effect of reducing bandwidth to zero. Note that rejecting a bandwidth request at call set-up means the call will not be allowed.

When there is a bandwidth request

If the bandwidth request is ≥ 512

Do reject the bandwidth request

For bandwidth = 0

P11: Reject Personal Calls If Busy

Rejects personal calls that occur when the callee is busy and plays a specified audio message to the caller. This prototype has the effect of reducing the number of calls that might have been answered by the callee.

When there is an incoming call

If the callee status is "busy" and the call_type is "personal"

Do reject the call and play the sound clip "busy.wav"

For calls_received $-=$ 1

6.4.3 Static Prototype Selection

The static selection process looks at each prototype and compares its effects against the controlled and uncontrolled values within each goal measure. If a prototype affects a variable used within a goal measure, it potentially contributes to that goal. The results of static prototype filtering against goals is shown in Figure 6.7. This shows the prototypes that potentially contribute to each goal, with a telephone symbol indicating applicability.

Prototype	**Call Cost**	**Interruption Time**	**Multimedia Use**	**Bandwidth Use**
Add Video On Bandwidth Request			☎	☎
Add Video To Incoming Call			☎	☎
Add Whiteboard To Department Calls			☎	☎
Disconnect After 60 Minutes	☎	☎		☎
Extend Call On Bandwidth Request	☎	☎	☎	☎
Forward Call If Busy		☎		
Limit Expensive Call Time	☎	☎		☎
Limit International Call Time	☎	☎		☎
Parameterised Call Duration	☎	☎		☎
Reject High Bandwidth			☎	☎
Reject Personal Calls If Busy		☎		

Figure 6.7: Static Telephony Prototype Selection

The following two examples demonstrate policy instantiation based on Figure 6.7. The examples use XML to explain the technical alterations to policy documents. For clarity, some document headers and other policy tags are omitted. Example 1 details prototype instantiation in support of a single goal and its conditions, while Example 2 shows the process for a different prototype supporting multiple goals.

Example 1: Call Forwarding Policy

This example shows the instantiation of prototype P06, which forwards a call when the callee is busy. The effect of this prototype is to decrement the calls received counter by one. The prototype supports one goal, which has one goal condition. The XML for this prototype is as follows.

```
<prototype
    owner="goalkeeper@cs.stir.ac.uk"
    applies_to="@cs.stir.ac.uk"
    effects="calls_received -= 1"
    id="Forward Call If Busy"
    enabled="true"
    changed="2008-04-21T10:20:59">
  <policy_rule>
    <trigger>call_incoming</trigger>
    <condition>
      <parameter>status</parameter>
      <operator>eq</operator>
      <value>busy</value>
    </condition>
    <action arg1=":secretary">
      forward_to(arg1)
    </action>
  </policy_rule>
</prototype>
```

From Figure 6.7, this prototype contributes to the goal “Minimise Interruption Time”. The condition on this goal states that the time must be between 12:30pm and 2pm. This condition is copied into the instantiated prototype, combined with an “and” operator. The new policy may only be executed when the original prototype condition (status is busy) and the goal condition (time is 12:30pm-2pm) are both true. The new policy is shown below in XML. The policy has a new “id” (line 6) to distinguish it from the prototype, and a new attribute listing the goal it supports (line 8). The condition block contains the original prototype condition (lines 13-17) and the inherited goal condition (lines 18-22).

```
<policy_document>
  <policy
      applies_to=″@cs.stir.ac.uk″
      changed=″2008-09-03T11:50:33″
      effects=″calls_received -= 1″
      enabled=″true″
      id=″Forward Call If Busy_pol″
      owner=″goalkeeper@cs.stir.ac.uk″
      supports_goal=″Minimise Interruption Time″>
    <policy_rule>
      <trigger>incoming_call</trigger>
      <conditions>
        <and/>
        <condition>
          <parameter>status</parameter>
          <operator>eq</operator>
          <value>busy</value>
        </condition>
        <condition>
          <parameter>hour</parameter>
          <operator>in</operator>
          <value>12:30:00..14:00:00</value>
        </condition>
      </conditions>
      <action arg1=″:secretary″>
        forward_to(arg1)
      </action>
    </policy_rule>
  </policy>
</policy_document>
```

Example 2: Parameterised Call Duration Policy

This example shows the instantiation of prototype P09 (“Parameterised Call Duration”) to support multiple goals. The XML for this prototype is as follows:

```
<prototype
    owner=″goalkeeper@cs.stir.ac.uk″
    applies_to=″@cs.stir.ac.uk″
    effects=″call_duration = $T″
    id=″Parameterised Call Duration″
    enabled=″true″
    changed=″2008-04-21T10:20:59″>
  <policy_rule>
    <trigger>call_incoming</trigger>
    <condition>
      <parameter>bandwidth</parameter>
      <operator>ge</operator>
      <value>512</value>
    </condition>
    <action arg1=″call_cut″ arg2=″$T″>
      set_timer(arg1,arg2)
    </action>
  </policy_rule>
</prototype>
```

From Figure 6.7, this prototype contributes to three goals: Minimise Call Cost (G01), Minimise Interruption Time (G02) and Maximise Bandwidth Use (G04). The first two goals have conditions but the third does not. Consequently, no goal conditions are copied when the prototype is instantiated.

The new policy is shown below in XML. The policy has a new id (line 6) and a new attribute listing the goals it supports (line 7). The policy condition block contains the original prototype condition only and has not been altered. Note that the parameter value “T”, representing the period of time after which to disconnect the call, is not given a value until runtime. The parameter appears in the effect attribute (line 4) and the policy action argument (line 15).

```
 0 <policy_document>
 1   <policy owner=″goalkeeper@cs.stir.ac.uk″
 2       applies_to=″@cs.stir.ac.uk″
 3       changed=″2008-09-03T11:50:34″
 4       effects=″call_duration = $T″
 5       enabled=″true″
 6       id=″Parameterised Call Duration_pol″
 7       supports_goal=″Minimise Call Cost,Minimise Interruption Time,Maximise Bandwidth Use″>
 8     <policy_rule>
 9       <trigger>call_incoming</trigger>
10       <condition>
11         <parameter>bandwidth</parameter>
12         <operator>ge</operator>
13         <value>512</value>
14       </condition>
15       <action arg1=″call_cut″ arg2=″$T″>
16         set_timer(arg1,arg2)
17       </action>
18     </policy_rule>
19   </policy>
20 </policy_document>
```

6.4.4 Dynamic Refinement

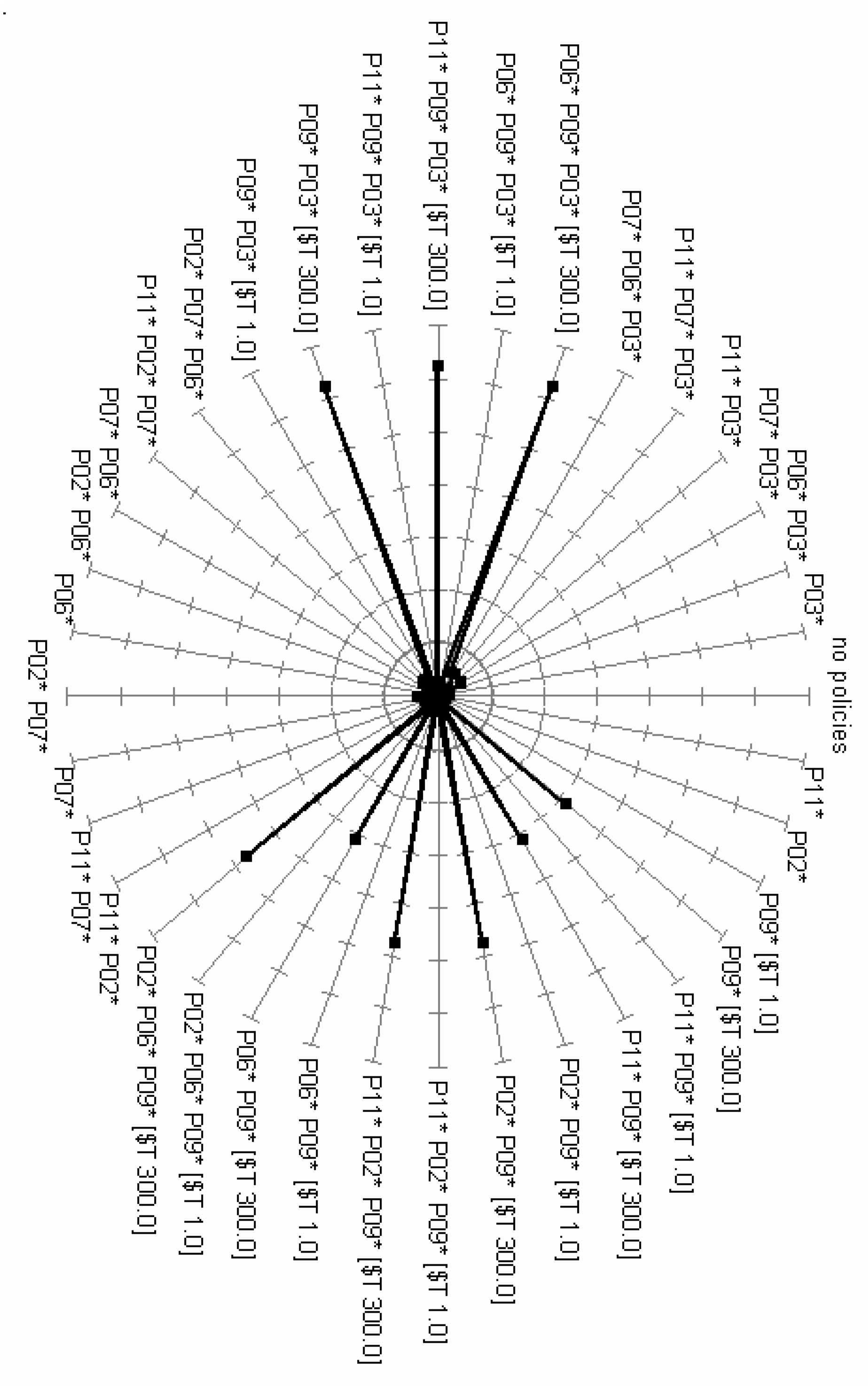
This section presents the results of run-time policy selection for telephony through an example with four scenarios: an incoming personal call, an incoming international call, a normal bandwidth request and a high bandwidth request. Each scenario describes an event under different conditions, triggering alternative policy sets in each case.

The results of run-time policy selection for each scenario are displayed in a radar graph. A radar graph is a polygon with policy combinations plotted on its vertexes using separate axes stemming from the same (central) point. The relative position and angle of the axes is not important. For the graphs shown, plotted values and numerical labels are omitted as the relationship between each policy combination is more meaningful in this context than the values obtained. Policies are labelled using the number of the prototype they were derived from, followed by an asterisk "*". For example, the policy P12* represents the policy derived from the prototype P12. The graph compares each non-conflicting policy combination in a triggered set using the value each yields when evaluated against the goal function. Conflicting policy combinations are policy sets that modify a common state variable, which makes it ineffective to evaluate the policy set against the goal function as there is no single predicted value for the state variable concerned. Taking the policy set in scenario 1 as an example, policies P07* and P09* both modify the variable "call_duration" and are therefore considered to be in conflict. During run-time policy selection, no combinations of P07* and P09* are considered, and these policies do not appear together in the combinations noted in the results graph. The best policy combination is the one that yields the highest value – identified by the value that is closest to the outermost edge of the graph.

Scenario 1: Incoming Personal Call

The first scenario is an incoming personal call when the callee is busy. The bandwidth required is 512 Kbps, the caller and callee are in the same local domain, the medium is "audio", and the call duration is zero as the call has yet to begin. Six policies are applicable under these conditions, including a parameterised policy. An optimal value for the parameter `$T` is allocated during the selection process. The extreme values for this parameter (which represents the call_duration) are 300.0 minutes and 1.0 minute. The result of the policy selection process for all non-conflicting policy combinations is shown in Figure 6.8.

The optimal policy selection is the set of policies "P06* P09* P03*" where the best value for the call_duration parameter is 300.0. P06* forwards the incoming call (as the callee is busy), P09* limits the call duration (the parameter $T is set to 300.0) and P03* adds a whiteboard for use during the call.

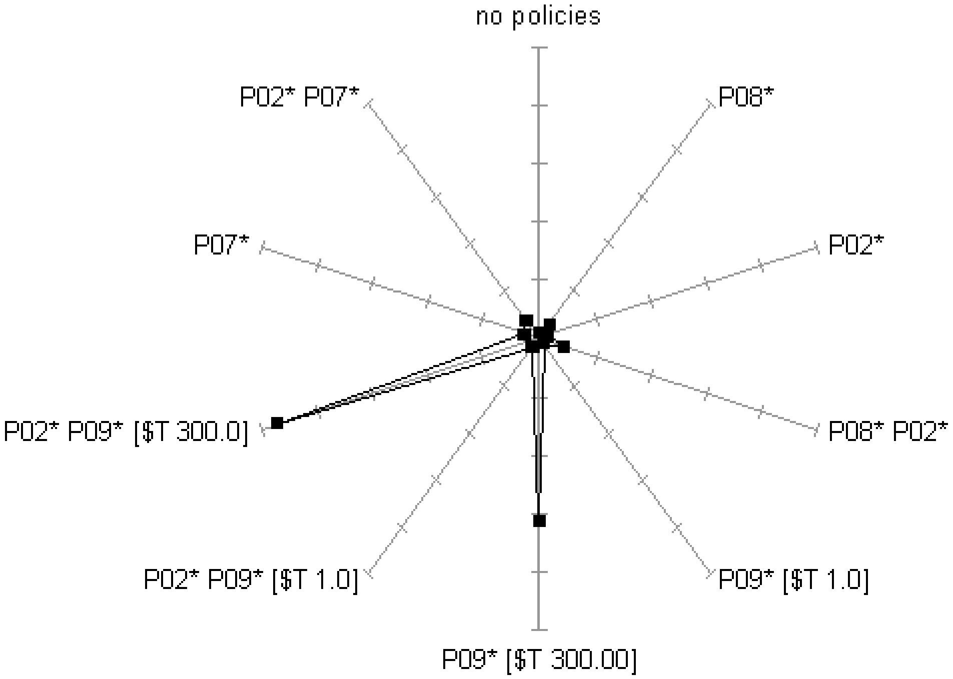


P02*	Add Video To Incoming Call
P03*	Add Whiteboard To Department Calls
P06*	Forward Call If Busy
P07*	Limit Expensive Call Time
P09*	Parameterised Call Duration
P11*	Reject Personal Calls If Busy

Figure 6.8: Scenario 1: Policy Selection for an Incoming Personal Call

Scenario 2: Incoming International Call

This scenario shows the policy selection for an incoming international call, where the caller is not in the same domain as the callee. The bandwidth requested is 512 Kbps, the medium is “audio”, the callee status is “available”, and the call duration is zero as the call has yet to begin. There are four policies applicable under these conditions, including a parameterised policy. An optimal value for the parameter `$T` is allocated during the selection process. The extreme values for this parameter (which represents the call_duration) are 300.0 minutes and 1.0 minute. The result of the policy selection process for all non-conflicting policy combinations is shown in Figure 6.9.



