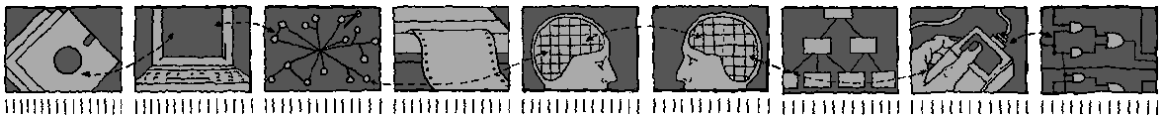
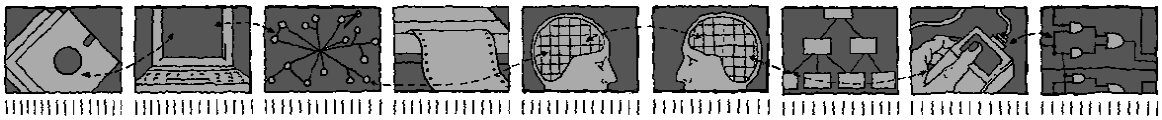


Department of Computing Science and Mathematics
University of Stirling



APPEL: An Adaptable and Programmable Policy Environment and Language

Kenneth J. Turner, Stephan Reiff-Marganiec, Lynne Blair, Gavin A. Cambpell and Feng Wang

Technical Report CSM-161

ISSN 1460-9673

April 2009

Department of Computing Science and Mathematics
University of Stirling

APPEL: An Adaptable and Programmable Policy Environment and Language

Kenneth J. Turner, Stephan Reiff-Marganiec, Lynne Blair, Gavin A. Cambpell and Feng Wang

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44 1786 467 421, Facsimile +44 1786 464 551
Email kjt@cs.stir.ac.uk, srm13@le.ac.uk, lb@comp.lancs.ac.uk,
gac@cs.stir.ac.uk, fw@cs.stir.ac.uk

Technical Report CSM-161

ISSN 1460-9673

April 2009

Abstract

The ACCENT project (Advanced Component Control Enhancing Network Technologies) developed a practical and comprehensive policy system for call control/Internet telephony. The policy system has subsequently been extended for management of sensor networks/wind farms and of home care/telecare.

This report focuses on APPEL (Adaptable and Programmable Policy Environment and Language). It provides an overview of the language, and presents the language in XML schema form. The core language has been instantiated for call control, for sensor networks, and for home care. Sample goals and policies of different kinds are provided to illustrate these applications.

Changes in Version 2

Relative to the version of November 2003, this Technical Report has been substantially revised as follows:

- The policy language schema has been presented first in generic form (section 2). It has been significantly simplified but also extended as described below. The policy language description has been substantially revised to match the new version of the language.
- The policy language has then been instantiated for the call control domain (section 3.1). The description of triggers, conditions and actions is now much more precise. These elements have been significantly revised as described below.
- The example call control policies have been revised to match the new definition of the language (section 3.2).

Changes in Version 3

Relative to version 2 of APPEL, the policy language has been updated in a large number of ways:

- elements are now listed in logical order
- groups have been created for *actions*, *conditions*, *policy_rules*, *triggers* such that a simple or compound element can be used (e.g. one *action* or *actions* containing combinations)
- by convention (i.e. not in the schema) a string may contain substrings preceded by '?' (meaning a template value to be made actual at definition time) or all in capitals (meaning a policy variable to be instantiated at run-time)
- *action_name*:
  - extracted as a simple type, i.e. no longer a sub-element of *action*
  - removed *connect*, *forward*, *handle_calls*, *interrupt*, *make_call*, *make_call_to*, *speak_to*, *transfer*
  - *add_media(medium)* → *add_medium(medium)*, *block_call* → *reject_call(reason)*, *publish_availability(addresses,subject)* → *note_availability(subject)*
  - added *add_party(address)*, *confirm_bandwidth*, *fork_to(address)*, *log_event(message)*, *note_presence(subject)*, *play_clip(URL)*, *reject_bandwidth(limit)*, *remove_medium(medium)*, *send_message(URL,message)*
- *actions*:
  - now an *action*, *actions*, or combinations of actions
  - added an *else* operator to mean conditional choice of actions immediately after conditions, or an *or* if there are none

- *acts*: removed for simplicity
- *alwaysnever_modality* → *relevance*
- *compop* → *operator*
  - for strings, '*v1 lt v2*' should mean *v1* is a strict substring of *v2*, etc.
- *conditions*: now a *condition*, *conditions*, or combinations of conditions
- *conds*: removed for simplicity
- *operator*:
  - *earlier* and *later* removed for simplicity and generality (use *between*, *gt*, etc. for times)
  - *in* → *between*, *notin* → *not_between* (use *eq*, *ne*)
- *param* → *parameter*
- *parameter*:
  - removed *busy*, *call*, *caller_id*, *forward_target*
  - *callcontent* → *call_content*, *calltype* → *call_type*, *media* → *medium*
  - added *active_content*, *bandwidth*, *capability_set*, *date*, *destination_address*, *network_type*, *priority*, *signalling_address*, *source_address*, *traffic_load*
- *policy*:
  - *appliesTo* → *applies_to*
  - added *profile*, *valid_from*, *valid_to* attributes
  - *lastchanged* → *changed*, and moved from *policydoc* to *policy*
- *policydoc* → *policy_document*
- *policyrule* → *policy_rule*
- *policyrules* →o *policy_rules*
  - *guarded_choice* → *guarded*, now with a nested not following condition
  - *unguarded_choice* → *unguarded*
- *polrules*: removed for simplicity
- *preference_opi_modality* → *preference*
  - removed *obligation*, *permission*, *interdiction*, *authorisation*
  - *prefer to* → *prefer*, *want not* → *do not want*, *wish not* → *do not wish*
  - used underscores instead of spaces in names
- *trigger_name*:
  - extracted as a simple type, i.e. no longer a sub-element of *trigger*
  - removed *busy*, *busy_incoming*, *busy_outgoing*, *call*, *call_incoming*, *call_outgoing*
  - *call attempt* → *call*, *incoming/outgoing* prefix → *_incoming/_outgoing as suffix*
  - moved *event* from *condition* to *trigger*
  - added *absent(address)*, *available(address)*, *bandwidth_request*, *present(address)*, *unavailable(address)*

- *triggers*:
  - changed infix *or* to prefix form
  - added *and* combination
- *value*:
  - a time should now have a colon, i.e. HH:MM:SS
  - a list can be a single element or comma-separated values, i.e. not a value in brackets
- *variable*: added as an alternative to i, with attributes *applies_to*, *id*, *owner*, *value*

Changes in Version 4

Relative to version 3 of APPEL, the following changes have been made:

- *actions*:
  - added *add_caller(method)*, with parameter *conference*, *hold*, *monitor*, *release* or *wait*
  - added *remove_party(address)*
  - *callee* removed as parameter established by *event*
- *relevance*: removed
- *variable*: prefix for variable values changed to ':'

Changes in Version 5

Relative to version 4 of APPEL, the following changes have been made:

- *xsd:token* used in place of *xsd:string* where it might be convenient to split element contents across lines
- *action*:
  - *add_caller(method)* added
  - *forward_to(Address)* action described
  - renamed *log_call* as *log_event*, which now takes a message rather than URL as parameter
  - *reject_call(reason)* parameter extended in meaning
- *operator*:
  - *between* and *not_between* renamed *in* and *out*
- *parameter*:
  - *day* added
  - *priority* now defined as a positive real
  - use of operators with parameters made explicit
- *policy*:
  - certain characters forbidden in policy identifiers
- *preference*:
  - *do_not_want*, *do_not_wish*, *want*, *wish* removed

- *resolution*:
  - resolution policies added
  - resolution associated triggers, conditions and actions defined
  - resolution policy examples added
- *trigger*:
  - *connect*, *disconnect*, *no_answer* (and their variants) now permit a *send_message* action
- *variable*:
  - certain characters forbidden in variable identifiers
  - variables no longer substituted inside *applies_to*
  - multiple variables may be substituted in a string by beginning and ending each variable with ':'
  - the *changed* attribute has been added

Changes in Version 6

Relative to the version of June 2005, this Technical Report has incorporated only small editorial revisions have been made.

Changes in Version 7

Relative to version 6 of APPEL, the policy language has been extended as follows to support timing, state, and the new application domains of sensor networks and home care.

- Chapter 1 now contains a combined introduction to the original ACCENT context and to APPEL.
- Section 2.1 explains how the APPEL schemas have been refactored into three levels to make them more modular. Most of chapter 2 describes the core level. Sections 2.13 and 2.14 describe how the core language is extended by regular policies and resolution policies respectively.
- Section 2.3 now clarifies that policies may refer to entities as well as people.
- Section 2.6 now uses a *trigger_argument* type for trigger arguments. A *trigger_name* is now the union of core, policy and domain definitions. The following changes have also been made:
  - A *trigger* may now have attributes *arg1..5*; this is to accommodate sensor network policies, where up to five arguments may be given. The use of this is illustrated in section 4.4.1.
  - The trigger *timer_expiry(identifier)* has been added. The use of this is illustrated in section 3.2.9.
- Section 2.7 now defines a *parameter_name* as the union of core, policy and domain definitions. The following changes have also been made:
  - A condition now takes two *operands* (a parameter or a value), so that all parameter and value combinations can be compared.
  - Epoch parameters have been removed from call control and defined as core parameters: *date*, *day* and *time*.
  - Days of the week are now numbered from 1 (Monday) to 7 (Sunday) so that time conditions on weekdays and weekends can be more readily defined.
- Section 2.8 now defines an *action_name* as the union of core, policy and domain definitions. The following changes have also been made:
  - An *action* may now have attributes *arg1..5*; this is to accommodate sensor network policies, where up to five arguments may be given. The use of this is illustrated in section 4.2.3.

  - Internal actions have been removed from call control and defined as core actions: *log_event(message)* and *send_message(URL,message)*.
  - Actions on variables *set_variable(identifier,expression)* and *unset_variable(identifier)* have been added. The use of these is illustrated in section 3.2.8.
  - Actions on timers *start_timer(identifier,period)*, *restart_timer(identifier)* and *stop_timer(identifier)* have been added. The use of these is illustrated in sections 3.2.9 and 3.4.7.

- Section 2.9 now clarifies where variables may be used.
- Section 2.10 now allows expressions with various operators and functions. The use of these is illustrated in sections 3.2.8, 3.4.8 and 4.4.2. White space in variable names is now disallowed.
- Section 2.11 adds the ability to have policy timers, and to have policies triggered by time conditions. The use of these is illustrated in sections 3.2.9 and 3.2.10.
- Section 2.12 adds the ability to store past triggers and actions automatically, and to check this history in policy conditions. The use of this is illustrated in section 3.2.11.
- Section 2.14 now binds *preference0..9* and *variable0..9* in resolution policies. This is to accommodate sensor network policies, where a resolution may have two triggers with up to five arguments each. The use of this is illustrated in, for example, section 3.4.
- The report has been restructured so that each application domain is now described in one chapter, currently chapter 3 for call control, chapter 4 for sensor networks, and chapter 5 for home care networks.

Changes in Version 8

Relative to the version of December 2007, this Technical Report has been revised as follows:

- All XML has now been reformatted.
- Times are now given as HH:MM:SS throughout.
- Chapter 1 has been renamed 'Introduction'. Chapter 6 has been added to give some brief conclusions.
- Section 2.3 introduces new *effect* and *supports_goal* attributes that support goal refinement.
- Section 2.7 now disallows the use of **in/out** when comparing a value and a parameter. In the case of a value *preference0..9* or *variable0..9*, a ':' prefix is now suggested (and has been used throughout).
- Section 2.10 now allows use of the '%' operator on numbers. The division operator '/' has now been defined explicitly for integer and floating point operands. The interpretation of a string as a floating point number has been specified. It is explicitly stated that a quoted string will never be interpreted as a number.
- Section 2.14.3 now refers to actions in general rather than to just call actions.
- Section 2.15 introduces prototype policies. Corresponding examples have been introduced for call control (section 3.5), sensor networks (section 4.5) and home care (section 5.5).
- Section 2.16 introduces goals. Corresponding examples have been introduced for call control (section 3.6), sensor networks (section 4.6) and home care (section 5.6).
- Section 3.1.2 now contains a common definition of trigger, condition and action parameters.
- Section 3.1.5 introduces a new *disconnect* action to clear the current call.
- Section 3.4.6 now contains the correct XML for the caller-medium conflict example.
- Sections 4.1.2 and 5.1.2 now define more specific values for triggers and actions in the home care and sensor network domains.
- The sensor network examples in section 4.2 and 4.4 have been updated.
- The home care examples in section 5.2 and 5.4 have been updated.

Contents

List of Figures

Chapter 1

Introduction

1.1 ACCENT Overview

The ACCENT project (Advanced Component Control Enhancing Network Technologies, *http://www.cs.stir.ac.uk/accent*) developed a practical and comprehensive policy system for call control/Internet telephony. The policy system has subsequently been extended for management of sensor networks/wind farms on the PROSEN project (Proactive Control of Sensor Networks, *http://www.prosen.org.uk*). The policy system has also been extended for management of home care/telecare on the MATCH project (Mobilising Advanced Technologies for Care at Home, *http://www.match-project.org.uk*).

Other policy description languages exist, and were investigated during the work on APPEL[1] (Adaptable and Programmable Policy Environment and Language, the 'A' originally referring to the ACCENT project). An example of a well-supported and popular policy language for access control and system configuration is Ponder [5], but there are many other approaches (e.g. [1, 6]). However, these other languages have not been found to be suitable for the kinds of domains considered here [7].

This document focuses on the APPEL policy language, and elaborates what is in [7]. The following sections provide an overview of the language, discuss concepts for various kinds of policies, and discuss syntax and semantics by reference to the XML schemas. In subsequent chapters, the use of APPEL in various application domains is described and illustrated.

[7, 8, 11] give some general background to the ACCENT project. There are technical reports describing the ACCENT Policy Server [9] and the APPEL Policy Wizard [10]. Ontologies for the policy wizard are discussed in [2, 3, 4].

1.2 APPEL Overview

The APPEL policy language is defined by an XML grammar that appears in chapter 2. XML has been chosen due to its good support with respect to tools and parsers. Also, it is the common standard for interchange of structured data in distributed systems.

When APPEL was designed, it was hoped that a good language design could rule out many of the possible policy conflicts. However, it is likely that no policy language can be designed to avoid conflicts while still remaining sufficiently expressive. Features (in the sense of call control) are different in a number of ways from policies, e.g. policies are more abstract, higher level, declarative and user-oriented. Policies should also be written by lay end users and not domain experts (as one might be expected with traditional features).

Another major aspect considered when designing the language was that it should be usable and useful to a number of users, ranging from lay end users to system administrators to domain experts. The language is as expressive as deemed necessary to fulfil the requirements of the most knowledgeable users (with respect to the application domain), while the complexity is hidden behind suitable interfaces for the less knowledgeable users. User-friendly wizards have been designed for formulating policies [10]. These allow lay users to specify policies by choosing predefined templates and customising them. A subset of natural language is used rather than XML. However, more knowledgeable users can work with XML directly.

[1] The name derives from the French 'appel' meaning a call.

Although APPEL was originally aimed at call control, it was designed to be extensible. Indeed, it has now been used for policy-based management of sensor networks and of home care. The policy language splits the issues in a relatively clean way. The core of the language is domain-independent, i.e. the structure of policies and policy rules as well as their combinations. The body of a policy rule defines triggers, conditions and actions. The general concepts here are domain-independent, though the specific names used in these elements are defined for each domain.

Chapter 2

Core Language

2.1 Language Introduction

The syntax of ACCENT is defined by an XML schema. The semantics is given by the implementation of the *PolicyEvaluate* class (see [9]).

A policy grammar is defined using standard XML Schema notation. XML schemas have the same purpose as XML DTDs, namely specifying what is syntactically legal for a document adhering to the schema. They can be used for validation purposes. XML schemas allow for richer data types than XML DTDs. The XML schema notation uses nesting to embed elements in each other. The keyword *ref* following an element tag shows that the specific element is defined at another place (in this case, to locations within the same schema). *xsd:sequence* indicates that all embedded elements must occur (in the specified order) and *xsd:choice* indicates that one of the listed elements has to be chosen. However, these restrictions can be modified by *use=required* and by allowing elements to have a cardinality specified by *minOccurs* and *maxOccurs*. In the graphical representations (from XMLSpy), sequences are denoted by a line with several dots,and choice by what looks like a switch.

Schemas for APPEL are defined in a modular fashion at three levels as illustrated in figure 2.1:

Core: The core level defines the framework that applies to all domains. This includes universal triggers, condition parameters and actions.

Policy: The policy level extends the core level in two directions: for regular policies and for resolution policies. This mainly adds further triggers, condition parameters and actions, while remaining domain-independent.

Domain: The domain level extends the policy level for specific application domains. This adds further triggers, condition parameters and actions for particular domains. Examples of application domains are call control, sensor networks and home care.

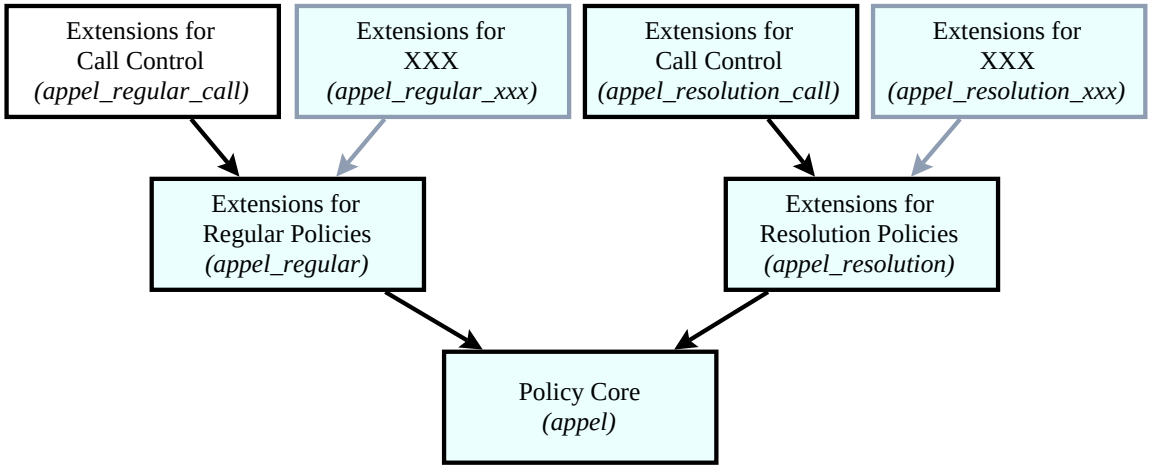


Figure 2.1: Schema Levels

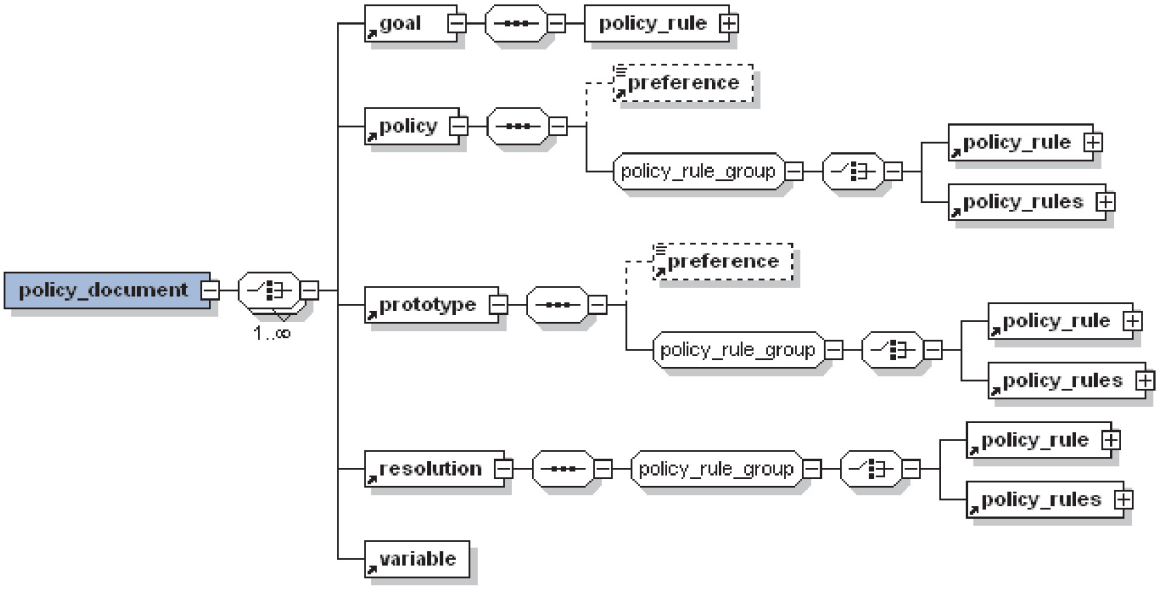


Figure 2.2: Policy Document

The schemas shown in figure 2.1 are available from *www.cs.stir.ac.uk/schemas*.

This chapter describes the structure of the core and policy schemas. Chapters 3, 4 and 5 respectively describe the application of the policy language for call control/Internet telephony, sensor network/wind farm management, and home care/telecare management.

2.2 Policy Document

A policy document (see figure 2.2) is the highest level at which one or more goals, regular policies, prototype policies, resolution policies and policy variables may be defined.

```
<xsd:element name="policy_document">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="goal"/>
      <xsd:element ref="policy"/>
      <xsd:element ref="prototype"/>
      <xsd:element ref="resolution"/>
      <xsd:element ref="variable"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

2.3 Policy

A policy is the main object of interest in the policy description language. A policy has a number of attributes: *owner* gives the address of the person or entity that the policy belongs to, while *applies_to* gives the address of the person, entity or group that the policy applies to. The term 'entity' is used here to refer to any system component such as a device or an application.

Normally the *owner* and *applies_to* attributes are the same. However in an enterprise, this might not be so: a policy might have been created (and be owned) by a system administrator, but apply to one or more people in the enterprise.

In general, *applies_to* refers to a domain. It can be one person or entity, one domain, or a list of users, entities and domains. It can also refer to a domain symbolically by citing its variable name (see section 2.9). The following examples illustrate the idea:

ken@cs.stir.ac.uk: means 'ken' as a particular user

@cs.stir.ac.uk: means anyone in the Stirling Computing Science department

@stir.ac.uk: means anyone in Stirling University

ken@cs.stir.ac.uk,lynne@comp.lancs.ac.uk,@accent.org.uk: means users 'ken' or 'lynne' or anyone in the accent.org.uk domain

@: means everyone (though the current version of the policy server stops at the top-level domain: @domain)

Each policy has an *id* to identify it uniquely for each owner. An identifier must not contain the characters '?', '/', '[' or ']'. A policy identifier may contain embedded spaces, and may also contain non-Latin characters (i.e. from the Unicode character set). The *changed* attribute indicates when a policy was last altered. (A date/time in XML format has the form of a date YYYY-MM-DD, the letter 'T' (for time), and then a time HH:MM:SS.) Selection of a policy is subject to the following rules:

- Each policy can be *enabled* or not, determining whether the system should consider the policy.
- Policies can be optionally associated with a *profile*. This is simply an identifier that groups policies, e.g. those for 'at the office' or those for 'at home'. A policy with an empty profile is always applicable, while one with a non-empty profile must match the user's current profile.
- The validity of a policy can be specified by a starting date and time (*valid_from*) and a finishing date and time (*valid_to*). Either or both of the start and finish values may be given; the values are inclusive.

The *effect* attribute is optionally used to define the abstract effects of a policy. This is mainly for use with prototype policies, and is described in section 2.15. However because regular policies can be derived from prototype policies, the attribute is also permitted for regular policies.

The *supports_goal* attribute is optionally used to state the goals to which a policy contributes. Again, this is a result of goal refinement and is used to distinguish ordinary policies from those created by instantiating prototype policies. This attribute is a comma-separated list of goal identifiers (white space being irrelevant).

Regular policies have the following structure:

```
<xsd:element name="policy">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="preference" minOccurs="0"/>
      <xsd:group ref="policy_rule_group"/>
    </xsd:sequence>
    <xsd:attribute name="applies_to" type="xsd:string" use="required"/>
    <xsd:attribute name="changed" type="xsd:dateTime" use="required"/>
    <xsd:attribute name="effect" type="xsd:string"/>
    <xsd:attribute name="enabled" type="xsd:boolean" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="owner" type="xsd:string" use="required"/>
    <xsd:attribute name="profile" type="xsd:string"/>
    <xsd:attribute name="supports_goal" type="xsd:string"/>
    <xsd:attribute name="valid_from" type="xsd:dateTime"/>
    <xsd:attribute name="valid_to" type="xsd:dateTime"/>
  </xsd:complexType>
</xsd:element>
```

Resolution policies differ from regular policies in having no *effect* attribute (used for prototype policies), no *preference* attribute (no recursive resolution), no *profile* attribute (not relevant), and no *supports_goal* attribute (used only for derived policies):

```
<xsd:element name="resolution">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="policy_rule_group"/>
    </xsd:sequence>
    <xsd:attribute name="applies_to" type="xsd:string" use="required"/>
```

```
    <xsd:attribute name="changed" type="xsd:dateTime" use="required"/>
    <xsd:attribute name="enabled" type="xsd:boolean" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="owner" type="xsd:string" use="required"/>
    <xsd:attribute name="valid_from" type="xsd:dateTime"/>
    <xsd:attribute name="valid_to" type="xsd:dateTime"/>
  </xsd:complexType>
</xsd:element>
```

Prototype policies differ from regular policies in having a mandatory *effect* attribute (used to derive policies), and in having no *supports_goal* attribute (used only for derived policies):

```
<xsd:element name="prototype">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="preference" minOccurs="0"/>
      <xsd:group ref="policy_rule_group"/>
    </xsd:sequence>
    <xsd:attribute name="applies_to" type="xsd:string" use="required"/>
    <xsd:attribute name="changed" type="xsd:dateTime" use="required"/>
    <xsd:attribute name="effect" type="xsd:string" use="required"/>
    <xsd:attribute name="enabled" type="xsd:boolean" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="owner" type="xsd:string" use="required"/>
    <xsd:attribute name="profile" type="xsd:string"/>
    <xsd:attribute name="valid_from" type="xsd:dateTime"/>
    <xsd:attribute name="valid_to" type="xsd:dateTime"/>
  </xsd:complexType>
</xsd:element>
```

2.4 Policy Modality

A policy may contain a modality in the form of a preference. This modality is principally for use by the conflict resolution engine. The optional preference of a policy states how strongly the policy definer feels about it. Omitting the preference means that the policy definer is neutral about this. From strongly positive to strongly negative, the ordering is *must*, *should*, *prefer*, empty, *prefer_not*, *should_not*, *must_not*.

```
<xsd:element name="preference">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="must"/>
      <xsd:enumeration value="must_not"/>
      <xsd:enumeration value="prefer"/>
      <xsd:enumeration value="prefer_not"/>
      <xsd:enumeration value="should"/>
      <xsd:enumeration value="should_not"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

2.5 Policy Rules

A policy rule group (see figure 2.2) is either a single policy rule or a composite of subsidiary policy rules.

```
<xsd:group name="policy_rule_group">
  <xsd:choice>
    <xsd:element ref="policy_rule"/>
    <xsd:element ref="policy_rules"/>
  </xsd:choice>
</xsd:group>
```

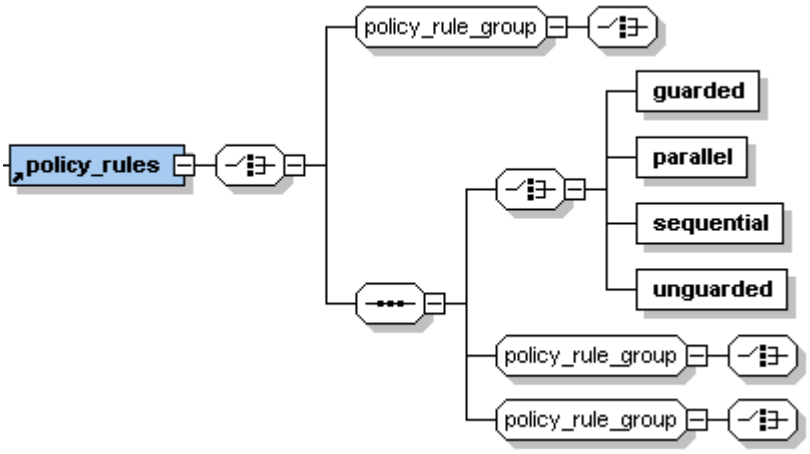


Figure 2.3: Policy Rules

A composite list of policy rules (see figure 2.3) is normally used when an operator governs these rules. However groups with just one rule may be nested within each other if desired.