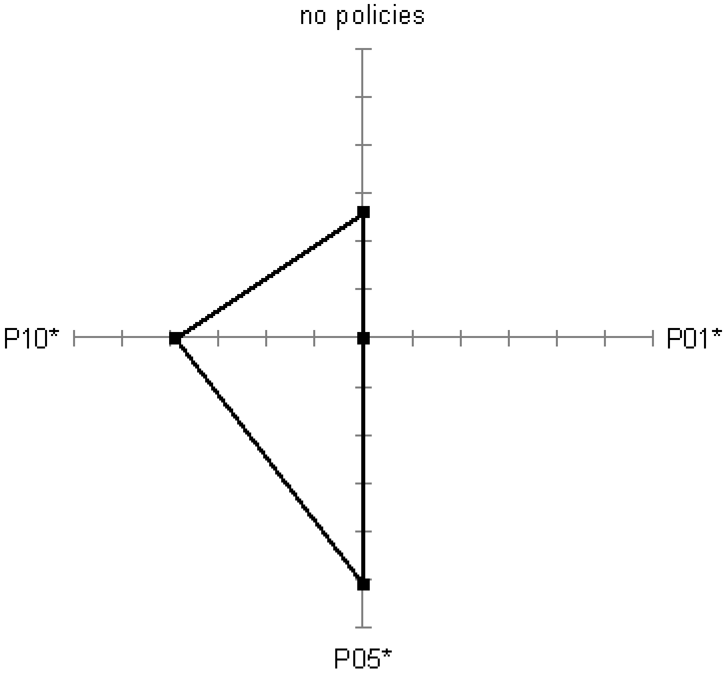
P02*	Add Video To Incoming Call
P07*	Limit Expensive Call Time
P08*	Limit International Call Time
P09*	Parameterised Call Duration

Figure 6.9: Scenario 2: Policy Selection for an Incoming International Call

The best policy selection is P09* and P02* with a parameter value of 300.0 for the variable representing the duration of the call in P09*. P09* limits the call time to the chosen parameter value of 300 minutes, and P02* adds video to the call.

Scenario 3: Normal Bandwidth Request

This scenario describes a request for bandwidth. The requested amount is 512 Kbps and the current medium is “audio”. As this trigger occurs during the call setup phase, the call duration is zero. There are three policies applicable under these conditions, and the result of the policy selection process for all non-conflicting policy combinations is shown in Figure 6.10.



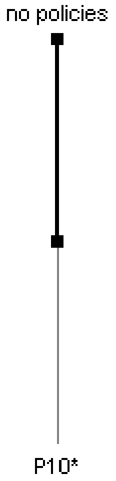
P01* Add Video On Bandwidth Request
P05* Extend Call On Bandwidth Request
P10* Reject High Bandwidth

Figure 6.10: Scenario 3: Policy Selection for a Normal Bandwidth Request

As the effects of all three policies modify bandwidth, they may only be selected individually without conflict. The optimal policy to select is P05*, which adds ten minutes to the current call duration. This is favoured over rejecting the bandwidth altogether in the case of P10* or adding video in the case of P01*.

Scenario 4: High Bandwidth Request

This scenario deals with a request for a high amount of bandwidth – 1000 Kbps. All other state variables are the same as for scenario three. Under these conditions, only one policy is applicable, “Reject High Bandwidth”: this is P10* in the results graph shown in Figure 6.11.



P10* Reject High Bandwidth

Figure 6.11: Scenario 4: Policy Selection for a High Bandwidth Request

With only one goal-related policy, there are just two options to consider: the result of the goal evaluation function for no policies (current state unchanged), and the result for executing the single policy. The optimal result in this case is to select no policies. Selecting P10* negatively contributes to the system goals when evaluated across all goal measures. Although the bandwidth request is for a high amount, rejecting the bandwidth (say, to minimise the call cost measure) has been deemed less helpful to the system goals as a whole since it contributes more toward maximising bandwidth and multimedia use.

6.5 Application 2: Goals for Sensor Networks

The second application of the goal-directed approach is sensor network and wind turbine management. Section 6.5.1 describes the domain, while section 6.5.2 lists the goals and prototypes devised. The process of static and dynamic policy selection is demonstrated in sections 6.5.3 and 6.5.4 respectively.

6.5.1 Goal Domain Information

The domain for goal-directed management of sensor network and wind turbine management is based on the Appel policy language specialisation described in section 3.2. As explained in section 2.2.3, a wind turbine is continually monitored using a controller within the nacelle. Communication between a turbine controller and the policy system is assumed via some network link. Goals for wind turbine management assume that the parameters modifiable by the turbine controller may be set via policy actions. It is also assumed that these values are periodically reported to the policy system so as to trigger policies. These assumptions were taken into account when compiling goals and goal measures.

State Variables

Goal measures and their relationships with controlled and uncontrolled variables are defined within the ontology for sensor networks (discussed previously in section 4.6). Figure 6.12 details the controlled state variables, and Figure 6.13 the uncontrolled state variables. Each variable has an associated maximum, minimum and default value.

Referring to Figures 6.12 and 6.13, sampling and reporting frequencies are integer values representing time intervals in minutes. The sampling frequency is the interval between each measurement, and the reporting frequency is the interval between each report of the measurements made. Note that the "rainfall_sampling_freq" is either 1 (on) or 0 (off) as rainfall is measured continually when the rainfall sensor is switched on. Variables that refer to a "drain" represent the power consumption of a particular component. Zero indicates no drain (the sensor is switched off) and 1 indicates the sensor is operational. The maximum, minimum and default values for turbine components are estimations made for the purposes of simulation and are intended only as a plausible guide.

Goals, Measures and the Evaluation Function

The following goals devised support both sensor and wind turbine management views. There are five goals, based on collaborative input from the Prosen project and a study of wind farm and turbine operation:

G01 Minimise Sensor Battery Drain

G02 Maximise Sensor Data Yield

G03 Minimise Turbine Component Damage

G04 Maximise Energy Generated From Turbines

G05 Maximise Turbine Monitoring

G01 and G02 are specific to sensor network nodes, while G03, G04 and G05 support wind turbine management and monitoring. Each set of goals is achieved using policies specific to either sensor network configuration or turbine management respectively. Consequently, the measure for each goal refers to either sensor node or wind turbine variables. G01 aims to get the most data from the sensor network in terms of the measurements sampled from each sensor and the frequency of sensor reports to the policy system. G02 aims to conserve battery power sensibly when the power is falling, helping prolong sensor node life until it is serviced. This goal measures power consumption by the sensor sampling and reporting rates as well as the status of each sensor. G03 aims to limit and prevent mechanical damage to wind turbines. Factors contributing to this goal measure include vibrations in the rotor blades and nacelle, component temperatures, brake status, rotor speed, and the number of power cable twists within the tower. G04 aims to manage the system so maximum power is generated from a turbine, wind permitting. It is measured through consideration of possible power output and the power drain on the turbine from internal components. G05 aims to detect potential faults or significant conditions through increased monitoring of its

Controlled Variables

Variable Name	Description	Max	Min	Default
		Value	Value	Value
anemometer_reporting_freq	Every x minutes	1	60	5
temperature_reporting_freq	Every x minutes	1	60	5
rainfall_reporting_freq	Every x minutes	1	60	10
humidity_reporting_freq	Every x minutes	1	60	15
soilmoisture_reporting_freq	Every x minutes	1	60	15
anemometer_sampling_freq	Every x minutes	1	15	5
temperature_sampling_freq	Every x minutes	1	15	5
rainfall_sampling_freq	On or off	1	0	1
humidity_sampling_freq	Every x minutes	1	20	5
soilmoisture_sampling_freq	Every x minutes	1	15	5
anemometer_drain	On or off	1	0	1
temperature_drain	On or off	1	0	1
rainfall_drain	On or off	1	0	1
humidity_drain	On or off	1	0	1
soilmoisture_drain	On or off	1	0	1
gsm_modem_drain	On or off	1	0	1
rotor_speed	RPM	25	0	0
blade_pitch_angle	degrees	90	0	0
yaw_angle	degrees	180	0	0
generator_cooling_fan_speed	RPM	5000	0	0
rotor_brake_status	On or off	1	0	0
yaw_brake_status	On or off	1	0	0
internal_heater_power	Volts (V)	230	0	0
gearbox_bearing_report_freq	Every x minutes	1	60	10
blade_vibration_report_freq	Every x minutes	1	60	10
generator_report_freq	Every x minutes	1	60	10
turbine_anemometer_report_freq	Every x minutes	1	60	5
nacelle_temperature_report_freq	Every x minutes	1	60	5
nacelle_vibration_report_freq	Every x minutes	1	60	10
blade_pitch_report_freq	Every x minutes	1	60	10

Figure 6.12: Sensor and Turbine Controlled Variables

Uncontrolled Variables				
Variable Name	**Description**	**Max Value**	**Min Value**	**Default Value**
sensor_battery_voltage	Volts(V)	12	0.0	12
wind_speed	m/s	50	0	0
rainfall	mm/hour	100	0	0
humidity	%	100	0	0
temperature	degrees Celsius	40	-20	15
soil_moisture	mho	10^{-4}	10^{-7}	10^{-6}
generator_voltage	Volts(V)	690	0	0
generator_temperature	degrees Celsius	100	-10	30
gearbox_oil_temperature	degrees Celsius	100	-10	30
gearbox_bearings_temperature	degrees Celsius	100	-10	30
power_cable_twists	no. of full twists	10	0	0
wind_direction	degrees	360	0	0
wind_speed_turbine	m/s	100	0	1
blade_vibration_size	mm	100	0	0
blade_vibration_frequency	Hertz (Hz)	100	0	0
nacelle_vibration_size	mm	100	0	0
nacelle_vibration_frequency	Hertz (Hz)	100	0	0
rotor_brake_lining_thickness	mm	10	2	10
nacelle_temperature	degrees Celsius	40	-20	15

Figure 6.13: Sensor and Turbine Uncontrolled Variables

components. This goal is measured using the reporting frequencies for monitored values of turbine components.

The goal evaluation function is the weighted sum of these five goals, although a more complex combination could be used. Minimising a goal measure is associated with a minus ("−") sign, and maximising is associated with a plus sign ("+"). Weights are utilised to counteract situations where altering a single variable makes no difference to the function value. This occurs when a variable change in a goal measure being maximised is cancelled out in another measure being minimised. Without weightings, there is a risk that no policies are selected as the function value appears not to change.

Weightings have been allocated assuming that, for sensor network nodes, battery drain is more important than data yield (0.7 vs. 0.3), and for wind turbine management, preventing component damage is given priority over energy yield and turbine monitoring, which are deemed equally important (0.4 vs. 0.3 vs. 0.3). In this case, the allocated weights sum to 1, but as the weights are relative this may not necessarily be the case in every domain. As goals for sensor networks and goals for turbine management do not share any common state variable in their measures, the two groups of goals are not likely to conflict and can be weighted independently. The goals, measures and their weightings are shown in the evaluation function overleaf.

Evaluation function:

= G01 + G02 + G03 + G04 + G05

= − (0.7 * battery_drain) + (0.3 * data_yield) − (0.4 * component_damage)
+ (0.3 * energy_generated) + (0.3 * turbine_monitoring)

= − (0.7 *
(gsm_modem_drain
+ anemometer_drain * (1/anemometer_sampling_freq + 1/anemometer_reporting_freq)
+ temperature_drain * (1/temperature_sampling_freq + 1/temperature_reporting_freq)
+ rainfall_drain * (1/rainfall_sampling_freq + 1/rainfall_reporting_freq)
+ humidity_drain * (1/humidity_sampling_freq + 1/humidity_reporting_freq)
+ soilmoisture_drain * (1/soilmoisture_sampling_freq + 1/soilmoisture_reporting_freq)))

+ (0.3 *
(1/anemometer_sampling_freq + 1/anemometer_reporting_freq
+ 1/temperature_sampling_freq + 1/temperature_reporting_freq
+ 1/rainfall_sampling_freq + 1/rainfall_reporting_freq
+ 1/humidity_sampling_freq + 1/humidity_reporting_freq
+ 1/soilmoisture_sampling_freq + 1/soilmoisture_reporting_freq))

− (0.4 *
(rotor_brake_status + blade_pitch_angle + rotor_speed + blade_vibration_size
+ blade_vibration_frequency + yaw_brake_status − yaw_angle
− rotor_brake_lining_thickness + nacelle_vibration_size + nacelle_vibration_frequency
+ power_cable_twists))

+ (0.3 *
(rotor_brake_status + blade_pitch_angle + rotor_speed − yaw_angle
− generator_cooling_fan_speed/100 − internal_heater_drain/100))

+ (0.3 *
((1/gearbox_bearing_report_freq) + (1/blade_vibration_report_freq)
+ (1/generator_report_freq) + (1/turbine_anemometer_report_freq)
+ (1/nacelle_temperature_report_freq) + (1/nacelle_vibration_report_freq)
+ (1/blade_pitch_report_freq)))

6.5.2 Implemented Goals and Prototypes

This section outlines goals and prototype policies for sensor and turbine management, using pseudocode for readability. Applicable goal conditions are preceded with "Provided". Effects are preceded by "For". The triggers, conditions, actions and effects are given for each prototype. A selection of goals and prototypes include XML notation to clarify their syntax (the format of goals and prototypes was explained previously in section 6.3.2). A high-level description of each goal/prototype is also provided.

Goals

G01: Maximise Data Yield from the Sensor Network

This goal aims to gather as much information as possible about each sensor and the environment they monitor. This might be achieved through actions such as increasing the sampling and reporting frequencies of individual sensors.

Maximise Data Yield

```
<?xml version=″1.0″ encoding=″UTF-8″?>
<policy_document
    xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
    xsi:noNamespaceSchemaLocation=″http://www.cs.stir.ac.uk/schemas/appel_goal.xsd″>
  <goal
      owner=″goalkeeper@cs.stir.ac.uk″
      applies_to=″goalkeeper@cs.stir.ac.uk″
      id=″Maximise Data Yield″
      enabled=″true″
      changed=″2008-04-21T10:20:59″>
    <policy_rule>
      <action arg1=″data_yield″>maximise(arg1)</action>
    </policy_rule>
  </goal>
</policy_document>
```

G02: Minimise Sensor Node Battery Drain

This goal aims to prolong sensor node life by minimising the battery drain where possible. This might be achieved by switching off particular sensors when they are not required and altering sensor sampling and reporting frequencies to reduce the battery drain from sampling and transmission of data respectively.

Minimise Battery Drain

G03: Minimise Turbine Component Damage and Failure

This goal aims to limit and prevent mechanical damage to turbines. This might be achieved by lowering vibrations in the blades and nacelle, taking action when component temperatures become too high, taking preventative action when brakes show stress or signs of failure and avoiding unnecessary strain on mechanical components when power output is very low.

Minimise Component Damage

G04: Maximise Energy Generated From A Turbine

This goal aims to configure components so that maximum power is generated (wind permitting). This might be achieved through actions such as altering rotor blade pitch and rotor yaw, or choosing to keep the turbine operational in low winds rather than opt for a shutdown.

Provided the generator voltage $\leq$ 690V

Maximise Energy Generated

```
<?xml version=″1.0″ encoding=″UTF-8″?>
<policy_document
    xmlns:xsi=″http://www.w3.org/2001/XMLSchema-instance″
    xsi:noNamespaceSchemaLocation=″http://www.cs.stir.ac.uk/schemas/appel_goal.xsd″>
  <goal
      owner=″goalkeeper@cs.stir.ac.uk″
      applies_to=″goalkeeper@cs.stir.ac.uk″
      id=″Maximise Energy Generated″
      enabled=″true″
      changed=″2008-04-21T10:20:59″>
    <policy_rule>
      <action arg1=″energy_generated″>maximise(arg1)</action>
    </policy_rule>
  </goal>
</policy_document>
```

G05: Maximise Turbine Component Monitoring

This goal aims to maximise monitoring of turbine components to detect potential faults. This might be achieved by increasing the frequency of reports obtained from a turbine controller on the status of particular components.