Pairs of policy rules may be combined by guarded choice, unguarded choice, parallel application or sequence. More complex constructions can be obtained by nesting policy rules.

```
<xsd:element name="policy_rules">
  <xsd:complexType>
    <xsd:choice>
      <xsd:group ref="policy_rule_group"/>
      <xsd:sequence>
        <xsd:choice>
        <xsd:element name="guarded"/>
        <xsd:element name="parallel"/>
        <xsd:element name="sequential"/>
        <xsd:element name="unguarded"/>
        </xsd:choice>
        <xsd:group ref="policy_rule_group"/>
        <xsd:group ref="policy_rule_group"/>
      </xsd:sequence>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Policy rules can optionally be combined in pairs with a number of operators:

guarded: When two policy rules are joined by the guarded choice operator, the execution engine will first evaluate the nested condition. If the guard evaluates to 'true', the first of the two rules will be applied, otherwise the second. Clearly once the guard has been evaluated it is necessary to decide whether the individual rule is applicable and whether there is no conflict that prohibits enforcement. Once a guarded choice has been made, it is not undone even if the resulting rule is not followed.

```
<xsd:element name="guarded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="condition_group"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

unguarded: Unguarded choice provides more flexibility, as both parts will be tested for applicability. If only one of the two policy rules is applicable, this will be chosen. If both are applicable, the system can choose one at random; *the current policy server implementation will select the first rule*. The conflict resolution mechanism might still inhibit one or both branches, reducing the system's choice.

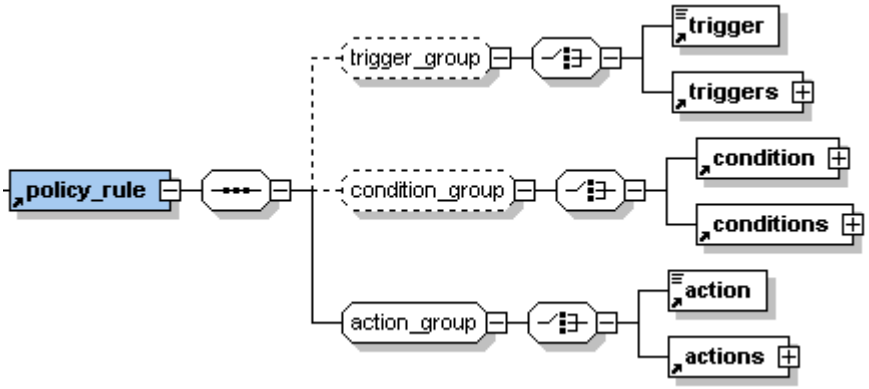


Figure 2.4: Policy Rule

sequential: With sequentially composed policy rules, the rules are enforced in the specified order. That is we traverse the structure, determining whether the first rule is applicable. If so we apply the first rule, otherwise we check the second rule. Note that the second rule will only be checked if the first rule is not applicable.

parallel: Parallel composition of two rules allows for a user to express an indifference with respect to the order of two rules. Both rules are enforced, but the order in which this is done is not important. An example would be a rule that leads to the attempt to add a video channel and the logging of the fact that this is attempted: we would allow the system to resolve the order of the two actions.

A policy rule (see figure 2.4) defines a trigger and a condition that need to hold in order to lead to an action being applied. Triggers are external events such as a call arriving or someone becoming available. Conditions depend on information derived from the triggering event, such as the caller or the subject of an event. A policy rule may omit a trigger, i.e. act as a goal. A policy may also omit a condition, i.e. not depend on information established by a trigger. However, an action always needs to be specified.

```
<xsd:element name="policy_rule">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:group ref="trigger_group" minOccurs="0"/>
      <xsd:group ref="condition_group" minOccurs="0"/>
      <xsd:group ref="action_group"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

2.6 Triggers

A trigger group (see figure 2.4) is either a single trigger or a composite of subsidiary triggers.

```
<xsd:group name="trigger_group">
  <xsd:choice>
    <xsd:element ref="trigger"/>
    <xsd:element ref="triggers"/>
  </xsd:choice>
</xsd:group>
```

A composite list of triggers (see figure 2.5) is normally used when an operator governs these triggers. However groups with just one trigger may be nested within each other if desired.

One or more triggers might enable a policy. The *and* and *or* combinations require both or either of the triggers to be active. If the combination of actual trigger events permits the policy rule, then its condition is considered. Goals (which have no triggers) are always implicitly activated. A conflict resolution engine might decide that a policy, despite having been triggered and satisfying the conditions, might not be applicable as it causes conflicts with more desirable policies.

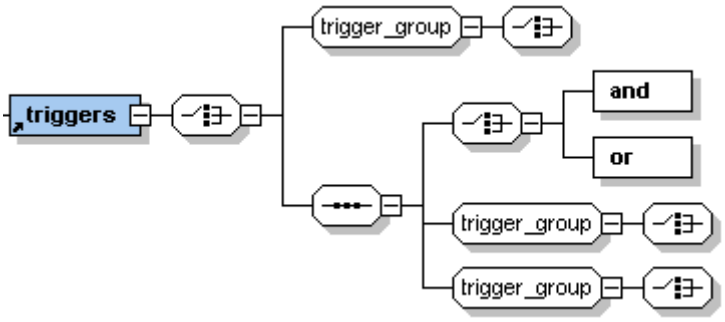


Figure 2.5: Triggers

```
<xsd:element name="triggers">
  <xsd:complexType>
    <xsd:choice>
      <xsd:group ref="trigger_group"/>
      <xsd:sequence>
        <xsd:choice>
          <xsd:element name="and"/>
          <xsd:element name="or"/>
        </xsd:choice>
        <xsd:group ref="trigger_group"/>
        <xsd:group ref="trigger_group"/>
      </xsd:sequence>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

A trigger has a name and optional arguments. The policy language does not statically enforce a match between a trigger and its arguments. Conventionally, a trigger is named *trigger(arg1)*, *trigger(arg1,arg2)* and the like. To be correct, the corresponding arguments must be given.

```
<xsd:element name="trigger">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="trigger_name">
        <xsd:attribute name="arg1" type="trigger_argument"/>
        <xsd:attribute name="arg2" type="trigger_argument"/>
        <xsd:attribute name="arg3" type="trigger_argument"/>
        <xsd:attribute name="arg4" type="trigger_argument"/>
        <xsd:attribute name="arg5" type="trigger_argument"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

At core level, trigger arguments are unrestricted strings:

```
<xsd:simpleType name="trigger_argument">
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
```

Trigger names are defined for the three levels in figure 2.1. At core level, these are unrestricted tokens; the policy and domain levels redefine these trigger names as required. Triggers that are generated by the policy system itself are considered to be internal.

```
<xsd:simpleType name="trigger_name">
  <xsd:union memberTypes="trigger_core trigger_domain trigger_policy"/>
</xsd:simpleType>

<xsd:simpleType name="trigger_core">
```

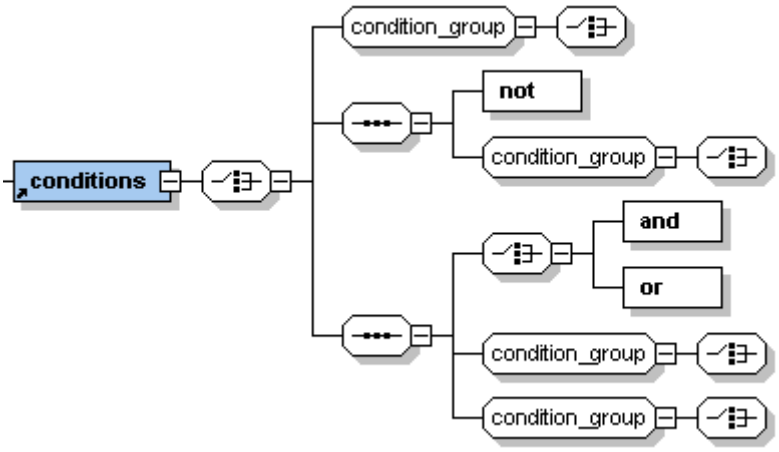


Figure 2.6: Conditions

```
  <xsd:restriction base="xsd:token"/>
</xsd:simpleType>

<xsd:simpleType name="trigger_policy">
  <xsd:restriction base="xsd:token"/>
</xsd:simpleType>

<xsd:simpleType name="trigger_domain">
  <xsd:restriction base="xsd:token"/>
</xsd:simpleType>
```

2.7 Conditions

A condition group (see figure 2.4) is either a single condition or a composite of subsidiary conditions.

```
<xsd:group name="condition_group">
  <xsd:choice>
    <xsd:element ref="condition"/>
    <xsd:element ref="conditions"/>
  </xsd:choice>
</xsd:group>
```

A composite list of conditions (see figure 2.6) is normally used when an operator governs these conditions. However groups with just one condition may be nested within each other if desired.

Conditions may be combined with the usual boolean *and*, *or* and *not* operators.

```
<xsd:element name="conditions">
  <xsd:complexType>
    <xsd:choice>
      <xsd:group ref="condition_group"/>
      <xsd:sequence>
        <xsd:element name="not"/>
        <xsd:group ref="condition_group"/>
      </xsd:sequence>
      <xsd:sequence>
        <xsd:choice>
          <xsd:element name="and"/>
          <xsd:element name="or"/>
        </xsd:choice>
        <xsd:group ref="condition_group"/>
        <xsd:group ref="condition_group"/>
      </xsd:sequence>
```

```
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
```

A condition has a first operand, an operator (that performs the check), and a second operand. An operand is either a parameter (a named value established by a trigger) or a value (a literal or an expression). It is possible to add further parameters by adding new environment classes to the policy server [9].

All four combinations of parameter and value are permitted syntactically, but special considerations apply:

parameter-parameter: This combination is meaningful only for resolution policies. A parameter must be one of *preference0* to *preference9* or *variable0* to *variable9*.

parameter-value, value-parameter: The name of the parameter determines the nature of the condition that is checked. For example, a *day* parameter is checked as day number against an integer value.

value-value: In this case, there is no parameter to determine the nature of the condition. If a value is enclosed in single quotes it is treated as a string, otherwise as an expression (see section 2.10). The result of evaluating an expression is treated as a number or as a string depending on whether it has the format of a floating-point number.

The names *preference0..9* and *variable0..9* are treated specially in that the ':' prefix that would normally indicate a variable is implied can therefore omitted. In the context of a value, it may be preferable to use the ':' prefix to make it clear that a variable is being used.

```
<xsd:element name="condition">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="operand"/>
      <xsd:element ref="operator"/>
      <xsd:element ref="operand"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="operand">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="parameter"/>
      <xsd:element ref="value"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Parameter names are defined for the three levels in figure 2.1. At core level, these are unrestricted tokens though certain history functions may be used (see section 2.12). The policy and domain levels redefine these parameter names as required.

```
<xsd:element name="parameter" type="parameter_name"/>

<xsd:simpleType name="parameter_name">
  <xsd:union memberTypes="parameter_core parameter_domain parameter_policy"/>
</xsd:simpleType>

<xsd:simpleType name="parameter_core">
  <xsd:restriction base="xsd:token"/>
</xsd:simpleType>

<xsd:simpleType name="parameter_policy">
  <xsd:restriction base="xsd:token"/>
</xsd:simpleType>

<xsd:simpleType name="parameter_domain">
  <xsd:restriction base="xsd:token"/>
</xsd:simpleType>
```

The operators that compare parameter values are fixed in the language. The meaning of the operators differs depending on the arguments they are applied to. The policy server has a flexible approach that allows parameter classes to be defined as part of the *environment* package (see [9]). For example, the standard operators have different meanings for address (e.g. *caller*) and epoch (e.g. *date*) values.

Operators are fixed in the language, though their interpretation varies according to context:

- The *eq*, *ge*, *gt*, *le*, *lt*, *ne* operators are equivalent to $=$, $\geq$, $>$, $\leq$, $<$, $\neq$. (In)equality has the obvious meaning. A comparison between a pair of values in numeric formats is numeric. If one or both values are not numeric, a string comparison (i.e. character ordering) is used. Thus *13 gt 1* is true (numeric comparison) but *13 gt '1'* is false (string comparison).

- The *in* and *out* operators are provided to check for inclusion or exclusion. For string values and certain parameters, *in*/*out* has the meaning of 'a substring of'/'not a substring of'. For other parameters, *in*/*out* has the meaning of 'among the values in a list'/'not among the values in a list'. These operators are not supported when the comparison is between a value and a parameter.

```
<xsd:element name="operator">
  <xsd:simpleType>
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="eq"/>
      <xsd:enumeration value="ge"/>
      <xsd:enumeration value="gt"/>
      <xsd:enumeration value="in"/>
      <xsd:enumeration value="le"/>
      <xsd:enumeration value="lt"/>
      <xsd:enumeration value="ne"/>
      <xsd:enumeration value="out"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

A value is just a (possibly empty) string:

```
<xsd:element name="value" type="xsd:token"/>
```

A list of values may be given, separated by commas without following spaces (e.g. '09:00:00,12:00:00' for a pair of times, or 'bob@acme.com,colin@acme.com,donald@acme.com' for a list of callers). In the case of date, day or time, a value may also be a range separated by '..' (e.g. 09:00:00..12:00:00 or 2005-02-03..2005-02-10). Value names can contain policy variables (see section 2.9).

2.8 Actions

An action group (see figure 2.4) is either a single action or a composite of subsidiary actions.

```
<xsd:group name="action_group">
  <xsd:choice>
    <xsd:element ref="action"/>
    <xsd:element ref="actions"/>
  </xsd:choice>
</xsd:group>
```

A composite list of actions (see figure 2.7) is normally used when an operator governs these actions. However groups with just one action may be nested within each other if desired.

```
<xsd:element name="actions">
  <xsd:complexType>
    <xsd:choice>
      <xsd:group ref="action_group"/>
      <xsd:sequence>
        <xsd:choice>
          <xsd:element name="and"/>
          <xsd:element name="andthen"/>
```

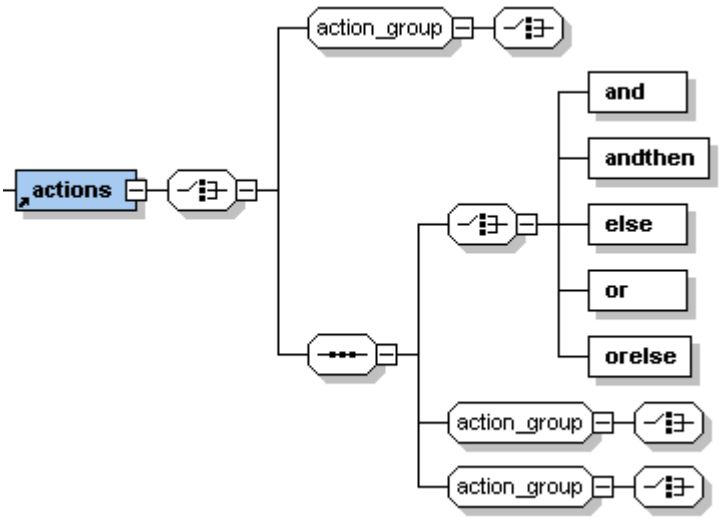


Figure 2.7: Actions

```
        <xsd:element name="else"/>
        <xsd:element name="or"/>
        <xsd:element name="orelse"/>
      </xsd:choice>
      <xsd:group ref="action_group"/>
      <xsd:group ref="action_group"/>
    </xsd:sequence>
  </xsd:choice>
 </xsd:complexType>
</xsd:element>
```

Operators can be used to define the order in which pairs of actions are applied.

and: This specifies that the policy should lead to the execution of both actions in either order. *The current policy server implementation will execute the actions in the order given*. However this is arbitrary, and indeed the actions could be executed in parallel. The conflict resolution engine can make use of the flexibility provided by this. However it must rearrange the order prior to execution by the policy server.

andthen: This is a stronger version of *and*, since the first action must precede the second in any execution. This means that the conflict resolution engine cannot rearrange the order. This might lead to a conflict resolvable only by not executing any of the specified actions.

or: This specifies that either one of the actions should be taken. *The current policy server implementation will always execute the first action*. Again, the conflict resolution engine could rearrange the order if the first action is not suitable. As for *and*, any reordering must occur before execution by the policy server.

orelse: This is the *or* operator with a prescribed order. It means that a user feels more strongly about the first action specified. However, *or else* implies a choice so the conflict resolution engine can decide to reverse the order.

else: If there is a condition and *else* is at the top level of actions, this has the effect of a conventional **if**...**else**. If the condition is true then the first action is chosen, otherwise the second action is chosen.

If there is no condition or *else* does not appear at the top level, *else* behaves like *or*. Note that this allows a single choice at the top level of a policy; multiple *else* operators cannot be used in a *switch*-like fashion.

An action has a name and optional arguments. The policy language does not statically enforce a match between an action and its arguments. Conventionally, an action is named *action(arg1)*, *action(arg1,arg2)* and the like. To be correct, the corresponding arguments must be given. Action arguments can refer to policy variables (see section 2.9).

```
<xsd:element name="action">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="action_name">
        <xsd:attribute name="arg1" type="xsd:string"/>
        <xsd:attribute name="arg2" type="xsd:string"/>
        <xsd:attribute name="arg3" type="xsd:string"/>
        <xsd:attribute name="arg4" type="xsd:string"/>
        <xsd:attribute name="arg5" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

Action names are defined for the three levels in figure 2.1. At core level, universal action names are given. Action names at policy and domain level are unrestricted tokens; these may be redefined as required. Actions that are executed by the policy system itself are considered to be internal.

```
<xsd:simpleType name="action_name">
  <xsd:union memberTypes="action_core action_domain action_policy"/>
</xsd:simpleType>

<xsd:simpleType name="action_core">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="log_event(arg1)"/>
    <xsd:enumeration value="restart_timer(arg1)"/>
    <xsd:enumeration value="send_message(arg1,arg2)"/>
    <xsd:enumeration value="set_variable(arg1,arg2)"/>
    <xsd:enumeration value="start_timer(arg1,arg2)"/>
    <xsd:enumeration value="stop_timer(arg1)"/>
    <xsd:enumeration value="unset_variable(arg1)"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="action_policy">
  <xsd:restriction base="xsd:token"/>
</xsd:simpleType>

<xsd:simpleType name="action_domain">
  <xsd:restriction base="xsd:token"/>
</xsd:simpleType>
```

Core actions have parameters as follows:

- an *identifier* parameter (see section 2.9).
- a *URL* parameter (one prefixed by 'file:', 'ftp:', 'http:', 'sip:', etc.).

Core actions are as follows, all of them internal:

log_event(message): This logs the date, time and message. A single line (ended with '\n') is appended to an event log for the user whose policy invokes this action. Call information may be interpolated into the message if available, e.g. 'starting call from :caller: to :callee'. This would result in a log entry such as:

2005-03-07 17:53:52 starting call from grace@cs.stir.ac.uk to ken@cs.stir.ac.uk

typically in a file such as ken@cs.stir.ac.uk.log.

restart_timer(identifier): This immediately restarts a timer with the specified identifier for its original period. If a timer with this identifier is not already active, the action is ignored. See section 2.11 for more details.

send_message(URL,message): This sends a message. The first argument is the address of the destination . The second argument is a message in a form that depends on the means of transmission. The form of the address dictates how the message is sent (e.g. 'mailto:bob@acme.com' or just 'bob@acme.com' would send an email message as a text string; 'tel:123456@gateway.com' might speak a message defined by the URL of a pre-recorded audio file).

set_variable(identifier,expression): This sets a variable with the specified identifier to the specified expression. An identifier-value binding is created if the variable did not exist previously, or the binding is updated if it did. See section 2.9 for more details.

start_timer(identifier,period): This immediately starts a timer with the specified identifier for the specified period (format HH:MM:SS). If a timer with this identifier is already active, the action is ignored. See section 2.11 for more details.

stop_timer(identifier): This immediately stops a timer with the specified identifier. If a timer with this identifier is not already active, the action is ignored. See section 2.11 for more details.

unset_variable(identifier): This removes a variable with the specified identifier. If the variable did not exist previously, the action is ignored. See section 2.9 for more details.

2.9 Variables

A policy variable is just a name for a value. By convention, variables may be named in upper case. Policy variables can be defined by a policy document, can be defined implicitly when an external trigger occurs (see section 2.12), and can be defined by a policy action (see section 2.8).

The top-level definition of a variable deliberately resembles that of a policy. Each variable has an *id* to identify it uniquely for each owner. An identifier must not begin with a digit, must not contain white space, and must not contain the characters '?', '/', '[', ']', ':' or ','. The identifier '*' is reserved for internal use (to mean all identifiers). An identifier may contain non-Latin characters (i.e. from the Unicode character set).

The value of a variable is what it stands for. As there is only ever one instance of a variable, assigning a new value overwrites the previous one. Variables may hold boolean, numeric (integer, floating point) or string values. Variables are dynamically typed as in scripting languages, i.e. they may hold different kinds of values at different times.

The *owner* of a variable is the address of the person or entity that defined the variable. Normally this will be the same as the *applies_to* attribute, but it could be different if one user (typically an administrator) defines variables that apply to others (typically ordinary users). As an example, an administrator could define the variable *holidays* that lists public holidays for everyone in the current year. The *changed* attribute is as for policies.

Within certain policy elements, variable names can appear individually, in ranges or in lists. The value of a variable is substituted when a policy is executed, not when it is defined. Variable names can be used in the following cases:

- a trigger *arg* attribute
- a condition *parameters*
- an action *arg* attribute
- an expression
- as the first argument of *set_variable* and *unset_variable*.

When used in an expression, the name of a variable is prefixed by ':' to obtain its value. If necessary (particularly when a variable appears inside a larger string), a variable name may be terminated by ':'. Since variables may appear in a range or list, a variable name is also terminated by '.' and ','.

Apart from policy variables, environment variables defined by the system (e.g. *callee*, *caller*, *date*, *day*, *time*, *trigger*, *user*) may also be substituted in arguments. An environment-defined variable is used in preference to a user-defined variable of the same name.

```
<xsd:element name="variable">
  <xsd:complexType>
    <xsd:attribute name="applies_to" type="xsd:string" use="required"/>
      <xsd:attribute name="changed" type="xsd:dateTime" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="owner" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string"/>
```

```
    </xsd:complexType>
  </xsd:element>
```

2.10 Expressions

Expression may be used in the following cases:

- as a *value*
- as the second argument of *set_variable*

Variable names inside an expression are replaced by their current values, e.g.:

```
<trigger arg1=":delay:">no_answer(arg1)</trigger>

<condition>
  <parameter>caller</parameter>
  <operator>in</operator>
  <value>:family,:friends,:relatives</value>
</condition>

<action arg1=":boss" arg2="I was at :place: when :caller: called at :time, :date">
  send_message(arg1,arg2)
</action>
```

A policy may set a variable to an expression by calling the action *set_variable(identifier,expression)*. On first creation, such a variable is given the *owner* and *applies_to* values of the policy that executes this action. The expression may be a literal value, the name of a variable, or a general expression:

- boolean: literals are *true* and *false*; operators are *and*, *or*, *not*, *eq*, *ne*, *lt*, *le*, *gt*, *ge*.
- numeric: literals are (signed) numbers (floating point and integer values); operators are '+' (unary, binary), '−' (unary, binary), '*', '/', '%' (modulus). Integer division yields an integer (e.g. '3/5' is 0). Division of other numbers yields a floating point number (e.g. '3.0/5.0' is 0.6).
- string: literals are characters in single quotes (using '' for a single quote inside a string) String functions are:

  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

  substr(string,start,count) with 0 as the start position

  In addition, a comma-separated string may be treated as an indexed list of sub-strings. If a sub-string needs to contain a comma, it must be given in square brackets; sub-strings may not be nested. Suppose *dates* holds the string '2007,Sep,[13,21,30]'. Then *:dates[0]* is the string '2007', *:dates[1]* is the string 'Sep', and *:dates[2]* is the string '[13,21,30]'. If the string index is out of range (negative, or beyond the end of the string), the indexed value is an empty string. Thus if *day* holds 'Monday' then *:day[0]* is the string 'Monday' and *:day[1]* is the empty string. An indexed string element may be used only as a value, and may not be assigned to (e.g. *set_variable(:dates[1],Oct)* is not allowed).

- variable value: the name of the variable prefixed with ':'.

Expressions are parsed left-to-right, with operators having identical precedence. Expressions may therefore need parentheses to disambiguate them. For example, '3+4*5' is parsed as '(3+4)*5' and must be written '3+(4*5)' for the conventional interpretation.

If a variable is used without having been given a value, it is interpreted according to context: false (boolean), 0 (integer), or empty (string). The associated operator or function determines how the value of a variable is interpreted:

- boolean: *false*, 0 and ′ ′ are interpreted as *false*; all other values are interpreted as *true*.
- integer: the leftmost valid integer value is used, e.g. *X* is interpreted as 0, *42X* is interpreted as 42.
- floating point: the leftmost valid double value is used, e.g. *0.1X* is interpreted as 0.1, *42.4X* is interpreted as 42.4.
- string: all values are permissible as strings. A quoted value such as ′123′ will be interpreted as a string and not as a number.

Here are examples of variable assignment and use:

```
set_variable(text,'test string')
set_variable(substituted,'substitution of a :text variable value')
set_variable(hasVowel,indexOf(:substituted,'aeiou') ne -1)
set_variable(validLength,(5 le length(:text)) and (length(:text) lt 15))
```

A policy may remove a variable binding with the action *unset_variable(identifier)*, causing any future use of the variable to act as if undefined.

2.11 Timers

Policies may make use of interval (count-down) timers. Each timer has an identifier that must be different from the identifiers used by variables. There is only ever one instance of a timer. 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. An active timer may be restarted or stopped.

The policy server keeps track of all active timers. Each timer is associated with its identifier, its starting time and period, and with the *owner* and *profile* of the defining policy. If the policy server shuts down (accidentally or deliberately), on a restart it checks the status of all timers. Timers that would have expired in the meantime are discarded. All other timers are treated as if time had passed normally.

In addition to explicit timers, policies with time conditions imply internal timer triggers. Policies like this are said to be time-based. They must have a non-negative preference (*must*, *should*, *prefer*, empty), no triggers, and only epoch conditions (*date*, *day*, *time*). When such a policy is defined, the policy server infers the time intervals during which the policy *may* be executed. An internal trigger at the start of each time interval causes such policies to be considered for execution. The associated *owner* and *profile* of such a trigger are determined by the defining policy. Whether a policy actually executes depends on its period of validity and on whether its condition holds. Timer resolution is presumed to be one second. However, due to scheduling constraints in the policy server, a timer trigger may not be processed to this level of accuracy.

When a policy is added to the policy store or is enabled, it is checked whether an internal timer should be created for it. Conversely, such a timer is removed when a policy is disabled or deleted.

In deciding whether to trigger a time-based policy, the policy server assumes that all time conditions can be met. As an example, suppose a policy condition requires the time to be 12:15:00 or 12:30:00. The policy server will infer that these are important times when the policy should be triggered; on both occasions the policy will execute. However, suppose the policy condition requires the time to be 12:15:00 *and* 12:30:00. Although the policy server will again trigger the policy at these times, the condition cannot in fact hold. To avoid the policy server having to make a full determination of timing considerations, it just determines a set of simple trigger times that should be used.

A time-based policy may specify a number of discrete epoch values with the *eq* and *in* operators, and epoch ranges with the *in* operator. Such a policy may not use other operators on epochs (*ne*, *lt*, *le*, *gt*, *ge*, *out*). In addition, a time condition may use only the *and* and *or* operators (i.e. *not* is disallowed). The reason for these restrictions is to ensure that all epoch ranges are well defined. A *date* or *day* parameter implies a triggering hour of 00:00:00, and eligibility for execution at any hour of that day.

As well as being triggered at the start of each time interval, a time-based policy must considered for execution whenever any external trigger occurs for the same owner. The time condition (and other factors) will determine whether the policy is actually executed. A time condition may lead to a policy being triggered multiple times (e.g. *day eq 3 or hour eq 12:00:00*). Care must be taken to ensure to ensure that repeated execution is appropriate.

Here are examples of time conditions in informal notation (rather than XML). Triggering refers to when an internal trigger occurs, eligibility refers to when an external trigger occurs. Although the policy wizard allows

short forms for times and dates (e.g. 23, 23:15, 2007, 2007-02), the policy wizard converts these to full HH:MM:SS and YYYY-MM-DD format (e.g. 23:00:00, 23:15:00, 2007-01-01, 2007-02-01).