Provided the wind speed $\geq$ 15 m/s

Maximise Turbine Monitoring

Prototypes

Prototype policies numbered P01 to P18 inclusive are specific to wind turbine management and monitoring. Prototypes P19 to P35 inclusive handle sensor network events and monitoring. All prototypes are numbered in no particular way, although they are grouped roughly by triggering event.

P01: Rotor Overspeed Protection

Protects the turbine rotor when the speed reaches a dangerous level (over 50 RPM) by applying the rotor brakes and pitching the blades out of the wind to reduce the speed. It also sends a warning to the operator. This detects a possible malfunction in the generator, which may have overheated or disconnected from the power grid causing the rotor to accelerate. This prototype has the effect of setting the rotor brake status to 1 (on) and the blade pitch angle to 0.

When the rotor speed goes above 50 RPM
If wind speed is less than 10 m/s
Do report rotor malfunction and apply rotor brakes and
pitch blades to 0 degrees
For rotor_brake_status = 1 and blade_pitch_angle = 0

P02: Shutdown Turbine Under High Vibration

Shuts the turbine down when there is high rotor blade vibration (over 20mm), to avoid stress damage to the turbine. The steps taken to initiate a shut down are to warn the operator and pitch the blades out of the wind – slowing the rotor speed. This prototype has the effect of setting the blade pitch angle to 0.

When the blade vibration size is over 20mm
Do warn operator of possible rotor malfunction and
pitch blades to 0 degrees
For blade_pitch_angle = 0

P03: Handle Low Blade Vibration

Takes action when low levels of rotor blade vibration (over 5mm) are detected. This prototype has the effect of decreasing the current blade pitch angle by 3 degrees, increasing the yaw angle by 10 degrees (both actions intended to angle the rotor out of the wind to lower friction), and setting the blade vibration report frequency to 3.

When the blade vibration size is over 5mm
If the blade pitch is ≥ 3 degrees and yaw angle is ≤ 80 degrees
Do decrease blade pitch by 3 degrees and increase yaw angle by 10 degrees
  and set blade vibration reporting to 3 minute intervals
For blade_pitch_angle −= 3 and yaw_angle += 10 and
  blade_vibration_report_freq = 3

P04: Handle Medium Blade Vibration

Takes action when medium levels of rotor blade vibration (over 15mm) are detected. This prototype has the effect of decreasing the current blade pitch angle by 10 degrees, increasing the yaw angle by 15 degrees (both actions intended to angle the rotor slightly out of the wind to lower friction), and setting the blade vibration report frequency to 1. The XML notation for this prototype is shown below.

When the blade vibration size is over 15mm
If the blade pitch is ≥ 10 degrees and the yaw angle is ≤ 75 degrees
Do decrease blade pitch by 10 degrees and increase yaw angle by 15 degrees
  and set blade vibration reporting to 1 minute intervals
For blade_pitch_angle −= 10 and yaw_angle += 15 and
  blade_vibration_report_freq = 1

```
<?xml version="1.0" encoding="UTF-8"?>
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel_prototype.xsd">
  <prototype
      owner="goalkeeper@cs.stir.ac.uk"
      applies_to="@cs.stir.ac.uk"
      effects="blade_pitch_angle -= 10,yaw_angle += 15,blade_vibration_report_freq = 1"
      id="Handle Medium Blade Vibration"
      enabled="true"
      changed=″2008-04-21T10:20:59″>
    <policy_rule>
      <trigger arg1=″blade_vibration_size″ arg2=″turbine″>
        device_in(arg1,arg2)
      </trigger>
      <conditions>
        <and/>
        <condition>
          <parameter>message_values</parameter>
          <operator>ge</operator>
          <value>15</value>
        </condition>
        <conditions>
          <and/>
          <condition>
            <parameter>:blade_pitch_angle</parameter>
            <operator>ge</operator>
            <value>10</value>
          </condition>
          <condition>
            <parameter>:yaw_angle</parameter>
            <operator>le</operator>
            <value>75</value>
          </condition>
        </conditions>
      </conditions>
      <actions>
        <and/>
        <action arg1=″set_parameter″ arg2=″turbine″
            arg3=″:entity_instance″ arg5=″[blade_pitch,-10]″>
          device_out(arg1,arg2,arg3,,arg5)
        </action>
        <actions>
          <and/>
          <action arg1=″set_parameter″ arg2=″turbine″
              arg3=″:entity_instance″ arg5=″[yaw_angle,+15]″>
            device_out(arg1,arg2,arg3,,arg5)
          </action>
          <action arg1=″set_parameter″ arg2=″turbine″
              arg3=″:entity_instance″ arg5=″[blade_vibration_reports,1]″>
            device_out(arg1,arg2,arg3,,arg5)
          </action>
        </actions>
      </actions>
    </policy_rule>
  </prototype>
</policy_document>
```

P05: Alter Yaw In High Blade Vibration

Takes action when a high level of vibration (over 20mm) is detected in the rotor blades. This prototype has the effect of setting the yaw angle to 90 degrees (intended to angle the rotor out of the wind to lower drive), and setting the yaw brake status to 1 (apply the yaw brakes).

When the blade vibration size is over 20mm

Do set yaw angle to 90 and apply yaw brakes

For yaw_angle = 90 and yaw_brake_status = 1

P06: Fix Repeating Yaw Direction

Helps the turbine recover when the rotor has yawed too many times in the same direction (indicated by the number of twists in the power cables in the turbine tower). This prototype has the effect of setting the blade pitch angle to 0 degrees (slowing and stalling the rotor speed) and setting the yaw angle to 90. Although not a measureable "effect", the final policy action sends a warning to the operator to resolve the problem – perhaps by yawing the turbine in the appropriate direction to untwist the cables.

When reported power cable twists is over 6

Do pitch blades to 0 degrees and reset the yaw angle to 90 degrees and
warn the operator of cable twist error

For blade_pitch_angle = 0 and yaw_angle = 90

P07: Blade Pitch Parameterised Normal Wind

Pitches the rotor blades when the wind speed is between 3 m/s to 15 m/s – normal conditions (neither too high nor too low). The pitch angle is a parameter (denoted by the "$" symbol). Its value is chosen at run-time during policy optimisation. This prototype has the effect of setting the blade pitch angle to a fixed value decided at run-time.

When the turbine anemometer reports

If the wind speed > 3 m/s and the wind speed ≤ 15 m/s

Do set blade pitch to $P degrees

For blade_pitch_angle = $P

P08: Cool Generator When Overheating

Cools the turbine generator when the generator temperature is reported to be over 80 degrees. This prototype has the effect of setting the generator cooling fan speed to

5000 RPM (revolutions per minute) and setting the generator reporting frequency to once every five minutes.

When the generator temperature is reported

If the temperature > 80 degrees

Do set generator cooling fan speed to 5000 RPM and
set generator reporting to 5 minute intervals

For generator_cooling_fan_speed = 5000 and
generator_report_freq = 5

P09: Shutdown Turbine In High Generator Voltage

Shuts down a turbine when its generator voltage is too high (over its maximum safe level of 690 Volts). The shut down is initiated by applying the rotor brakes and pitching the rotor blades completely out of the wind. This prototype has the effect of setting the brake status to 1 and the blade pitch angle to 0 degrees.

When the generator voltage > 690 volts

Do apply rotor brakes and pitch blades to 0 degrees

For rotor_brake_status = 1 and blade_pitch_angle = 0

P10: Reduce Yaw If Gearbox Oil High

Alters the rotor yaw angle when the turbine gearbox oil temperature is above a safe level (above 60 degrees). This prototype has the effect of increasing the current yaw angle by 18 degrees.

When the gearbox oil temperature is over 60 degrees

Do warn the operator and increase the yaw angle by 18 degrees

For yaw_angle += 18

P11: Set Bearing Reports To 5 Mins

Alters the frequency of gearbox oil temperature reports when the gearbox oil temperature is high (equal to or above 70 degrees). This prototype has the effect of setting the gearbox oil temperature reporting frequency to 5.

When the gearbox oil temperature is $\geq$ 70 degrees

Do set gearbox bearing temperature reporting to 5 minute intervals

For gearbox_bearing_report_freq = 5

P12: Shutdown Turbine In Gale Force Wind

Shuts down the turbine when gales are detected (wind speeds over 25 m/s). The shut down is initiated by pitching the rotor blades out of the wind and applying the rotor brake. The anemometer reporting frequency is also altered – to monitor wind speed and detect when the conditions are safe to resume turbine operation. This prototype has the effect of setting the blade pitch angle to 0 degrees, the rotor brake status to 1 and the turbine anemometer reporting frequency to 2.

When the turbine anemometer reports

If the wind speed > 25 m/s

Do pitch the blades to 0 degrees and apply rotor brakes and
set turbine anemometer reporting to 2 minute intervals

For blade_pitch_angle = 0 and rotor_brake_status = 1 and
turbine_anemometer_report_freq = 2

P13: Yaw Adjustment In High Wind

Adjusts the turbine yaw angle when high winds (above 20 m/s) are reported. This prototype has the effect of increasing the current turbine yaw angle by 30 degrees (turning the rotor out of the wind to reduce rotor speed).

When the turbine anemometer reports

If the wind speed > 20 m/s and the yaw angle < 10 degrees

Do warn operator and increase yaw angle by 30 degrees

For yaw_angle += 30

P14: Low Rotation In Low Wind

Ensures the turbine rotor blades continue to rotate at a low level in the case of little or no wind to power them. This is instead of shutting the turbine down (increasing strain on mechanical components from being idle for extended periods of time). This prototype has the effect of setting the blade pitch angle to 45 (pitching blades fully to make use of available wind), and setting the turbine anemometer reporting frequency to 10 (to monitor conditions and detect when wind speed increases).

When the turbine anemometer reports
If the wind speed < 1.5 m/s
Do pitch blades to 45 degrees and
  set turbine anemometer reporting to 10 minute intervals
For blade_pitch_angle = 45 and
  turbine_anemometer_report_freq = 10

P15: Shutdown Turbine In Very Low Wind

Shuts down the turbine when there is little or no wind to turn the rotor blades without assistance. This prototype has the effect of setting the blade pitch angle to 0 and the rotor brake status to 1 (applying the rotor brake to keep the rotor safely stationary).

When the turbine anemometer reports
If the wind speed < 0.5 m/s
Do pitch blades to 0 degrees and apply rotor brakes
For blade_pitch_angle = 0 and rotor_brake_status = 1

P16: Blade Pitch For Stronger Wind

Alters the blade pitch for strong (safe) wind conditions between 10 m/s and 15 m/s. This prototype has the effect of decreasing the the blade pitch angle by 20 degrees (to slow the rotor speed and protect the turbine components from overheating).

When the turbine anemometer reports
If the wind speed ≥ 10 m/s and the wind speed ≤ 15 m/s
  and the blade pitch > 40 degrees
Do reduce the blade pitch by 20 degrees
For blade_pitch_angle −= 20

P17: Blade Pitch For Low Wind

Alters the blade pitch for low winds (below 5 m/s). This prototype has the effect of increasing the the blade pitch angle by 15 degrees (to increase the rotor speed and generate more power).

When the turbine anemometer reports
If the wind speed $<$ 5 m/s and the blade pitch $<$ 31 degrees
Do increase the blade pitch by 15 degrees
For blade_pitch_angle += 15

P18: Heat Nacelle In Freezing Temperatures

Turns on internal nacelle heater in freezing conditions (when temperatures fall below -10) to prevent ice forming on mechanical components and causing damage or malfunction. This prototype has the effect of setting the internal nacelle heater to full power (230W) and the nacelle temperature reporting frequency to 4.

When the nacelle temperature ≤ -10
Do switch on internal heater and
  set nacelle temperature reporting to 4 minute intervals
For internal_heater_power = 230V and
  nacelle_temperature_report_freq = 4

P19: Battery Level Below 9 Volts

Prototypes P19 to P27 attempt to maintain the best operational capability of a node in the event of a failing battery. These prototypes configure the sensor node to either reduce sampling and/or reporting frequencies of individual sensors or switch sensors off depending on the detected battery level. Each sensor has an assumed priority based on the importance of its measurements and the power it consumes. For example, a rainfall sensor has the lowest priority as it consumes more power than any other sensor and its measurements are less important than, for example, temperature or wind speed. An anemometer has the highest priority as wind speed is deemed the most crucial measurement. Low priority, power hungry sensors are switched off first to conserve power, whereas sensors of most importance to data collection remain operational but have their sampling and reporting frequencies gradually decreased as the battery deteriorates. The maximum battery level is 12V. Different actions are required at set intervals of 9V, 7V, 5V, 3V and 2V.

Prototype P19 takes action when the sensor node battery falls below 9V, and sends a warning message to the operator console. This prototype has the effect of setting the temperature sampling frequency to once every 8 minutes, the soil moisture sampling frequency to once every 15 minutes and the humidity sampling frequency to once every 20 minutes. The XML for this prototype is shown below.