- *time eq 12:15:30* – triggered at 12:15:30 each day; eligible at 12:15:30 on any day.
- *time in 09:00:00,11:00:00* – triggered at 09:00:00 and 11:00:00 each day; eligible at 09:00:00 and 11:00:00 on any day.
- *time in 23:00:00..08:00:00,20:00:00* – triggered at 23:00:00 and 20:00:00 each day; eligible from 23:00:00 to 08:00:00 and at 20:00:00 on any day.
- *time eq 07:00:00 or time in 16:30:45,21:15:45* – triggered at 07:00:00, 16:30:45 and 21:15:45 each day; eligible at 07:00:00, 16:30:45 and 21:15:45 on any day.
- *time eq 09:15:00 or time eq 11:30:00* – triggered at 09:15:00 and 11:30:00 each day; eligible at 09:15:00 and 11:30:00 on any day.
- *time eq 09:15:00 and time eq 11:30:00* – triggered at 09:15:00 and 11:30:00 each day, though the condition can never hold; never eligible, as the condition is always false.
- *(date in 2006-12-01..2007-03-31) or (day eq 1)* – triggered at 00:00:00 on 1st December 2006, and at 00:00:00 every Monday; eligible at any hour from 1st December 2006 to 31st March 2007, and at any hour on a Monday.
- *(date in 2006-12-01..2007-03-31) and (day eq 1)* – triggered at 00:00:00 on 1st December 2006 (no effect as this was not a Monday), and at 00:00:00 every Monday (effective on 4th December 2006, ..., 26th March 2007); eligible at any hour on any Monday from 1st December 2006 to 31st March 2007 (i.e. 4th December 2006, ..., 26th March 2007).
- *(date eq 2006-12-15) or ((day eq 1) and (time eq 16:17:18))* – triggered at 00:00:00 on 15th December 2006, at 00:00:00 every Monday, and at 16:17:18 every day (the latter two combining to be effective at 16:17:18 on Mondays); eligible at any hour on 15th December 2006, and at 16:17:18 on Mondays.

2.12 History

2.12.1 Trigger History

External triggers are automatically recorded by the policy server as they are received (and prior to trigger processing). This includes the time the trigger occurred, the trigger name, and the trigger parameters. Such triggers are held individually in the policy store as *trigger* variables. The policy server periodically eliminates trigger variables that are older than a period specified as a server parameter.

All such variables are named *trigger*. Their *owner* and *applies_to* values are 'system'. Their *changed* value is an XML date-time for when the trigger was received by the policy system. The value of such a trigger variable is a copy of the trigger and its parameters.

Suppose a *device_in* trigger carries as parameters the message type *temperature_value*, the entity name 'gearbox', the entity instance 'turbine23', the reporting period '15', and a list of temperatures since the last reading: minimum 12.3°C, average 14.5°C, and maximum 19.2°C. The value of the recorded history variable will be:

device_in(temperature_value,gearbox,turbine23,15,[12.3,14.5,19.2])

The collection of stored *trigger* values constitutes a history that can be checked in a policy condition. This may use the following functions in a condition parameter:

trigger_count(trigger,period): This counts the number of matching triggers in the state history for the given period. The trigger has the same form as a policy trigger, except that an empty trigger parameter may be used to mean that any value is acceptable. For convenience, trailing empty parameters are omitted entirely. The period is either a non-negative integer *n* (during the last *n* minutes) or a time in HH:MM:SS format (since this time). An absolute time reference spans a maximum of 24 hours, e.g. if the time is currently 12:00:00 then 11:00:00 means 'since 11AM today' and 13:00:00 means 'since 1PM yesterday'.

Here are some examples of policy conditions that check the trigger history. These assume the same temperature trigger example as above.

- *trigger_count(device_in(),15)* – the number of device inputs during the past 15 minutes.
- *trigger_count(device_in(temperature_value),09:00:00)* – the number of temperature readings since 9AM.
- *trigger_count(device_in(temperature_value,gearbox,turbine23),30)* – the number of gearbox temperature readings for identifier 'turbine23' during the past 30 minutes.
- *trigger_count(device_in(temperature_value,,,20),22:30:00)* – the number of temperature readings for a period of 20 minutes since 10.30PM.
- *trigger_count(device_in(temperature_value,,turbine23,,[11.1]),15)* – the number of temperature readings for identifier 'turbine23' and minimum value 11.1°C during the past 15 minutes.
- *trigger_count(device_in(,,,,[,,28]),00:00:00)* – the number of inputs from any device with a third parameter value of 28 since midnight (this would mean a maximum of 28°C for a temperature reading).

2.12.2 Action History

Policy actions are automatically recorded by the policy server as they are issued (subsequent to conflict resolution). This includes the time the action occurred, the action name, and the action parameters. Such actions are held individually in the policy store as *action* variables. The policy server periodically eliminates action variables that are older than a period specified as a server parameter.

All such variables are named *action*. Their *owner* and *applies_to* values are 'system'. Their *changed* value is an XML date-time for when the action was issued by the policy system. The value of such an action variable is a copy of the action and its parameters.

Suppose a *device_out* action carries as parameters the message type *set_rule*, the entity name 'anemometer', the entity instance '3', the action period '20', and the action parameters 'wind_speed,15,alert_operator'. The value of the recorded history variable will be:

```
device_out(set_rule,anemometer,3,20,wind_speed,15,alert_operator)
```

The collection of stored *action* values constitutes a history that can be checked in a policy condition. This may use the following functions in a condition parameter:

action_count(action,period): This counts the number of matching actions in the state history for the given period. The action has the same form as a policy action, except that an empty action parameter may be used to mean that any value is acceptable. This function is very similar to *trigger_count* and is not illustrated here.

2.13 Extensions for Regular Policies

2.13.1 Triggers

Regular policies have triggers as follows:

```
<xsd:simpleType name="trigger_policy">
  <xsd:restriction base="trigger_policy">
    <xsd:enumeration value="timer_expiry(arg1)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy triggers are as follows, all of them internal:

timer_expiry(identifier): This occurs when a timer with the specified identifier has counted down to zero (see section 2.9. See section 2.11 for more details.

Category	Parameter	Operator
epoch	*date*, *day*, *time*	*eq*, *in* (time-based policy)
		eq, *ne*, *lt*, *le*, *gt*, *ge*, *in*, *out* (other policy)

Figure 2.8: Epoch Parameter Operators

2.13.2 Conditions

Regular policies have condition parameters as follows:

```
<xsd:simpleType name="parameter_policy">
  <xsd:restriction base="parameter_policy">
    <xsd:enumeration value="date"/>
    <xsd:enumeration value="day"/>
    <xsd:enumeration value="time"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy conditions have parameters as follows:

date: This means a date (format YYYY-MM-DD).

day: This means a day of the week (numbered from 1 as Monday to 7 as Sunday).

time: This means the current moment (format HH:MM:SS).

The operators permitted for each parameter are listed in figure 2.8. Some special rules for values are as follows:

- All parameters can be compared with a single value.
- An epoch value may also be a range of the form *start..finish*.
- For the epoch category, *in* and *out* are used with a comma-separated list of values to mean 'among' and 'not among'. That is, the parameter is checked for presence or absence in a list of values.

2.14 Extensions for Resolution Policies

2.14.1 Triggers

Resolution policies have triggers as follows, all of them internal. Since resolution policies are triggered by regular actions, the resolution triggers are identical to the core actions in section 2.8.

```
<xsd:simpleType name="trigger_policy">
  <xsd:restriction base="trigger_policy">
    <xsd:enumeration value="log_event(arg1)"/>
    <xsd:enumeration value="restart_timer(arg1)"/>
    <xsd:enumeration value="send_message(arg1,arg2)"/>
    <xsd:enumeration value="set_variable(arg1,arg2)"/>
    <xsd:enumeration value="start_timer(arg1,arg2)"/>
    <xsd:enumeration value="stop_timer(arg1)"/>
    <xsd:enumeration value="unset_variable(arg1)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Resolution policies have restricted trigger arguments. *variable0* to *variable9* are explicitly bound to the actual arguments of resolution triggers. *preference0* to *preference9* are implicitly bound to the preferences of the corresponding policies that triggered a resolution. Preferences are numbered from 0 in the same order as triggers appear in a resolution policies. For example, *preference0* will be assigned the preference associated with the first resolution trigger (i.e. the preference associated with this action), and *preference1* to that of the second resolution trigger.

```
<xsd:simpleType name="trigger_argument">
  <xsd:restriction base="trigger_argument">
    <xsd:pattern value="variable[0-9]"/>
  </xsd:restriction>
</xsd:simpleType>
```

2.14.2 Conditions

Resolution policies have condition parameters as follows. These are the values that are bound by resolution triggers.

```
<xsd:simpleType name="parameter_policy">
  <xsd:restriction base="parameter_policy">
    <xsd:pattern value="preference[0-9]"/>
    <xsd:pattern value="variable[0-9]"/>
  </xsd:restriction>
</xsd:simpleType>
```

Resolution operators are the standard ones, but interpreted for *preference* and *variable* values:

- If the first operator argument is *preference0* to *preference9*, the parameter and value are converted from preference names to numerical values. Preference names are numerically ranked as +3 for *must*, +2 for *should*, +1 for *prefer*, 0 for empty, -1 for *prefer_not*, -2 for *should_not*, -3 for *must_not*. The operators *eq*, *ne*, *lt*, *le*, *gt* and *ge* then perform a numerical comparison. The *in* and *out* operators should be read as 'in keeping with' and 'out of keeping with', i.e. similar and opposite respectively.

  Positive and negative preference values are considered to be opposites. A zero value is similar to a positive or a negative value. Thus *must* is similar to *should* or empty (operator *in*), and is opposite to *should_not* or *must_not* (operator *out*). Similarly *must_not* is similar to *should_not* or empty (operator *in*), and is opposite to *should* or *must* (operator *out*).

- If the first operator argument is *variable0* to *variable9*, the *eq* and *ne* operators may be used. The *in* and *out* operators are used with a text string to mean 'includes' and 'excludes'. That is, the value is checked to be a substring or not of the parameter.

2.14.3 Actions

Resolution policies have actions as follows that are added to the core ones defined in section 2.8. The following are generic resolution actions that apply in any domain, all of them internal.

```
<xsd:simpleType name="action_policy">
  <xsd:restriction base="action_policy">
    <xsd:enumeration value="apply_default"/>
    <xsd:enumeration value="apply_inferior"/>
    <xsd:enumeration value="apply_negative"/>
    <xsd:enumeration value="apply_newer"/>
    <xsd:enumeration value="apply_older"/>
    <xsd:enumeration value="apply_one"/>
    <xsd:enumeration value="apply_positive"/>
    <xsd:enumeration value="apply_stronger"/>
    <xsd:enumeration value="apply_superior"/>
    <xsd:enumeration value="apply_weaker"/>
  </xsd:restriction>
</xsd:simpleType>
```

Although conflicting actions are handled pairwise, a list of conflicting actions may be under consideration so the effect is to make a choice from an arbitrarily long list. A fundamental principle is that resolution selects just one of the conflicting actions. It is up to the policy server to make sure this happens, even if it requires making an arbitrary choice when resolution does not yield a unique result.

A resolution action is either specific (applicable in some domain) or generic (making a choice among the conflicting actions). Generic resolution actions are as follows:

apply_default: Although this action can be called explicitly, it is mainly intended for internal use by the policy server when resolution does not select just one of the conflicting actions. The default resolution is first to call *apply_stronger*. If this does not yield a unique action, *apply_newer* is called. If this still does not yield a unique action, *apply_one* is called.

apply_inferior: This chooses the action associated with the inferior domain. For example, an action specified by kjt@cs.stir.ac.uk will be chosen in preference to an action for bob@stir.ac.uk. The notion of 'inferior' is that the higher-level domain is a suffix of the lower-level domain (dividing the domain into parts at '.'). If the domains are identical or incomparable (e.g. the second party is lb@comp.lancs.ac.uk), this resolution will not eliminate either action.

apply_negative: This chooses the action associated with the more negative preference. If the preferences are identical, this resolution will not eliminate either action.

apply_newer: This chooses the action associated with the more recently defined policy. If the timestamps are identical, this resolution will not eliminate either action.

apply_older: This is the opposite of *apply_newer*.

apply_one: This arbitrarily chooses one of the actions. (The current implementation of the policy server in fact chooses the first action in the list.)

apply_positive: This is the opposite of *apply_negative*.

apply_stronger: This chooses the action associated with the stronger preference (disregarding the sign of the preference). For example, *must_not* will be selected over *should*. If the preferences are exact opposites, the more positive preference is selected. For example, *should* will be selected over *should_not*. If the preferences are identical, this resolution will not eliminate either action.

apply_superior: This is the opposite of *apply_inferior*.

apply_weaker: This is the opposite of *apply_stronger*, except that the more negative preference is selected in the case of exact opposites.

2.15 Extensions for Prototype Policies

Prototype policies are used to realise goals (see section 2.16). Prototype policies have almost the same structure as regular policies, except that the *effect* attribute is mandatory, and the *supports_goal* attribute does not apply. Triggers, conditions and actions are otherwise common to regular and prototype policies.

The *effect* attribute is used to define the abstract effects of a prototype policy. An effect is a comma-separated list of effect elements (white space being irrelevant). Each effect element has the form: *variable operator expression* (e.g. *duration += 10*):

variable: The system managed by policies is presumed to have a number of state ('environment') variables. (These are different from policy variables that are substituted in the body of a policy.) A controlled variable is one that the policy system can affect (e.g. the length of a call). An uncontrolled variable is one that the policy system cannot affect (e.g. the time of day). A derived variable is one that is defined in terms of the others (usually controlled and uncontrolled variables).

operator: The basic effect operators are '=' (set variable), '+=' (augment variable) and '-=' (diminish variable). In the special case that simultaneously enabled policies are not allowed to affect the same variable, the '+∼' and '-∼' variants are used to mean an exclusive change.

expression: A basic expression is just a literal numeric value. However, an effect can be parameterised by one parameter – a variable prefixed with '$' (e.g. *duration += $length*). Such a parameter is given an optimal value during goal refinement, and can be used in the actions (only) of the policy. An effect parameter can be combined using a single binary operator ('+', '-', '*', '/') with a literal numeric value (e.g. *duration += 2*$length*).

2.16 Extensions for Goals

Goals are similar to regular policies, but with certain changes and restrictions. Because goals are persistent, they have no trigger. Although a goal can have conditions, the absence of a trigger means that only generic conditions can be used (i.e. those described in section 2.13.2). A goal has a single action that maximises or minimises a goal named by its identifier.

The attributes of a goal are similar to those of regular policies, but omitting *effect* (used only to derive policies), *profile* (not relevant), and *supports_goal* (used only for derived policies) .