When a sensor node reports on battery voltage
If the battery voltage < 9V
Do set the temperature sampling frequency to 8 and
  set the soil moisture sampling frequency to 15 and
  set the humidity sampling frequency to 20 and
  send a message to the console:
 "Level 1 warning, node battery low, reducing sampling rates across sensors"
For temperature_sampling_freq = 8.0 and
  soilmoisture_sampling_freq = 15.0 and humidity_sampling_freq = 20.0

```
<?xml version="1.0" encoding="UTF-8"?>
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel_prototype.xsd">
  <prototype
      owner="goalkeeper@cs.stir.ac.uk"
      applies_to="@cs.stir.ac.uk"
      effects="temperature_sampling_freq = 8.0,soilmoisture_sampling_freq = 15.0,
        humidity_sampling_freq = 20.0"
      id="Battery Level Below 9 Volts"
      enabled="true"
      changed="2008-04-21T10:20:59">
    <policy_rule>
      <trigger arg1="battery_voltage" arg2="sensor">
        device_in(arg1,arg2)
      </trigger>
      <condition>
        <parameter>message_values</parameter>
        <operator>lt</operator>
        <value>9</value>
      </condition>
      <actions>
```

```
        <and/>
        <action arg1=″set_rule″ arg2=″sensor″
            arg3=″:entity_instance″ arg5=″[temp_samp_freq,8.0]″>
          device_out(arg1,arg2,arg3,,arg5)
        </action>
        <actions>
          <and/>
          <action arg1=″set_rule″ arg2=″sensor″
              arg3=″:entity_instance″ arg5=″[soil_samp_freq,15.0]″>
            device_out(arg1,arg2,arg3,,arg5)
          </action>
          <actions>
            <and/>
            <action arg1=″set_rule″ arg2=″sensor″
                arg3=″:entity_instance″ arg5=″[humidity_samp_freq,20.0]″>
              device_out(arg1,arg2,arg3,,arg5)
            </action>
            <action arg1=″message″ arg2=″console″
                arg5=″[Level 1 warning, node battery low, reducing sampling rates across sensors]″>
              device_out(arg1,arg2,,,arg5)
            </action>
          </actions>
        </actions>
      </actions>
    </policy_rule>
  </prototype>
</policy_document>
```

P20: Battery Level Below 7 Volts

Takes action when the sensor node battery falls under 7V, and notifies the operator via the console and by email. This prototype has the effect of setting the anemometer sampling frequency to once every 5 minutes and the temperature sampling frequency to once every 12 minutes.

```
When a sensor node reports on battery voltage
If the battery voltage < 7V
Do set the temperature sampling frequency to 12 and
   set the anemometer sampling frequency to 5 and
   send a message to the console:
     “Level 2 warning, node battery low, reducing sampling rates” and
   send a message to operator@wf.com:
     “Level 2 warning, node battery low, reducing sampling rates”
For anemometer_sampling_freq = 5.0 and
   temperature_sampling_freq = 12.0
```

P21: Battery Level Below 5 Volts

Takes action when the sensor node battery falls under 5V, and notifies the operator via the console. This prototype has the effect of switching off the rainfall and soil moisture sensors (rainfall and soil moisture drain set to 0), setting the anemometer sampling frequency to once every 10 minutes and setting the temperature sampling frequency to once every 15 minutes.

```
When a sensor node reports on battery voltage
If the battery voltage < 5V
Do switch off rainfall sensor and switch off soil sensor and
    set the temperature sampling frequency to 15 and
    set the anemometer sampling frequency to 10 and
    send a message to the console:
      “Node battery low, switching off rainfall sensor”
For rainfall_drain = 0.0 and soilmoisture_drain = 0.0 and
    anemometer_sampling_freq = 10.0 and temperature_sampling_freq = 15.0
```

P22: Battery Level Below 3 Volts

Takes action when the sensor node battery falls under 3V, and notifies the operator via the console. This prototype has the effect of switching off the temperature and humidity sensors (temperature and humidity drain set to 0) and setting the anemometer sampling frequency to once every 15 minutes.

```
When a sensor node reports on battery voltage
If the battery voltage < 3V
Do switch off temperature sensor and switch off humidity sensor and
    set the anemometer sampling frequency to 15 and
    send a message to the console:
      “Node battery low, switching off temperature sensor”
For temperature_drain = 0.0 and humidity_drain = 0.0 and
    anemometer_sampling_freq = 15.0
```

P23: Battery Level Below 2 Volts

Takes action when the sensor node battery falls under 2V. This is a critical power level for the node, therefore the remaining sensor (the anemometer) is switched off and the operator notified via console and SMS message. This prototype has the effect of switching off the anemometer (anemometer drain set to 0).

```
When a sensor node reports on battery voltage
If the battery voltage < 2V
Do switch off the anemometer and send a message to the console:
    “Node at critical power level, switching off anemometer” and
  send a message via SMS:
    “Node at critical power level, switching off anemometer”
For anemometer_drain = 0.0
```

P24: Sensor Reporting Battery Below 9 Volts

Attempts to conserve battery power by reducing the frequency of sensor reports when the sensor node battery falls under 9V. This prototype has the effect of setting the temperature reporting frequency to once every 30 minutes, the soil moisture reporting frequency to once every 40 minutes and the humidity reporting frequency to once every 40 minutes.

```
When a sensor node reports on battery voltage
If the battery voltage < 9V
Do set the temperature reporting frequency to 30 and
  set the soil moisture reporting frequency to 40 and
  set the humidity reporting frequency to 40
For temperature_reporting_freq = 30.0 and
  soilmoisture_reporting_freq = 40.0 and
  humidity_reporting_freq = 40.0
```

P25: Sensor Reporting Battery Below 7 Volts

Attempts to conserve battery power by reducing the frequency of sensor reports when the sensor node battery falls under 7V. This prototype has the effect of setting the temperature reporting frequency to once every 45 minutes and the anemometer reporting frequency to once every 10 minutes.

```
When a sensor node reports on battery voltage
If the battery voltage < 7V
Do set the temperature reporting frequency to 45 and
    set the anemometer reporting frequency to 10
For temperature_reporting_freq = 45.0 and
    anemometer_reporting_freq = 10.0
```

P26: Sensor Reporting Battery Below 5 Volts

Attempts to conserve battery power by reducing the frequency of sensor reports when the sensor node battery falls under 5V. This prototype has the effect of setting the temperature reporting frequency to once every 60 minutes and the anemometer reporting frequency to once every 15 minutes.

```
When a sensor node reports on battery voltage
If the battery voltage < 5V
Do set the temperature reporting frequency to 60 and
    set the anemometer reporting frequency to 15 and
For temperature_reporting_freq = 60.0 and
    anemometer_reporting_freq = 15.0
```

P27: Sensor Reporting Battery Below 3 Volts

Attempts to conserve battery power by reducing the frequency of sensor reports when the sensor node battery falls under 3V. This prototype has the effect of setting the anemometer reporting frequency to once every 20 minutes.

When a sensor node reports on battery voltage

If the battery voltage < 3V

Do set the anemometer reporting frequency to 20

For anemometer_reporting_freq = 20.0

P28: Increase Sampling In High Winds

Notifies the operator and increases the anemometer sampling frequency when the wind speed is 20 m/s or above, and the sensor node battery is at least 9V (if the battery is below 9V, the previous policies P19 - P27 may take better configuration actions). This prototype has the effect of setting the anemometer sampling frequency to once a minute.

When the anemometer reports

If the value is ≥ 20 m/sec and battery voltage ≥ 9V

Do set the anemometer sampling frequency to 1 and

send a message to the console:

"High wind, increasing anemometer sampling"

For anemometer_sampling_freq = 1.0

P29: Increase Reporting In High Winds

Notifies the operator and increases the anemometer reporting frequency when the wind speed is 20 m/s or above, and the sensor node battery is at least 9V (if the battery is below 9V, the previous policies P19 - P27 may take better configuration actions). This prototype has the effect of setting the anemometer reporting frequency to once a minute.

```
When the anemometer reports
If the value is $\geq$ 20 m/sec and battery voltage $\geq$ 9V
Do set the anemometer reporting frequency to 1 and
   send a message to the console:
     "High wind, increasing anemometer reporting"
For anemometer_reporting_freq = 1.0
```

P30: Reduce Sampling In Low Winds

Notifies the operator and reduces the anemometer sampling frequency when the wind speed is low (below 2 m/s), and the sensor node battery is at least 9V (if the battery is below 9V, the previous policies P19 - P27 may take better configuration actions). This prototype has the effect of setting the anemometer sampling frequency to once every 15 minutes.

```
When the anemometer reports
If the value is $<$ 2 m/sec and battery voltage $\geq$ 9V
Do set the anemometer sampling frequency to 15 and
   send a message to the console:
     "Low wind, reducing anemometer sampling"
For anemometer_sampling_freq = 15.0
```

P31: Reduce Reporting In Low Winds

Notifies the operator and reduces the anemometer reporting frequency when the wind speed is low (below 2 m/s), and the sensor node battery is at least 9V (if the battery is below 9V, the previous policies P19 - P27 may take better configuration actions). This prototype has the effect of setting the anemometer reporting frequency to once every 15 minutes.

When the anemometer reports
If the value is < 2 m/sec and battery voltage $\geq$ 9V
Do set the anemometer reporting frequency to 15 and
  send a message to the console:
   "Low wind, reducing anemometer reporting"
For anemometer_reporting_freq = 15.0

P32: Set Anemometer Rates Normal

Sets the anemometer sampling and reporting frequencies to their default values when the wind speed is normal (between 2 m/s and 20 m/s), and the sensor node battery is at least 9V (if the battery is below 9V, the previous policies P19 - P27 may take better configuration actions). This prototype has the effect of setting the anemometer sampling frequency to once every 3 minutes and the anemometer reporting frequency to once every 3 minutes.

When the anemometer reports
If the value is ≥ 2 m/sec and the value is < 20 m/sec and
  the battery voltage $\geq$ 9V
Do set the anemometer sampling frequency to 3 and
  set the anemometer reporting frequency to 3
For anemometer_sampling_freq = 3.0 and
  anemometer_reporting_freq = 3.0

P33: Increase Sampling In Extreme Temperatures

Increases temperature sensor sampling and reporting frequencies in extreme conditions – either very warm (30°C or above) or very cold (below 1°C), and the sensor node battery is at least 9V (if the battery is below 9V, the previous policies P19 - P27 may take better configuration actions). This prototype has the effect of setting the temperature sampling frequency to once every 1 minute and the temperature reporting frequency to once every 3 minutes.

When the temperature sensor reports

If (the value is ≥ 30 or the value is < 1) and

the battery voltage $\geq$ 9V

Do set the temperature sampling frequency to 1 and

set the temperature reporting frequency to 3 and

send a message to the console: "Increasing temperature sampling rate"

For temperature_sampling_freq = 1.0 and temperature_reporting_freq = 3.0

P34: Set Temperature Rates Normal

Sets the temperature sensor sampling and reporting frequencies to their default values when the temperature is normal (between 1°C and 30°C), and the sensor node battery is at least 9V (if the battery is below 9V, the previous policies P19 - P27 may take better configuration actions). This prototype has the effect of setting the temperature sampling frequency to once every 5 minutes and the temperature reporting frequency to once every 5 minutes.

When the temperature sensor reports

If (the value is < 30 and the value is ≥ 1) and

the battery voltage $\geq$ 9V

Do set the temperature sampling frequency to 5 and

set the temperature reporting frequency to 5

For temperature_sampling_freq = 5.0 and

temperature_reporting_freq = 5.0

P35: Alter Anemometer In High Winds

Alters the anemometer sampling and reporting frequencies when the wind speed is high (20 m/s or above), and the sensor node battery is at least 9V (if the battery is below 9V, the previous policies P19 - P27 may take better configuration actions). The sampling and reporting frequencies are parameter values (denoted by the "$" symbol). The values for $S and $R are chosen at run-time during policy optimisation. This prototype has the effect of setting the anemometer sampling frequency to a fixed value decided at run-time and, similarly, the anemometer reporting frequency to a fixed value decided at run-time.

```
When the anemometer reports
If the value is ≥ 20 m/sec and battery voltage < 9V
Do set the anemometer sampling frequency to $S and
   set the anemometer reporting frequency to $R and
   send a message to the console:
     "High winds, altered anemometer configuration"
For anemometer_sampling_freq = $S and
   anemometer_reporting_freq = $R
```

6.5.3 Static Refinement

The static selection process looks at each prototype and compares its effects against the controlled and uncontrolled values within each goal measure. If a prototype affects a variable used within a goal measure, it potentially contributes to that goal. The result of static prototype selection is shown in Figure 6.14. This shows which prototypes potentially contribute to each goal, with a star indicating applicability. Prototypes are listed alphabetically by name, with turbine-specific prototypes first followed by sensor-specific.

Prototype	**Battery Drain**	**Data Yield**	**Component Damage**	**Energy Generated**	**Turbine Monitoring**
Alter Yaw In High Blade Vibration			★	★	
Blade Pitch For Low Wind			★	★	
Blade Pitch For Stronger Wind			★	★	
Blade Pitch Parameterised Normal Wind			★	★	
Cool Generator When Overheating				★	★
Fix Repeating Yaw Direction			★	★	
Handle Low Blade Vibration			★	★	★
Handle Medium Blade Vibration			★	★	★
Heat Nacelle In Freezing Temperatures				★	★
Low Rotation In Low Wind			★	★	★
Reduce Yaw If Gearbox Oil High			★	★	
Rotor Overspeed Protection			★	★	
Set Bearing Reports To 5 Mins					★
Shutdown Turbine High In Generator Voltage			★	★	
Shutdown Turbine In Gale Force Wind			★	★	★
Shutdown Turbine In Very Low Wind			★	★	
Shutdown Turbine Under High Vibration			★	★	
Yaw Adjustment In High Wind			★	★	
Alter Anemometer In High Winds	★	★			
Battery Level Below 2 Volts	★				
Battery Level Below 3 Volts	★	★			
Battery Level Below 5 Volts	★	★			
Battery Level Below 7 Volts	★	★			
Battery Level Below 9 Volts	★	★			
Increase Reporting In High Winds	★	★			
Increase Sampling In Extreme Temperatures	★	★			
Increase Sampling In High Winds	★	★			
Reduce Reporting In Low Winds	★	★			
Reduce Sampling In Low Winds	★	★			
Sensor Reporting Battery Below 3 Volts	★	★			
Sensor Reporting Battery Below 5 Volts	★	★			
Sensor Reporting Battery Below 7 Volts	★	★			
Sensor Reporting Battery Below 9 Volts	★	★			
Set Anemometer Rates Normal	★	★			
Set Temperature Rates Normal	★	★			

Figure 6.14: Static Turbine and Sensor Prototype Selection

Referring to Figure 6.14, the following example demonstrates the process of prototype instantiation for “Cool Generator When Overheating” (P08 in the prototypes listed in the previous section), which cools the turbine generator when an overheat is detected. The XML for this prototype is as follows:

```
<prototype
    owner="goalkeeper@cs.stir.ac.uk"
    applies_to="@cs.stir.ac.uk"
    effects="generator_cooling_fan_speed = 5000,generator_report_freq = 5"
    id="Cool Generator When Overheating"
    enabled="true"
    changed="2008-10-05T13:48:54">
  <policy_rule>
    <trigger arg1="generator_temperature" arg2="turbine">
      device_in(arg1,arg2)
    </trigger>
    <condition>
      <parameter>parameter_values</parameter>
      <operator>gt</operator>
      <value>80</value>
    </condition>
    <actions>
      <and/>
      <action arg1="set_parameter" arg2="turbine"
          arg3=":entity_instance" arg5="[generator_fan_speed,5000]">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
      <action arg1="set_parameter" arg2="turbine"
          arg3=":entity_instance" arg5="[generator_reports,5]">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
    </actions>
  </policy_rule>
</prototype>
```

Referring to Figure 6.14, this prototype potentially contributes to two goals, Maximise Energy Generated (G04) and Maximise Turbine Monitoring (G05). As both these goals have associated conditions, the conditions from each are copied when the prototype is instantiated.

The instantiated prototype is shown below in XML. This new policy has a modified id (line 4) and a new attribute listing the goals it supports (line 7). The policy condition block contains the original prototype condition (lines 14-18) and the conditions inherited from the goals it supports: the condition on G04 (lines 21-25) and the condition on G05 (lines 26-30). Note the way in which these conditions are combined. The original condition is combined with the goal conditions using “and” and the goal conditions combined using “or”. Once triggered, the policy is executable only if its original condition plus either one of the two goal conditions are satisfied.

```
<policy
    owner="goalkeeper@cs.stir.ac.uk"
    applies_to="@cs.stir.ac.uk"
    effects="generator_cooling_fan_speed = 5000,generator_report_freq = 5"
    id="Cool Generator When Overheating_pol"
    enabled="true"
    changed="2008-10-05T14:36:07"
    supports_goal="Maximise Energy Generated,Maximise Turbine Monitoring">
  <policy_rule>
    <trigger arg1="generator_temperature" arg2="turbine">
      device_in(arg1,arg2)
    </trigger>
    <conditions>
      <and/>
      <condition>
        <parameter>parameter_values</parameter>
        <operator>gt</operator>
        <value>80</value>
      </condition>
      <conditions>
        <or/>
        <condition>
          <parameter>:generator_voltage</parameter>
          <operator>le</operator>
          <value>690</value>
        </condition>
        <condition>
          <parameter>:wind_speed</parameter>
          <operator>ge</operator>
          <value>15</value>
        </condition>
      </conditions>
    </conditions>
    <actions>
      <and/>
      <action arg1="set_parameter" arg2="turbine"
          arg3=":entity_instance" arg5="[generator_fan_speed,5000]">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
      <action arg1="set_parameter" arg2="turbine"
          arg3=":entity_instance" arg5="[generator_reports,5]">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
    </actions>
  </policy_rule>
</policy>
```

6.5.4 Dynamic Refinement

This section demonstrates run time policy selection for sensor network and wind turbine management. There are four scenarios documenting the results of event handling and policy selection for two sensor network triggers and two wind turbine triggers. Triggers to the policy system take the form of the standard "device_in" trigger for sensor network policies described in Chapter 3. For example, turbine T03 may notify the policy system that the blade pitch angle at 1pm was 30.8 degrees using a message in the following format:

```
device_in(blade_pitch_angle,turbine,T03,13:00:00,[30.8])
```

Each scenario is presented in a similar fashion. The trigger is given in the format above together with the values of any other relevant state variables, followed by a graph of the results obtained for each applicable policy combination. For an explanation of the graph and how to interpret the results refer to the previous section 6.4.4. Each scenario concludes with an evaluation of the optimal result obtained.

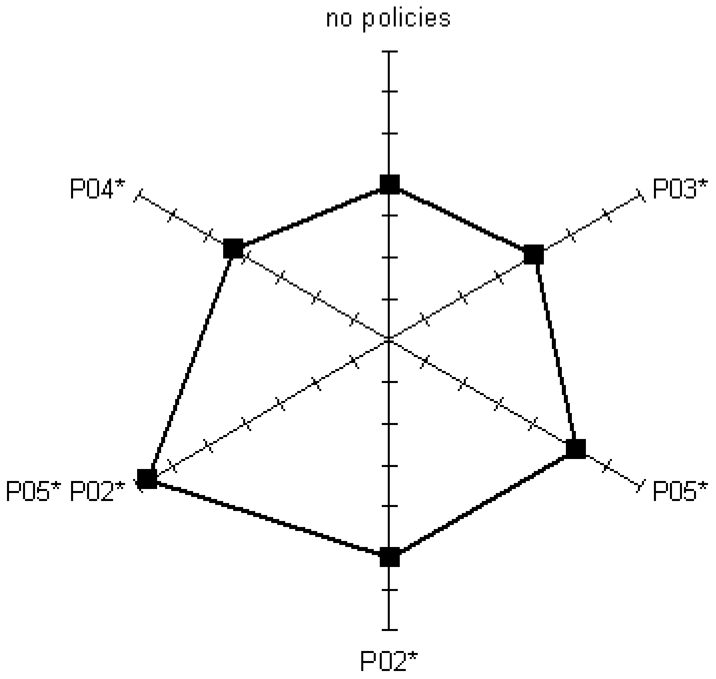
Scenario 1: Wind Turbine Blade Vibration

This scenario demonstrates policy selection when a turbine blade vibration size report is received. The trigger is:

```
device_in(blade_vibration_frequency,turbine,T03,12:30:21,[22.0])
```

The blade vibration size is 22.0, from the turbine T03, measured at the time of 12:30:21. This example assumes the current blade pitch angle is 42 degrees and the yaw angle is 15 degrees. Under these conditions, four goal-related policies are applicable. The result of goal evaluation for these policies is shown in Figure 6.15.

As the trigger was a high level of blade vibration size, suitable policies should be aiming to take action that prevents further damage or mechanical failure. The optimal result in this case is to select policies P05* and P02*. P05* sets the yaw angle of the



P02*	Shutdown Turbine Under High Vibration
P03*	Handle Low Blade Vibration
P04*	Handle Medium Blade Vibration
P05*	Alter Yaw In High Blade Vibration

Figure 6.15: Scenario 1: Policy Selection for Wind Turbine Blade Vibration

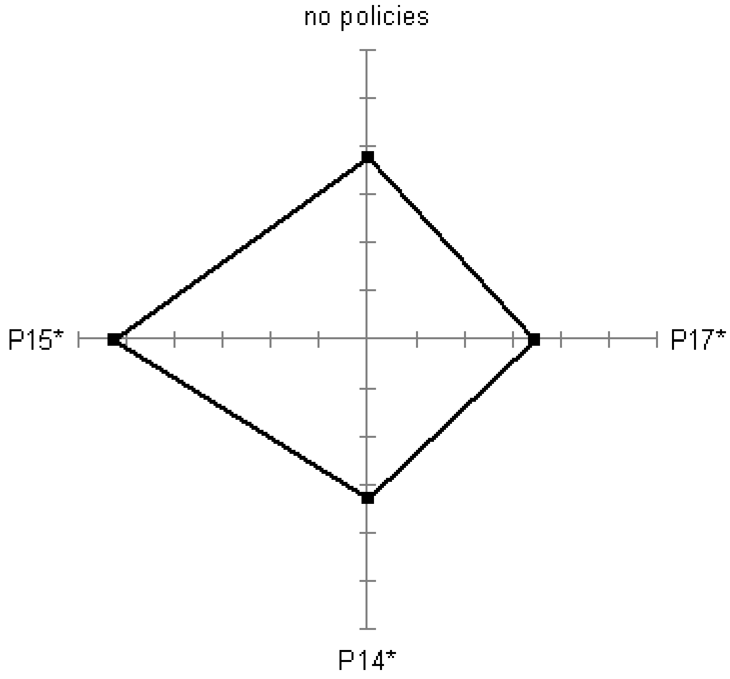
turbine rotor to 90 degrees (angling the rotor out of the wind). P02* slows the rotor speed by pitching the rotor blades to 0 degrees (angling the blades out of the wind). When measured across all goals, these actions are preferable over increasing the blade pitch and yaw angle by smaller amounts as in policies P03* and P04* – actions more appropriate for lower levels of blade vibration.

Scenario 2: Turbine Anemometer Report

This scenario demonstrates policy selection when a turbine anemometer reports the detected wind speed. The trigger is:

```
device_in(anemometer,turbine,T01,16:10:51,[0.4])
```

The reported wind speed is 0.4 m/s, from the turbine T01, measured at the time of 16:10:51. This example assumes the current blade pitch angle is 35 degrees and the yaw angle is 43 degrees. Under these conditions, three goal-related policies are applicable. The result of goal evaluation for these policies is shown in Figure 6.16.



P14*	Low Rotation In Low Wind
P15*	Shutdown Turbine In Very Low Wind
P17*	Blade Pitch For Low Wind

Figure 6.16: Scenario 2: Policy Selection for Turbine Anemometer Report

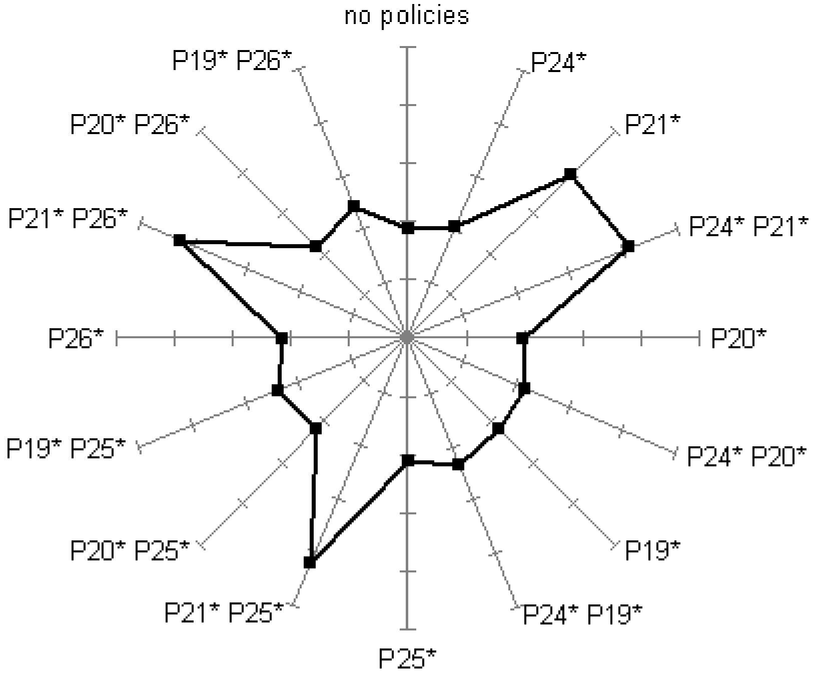
The trigger was a report that there was little wind. The optimal policy to select was P15*, which shuts the turbine down by applying the rotor brakes and pitching the rotor blades to 0 degrees (to stop the rotor rotating in any available wind). This is deemed more favourable than allowing the turbine to rotate slowly (P14*), or increasing the blade pitch in an attempt to operate more efficiently (P17*).

Scenario 3: Sensor Node Battery Voltage

This scenario demonstrates policy selection when a sensor node reports the battery voltage level. The trigger is:

```
device_in(battery_voltage,sensor,S02,19:35:05,[4.8])
```

The battery voltage is 4.8 Volts, from sensor node S02, measured at the time of 19:35:05. Under these conditions, six goal-related policies are applicable. The result of goal evaluation for these policies is shown in Figure 6.17.



P19*	Battery Level Below 9 Volts
P20*	Battery Level Below 7 Volts
P21*	Battery Level Below 5 Volts
P24*	Sensor Reporting When Battery Below 9 Volts
P25*	Sensor Reporting When Battery Below 7 Volts
P26*	Sensor Reporting When Battery Below 5 Volts

Figure 6.17: Scenario 3: Policy Selection for Sensor Node Battery Voltage

The trigger is a report that the battery voltage was at a low level. All the triggered policies take action toward conserving power on a node to varying degrees. The optimal set of policies selected is P26* and P21*. P26* sets the reporting frequencies of sensors on the node to values recommended when the battery is below 5 volts, and P21* sets the sampling rates of sensors to recommended levels in the same circumstances, and also switches some sensors off. This selection is logically what is expected under the presented conditions. However, without goal-directed policy selection, any possible combination of the six triggered policies might have been executed. From the graph, although the combination of P26* and P21* yields the best value (20.113), there are three other selections which are marginally worse (P21* alone (19.987), P24* and P21* (20.056), and P21* and P25* (20.097)). All these options include policy P21* but consider the impact of executing different combinations of policies that alter sensor reporting frequencies (P24*, P25* and P26*). Although there is small difference between the values, the value for the evaluation function (that takes into account all goal measures) is highest using the combination including P26*.

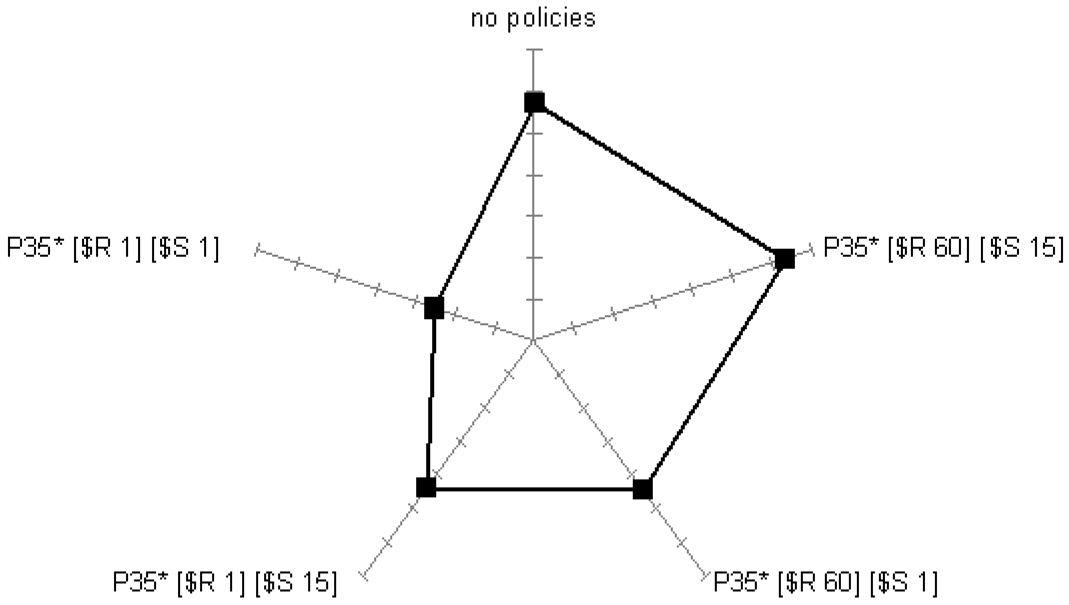
Scenario 4: Sensor Anemometer Report

This scenario demonstrates policy selection when a sensor node reports the wind speed from the anemometer. The trigger is:

```
device_in(anemometer,sensor,S15,07:45:00,[22.0])
```

The reported wind speed is 22 m/s, from the sensor node S15, measured at the time of 07:45:00. Note the battery voltage on this node is recorded as 7.5 volts in the policy store – the battery is getting low. Under these conditions, only one goal-related policy is applicable. The result of goal evaluation is shown in Figure 6.18.

The trigger is a report of high wind speed from a sensor node with a low battery. As the triggered policy is parameterised, the policy selection process considers the optimal value to assign. The parameters represent the anemometer reporting frequency ($R) and the anemometer sampling frequency ($S). The extreme values for $R are 1 and 60. The extreme values for $S are 1 and 15. The optimal solution sets the anemometer to report once every 60 minutes (R = 60) and to sample every 15 minutes (S = 15).



P35* Alter Anemometer In High Winds

Figure 6.18: Scenario 4: Policy Selection for Sensor Anemometer Report

6.6 Conclusion

6.6.1 Chapter Summary

This chapter presented a new approach to managing systems via high-level goals. The approach was outlined together with an implementation that was illustrated for the domains of Internet telephony and sensor network/wind turbine management.

The introductory section summarised goal-based techniques in computing and prior work where goals have been used in policy-based environments. Goal and policy refinement was explained and the differences between existing work in this field and the approach in this chapter outlined. No concrete methodology to define and refine goals into policies has been developed by others and no complete system has been implemented and demonstrated for multiple application domains.

In the presented goal refinement approach, goals are high-level objectives which are refined into sets of policies that help achieve them. Goals and prototypes (template goal-related policies) are defined using the Appel policy language. The state of the system is described using a set of controlled and uncontrolled variables which may be modified based on events received by the policy system and actions performed as a result of policy execution. Each goal has an associated numerical measure that includes these state variables. A goal evaluation function is defined which combines and optionally weights individual goal measures. Goal refinement is achieved using a two-stage process. Statically, prototypes are instantiated as executable policies to support goals. At run-time, triggered sets of goal-related policies are optimised, and parameterised if applicable. The outcome of run-time goal optimisation is a subset of initially triggered policies which, when executed, contribute most to the goals of the system. The developed goal system was integrated and evaluated with the existing Accent policy system.

6.6.2 Evaluation

The goal-directed approach to policy-based management presented in this chapter has been shown to be valuable in automatically managing and configuring a system based on a collection of high-level aims. In general, the goal approach is applicable to any event-based system environment manageable by policies. The approach is evaluated through consideration of its strengths, limitations, and future work.