```
<xsd:element name="goal">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="policy_rule">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:group ref="condition_group" minOccurs="0"/>
            <xsd:element name="action">
              <xsd:complexType>
                <xsd:simpleContent>
                  <xsd:extension base="action_goal">
                    <xsd:attribute name="arg1" type="xsd:string"/>
                  </xsd:extension>
                </xsd:simpleContent>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="applies_to" type="xsd:string" use="required"/>
    <xsd:attribute name="changed" type="xsd:dateTime" use="required"/>
    <xsd:attribute name="enabled" type="xsd:boolean" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="owner" type="xsd:string" use="required"/>
    <xsd:attribute name="valid_from" type="xsd:dateTime"/>
    <xsd:attribute name="valid_to" type="xsd:dateTime"/>
  </xsd:complexType>
</xsd:element>
```

A goal maximises or minimises a goal measure. This is a numerical quantity defined by a formula in the domain ontology (i.e. outside the policy language). The formula is an arbitrary calculation using the state variables discussed in section 2.15. However for practicality, goal measures are usually linear weighted sums. For example, a goal might aim to minimise the goal measure *interruption_time*. This might be defined in terms of the controlled variables *call_received* (number of calls received today) and *call_duration* (length of current call in seconds). The ontology definition of this measure might be:

```
0.02 * call_received * call_duration
```

The scaling factor of 0.02 here is chosen so that all goal measures have similar values for typical values of the state variables. Indeed, the choice of scaling factors can be automated on this basis.

Typically, multiple goals have to be satisfied; these often conflict with each other. Goal measures are therefore combined in an overall evaluation function. This can be an arbitrary formula, but is usually a linear weighted sum. Goal measures to be maximised have a positive sign, while those to be minimised have a negative sign. Goals may be weighted, though analysis of various domains has shown that a weight of 1.0 is usually satisfactory. The overall evaluation function is defined by an ontology (i.e. outside the policy language). An example might be:

```
-call_cost_weight * call_cost +
-call_interruption_weight * interruption_time          +
+call_multimedia_weight * multimedia_use +
+call_network_weight * network_use;
```

Chapter 3

Call Control

3.1 Regular Policies for Call Control

3.1.1 Introduction

This section is specific to regular policies for call control, as it discusses the triggers, conditions and actions that are used in this domain. The policy server employs a terminology mapping between information in the underlying communications system and information in the policy system [9]. If the policy language were used in a context different from call control, it would be these elements that were adapted.

For call control, regular policies are extended with domain-specific triggers, conditions and actions. The relationship among these is shown in figure 3.1. Core triggers (section 2.6), condition parameters (section 2.7) and actions (section 2.8) are also applicable. Certain elements (marked ‡) apply to only some communications layers. The combination rules are as follows:

- Triggers may be combined with *and* or *or*. Internal triggers (marked † in figure 3.1) may be combined with any other triggers. At most one external trigger (no † in figure 3.1) may appear underneath an *and*.
- Triggers establish environment variables that are used as condition parameters. If a combination of triggers is provided, *and* establishes the union of the environment variables, while *or* establishes their intersection.
- The environment variables established by one of the *connect* or *no_answer* triggers depend on the underlying communications layer.
- Triggers allow certain actions to occur. If a combination of triggers is provided, *and* allows the union of the actions, while *or* allows their intersection.

3.1.2 Common Parameters and Environment Variables

Regular policy conditions can use the following parameters: *date*, *day* and *time* (section 2.13.2). Regular policies for call control can also use a common set of parameters within triggers, conditions and actions. These parameters are outlined below together with examples of their use:

address: This identifies a user through an email-like address (e.g. ken@cs.stir.ac.uk).

Address: This identifies a user through an email-like address or a telephone-like number (e.g. 1235678, 94723*, 441786467423).

3.1.3 Triggers

Regular policies for call control have triggers as follows. These are added to the regular policy triggers in section 2.13.1. Where triggers also exist in outgoing and incoming forms, the plain form means both situations. ‘Outgoing’ means that a user governed by the policy server has initiated something. ‘Incoming’ means that a user governed by the policy server has to respond to something. Normally these correspond to ‘local request’ and

Trigger	Parameters Established	Actions Permitted
internal†		note_availability, note_presence, send_message
absent†		log_event, note_presence, send_message
available†	topic (empty if *availability* variable is 'true')	connect_to, log_event, note_availability, send_message
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, log_event, note_availability, note_presence, play_clip, reject_call, remove_medium, remove_party, send_message
disconnect, disconnect_incoming, disconnect_outgoing	callee, caller, medium, network_type	log_event, note_availability, note_presence, play_clip, send_message
event	caller, network_type, topic	note_availability, note_presence, send_message
present†	location (empty if *presence* variable is 'true')	connect_to, log_event, note_presence, send_message
register, register_incoming, register_outgoing	caller, network_type	note_presence, reject_call
timer_expiry†		log_event, note_presence, send_message
unavailable†		log_event, note_availability, send_message

† internal trigger that may be combined with any other
‡ available with only certain communications systems

Figure 3.1: Relationship between Triggers, Conditions and Actions

'remote request' respectively. However if both users are governed by the same policy server, 'outgoing' refers to the initiator and 'incoming' refers to the responder.

```
<xsd:simpleType name="trigger_domain">
  <xsd:restriction base="trigger_domain">
    <xsd:enumeration value="absent(arg1)"/>
    <xsd:enumeration value="available(arg1)"/>
    <xsd:enumeration value="bandwidth_request"/>
    <xsd:enumeration value="connect"/>
    <xsd:enumeration value="connect_incoming"/>
    <xsd:enumeration value="connect_outgoing"/>
    <xsd:enumeration value="disconnect"/>
    <xsd:enumeration value="disconnect_incoming"/>
    <xsd:enumeration value="disconnect_outgoing"/>
    <xsd:enumeration value="event"/>
    <xsd:enumeration value="no_answer(arg1)"/>
    <xsd:enumeration value="no_answer_incoming(arg1)"/>
    <xsd:enumeration value="no_answer_outgoing(arg1)"/>
    <xsd:enumeration value="present(arg1)"/>
    <xsd:enumeration value="register"/>
    <xsd:enumeration value="register_incoming"/>
    <xsd:enumeration value="register_outgoing"/>
    <xsd:enumeration value="unavailable(arg1)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy triggers for call control are as follows:

absent(address): This is triggered when the *presence* variable for the specified address becomes 'false'. An empty address means the user to whom the policy applies.

available(address): This is triggered when the *availability* variable for the specified address becomes something other than 'false'. The value 'true' means generally available for an unspecified topic, while a string value defines a specific topic. An empty address means the user to whom the policy applies. If multiple *available* triggers appear in the same policy rule, it is undefined which topic value is set.

bandwidth_request: This occurs if the policy server is informed of a bandwidth request (by an H.323 gatekeeper).

connect, connect_incoming, connect_outgoing: These occur if the policy server is informed by the communications layer of a request to establish a connection.

disconnect, disconnect_incoming, disconnect_outgoing: These occur if the policy server is informed by the communications layer of a request to break an established connection.

event: This is a catch-all for any event provided by an agency external to the policy system. For example a security system might generate an event due to a security violation.

no_answer(period), no_answer_incoming(period), no_answer_outgoing(period): These occur if the policy server is informed by the communications layer that a connection attempt has been not been answered within some period. The argument is the timeout period (measured in seconds).

present(address): This is triggered when the *presence* variable for the specified address becomes something other than 'false'. The value 'true' means generally present in an unspecified location, while a string value defines a specific location. An empty address means the user to whom the policy applies. If multiple *present* triggers appear in the same policy rule, it is undefined which location value is set.

register, register_incoming, register_outgoing: These occur if the policy server is informed by the communications layer of a request to register with the communications system.

unavailable(address): This is triggered when the *availability* variable for the specified address becomes 'false'. An empty address means the user to whom the policy applies.

3.1.4 Conditions

Regular policies for call control have condition parameters as follows:

```
<xsd:simpleType name="parameter_domain">
  <xsd:restriction base="parameter_domain">
    <xsd:enumeration value="active_content"/>
    <xsd:enumeration value="bandwidth"/>
    <xsd:enumeration value="call_content"/>
    <xsd:enumeration value="call_type"/>
    <xsd:enumeration value="callee"/>
    <xsd:enumeration value="caller"/>
    <xsd:enumeration value="capability"/>
    <xsd:enumeration value="capability_set"/>
    <xsd:enumeration value="cost"/>
    <xsd:enumeration value="destination_address"/>
    <xsd:enumeration value="device"/>
    <xsd:enumeration value="location"/>
    <xsd:enumeration value="medium"/>
    <xsd:enumeration value="moment"/>
    <xsd:enumeration value="network_type"/>
    <xsd:enumeration value="priority"/>
    <xsd:enumeration value="quality"/>
    <xsd:enumeration value="role"/>
    <xsd:enumeration value="signalling_address"/>
    <xsd:enumeration value="source_address"/>
    <xsd:enumeration value="topic"/>
    <xsd:enumeration value="traffic_load"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy condition parameters for call control are as follows:

active_content: This means the active media content. It is defined only for H.323.

bandwidth: This is the requested bandwidth as a positive real (measured in Kbps). It is currently defined only for H.323.

call_content: This is the content of a call (e.g. 'Beatles Album', 'Shrek video'). This is an open-ended string that will be defined only if the caller provides it.

call_type: This is the type of a call (e.g. 'emergency', 'long-distance'). This is an open-ended string that will be defined only if the caller provides it.

callee: This is the responder to a call, identified by *address*.

caller: This is the initiator of a call or an event, identified by *address*.

capability: This is the capability of the caller to engage in a call (e.g. 'short discussion only', 'English speaker'). This is an open-ended string that will be defined only if the caller provides it.

capability_set: This is the set of technical functions supported by the caller's equipment. It is defined only for H.323.

cost: This is the cost per minute of a call as a positive real. The unit of currency is arbitrary, but must be known to the caller or the callee. This value will be defined only if the communications layer provides it.

destination_address: This is the destination system, identified by domain name or IP address (e.g. gw.acme.com or 192.168.0.1). It is defined only for H.323.

device: This is the device being used for a call (e.g. 'cell phone', 'PDA'). This is an open-ended string that will be defined only if the caller provides it.

Category	Parameter	Operator
address	*callee*, *caller*, *caller_id*, *destination_address*, *signalling_address*, *source_address*	*eq*, *ne*, *in*, *out*
amount	*bandwidth*, *cost*, *priority*, *traffic_load*	*eq*, *ne*, *lt*, *le*, *gt*, *ge*
description	*active_content*, *call_content*, *call_type*, *capability*, *capability_set*, *device*, *event*, *location*, *medium*, *network_type*, *quality*, *role*, *topic*	*eq*, *ne*, *in*, *out*

Figure 3.2: Address, Amount and Description Parameter Operators

location: This is the location of the caller (e.g. 'downtown', 'Stirling'). This is an open-ended string that will be defined only if the caller provides it.

medium: This is the medium being used for the call. The permissible value is a list of 'audio', 'video', 'white-board'. This is typically provided by the communications layer.

network_type: This is the type of the underlying communications layer (e.g. 'H.323', '7000' (Mitel 7000 ICS), 'SIP'). This is an open-ended string defined by the specific communications layer.

priority: This is the priority assigned to a call as a positive real. Priorities conventionally range from 1 (lowest) to 9 (highest), but no particular values are enforced. This value will be defined only if the caller provides it.

quality: This is the quality of service expected of a call (e.g. 'loss rate below 1%', 'response time below 20ms'). This is an open-ended string that will be defined only if the communications layer provides it.

role: This is the role assumed by the caller (e.g. 'boss', 'spouse'). This is an open-ended string that will be defined only if the caller provides it.

signalling_address: This is the signalling system, identified by domain name or IP address (e.g. gw.acme.com or 192.168.0.1). It is defined only for H.323.

source_address: This is the source system, identified by domain name or IP address (e.g. gw.acme.com or 192.168.0.1). It is defined only for H.323.

topic: This is the subject of a call or an event (e.g. 'project budget' or 'intruder alert'). This is an open-ended string that will be defined only if the caller or event system provides it.

traffic_load: This is the current traffic load as a positive real (measured in Kbps). It is defined only for H.323.

The operators permitted for each parameter are listed in figure 3.2; the use of other operators will yield a result, but this may not be useful. Some special rules for values are as follows:

- For the address category, *in* and *out* are used with a comma-separated list of values to mean 'among' and 'not among'. That is, the parameter is checked for presence or absence in a list of values.
- For the description category, *in* and *out* are used with a text string to mean 'includes' and 'excludes'. That is, the value is checked to be a substring or not of the parameter.

3.1.5 Actions

Regular policies for call control have actions as follows. These are added to the regular policy actions in section 2.8. Actions are requests to the underlying communications layer. If the policy *preference* is *must_not*, an action argument can be omitted to mean any argument value. This allows an easy way to state policies like 'emergency calls must not be forwarded' using *forward_to* without a specific destination argument.