Strengths

The approach differs in many ways from existing goal-directed methods discussed in section 6.1.6. A key difference between this approach and that of other goal-directed policy approaches is that, here, numerical analysis and optimisation is used instead of formal methods to define and refine goals into policies that achieve them. This is advantageous as domain experts rather than formalists can specify goals, weightings and measures with no need for prior knowledge of formal specifications. The absence of formal notation is more user-friendly and allows the goal environment to be modelled and understood by a wider range of people in an organisation – the mathematical and logical symbols used in formal methods is not as easy to interpret or explain.

Goals, prototypes and policies are defined using the common language of Appel.

As goals are refined directly into policies, conflict detection and resolution are addressed automatically within the Accent system, requiring no additional mechanism. In addition, the common syntax between goals, prototypes and policies is more user-friendly as it allows goals to be defined in a similar way to policies, which is simpler than forcing users to learn a separate language for each. Furthermore, the goal system has been designed and implemented to operate independently of the core Accent policy system. This means that goals are not essential for a domain specialisation of the policy system – policies alone may be used to manage a system without additional goal-direction. This is a strength of the approach as it allows the policy system and goal system to be extended and maintained independently, and does not prevent the policy system being used in domains where goal-direction is not desired.

Other goal and policy-based approaches, such as [55] and [93], use policy hierarchies (layers of policy abstraction). This is not as flexible as the presented approach, as goals and policies are more tightly associated (changes in lower-level policies have an affect on higher-level goals and policies). Here, goal-derived policies are not intended to be changed once deployed. The policy system thus stores and deploys both goal-derived policies and user-defined policies together and treats them similarly.

Existing approaches have been applied to limited real life domains other than network service management scenarios, thus failing to demonstrate the benefits that high-level goals can bring to system management over the use of policies alone. A major strength of the approach presented in this thesis demonstrates a system capable of goal direction using policies in more general domains – specifically, Internet telephony and sensor network/wind turbine management.

Limitations and Further Work

Although advantages of using non-formal methods of goal definition and refinement have been discussed, one weakness of this is that verification of policy solutions is not automatic. In the presented approach, trial and error is required when defining and testing the goal evaluation function and goal weightings. A domain expert must also check an appropriate set of policies (prototypes) are written to ensure contribution to all defined goals. Means of overcoming these issues include more rigorous evaluation of goals and prototypes when they are defined. For example, goal measures and prototypes could be automatically cross-referenced (to identify any goals that have few or no contributing policies), and prototypes grouped by trigger to determine cases where particular policies might be idle (triggered but never executed as other policies persistently score higher values).

As both user-defined policies and goal-related policies can be triggered together, it is possible they could conflict. This requires suitable resolution policies set up to handle such conflicts (e.g. user-defined policies may get priority over goal-derived policies). Goals themselves may conflict with one another but this may be determined through inspection of their measures when goals are defined. Conflicts between goals and policies are independently detectable (and resolvable) and are unrelated.

With regard to the detection of potential conflicts between both goals and the prototypes selected during the static analysis stage, conflicts might be detectable by comparing goal conditions and examining the actual triggers, conditions and actions of prototypes in addition to just their effects. An enhancement to the goal system might also include a mechanism to generate user-friendly explanations for changes in system behaviour resulting from the goal-refinement process. For example, this might provide feedback to defend why a particular action has been taken over others in relation to the goals it achieves. This feedback might also be useful in determining whether the weights applied to each goal are suitable or should be altered to achieve more desirable system behaviour.

The presented approach is flexible in that it offers a generic framework that may be adapted to use any optimisation algorithm. The optimisation method described in section 6.3.4 is sufficient for the demonstrated application domains of telephony and sensor networks as both domains utilise a relatively small number of policies and parameters. However, should the number of distinct parameters to optimise be larger for other domains, this approach is less efficient and the time overhead to process combinations would be undesirable. Under such circumstances, the use of a more powerful optimisation approach would be required, such as a Genetic Algorithm.

One improvement to this approach is to obtain more optimal parameter values. Currently, the approach considers just two values for each parameter (its maximum allowed value, and its minimum allowed value). To find the optimal parameter value would require a heavy-weight optimisation algorithm in order to compute solutions using a larger selection of possible values efficiently in real time. As mentioned previously, this is possible by implementing a different optimisation algorithm and plugging this into the goal system framework. However, there must always be a trade-off between the degree of optimisation and the processing overhead incurred.

In the presented approach, policy optimisation uses an evaluation function which is a weighted sum of all goal measures. Although the test scenarios documented in this chapter have yielded feasible optimal policy sets, it is plausible to question the impact small changes to the goal evaluation function might have on the results. For example, minor alterations to goal weightings or measures might yield a different optimal set of policies. Sensitivity analysis is therefore required in order to investigate this further. Such analysis should consider inputs to the evaluation function (i.e. measures, variables and weightings) and track the effect that changes in these inputs have on the resulting values (i.e. the optimal policy set). A means of automating this analysis would improve the approach and provide assurance as to the correctness of the optimal solutions found.

The goal system adds an overhead to the policy system during run-time optimisation. Domain depending, such a delay may reduce efficiency of the overall goal-directed, policy-based approach. Ways of improving this might be to detect and cache commonly triggered policy sets and omit particular combinations of policies which clash in their effects. Ultimately, this requires further practical testing and investigation.

Finally, while the presented approach has been demonstrated for Internet telephony and sensor network/wind turbine management, it would benefit from application to further domains, testing its scalability using a larger set of goals and prototypes, and experimenting with other optimisation techniques. Extending the range of applications would provide stronger evidence of the generality of the approach.

Chapter 7

Conclusion

The components of the goal-directed, policy-based approach proposed by this thesis have been presented within the past four chapters: policy-based management, a framework of generic and domain-specific ontologies, a tool to detect and filter policy conflicts, and a goal refinement approach. This final chapter considers the work of this thesis as a whole, evaluating the combined effort of the individual approaches, systems and tools.

Section 7.1 summarises the work presented. Section 7.2 reconsiders the objectives of the thesis outlined in the introduction and describes the achievements of this thesis work. Section 7.3 discusses the strengths of the overall approach, while section 7.4 describes the limitations of the presented work. Section 7.5 explores ways in which the whole approach may be improved upon and taken further through future work.

7.1 Thesis Summary

This thesis presented a variety of generic techniques and tools to achieve high-level system management using goals and policies. The overall approach built on an existing policy-based framework known as Accent (Advanced Component Control Enhancing Network Technologies [13]), and its associated policy description language called Appel (Adaptable and Programmable Policy Environment and Language [91]).

The approach has been applied to the domains of Internet telephony and sensor network/wind turbine management, the background and context of which were given in Chapter 2. The Appel policy language was previously specialised for Internet telephony as part of the Accent project, but reused in this work to evaluate the new techniques and tools. The domain of sensor network/wind turbine management was studied as part of the Prosen project (Proactive Condition Monitoring of Sensor Networks, `http://www.prosen.org.uk`). While the original Accent system demonstrated the use of policies for telephony, goals have not been applied in this domain prior to this work. Neither goals nor policies have been applied to sensor networks in the context of sensor network/wind turbine management. This thesis has demonstrated the ability to achieve high-level goal-directed, policy-based management in both these domains.

A policy language for sensor network/wind turbine management was devised as a specialisation of Appel and outlined in Chapter 3 together with examples of its use. In Chapter 5, the existing Accent system support for policy conflict handling was enhanced using an automated method to filter potential conflicts and generate template resolutions. A tool named Recap (Rigorously Evaluated Conflicts Among Policies) implements the approach.

Chapter 6 described the goal-directed approach. A goal language and system have been developed and implemented to work in conjunction with the Accent policy system. Goals are defined as an extension of Appel. Goal refinement into executable policies is achieved in a two-stage process. Statically, template policies known as prototypes are filtered against goals. Prototypes contributing to a goal are instantiated as executable policies. At run time, as the policy system is informed of system events, the applicable goal-related policies are optimised against a goal evaluation function, and are optionally parameterised based on the current state of the system.

The overall approach was generalised using OWL ontologies that model domain-specific knowledge. Chapter 4 described the ontologies devised to model the generic and domain-specific aspects of the Appel policy language, and how these ontologies are used within the Accent system to generalise the policy wizard user interface that was previously hard-coded for telephony. A system named Poppet (Policy Ontology Parser Program – Extensible Translation) was developed to enable applications (such as the goal system, policy system and conflict filtering tool) to utilise information within ontologies. Ontologies were devised to model the domains of Internet telephony and sensor network/wind turbine management.

7.2 Achievements

This thesis aimed to show that a system may be effectively managed using high-level goals and policies. Combined, the techniques and tools presented here have shown this to be the case in their application for Internet telephony and sensor network/wind turbine management.

The objectives of this thesis were outlined in Chapter 1. These objectives are now reconsidered together with explanation of how each has been achieved:

To define a language through which goals may be represented.

A goal language has been developed based on an extension of the Appel policy language. The syntax for a goal is a simplified version of a policy. A goal has a single measure (a function comprising constants and system state variables) which defines how a numerical value for the goal may be calculated, and a direction – namely "Maximise" or "Minimise". A goal may also have optional conditions which act as pre-conditions on its achievement. Goals for the domains of Internet telephony, sensor networks, and wind turbine management have been defined and demonstrated in this thesis work.

To design and implement a process of refining a set of high-level goals into a set of policies that achieve them.

A generic approach has been developed to achieve goal-directed configuration within a policy-based environment. The approach has been implemented for the Accent policy system and Appel policy language, but could in theory be reimplemented for other policy frameworks.

The goal domain is described using a set of controlled and uncontrolled variables which are modified based on events received by the policy system and actions performed as a result of policy execution. Each goal has an associated numerical measure that includes these state variables. A goal evaluation function combines and optionally weights individual goal measures so that multiple goals may be achieved simultaneously.

A two-stage algorithm has been designed and implemented as a system to refine goals into policies. Offline, prototypes (template policies) are instantiated as executable policies to support goals. Then at run time, triggered sets of goal-related policies are optimised, and parameterised if applicable. The outcome of run-time goal optimisation is a subset of initially triggered policies which, when executed, contribute most to the goals of the system. The developed goal system has been integrated and evaluated with the existing ACCENT policy system. It is also demonstrated for the domains of Internet telephony and sensor network/wind turbine management.

To define a policy language for the domain of sensor network management.

The APPEL policy language has been specialised for the new domain of sensor networks (described in section 3.4). The language supports policies for sensor networks in general, and is used in this thesis to define policies for environmental sensors and wind turbine control. The language is generic and flexible to support a wide range of sensor event notifications, and to configure devices not specific to any one type of sensor or to the wind power domain. The language builds on the core APPEL policy language described in section 3.2. Instead of a wide range of complex triggers and actions, the language uses a single trigger (*device_in*) and action (*device_out*) which may be parameterised accordingly for each entity. This approach defines a single common message format for use across all interfacing entities. The *device_in* trigger is used by external entities to notify the policy system of significant events (e.g. sensor data readings, warnings and network status information) and the *device_out* action is used in policies to configure sensors and send messages. The language is deliberately simple, but powerful in that it allows for reuse across multiple sensor-based environments, and is extensible to suit changes in underlying sensor network architecture. For example, avoiding hard-coded triggers and actions allows different sensors to be added to the network over time, or the capabilities of a sensor to be modified, without the need to change the policy language.

To enhance existing policy conflict handling using automated filtering of policy actions.

An approach has been implemented that automatically identifies and filters potential conflicts between policy actions at the policy language specification stage. In the original ACCENT policy system, conflict analysis and resolution generation mechanisms were manual, which made conflict identification time-consuming and error prone. The new approach concentrates on attempting to identify conflict prone actions at the earliest possible point when the policy language is either first defined, or being revised. The approach is non-formal, user-friendly, scalable and domain-independent. Each policy action is defined to have one or more *effects* on the managed system (e.g. on physical resources or abstract aspects of the system environment), and conflicting actions (and their parameters) are deemed potentially conflicting when they share a common effect. A second stage of the approach insists on human inspection in order to confirm detected conflicts or detect subtle domain conflicts that may have been missed.

A tool, named RECAP (Rigorously Evaluated Conflicts Among Policies) implements the approach and is presented in Chapter 5. In addition to analysing and displaying conflicts, RECAP also generates skeleton resolution policies (to handle conflicts at run time). This approach enhances existing conflict handling in the ACCENT system by automating the previously manual process of language analysis, and reduces the risk of missing potential conflicts. Use of the approach and tool has been demonstrated for two policy language specialisations – Internet telephony and sensor networks.

To generalise the whole approach by modelling domain-specific knowledge in ontologies.

A series of interrelated ontologies (described in Chapter 4), have been developed to model the core and domain-specific elements of the APPEL policy language using OWL (the Web Ontology Language [19]). Three types of ontologies were developed: *GenPol*, *WizPol*, and various domain ontologies. The *GenPol* ontology defines the core aspects of a policy in APPEL, the *WizPol* ontology defines additional user interface aspects (including formatting and display information), and a domain-specific ontology extends the constructs of both *GenPol* and *WizPol* to define the policy language for a specific application area. The approach has been demonstrated through the creation of domain ontologies for Internet telephony and sensor network/wind turbine management.

Domain information in these ontologies is used in the goal approach, the policy system and the RECAP conflict filtering tool. The ACCENT policy wizard user interface was previously developed for the domain of telephony. Using the designed ontologies, the policy wizard has been re-engineered to utilise ontology-defined language constructs as opposed to hard-coded domain information.

The result of this work is a domain-independent user interface to manage policies within the ACCENT system, and a flexible and extensible ontology model of the policy language.

To develop supporting software tools to integrate and utilise ontology-defined knowledge within the generic policy and goal systems and related components.

Integration of ontology-defined knowledge within the goal system, policy system and RECAP conflict filtering tool has been achieved using a custom-built parser named POPPET (Policy Ontology Parser Program – Extensible Translation). POPPET parses an OWL ontology and provides a generic API through which an application can query an ontology and extract information. The POPPET system is described in section 4.7.1.

7.3 Strengths

From the end user's perspective, the ability to express a system aim as a goal provides more powerful and autonomous high-level management and control of a system than was previously possible using policies alone. This additional level of abstraction from the mechanics of a system also appeals to less technical administrators. Using goals and policies, proactive system management is enhanced, as goals may be used to alter monitoring and dynamic configuration of the system based on the current state of the environment.

Success of the approach of this thesis is measured in the combined functionality of the developed techniques and tools. The goal system, policy system (including the policy wizard), and conflict filtering tool are domain-independent (neither contains hard-coded domain details), and are customisable through information defined within a domain ontology. This allows for reuse in new event-driven system environments. Reuse of the approach has been demonstrated in this thesis for two separate domains – telephony and sensor network/wind turbine management.

This thesis work has investigated and applied high-level goals and policies within the context of a wind farm, an area where these concepts have never previously been explored. Through the use of goals and policies, the approach in this thesis provides 'proactive' condition monitoring – the capability to detect potential faults and take action before they cause damage or failure. This is an advance in this area as current wind turbine monitoring is largely 'reactive' in nature. Using a goal-directed, policy-based approach in this domain is novel, and has shown through the examples in this thesis to be a viable means of achieving high-level management of wind turbines.

In addition, the approaches of this thesis have been used within the domain of home care networks, as part of ongoing work within the MATCH project (Mobilising Advanced Technologies for Care at Home, `http://www.match-project.org.uk`). This project is investigating new devices and services to support users in a home care setting. The Appel policy language specialisation for sensor networks (described in section 3.4) has been reused to define policies for home care networks. Goals for home care have similarly been defined and the goal approach has been used for goal-driven policy support within MATCH. An ontology for home care networks, which builds on the existing policy ontology framework discussed in section 4.4, has been defined and the Accent policy wizard customised for this domain.