```
<xsd:simpleType name="action_domain">
  <xsd:restriction base="action_domain">
    <xsd:enumeration value="add_caller(arg1)"/>
    <xsd:enumeration value="add_medium(arg1)"/>
    <xsd:enumeration value="add_party(arg1)"/>
    <xsd:enumeration value="confirm_bandwidth"/>
    <xsd:enumeration value="connect_to(arg1)"/>
    <xsd:enumeration value="fork_to(arg1)"/>
    <xsd:enumeration value="forward_to(arg1)"/>
    <xsd:enumeration value="note_availability(arg1)"/>
    <xsd:enumeration value="note_presence(arg1)"/>
    <xsd:enumeration value="play_clip(arg1)"/>
    <xsd:enumeration value="reject_call(arg1)"/>
    <xsd:enumeration value="reject_bandwidth(arg1)"/>
    <xsd:enumeration value="remove_medium(arg1)"/>
    <xsd:enumeration value="remove_party(arg1)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy actions for call control are as follows:

add_caller(method): This adds a new caller to an existing call using the specified method. The action is ignored if the callee is not in an existing call. The argument is the method for adding the caller: *conference* (conference in new caller), *hold* (other party held, new caller connected), *monitor* (new caller listens to current call), *release* (other party disconnected, new caller connected) or *wait* (new caller not connected until call clears). The effect is to modify the current call. Suppose A calls B while B is talking to C. A might be conferenced into the call, C might be held and A connected to B, A might silently monitor the call from B to C, C might be disconnected and A connected to B, or A might be connected to B only after the call from B to C clears.

add_medium(medium): This adds a new medium to the call (or ignores the request if the medium is already in use). The argument is the medium name; see the *medium* parameter in section 3.1.4 for the permissible values. For example, video might be added to an audio call.

add_party(Address): This adds a new party to the call (or ignores the request if the party is already in the call). The argument is the callee address. The effect is to bring about a conference call. For example when A calls B, then B might add C to the call.

confirm_bandwidth: This confirms that the requested bandwidth has been allocated. It applies to H.323 only.

connect_to(Address): This initiates a new and independent call. The caller is the user executing the policy; the argument is the callee address.

disconnect: This forcibly disconnects the current call (if any).

fork_to(Address): This adds an alternative leg to the call. The argument is a caller address (if this is an outgoing call) or a callee address (if this is an incoming call). The effect is that both the current call leg and the alternative call leg are tried in order to reach the party. Whether these are tried in sequence or in parallel depends on the underlying communications layer. However only one can be successful in completing the call.

forward_to(Address): This changes the destination of the call. The argument is the new callee address, as if this had been chosen explicitly by the caller. The forwarding address might identify a program (e.g. an auto-attendant or a voicemail inbox) rather than a person.

note_availability(topic): This updates the *availability* variable for the user whose policy is being executed. The argument is just a topic as a string; see the *available* parameter in section 3.1.3 for the permissible values.

note_presence(location): This updates the *presence* variable for the user whose policy is being executed. The argument is a location as a string; see the *present* parameter in section 3.1.3 for the permissible values.

play_clip(URL): This plays the specified media clip to the caller (e.g. a recorded audio message or a video clip). The argument may be the audio data (raw binary encoded in 'base 64' format) or a URL (e.g. 'file:/tmp/message.wav' or 'http://www.acme.com/hello.avi').

reject_call(reason): This rejects a call, i.e. prevents it from completing. This is used to refuse an incoming call or to abandon an outgoing call. In this context, a call means a call attempt or a registration attempt. The argument is an open-ended string providing a reason for the rejection (e.g. 'too busy currently'). An empty string may be used, perhaps because the precise reason for rejection should not be given. The interpretation of the argument depends on the communications layer. It might be sent as text in a response (e.g. SIP), might be spoken (using Text-To-Speech), or might identify an auto-attendant extension number to speak a message (e.g. Mitel 7000).

reject_bandwidth(limit): This reports the requested bandwidth has not been allocated. The argument is the bandwidth limit that has been exceeded. It applies to H.323 only.

remove_medium(medium): This removes a medium from the call (or ignores the request if the medium is not already in use). The argument is the medium name; see the *medium* parameter in section 3.1.4 for the permissible values. For example, video might be removed from a videoconference call.

remove_party(Address): This removes a conference party from the call (or ignores the request if the party is not already in the call or is the sole other party). The argument is the party address. The effect is to prevent a conference call. For example when A conferences in B and then calls C, then B might remove C from the call before accepting the call from A.

3.2 Example Regular Policies for Call Control

The following call control policies illustrate what can be done with the language. They should provide an insight in the use of the policy language for real examples. The examples are chosen such that they highlight the main aspects of the language and still appear realistic, rather than being contrived.

Each example is introduced briefly by its natural language meaning. This introduction also draws attention to some details of the policy language where appropriate.

An XML wrapper is required for a call control policy in the following 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_document>
```

3.2.1 Forward if Busy

Forward to bob@cs.stir.ac.uk if there is an incoming connection to ken@cs.stir.ac.uk while this user is busy. The empty argument for *unavailable* means the current user is unavailable, i.e. busy.

```
<policy owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
 id="Forward if busy" enabled="true" changed="2004-08-02T11:20:05">
  <policy_rule>
    <triggers>
      <and/>
      <trigger>connect_incoming</trigger>
      <trigger arg1="">unavailable(arg1)</trigger>
    </triggers>
    <action arg1="bob@cs.stir.ac.uk">forward_to(arg1)</action>
  </policy_rule>
</policy>
```

3.2.2 Forward Incoming Calls to Grace

This policy specifies that incoming connections for ken@cs.stir.ac.uk *should* be forwarded to grace@cs.stir.ac.uk during the dates 24th December 2004 to 5th January 2005 inclusive.

```
<policy owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
 id="Forward Incoming to Grace" enabled="true"
 valid_from="2004-12-24T00:00:00" valid_to="2005-01-05T23:59:00"
 changed="2004-08-12T11:33:00">
  <preference>should</preference>
  <policy_rule>
    <trigger>connect_incoming</trigger>
    <action arg1="grace@cs.stir.ac.uk">forward_to(arg1)</action>
  </policy_rule>
</policy>
```

3.2.3 Never forward to Mary

This policy applies to everyone in the cs.stir.ac.uk domain. It says that calls from anyone or by ken@cs.stir.ac.uk *must not* be forwarded to mary@plc.co.uk.

```
<policy owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
 id="Never forward to Mary" enabled="true" changed="2004-08-12T11:43:00">
  <preference>must_not</preference>
  <policy_rule>
    <trigger>connect</trigger>
    <conditions>
      <or/>
      <condition>
        <parameter>caller</parameter>
        <operator>eq</operator>
        <value>@</value>
      </condition>
      <condition>
        <parameter>callee</parameter>
        <operator>eq</operator>
        <value>ken@cs.stir.ac.uk</value>
      </condition>
    </conditions>
    <action arg1="mary@plc.co.uk">forward_to(arg1)</action>
  </policy_rule>
</policy>
```

3.2.4 Never forward Emergency Calls

This policy specifies that emergency calls *must not* be forwarded. This policy applies to @cs.stir.ac.uk, i.e. anyone in this domain. Note also that the forwarding address (*arg1*) is irrelevant and is omitted.

```
<policy owner="ken@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
 id="Never forward Emergency Calls" enabled="true"
 changed="2004-08-12T11:46:00">
  <preference>must_not</preference>
  <policy_rule>
    <trigger>connect</trigger>
    <condition>
      <parameter>call_type</parameter>
      <operator>eq</operator>
      <value>emergency</value>
    </condition>
    <action arg1="">forward_to(arg1)</action>
  </policy_rule>
</policy>
```

3.2.5 Voicemail on Busy or No Answer

This policy states that incoming calls should be forwarded to my voicemail when I am busy or don't answer within 5 rings. This policy has two combinations of trigger events but does not specify any conditions.

```
<policy owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
 id="Voicemail on Busy or No Answer" enabled="true"
 changed="2004-08-02T11:19:00">
  <policy_rule>
    <triggers>
      <or/>
      <trigger arg1="">unavailable(arg1)</trigger>
      <trigger arg1="5">no_answer_incoming(arg1)</trigger>
    </triggers>
    <action arg1="http://voicemail.co.uk/~ken">forward_to(arg1)</action>
  </policy_rule>
</policy>
```

3.2.6 Available for Java

This policy states that lecturers (domain @lecturers.cs.stir.ac.uk) are available for discussions about Java. Note that this policy has no triggers and no conditions, i.e. it is executed immediately on definition.

```
<policy owner="ken@cs.stir.ac.uk" applies_to="@lecturers.cs.stir.ac.uk"
 id="Available for Java" enabled="true"
 changed="2004-07-28T23:18:00">
  <policy_rule>
    <action arg1="Java">note_availability(arg1)</action>
  </policy_rule>
</policy>
```

3.2.7 Complex Busy and No Answer Handling

The next example is deliberately complex. It defines variables for use in the policy, though these might be defined separately rather than statically. By convention, policy variables could be named in upper case (as is common in programming languages for macros).

`Acme` jo@acme.com or bob@acme.com

`home` 01786 832 999, reached via a PSTN gateway (prefix 9)

`hours` 9AM to 5PM

`mobile` 0778 012 3456, reached via a PSTN gateway (prefix 9)

`office` 7423, reached via a PSTN gateway (prefix 9, then a short dialling code)

`voicemail` 2004*, the format for the voicemail inbox of extension 2004

This policy uses sequential and parallel policies to do the following:

```
    when I am busy
  and
    when there is a call
  do forward the call to voicemail

failing that

    when an incoming call is not answered after 5 seconds
      if the caller is not Acme
    and
      if the caller is not tom@uni.edu
      do forward the call to home
```

```
    else
        do forward the call to office
      or else
        do forward the call to mobile

  simultaneously

    when there is a call
      if the call type is business
    and
      if the hour is between hours
      do log an office hours call
    else
      do log an out of hours call
```

A period for the validity of this policy is set: from 24th February 2007 4PM to 4th March 2007 4PM. It will not be triggered outside this period.

```
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation=
  "http://www.cs.stir.ac.uk/schemas/appel_regular_call.xsd">

 <variable id="Acme" owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
  value="jo@acme.com,bob@acme.com" changed="2007-02-23T23:50:00"/>

 <variable id="home" owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
  value="9901786832999" changed="2007-02-23T23:50:00"/>

 <variable id="hours" owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
  value="09:00:00..17:00:00" changed="2007-02-23T23:50:00"/>

 <variable id="mobile" owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
  value="9907780123456" changed="2007-02-23T23:50:00"/>

 <variable id="office" owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
  value="97423" changed="202007-02-23T23:50:00"/>

 <variable id="voicemail" owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
  value="7423*" changed="2007-02-23T23:50:00"/>

 <policy owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
  id="Busy, No Answer, Log" enabled="true"
  valid_from="2007-02-24T16:00:00" valid_to="2007-03-04T16:00:00"
  changed="2007-02-23T23:50:00">
   <policy_rules>
     <sequential/>
     <policy_rule>
       <triggers>
         <and/>
         <trigger arg1="">unavailable(arg1)</trigger>
         <trigger>connect</trigger>
       </triggers>
       <action arg1=":voicemail">forward_to(arg1)</action>
     </policy_rule>
     <policy_rules>
       <parallel/>
       <policy_rule>
         <trigger arg1="5">no_answer_incoming(arg1)</trigger>
         <conditions>
           <and/>
           <condition>
             <parameter>caller</parameter>
```

```
            <operator>ne</operator>
            <value>:Acme</value>
          </condition>
          <condition>
            <parameter>caller</parameter>
            <operator>ne</operator>
            <value>tom@uni.edu</value>
          </condition>
        </conditions>
        <actions>
          <else/>
          <action arg1=":home">forward_to(arg1)</action>
          <actions>
            <orelse/>
            <action arg1=":office">forward_to(arg1)</action>
            <action arg1=":mobile">forward_to(arg1)</action>
          </actions>
        </actions>
      </policy_rule>
      <policy_rules>
        <policy_rule>
          <trigger>connect</trigger>
          <conditions>
            <and/>
            <condition>
              <parameter>call_type</parameter>
              <operator>eq</operator>
              <value>business</value>
            </condition>
            <condition>
              <parameter>time</parameter>
              <operator>in</operator>
              <value>:hours</value>
            </condition>
          </conditions>
          <actions>
            <else/>
            <action arg1="office hours call">log_event(arg1)</action>
            <action arg1="out of hours call">log_event(arg1)</action>
          </actions>
        </policy_rule>
      </policy_rules>
    </policy_rules>
  </policy>

</policy_document>
```

3.2.8 Talking Status

Suppose that bob@cs.stir.ac.uk wishes to note he is talking whenever a connection is made. The first rule below sets the *talking* variable to *true* when a call connects. The second rule unsets this variable on disconnection.

```
<policy owner="bob@cs.stir.ac.uk" applies_to="bob@cs.stir.ac.uk"
 id="Talking status" enabled="true" changed="2007-02-23T23:18:10">
  <policy_rules>
    <sequential/>
    <policy_rule>
      <trigger>connect</trigger>
      <action arg1="talking" arg2="true">set_variable(arg1,arg2)</action>
    </policy_rule>
```

```
    <policy_rule>
      <trigger>disconnect</trigger>
      <action arg1="talking">unset_variable(arg1)</action>
    </policy_rule>
  </policy_rules>
</policy>
```

3.2.9 Call Timer

Suppose that ken@cs.stir.ac.uk likes to log calls that exceed 10 minutes. The first rule below states than on connection a *duration* timer is started for 10 minutes. The second rule states that on disconnection this timer is stopped. The third rule says that if the *duration* timer expires, a lengthy call from the caller is logged.

```
<policy owner="ken@cs.stir.ac.uk" applies_to="ken@cs.stir.ac.uk"
 id="Lengthy call timer" enabled="true" changed="2007-02-23T23:08:41">
  <policy_rules>
    <sequential/>
    <policy_rule>
      <trigger>connect</trigger>
      <action arg1="duration" arg2="00:10:00">start_timer(arg1,arg2)</action>
    </policy_rule>
    <policy_rules>
      <sequential/>
      <policy_rule>
        <trigger>disconnect</trigger>
        <action arg1="duration">stop_timer(arg1)</action>
      </policy_rule>
      <policy_rule>
        <trigger arg1="duration">timer_expiry(arg1)</trigger>
        <action arg1="Lengthy call from :caller">log_event(arg1)</action>
      </policy_rule>
    </policy_rules>
  </policy_rules>
</policy>
```

3.2.10 Working Period Log

Suppose that don@cs.stir.ac.uk has working hours from 9AM to 5PM every day. At 9AM he wishes to log that he has started work. If an external trigger occurs (e.g. a connection), he wishes to log that he was working at this point. This policy does not log the end of the working hours at 5PM, but could be extended to do so with an additional policy rule that is triggered at this point.

Note that the following policy rule has no external trigger. It is initially triggered at 9AM to log the start of work. If any external trigger for the user occurs between 9AM and 5PM, this rule is eligible and logs that the user was working at this time.

```
<policy owner="don@cs.stir.ac.uk" applies_to="don@cs.stir.ac.uk"
 id="Working period log" enabled="true" changed="2007-02-23T23:21:59">
  <policy_rule>
    <condition>
      <parameter>time</parameter>
      <operator>in</operator>
      <value>09:00:00..17:00:00</value>
    </condition>
    <action arg1="Working at :time on :date">log_event(arg1)</action>
  </policy_rule>
</policy>
```

3.2.11 Polite Availability Check

Suppose that kjt@cs.stir.ac.uk prefers not to disturb bob@cs.stir.ac.uk unless it appears he is not very busy. If Bob announces his availability, Ken's policy checks how many times Bob has done this in the past two hours. If it is more than twice, then Bob is probably not having a busy day. In this case, a message is sent to Bob asking if Ken can drop by for a chat.