More technically, individual components (the goal system, the policy system, the Recap tool, the Poppet parser and the ontologies) all communicate using socket connections which allow them to either run on one physical machine or be distributed across several. The flexibility of this architecture eases run-time administration and allows one system component to be extended or altered without adversely impacting others.

The approach does not use formal methods of analysis in any of its components. Instead, non-formal methodologies are used, which are less technically challenging (and hence more user friendly), and less likely to suffer from the scalability and performance issues associated with formal methods in a run-time environment.

7.4 Limitations

Although the approach is domain-independent, its reuse is constrained to event-driven systems that require a degree of autonomy in their control. Specifically, compatible systems must be capable of generating and reporting events (changes in the system environment that act as policy triggers) and must also permit dynamic configuration of its components (services, resources or variables in the system environment must be alterable via policy actions).

In turn, a suitable interface must be devised to enable the policy system to communicate with the managed system. This is additional work that must be carried out before the policy-based approach may be used, and requires expertise from technical experts to ensure that messages to and from the managed system are interpreted correctly within the policy system.

Due to cost and risk factors, it was not feasible to deploy the goal-directed, policy-based system within a live wind farm environment. Plausible goal and policy scenarios were therefore devised to mimic interaction between the policy system and wind turbines, in order to test and evaluate the approaches of the thesis. While these simulations are adequate to demonstrate and initially evaluate the approach, the inability to conduct trials in the target environment does mean evaluation is less than complete. There may be undiscovered usability or performance issues, or pitfalls in real-time efficiency of using goals and policies within the wind farm domain. Until further practical testing and deployment is carried out, full evaluation of the approach for wind turbine management cannot be made.

7.5 Further Work

The approaches in this thesis are readily applicable to new domains. To further demonstrate this claim, the techniques and tools may be applied in new fields through specialisation of the Appel policy language and the creation of a new domain ontology. The sensor network policy language is itself independent of a particular implementation and may be reused for new sensor network applications.

Of benefit would be a longer, more rigorous practical deployment of the goal-directed approach, to enhance the results reported here, and test the durability of the approach over time. Deployment for Internet telephony should take place in an environment with sufficient numbers of individual policy users, volume of call traffic and network activity so that goal and policy system performance may be challenged. Practical testing within a sensor network environment, or that of wind farms, should also be similarly arranged to test the robustness of the approach.

Scalability of the goal-directed approach and the conflict filtering tool may be evaluated further using a policy language with a large set of policy actions (more than, say, the 21 actions in the language for telephony). Additionally, experimentation with larger and more diverse sets of goals, prototypes and user-defined policies would test the robustness of the approach. Further scalability tests could include studying a large number of user-defined policies (e.g. 200) in a multi-user environment. This could investigate the impact in terms of potential conflicts between policies (goal-derived and user-defined), as well as the overall efficiency of the goal-directed approach.

Enhancements and improvements specific to individual techniques and tools of the overall approach are discussed in each main chapter. However, significant future goal-related work worth reiterating is further investigation towards the impact on optimal policy selection resulting from minor alterations to the goal evaluation function, for the domains of Internet telephony and sensor network/wind turbine management. This could use a form of sensitivity analysis based on the test cases documented in this thesis. The specifics of this are discussed in detail in section 6.6.

7.6 Concluding Remarks

During the past two decades, a range of goal-based systems and policy-based languages, systems, architectures and tools have been proposed and developed. However, despite the evolution of such approaches, there is currently no real large-scale deployment of goal-directed, policy-based management. Part of the reason for this may be due to a lack of a readily implementable, generic framework that is easily customisable to new domains. It is my hope that, together, the approaches and tools presented in this thesis will contribute to the development and adoption of such an environment in the future.

Appendix A

Wind Turbine Blade Pitch and Yaw Configuration

Chapter 3 describes a policy language for sensor networks in the context of a wind farm. Policies are used to monitor and configure wind turbines using a set of common parameters. This appendix clarifies exactly how blade pitch and yaw angle are configured in the examples in this thesis, as the interpretation of these parameter values vary in turbine literature.

Rotor Blade Pitch

The rotor blades are pitched into or out of the wind to control the rotation speed and the generated power. The maximum pitch drive yield is between 0 (zero) and 90 degrees – typically 45 degrees, although this may vary slightly depending on the particular blade design. Pitching the blades to 0 degrees signals minimal power output (the wind passes round the blades), and pitching to 45 degrees signals maximum available power output (the wind catches the blades and forces the rotor to turn), as shown in Figure A.1. To reduce or increase drive, the blade pitch angle is reduced or increased respectively.

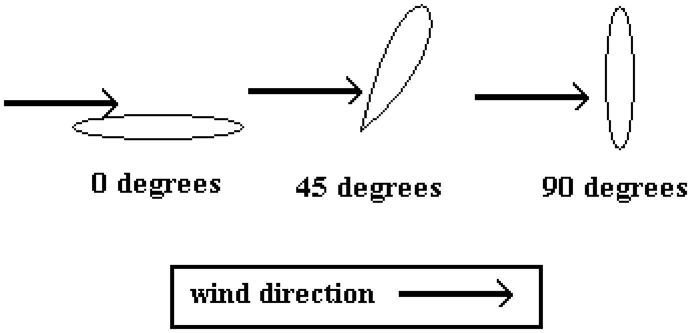


Figure A.1: Blade Pitch and Wind Direction

Rotor Yaw Angle

The rotor yaw angle rotates the blades into or out of the wind to alter the generated power. The angle of yaw is relative to the wind direction – 0 degrees signals the blades are facing directly into the wind (maximum power output), and 90 degrees signals the blades are facing out of the wind (minimal power output). To reduce or increase power output, the yaw angle is reduced or increased by an angle between 0 (zero) and 90 degrees respectively.

Appendix B

GenPol and *WizPol* Properties

Chapter 4 describes how the Appel policy language and Accent policy wizard interface features are modelled within separate OWL [19] ontologies named *GenPol* and *WizPol* respectively. The properties defined for each of these ontologies are shown here together with a brief description of their use. Italicised phrases refer to specific policy language concepts modelled as ontology classes that are explained within Chapter 4.

B.1 *GenPol* Ontology Properties

OWL Property Name	**Description of Use**
hasAction	A *PolicyRule* has at least one *Action*
hasActionArgument	An *Action* may have an *ActionArgument*
hasCondition	A *PolicyRule* has a *Condition*
hasConditionOperator	A *Condition* has a *ConditionOperator*
hasConditionParameter	A *Condition* has a *ConditionParameter*
hasConditionValue	A *Condition* has a *ConditionValue*
hasPermissibleAction	A *TriggerEvent* has some permissible *Action*(s)
hasPermissibleParameter	A *TriggerEvent* has some permissible *ConditionParameter*(s)
hasPolicy	A *PolicyDocument* has at least one *Policy*
hasPolicyAttribute	A *Policy* has some *PolicyAttribute*
hasPolicyRule	A *Policy* has at least one *PolicyRule*
hasPolicyVariable	A *Policy* has some *PolicyVariable*
hasPolicyVariableAttribute	A *PolicyVariable* may have some *PolicyVariableAttribute*(s)
hasTriggerArgument	A *TriggerEvent* may have a *TriggerArgument*
hasTriggerEvent	A *PolicyRule* has zero or more *TriggerEvent*(s)

B.2 *WizPol* Ontology Properties

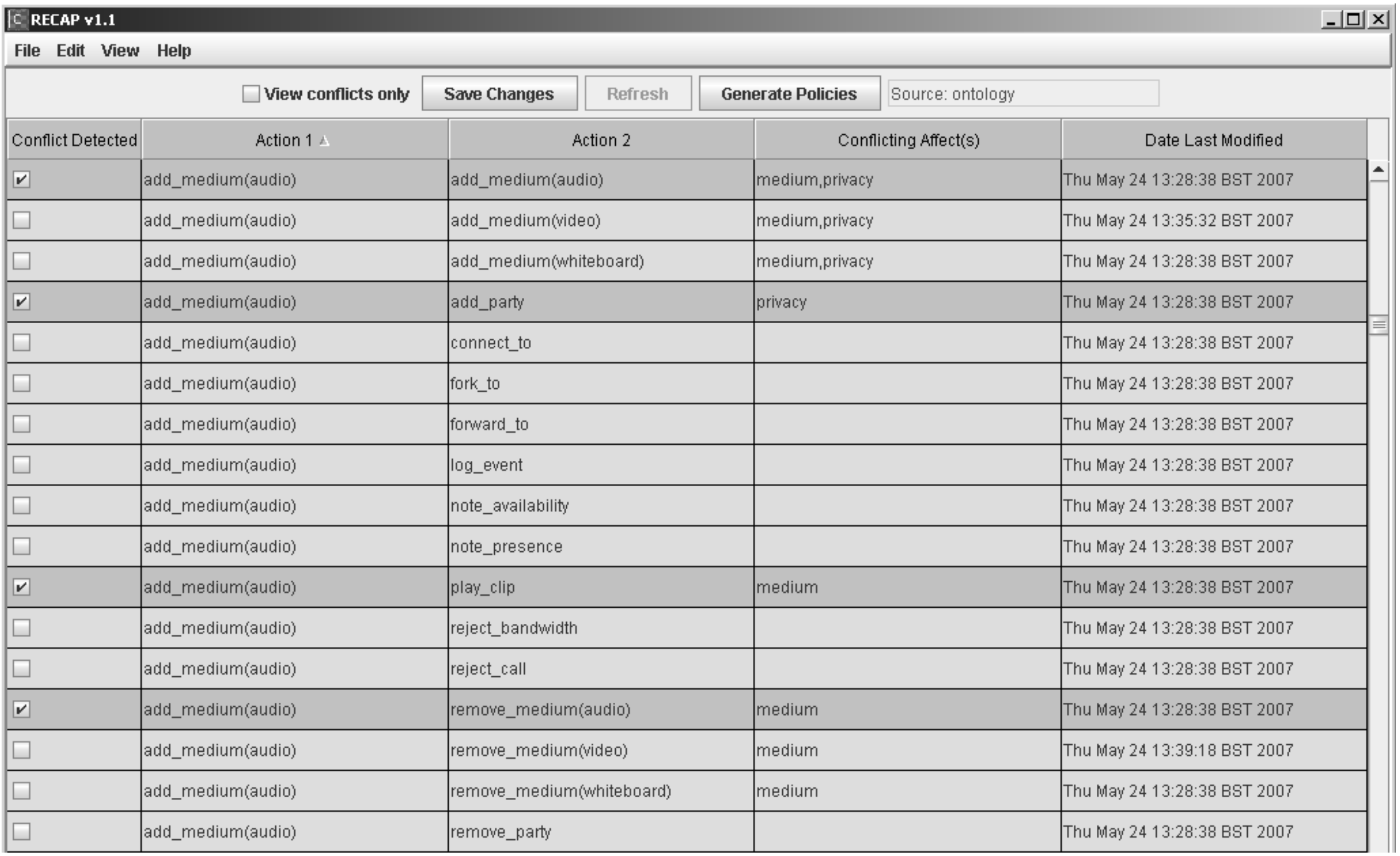
OWL Property Name	Description of Use
hasAbilityToQuery	Can be applied to any class in a domain-specific ontology to indicate a form of relationship with an internally classed variable.
hasCategory	Used to categorise triggers, condition parameters, actions and operators
hasDataType	Used to relate a particular defined subclass of *DataType* to a *TriggerArgument* or *ActionArgument*
hasInternalUse	Used to categorise a domain-specific trigger, condition parameter or action as internal in its use in a domain
hasUnitType	Used to associate particular units for display alongside a *TriggerArgument*, *ActionArgument* or *ConditionParameter*
hasUserLevel	Used to group triggers, conditions, actions and operators according to user level applicability
matchValue	This is an annotation property which has a special function and is a form of meta data. In OWL, this type of property acts as a class attribute rather than a restriction. It is applied in a similar way to the *rdfs:label*, *rdfs:comment* or *owl:versionInfo* predefined annotations defined for each class. The **matchValue** is used to define an alternative action or trigger class in a policy depending on the input value of an argument for a trigger or action. It contains a literal string value that links it with another ontology class. The string is interpreted and processed by an application reading the ontology.

Appendix C

Recap Tool Guide

This appendix gives a brief overview and example of the Recap tool (Rigorously Evaluated Conflicts Among Policies) that was developed to automate the policy conflict filtering algorithm described in Chapter 5. A screen-shot of the tool user interface is shown in Figure C.1. Taking the first line as an example, the tool shows pairs of actions (*add_medium(audio)* and *add_medium(audio)*), why they conflict (shared effect on *medium* and *privacy*), and when this conflict was identified (either automatically or manually).

The tool automatically constructs an underlying matrix of all policy action pairs and highlights those deemed to be potential conflicts. Potential conflicts are displayed along with a note of their common effects. Each action pairing appears as a single row in a table. Rows containing a suspected conflict appear in darker shading and the check-box in the left-most column is checked. Users may explore the matrix by scrolling up and down the table, confirming or refining each conflicting action pair. If on closer inspection the user decides there is no conflict, the pairing in question can be flagged as conflict-free by unchecking the check-box in the associated row. The table may be sorted in either ascending or descending order by clicking on any column header (e.g. by flagged conflict, first or second action, conflicting effect(s) or the date an entry was last modified).



Conflict Detected	Action 1	Action 2	Conflicting Affect(s)	Date Last Modified
☑	add_medium(audio)	add_medium(audio)	medium,privacy	Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	add_medium(video)	medium,privacy	Thu May 24 13:35:32 BST 2007
□	add_medium(audio)	add_medium(whiteboard)	medium,privacy	Thu May 24 13:28:38 BST 2007
☑	add_medium(audio)	add_party	privacy	Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	connect_to		Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	fork_to		Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	forward_to		Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	log_event		Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	note_availability		Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	note_presence		Thu May 24 13:28:38 BST 2007
☑	add_medium(audio)	play_clip	medium	Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	reject_bandwidth		Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	reject_call		Thu May 24 13:28:38 BST 2007
☑	add_medium(audio)	remove_medium(audio)	medium	Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	remove_medium(video)	medium	Thu May 24 13:39:18 BST 2007
□	add_medium(audio)	remove_medium(whiteboard)	medium	Thu May 24 13:28:38 BST 2007
□	add_medium(audio)	remove_party		Thu May 24 13:28:38 BST 2007

Figure C.1: Screen-shot of RECAP

Bibliography

[1] 3APL: An Abstract Agent Programming Language. http://www.cs.uu.nl/3apl. Valid October 2008.

[2] DAML+OIL Language Reference. http://www.daml.org/2001/03/daml+oil-index. Valid December 2008.

[3] DARPA Agent Markup Language (DAML) Program. http://www.daml.org. Valid December 2008.

[4] Jambalaya knowledge-base visualiser plug-in for Protege. http://www.thechiselgroup.org/jambalaya. Valid October 2008.

[5] LM Blade Monitoring Ltd. Home page. http://www.lmglasfiber.com. Valid June 2008.

[6] Ontology Inference Layer (OIL). http://www.ontoknowledge.org/oil. Valid December 2008.

[7] OWLViz Graphical ontology visualiser plug-in for Protege. http://www.co-ode.org/downloads/owlviz. Valid October 2008.

[8] Pellet OWL DL Reasoner. http://pellet.owldl.com. Valid October 2008.

[9] Protege ontology editor and knowledge-base framework. http://protege.stanford.edu. Valid October 2008.

[10] RacerPro reasoner and inference server. http://www.racer-systems.com. Valid October 2008.

[11] SKF Engineering Group. http://www.skf.com. Valid October 2008.

[12] Smart Fibres Ltd. http://www.smartfibres.com. Valid October 2008.

[13] Accent project home page. http://www.cs.stir.ac.uk/accent. Valid November 2008.

[14] The Danish wind industry association. Home page (English). http://www.windpower.org/en/core.htm. Valid December 2008.

[15] The eXtensible Markup Language (XML). http://www.w3.org/XML. Valid December 2008.

[16] The Jena Semantic Web Framework. http://jena.sourceforge.net. Valid October 2008.

[17] The Resource Description Framework (RDF). http://www.w3.org/RDF. Valid December 2008.

[18] The Web Ontology Language (OWL) Semantics and Abstract Syntax. http://www.w3.org/TR/owl-semantics. Valid December 2008.

[19] The Web Ontology Language Reference (OWL). http://www.w3.org/TR/owl-ref. Valid December 2008.

[20] Unified Modelling Language (UML). http://www.uml.org. Valid December 2008.

[21] University of Stirling XML Schemas and Ontologies Repository. http://www.cs.stir.ac.uk/schemas. Valid December 2008.

[22] WonderWeb OWL Ontology Validator. http://www.mygrid.org.uk/OWL/Validator. Valid June 2008.

[23] CREDITI: A KAOS tutorial. http://www.objectiver.com/download/documents/KaosTutorial.pdf, Sept. 1993. Valid September 2008.

[24] WindCon: SKF Condition Monitoring System. http://www.engineeringtalk.com/news/skf/skf214.html., June 2003.

[25] BWEA. Briefing Sheet: Wind and the UK's 10% Target. http://www.bwea.com/energy/briefing-sheets.html., Sept. 2005.

[26] BWEA. Briefing Sheet: Wind Turbine Technology. http://www.bwea.com/energy/briefing-sheets.html., Sept. 2005.

[27] Prosen Consortium. prosen project proposal, Jan. 2005. EPSRC, Swindon.

[28] Fibre Optics Magazine. SCADA Overview. http://www.fiber-optics.info/articles/SCADA-overview.htm., June 2006.

[29] D. Agrawal, K.-W. Lee, and J. Lobo. Policy-based management of networked computing systems. *Network and Service Management*, 43(10):69–75, Oct. 2005.

[30] D. Alrajeh, A. Russo, and S. Uchitel. Inferring operational requirements from scenarios and goal models using inductive learning. In *International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[31] D. Amyot, L. Charfi, N. Gorse, T. Gray, L. M. S. Logrippo, J. Sincennes, B. Stepien, and T. Ware. Feature description and feature interaction analysis with use case maps and LOTOS. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 274–289. IOS Press, Amsterdam, Netherlands, May 2000.

[32] Anon. Managing the Wind: Reducing Kilowatt-Hour Costs With Condition Monitoring. *Refocus*, 6(3):48–51, May/June 2005.

[33] P. K. Au and J. M. Atlee. Evaluation of a state-based model of feature interactions. In P. Dini, R. Boutaba, and L. M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 153–167. IOS Press, Amsterdam, Netherlands, June 1997.

[34] A. Bandara. *A Formal Approach to Analysis and Refinement of Policies.* PhD thesis, Imperial College, UK, July 2005.

[35] A. K. Bandara, E. C. Lupu, J. Moffett, and A. Russo. A goal-based approach to policy refinement. In *IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2004)*, pages 229–239. IEEE Press, 2004.

[36] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-278, MITRE Corporation, Bedford, MA, Apr. 1977.

[37] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, Hanscom AFB, Bedford, MA, Apr. 1977.

[38] J. Blom, R. Bol, and L. Kempe. Automatic detection of feature interactions in temporal logic. In K. E. Cheng and T. Ohta, editors, *Proc. 3rd. International Workshop on Feature Interactions in Telecommunications*, pages 1–19. IOS Press, Amsterdam, Netherlands, 1995.

[39] R. Boutaba and I. Aib. Policy-based management: A historical perspective. *Journal of Network and Systems Management*, 15(4):447–480, Dec. 2007.

[40] J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. The COPS (Common Open Policy Service) Protocol. RFC 2748, IETF, Jan. 2000.

[41] R. M. M. Braga, M. Mattoso, and C. M. L. Werner. The use of mediation and ontology technologies for software component information retrieval. In *Proc. of the 2001 Symposium on Software Reusability*, pages 19–28, New York, NY, USA, 2001. ACM.

[42] N. Brownlee. SRL: A language for describing traffic flows and specifiing actions for flow groups. IETF Internet Draft, Aug. 1999. Expired February 2000.

[43] M. Cain. Managing run-time interactions between call-processing features. *IEEE Communications Magazine*, pages 44–50, Feb. 1992.

[44] M. H. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41:115–141, Jan. 2003.

[45] E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine*, 31(8):18–23, Aug. 1993.

[46] J. Cameron, K. Cheng, F. J. Lin, H. Liu, and B. Pinheiro. A formal AIN service creation, feature interactions analysis and management environment: An industrial application. In P. Dini, R. Boutaba, and L. M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 342–346. IOS Press, Amsterdam, Netherlands, June 1997.

[47] G. A. Campbell. An Overview of Ontology Application for Policy-Based Management using POPPET. Technical Report CSM-168, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

[48] G. A. Campbell. Ontology for Call Control. Technical Report CSM-170, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

[49] G. A. Campbell. Ontology Stack for a Policy Wizard. Technical Report CSM-169, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

[50] G. A. Campbell. Ontologies for Resolution Policy Definition and Policy Conflict Detection. Technical Report CSM-172, Department of Computing Science and Mathematics, University of Stirling, UK, Feb. 2007.

[51] G. A. Campbell. Sensor network policy conflicts. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th International Conference on Feature Interactions in Software and Communications Systems*, France, Sept. 2007. IMAG Laboratory, University of Grenoble.

[52] G. A. Campbell and K. J. Turner. Ontologies to support Call Control Policies. In D. C. N. Meghanathan and Y. Takasaki, editors, *AICT'07*, 3rd Advanced International Conference on Telecommunications, pages 5.1–5.6. IEEE Computer Society, May 2007.

[53] G. A. Campbell and K. J. Turner. Policy Conflict Filtering for Call Control. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th International Conference*

on Feature Interactions in Software and Communications Systems, pages 93–108, France, Sept. 2007. IMAG Laboratory, University of Grenoble.

[54] G. A. Campbell and K. J. Turner. Goals and Policies for Sensor Network Management. In M. Benveniste, B. Braem, C. Dini, G. Fortino, R. Karnapke, J. L. Mauri, and M. S. H. Monsi, editors, *Proc. 2nd International Conference on Sensor Technologies and Applications (SENSORCOMM'08)*, pages 354–359. IEEE Computer Society, Los Alamitos, California, Aug. 2008.

[55] J. Chen, Z. Zhao, D. Qu, and P. Zhang. A policy-based approach for reconfiguration management and enforcement in autonomic communication systems. *Wireless Personal Communications Magazine, IEEE*, 45(2):145–161, 2008.

[56] J. Chomicki, J. Lobo, and S. Naqvi. A logical programming approach to conflict resolution in policy management. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proc. Principles of Knowledge Representation and Reasoning*, pages 121–132. Morgan Kaufmann, 2000.

[57] N. Damianou, E. C. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Policy Workshop 2001*, number 1995 in Lecture Notes in Computer Science. Springer, Berlin, Germany, Jan. 2001.

[58] A. Daneels and W. Salter. What is SCADA? In *International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 339–343, Trieste, Italy, Oct. 1999.

[59] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *4th ACM Symposium on the Foundations of Software Engineering (FSE4)*, pages 179–190, 1996.

[60] S. Davy, B. Jennings, and J. Strassner. The policy continuum – policy authoring and conflict analysis. *Computer Communications*, 31:2981–2995, Aug. 2008.

[61] P. Dini, A. Clemm, T. Gray, F. J. Lin, L. Logrippo, and S. Reiff-Marganiec. Policy-enabled mechanisms for feature interactions: Reality, expectations, challenges. *Computer Networks*, 45(5):585–603, Mar. 2004.

[62] A. P. Felty and K. S. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology*, 12(1):3–27, Jan. 2003.

[63] D. Ferraiolo and R. Kuhn. Role-based access controls. In *In Proceedings of 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[64] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[65] M. Frappier, A. Mili, and J. Desharnais. Detecting feature interactions on relational specifications. In P. Dini, R. Boutaba, and L. M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 123–137. IOS Press, Amsterdam, Netherlands, June 1997.

[66] N. Fritsche. Runtime resolution of feature interactions in architectures with separated call and feature control. In K. E. Cheng and T. Ohta, editors, *Proc. 3rd. International Workshop on Feature Interactions in Telecommunications*, pages 43–63. IOS Press, Amsterdam, Netherlands, 1995.

[67] S. Godik and T. Moses. eXtensible Access Control Markup Language (XACML) Version 1.0. OASIS, Feb. 2003.

[68] M. Heisel and J. Souquiéres. A heuristic approach to detect feature interactions in requirements. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 165–171. IOS Press, Amsterdam, Netherlands, Sept. 1998.

[69] P. Jackson. *Introduction to Expert Systems.* Addison-Wesley, Harlow, England, third edition, 1999.

[70] L. Kagal. Rei: A policy language for the me-centric project. Technical Report HPL-2002-270, HP Labs, 2002.

[71] D. O. Keck. A tool for the identification of interaction-prone call scenarios. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 276–290. IOS Press, Amsterdam, Netherlands, Sept. 1998.

[72] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, pages 779–796, Oct. 1998.

[73] S. Keoh, K. Twidle, N. Pryce, E. Lupu, A. Schaeffer Filho, N. Dulay, M. Sloman, S. Heeps, S. Strowes, and J. Sventek. Policy-based Management for Body-Sensor Networks. In *Proc. 4th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2007)*, pages 92–98, Aachen, Germany, Mar. 2007.

[74] K. Kimbler. Addressing the interaction problem at the enterprise level. In P. Dini, R. Boutaba, and L. M. S. Logrippo, editors, *Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks*, pages 13–22. IOS Press, Amsterdam, Netherlands, June 1997.

[75] C. Klein, C. Prehofer, and B. Rumpe. Feature specification and refinement with state transition diagrams. In P. Dini, R. Boutaba, and L. M. S. Logrippo, editors,

Proc. 4th. International Workshop on Feature Interactions in Telecommunication Networks, pages 284–297. IOS Press, Amsterdam, Netherlands, June 1997.

[76] B. W. Lampson. Protection. *Proc. of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, Mar. 1971.

[77] A. F. Layouni, L. Logrippo, and K. J. Turner. Conflict detection in call control using first-order logic model checking. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th. Feature Interactions in Telecommunications and Software Systems*, pages 66–82. IOS Press, Amsterdam, Netherlands, May 2008.

[78] J. Lee. What is ontology? IBM Research `http://www.alphaworks.ibm.com/contentnr/semanticsfaqs`, Aug. 2004.

[79] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. FSE'10 -10th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Charleston, Nov. 2002.

[80] J. Lobo, R. Bhatia, and S. Jaqvi. A policy description language. In *Proc. National Conference of the American Association for Artificial Intelligence*. Orlando, Florida, USA, July 1999.

[81] E. C. Lupu and M. Sloman. Conflict analysis for management policies. In *Proc. 5th. International Symposium on Integrated Network Management*, pages 430–443. Chapman-Hall, London, UK, 1997.

[82] J. Moffett and M. S. Sloman. Policy hierarchies for distributed systems management. *IEEE Journal of Selected Areas in Communications*, 11(9):404–14, 1993.

[83] C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic based detection of conflicts in Appel policies. In A. Movaghar and J. Rutten, editors, *Proc. Int. Symposium on Fundamentals of Software Engineering*, volume 4767 of *Lecture Notes in Computer Science*, pages 257–271. Springer, Berlin, Germany, Oct. 2007.

[84] M. Nakamura, T. Kikuno, J. Hassine, and L. M. S. Logrippo. Feature interaction filtering with Use Case Maps at requirements stage. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 163–178. IOS Press, Amsterdam, Netherlands, May 2000.

[85] E. P. D. Pednault. ADL and the State-Transition Model of Action. *Logic Computation*, 4(5):467–512, 1994.

[86] M. C. Plath and M. D. Ryan. Defining features for CSP: Reflections on the feature interaction contest. In S. T. Gilmore and M. D. Ryan, editors, *Language Constructs for Describing Features – Proceedings of the FIREworks Workshop*, pages 163–175. Springer, Berlin, Germany, Jan. 2001.

[87] C. Prehofer. Play-and-play composition of features and feature interactions with Statechart diagrams. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 43–58. IOS Press, Amsterdam, Netherlands, June 2003.

[88] P. Procter, editor. *Cambridge International Dictionary of English.* Cambridge University Press, Cambridge, UK, 2001.

[89] S. Reiff-Marganiec and K. J. Turner. Feature interaction in policies. *Computer Networks*, 45(5):569–584, Mar. 2004.

[90] S. Reiff-Marganiec and K. J. Turner. The ACCENT policy server. Technical Report CSM-164, Department of Computing Science and Mathematics, University of Stirling, UK, Dec. 2005.

[91] S. Reiff-Marganiec, K. J. Turner, L. Blair, G. A. Campbell, and F. Wang. APPEL: An adaptable and programmable policy environment and language. Technical Report CSM-161, Department of Computing Science and Mathematics, University of Stirling, UK, Dec. 2008.

[92] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou. A functional solution for goal-ooriented policy refinement. In *Proc. 7th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 133–144, Washington, DC, USA, 2006. IEEE Computer Society.

[93] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou. A methodological approach toward the refinement problem in policy-based management systems. *Communications Magazine*, 44(10):60–68, 2006.

[94] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, and A. L. Lafuente. Using linear temporal model checking for goal-oriented policy refinement frameworks. In *Proc. 6th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 181–190, Washington, DC, USA, 2005. IEEE Computer Society.

[95] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, New Jersey, second edition, 2003.

[96] G. N. Stone, B. Lundy, and G. G. Xie. Network policy languages: A survey and a new approach. In *IEEE Network*, volume 15(1), pages 10–21. IEEE, Jan./Feb. 2001.

[97] V. Sugumaran and V. C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, 2003.

[98] K. J. Turner and L. Blair. Policies and conflicts in call control. *Computer Networks*, 51(2):496–514, Feb. 2007.

[99] K. J. Turner, G. A. Campbell, and F. Wang. Policies for Sensor Networks and Home Care Networks. In M. Erradi, editor, *Proc. 7th. Int. Conf. on New Technologies for Distributed Systems*, pages 273–284. Cana Print, Rabat, Morocco, June 2007.

[100] H. Velthuijsen. Distributed artificial intelligence for runtime feature-interaction resolution. *Computer*, pages 48–55, Aug. 1993.

[101] X. Wu and H. Schulzrinne. Handling feature interactions in the language for end system services. *Computer Networks*, 51(2):515–535, 2007.

[102] T. Yoneda and T. Ohta. A formal approach for definition and detection of feature interactions. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 202–216. IOS Press, Amsterdam, Netherlands, Sept. 1998.