```
<policy owner="kjt@cs.stir.ac.uk" applies_to="kjt@cs.stir.ac.uk"
 id="Polite availability check" enabled="true" changed="2007-03-06T13:42:25">
  <policy_rule>
    <trigger arg1="bob@cs.stir.ac.uk">available(arg1)</trigger>
    <condition>
      <parameter>count(available(bob@cs.stir.ac.uk),00:02:00)</parameter>
      <operator>gt</operator>
      <value>2</value>
    </condition>
    <action arg1="bob@cs.stir.ac.uk" arg2="Can I drop by for a chat?">
      send_message(arg1,arg2)
    </action>
  </policy_rule>
</policy>
```

3.3 Resolution Policies for Call Control

3.3.1 Introduction

This section is specific to resolution policies for call control, as it discusses the triggers, conditions and actions that are used in this domain.

For call control, resolution policies are extended with domain-specific triggers, conditions and actions. The relationship among these is much simpler than for regular policies. Any action may be associated with a trigger. The actual parameters of these actions must be literal values, or the values of *variable0* to *variable9* (if a trigger has bound them).

3.3.2 Triggers

Resolution policies for call control have triggers as follows. These are added to the resolution policy triggers in section 2.14.1 and to the core triggers in section 2.6. Since resolutions are triggered by actions, resolution policy triggers are identical to regular policy actions for call control (see section 3.1.5).

```
<xsd:simpleType name="trigger_domain">
  <xsd:restriction base="trigger_domain">
    <xsd:enumeration value="add_caller(arg1)"/>
    <xsd:enumeration value="add_medium(arg1)"/>
    <xsd:enumeration value="add_party(arg1)"/>
    <xsd:enumeration value="confirm_bandwidth"/>
    <xsd:enumeration value="connect_to(arg1)"/>
    <xsd:enumeration value="fork_to(arg1)"/>
    <xsd:enumeration value="forward_to(arg1)"/>
    <xsd:enumeration value="note_availability(arg1)"/>
    <xsd:enumeration value="note_presence(arg1)"/>
    <xsd:enumeration value="play_clip(arg1)"/>
    <xsd:enumeration value="reject_call(arg1)"/>
    <xsd:enumeration value="reject_bandwidth(arg1)"/>
    <xsd:enumeration value="remove_medium(arg1)"/>
    <xsd:enumeration value="remove_party(arg1)"/>
  </xsd:restriction>
</xsd:simpleType>
```

3.3.3 Conditions

Resolution policies for call control have the same condition parameters as resolution policies in section 2.14.2.

3.3.4 Actions

Resolution policies for call control have actions as follows. These are added to the resolution policy actions in section 2.14.3 and to the core actions in section 2.8.

```
<xsd:simpleType name="action_domain">
  <xsd:restriction base="action_domain">
    <xsd:enumeration value="add_caller(arg1)"/>
    <xsd:enumeration value="add_medium(arg1)"/>
    <xsd:enumeration value="add_party(arg1)"/>
    <xsd:enumeration value="apply_callee"/>
    <xsd:enumeration value="apply_caller"/>
    <xsd:enumeration value="confirm_bandwidth"/>
    <xsd:enumeration value="connect_to(arg1)"/>
    <xsd:enumeration value="fork_to(arg1)"/>
    <xsd:enumeration value="forward_to(arg1)"/>
    <xsd:enumeration value="note_availability(arg1)"/>
    <xsd:enumeration value="note_presence(arg1)"/>
    <xsd:enumeration value="play_clip(arg1)"/>
    <xsd:enumeration value="reject_call(arg1)"/>
    <xsd:enumeration value="reject_bandwidth(arg1)"/>
    <xsd:enumeration value="remove_medium(arg1)"/>
    <xsd:enumeration value="remove_party(arg1)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Most of these actions are those of regular policies for call control in section 3.1.5. However, the following actions are additional for resolution:

apply_callee: This chooses the callee's call action.

apply_caller: This chooses the caller's call action.

3.4 Example Resolution Policies for Call Control

The following call conflict policies illustrate what can be done with the language. They should provide an insight in the use of the policy language for real examples. The examples are chosen such that they highlight the main aspects of the language and still appear realistic, rather than being contrived.

Each example is introduced briefly by its natural language meaning. This introduction also draws attention to some details of the policy language where appropriate.

An XML wrapper is required for a call conflict policy in the following 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_document>
```

A resolution policy typically contains two triggers, each of which defines explicitly a *variable* and implicitly a *preference*. In general, there are four cases to be considered by a resolution depending on whether the variables are equal/unequal and whether the preferences are similar/opposite. This can often be simplified. In some cases it is sufficient to check for the equal/similar combination; the other combinations are handled by default. In other cases, the equal/opposite and unequal/similar combinations may be handled. In unusual cases, all four combinations may need individual treatment. Since the **else** operator can be used with only two alternative actions, multiple resolutions may therefore be required for the same trigger combination.

There are other variations on the basic case. Resolution policies may have triggers with no or two arguments, so there may not be exactly two *variable* values. Resolution policies may also have more than two triggers.

3.4.1 Call Fork-Fork Conflict – Generic Resolution

Virtually any call control action may conflict with itself if its arguments are the same and the preferences of each party are opposite. As an example, suppose one party wishes to fork the call to an alternative address (e.g. to try a home number in addition to the dialled office number). Suppose the other party does not wish to fork the call to this address (e.g. because the callee must be called only in the office). The following detects this conflict, and resolves it by choosing the stronger of the two preferences.

This resolution explicitly deals with only the equal/opposite case. The equal/similar case is not explicitly handled; 'must fork to address *A*' and 'should fork to address *A*', for example, will result in forking to *A* since one of the two equivalent actions will be selected by default. The unequal/opposite case is not explicitly handled; 'must fork to address *A*' and 'should not fork to address *B*', for example, will result in forking to only *A* since actions with negative preferences are not performed. The unequal/similar case is not explicitly handled; 'must fork to address *A*' and 'should fork to address *B*', for example, will result in forking to both *A* and *B* since both actions are compatible.

```
<resolution id="Call fork-fork conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2008-07-10T20:40:00">
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1="variable0">fork_to(arg1)</trigger>
      <trigger arg1="variable1">fork_to(arg1)</trigger>
    </triggers>
    <conditions>
     <and/>
     <condition>
       <parameter>variable0</parameter>
       <operator>eq</operator>
       <parameter>variable1</parameter>
     </condition>
     <condition>
       <parameter>preference0</parameter>
       <operator>out</operator>
       <value>:preference1</value>
     </condition>
    </conditions>
    <action>apply_stronger</action>
  </policy_rule>
</resolution>
```

Similar resolutions could be defined for pairs of *add_caller*, *add_party*, *add_medium*, etc.

3.4.2 Call Forward-Forward Conflict – Generic Resolution

Call forwarding is another example of a call control action conflicting with itself. However, the resolution is more complex. There is conflict if the forwarding addresses are the same and the preferences are opposite (equal/opposite case), or if the forwarding addresses differ and the preferences are similar (unequal/similar case) The resolution given here is to apply the caller's preference. The equal/similar and unequal/opposite cases are handled by default much as described in section 3.4.1.

```
<resolution id="Call forward-forward conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2008-07-10T20:40:00">
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1="variable0">forward_to(arg1)</trigger>
      <trigger arg1="variable1">forward_to(arg1)</trigger>
    </triggers>
    <conditions>
```

```
      <or/>
      <conditions>
       <and/>
       <condition>
         <parameter>variable0</parameter>
         <operator>eq</operator>
         <parameter>variable1</parameter>
       </condition>
       <condition>
         <parameter>preference0</parameter>
         <operator>out</operator>
         <value>:preference1</value>
       </condition>
      </conditions>
      <conditions>
        <and/>
        <condition>
          <parameter>variable0</parameter>
          <operator>ne</operator>
          <parameter>variable1</parameter>
        </condition>
        <condition>
          <parameter>preference0</parameter>
          <operator>in</operator>
          <value>:preference1</value>
        </condition>
      </conditions>
    </conditions>
    <action>apply_stronger</action>
  </policy_rule>
</resolution>
```

Similar resolutions could be defined for pairs of *note_availability*, *note_presence*, *reject_call*, etc.

3.4.3 Medium Add-Remove Conflict – Generic Resolution

A number of call control actions are inverses of each other, and are an obvious source of conflict. For example, a conflict arises if one party wishes to add some medium to the call while the other party wishes to remove this. The following checks if the medium in question is the same for both actions, and whether the associated preferences are similar. If so, it selects the weaker preference.

This resolution explicitly deals with only the equal/similar case. The equal/opposite case is not explicitly handled; 'must add medium *M*' and 'should not remove medium *M*', for example, will result in adding *M* since actions with negative preferences are not performed. The unequal cases are handled by default much as described in section 3.4.1.

```
<resolution id="Medium add-remove conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2008-07-10T20:40:00">
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1="variable0">add_medium(arg1)</trigger>
      <trigger arg1="variable1">remove_medium(arg1)</trigger>
    </triggers>
    <conditions>
      <and/>
      <condition>
        <parameter>variable0</parameter>
        <operator>eq</operator>
        <parameter>variable1</parameter>
      </condition>
```

```
      <condition>
        <parameter>preference0</parameter>
        <operator>in</operator>
        <value>:preference1</value>
      </condition>
    </conditions>
    <action>apply_stronger</action>
  </policy_rule>
</resolution>
```

Similar resolutions could be defined for *add_party* vs. *remove_party*, *confirm_bandwidth* vs. *reject_bandwidth*, etc.

3.4.4 Call Fork-Reject Conflict – Generic Resolution

Sometimes, conflicting actions are not simple inverses. For example, a conflict arises if one party wishes to fork the call while the other wishes to reject it. Only the preferences of the parties are relevant to resolution: if they are similar, the superior party's action is selected by the following:

This resolution explicitly deals with only the equal/similar case. The equal/opposite case is not explicitly handled; 'must fork to address *A*' and 'should not reject call for reason *R*', for example, will result in forking to *A* since actions with negative preferences are not performed. The unequal cases are irrelevant since the *variable* values need not be checked.

```
<resolution id="Call fork-reject conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2008-07-10T20:40:00">
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1="variable0">fork_to(arg1)</trigger>
      <trigger arg1="variable1">reject_call(arg1)</trigger>
    </triggers>
    <condition>
      <parameter>preference0</parameter>
      <operator>in</operator>
      <value>:preference1</value>
    </condition>
    <action>apply_stronger</action>
  </policy_rule>
</resolution>
```

Similar resolutions could be defined for *fork_to* vs. *forward_to*, *forward_to* vs. *reject_call*, etc.

3.4.5 Bandwidth Confirm-Reject Conflict – Specific Resolution

This example is a straightforward conflict: one party wishes to confirm the requested bandwidth, while the other wishes to reject the request. The resolution this time is specific: the bandwidth request is confirmed, and the conflict is noted in the confirmer's event log. This resolution explicitly deals with only the equal/similar case. Other cases are handled by default much as described in section 3.4.1.

```
<resolution id="Bandwidth confirm-reject conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2008-07-10T20:40:00">
  <policy_rule>
    <triggers>
      <and/>
      <trigger>confirm_bandwidth</trigger>
      <trigger arg1="variable1">reject_bandwidth(arg1)</trigger>
    </triggers>
    <condition>
      <parameter>preference0</parameter>
      <operator>in</operator>
```

```
      <value>:preference1</value>
    </condition>
    <actions>
      <and/>
      <action>confirm_bandwidth</action>
      <action arg1="Overruled bandwidth conflict of :caller: and callee">
        log_event(arg1)
      </action>
    </actions>
  </policy_rule>
</resolution>
```

3.4.6 Caller-Medium Add-Add – Specific Resolution

Suppose that one party wishes to add video to the call, while the other wishes include a third party in the call (*add_caller*). This might be considered undesirable, since the third party would be able to view the call parties and their workplaces. The resolution is to allow both actions, but to conference in bob@cs.stir.ac.uk to oversee the call (*add_party*). Note that the triggers and actions are all of different types. This resolution explicitly deals with only the equal/similar case. Other cases are handled by default much as described in section 3.4.1.

```
<resolution id="Caller-Medium add-add conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2008-07-10T20:40:00">
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1="variable0">add_caller(arg1)</trigger>
      <trigger arg1="variable1">add_medium(arg1)</trigger>
    </triggers>
    <conditions>
      <and/>
      <condition>
       <parameter>variable1</parameter>
       <operator>eq</operator>
       <value>video</value>
      </condition>
      <condition>
        <parameter>preference0</parameter>
        <operator>in</operator>
        <value>:preference1</value>
      </condition>
    </conditions>
    <actions>
      <and/>
      <actions>
        <and/>
        <action arg1="variable0">add_caller(arg1)</action>
        <action arg1="variable1">add_medium(arg1)</action>
      </actions>
      <action arg1="bob@cs.stir.ac.uk">add_party(arg1)</action>
    </actions>
  </policy_rule>
</resolution>
```

3.4.7 Timer Start-Stop Conflict – Specific Resolution

Timer actions can give rise to conflicts. The following checks if the timer identifier is the same for both actions, and whether the associated preferences are similar. If so, it chooses to start the timer identified by *variable0*.

This resolution explicitly deals with only the equal/similar case. The equal/opposite case is not explicitly handled; 'must start timer *T*' and 'should not stop timer *T*', for example, will result in timer *T* being started. The unequal cases are handled by default much as described in section 3.4.1.

```
<resolution id="Timer start-stop conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2008-07-10T20:40:00">
  <policy_rule>
    <triggers>
      <and/>
    <trigger arg1="variable0" arg2="variable2">start_timer(arg1,arg2)</trigger>
    <trigger arg1="variable1">stop_timer(arg1)</trigger>
    </triggers>
    <conditions>
      <and/>
      <condition>
        <parameter>variable0</parameter>
        <operator>eq</operator>
        <parameter>variable1</parameter>
      </condition>
      <condition>
        <parameter>preference0</parameter>
        <operator>in</operator>
        <value>:preference1</value>
      </condition>
    </conditions>
    <action arg1=":variable0" arg2=":variable2">start_timer(arg1,arg2)</action>
  </policy_rule>
</resolution>
```

Similar resolutions could be defined for *start_timer* vs. *start_timer*, *start_timer* vs. *restart_timer*, etc.

3.4.8 Variable Set-Set Conflict – Specific Resolution

Setting variables can give rise to conflicts. The following checks if the variable identifier is the same for both actions, but the values being assigned are different. If so it logs the situation and performs no assignment, otherwise it performs both assignments (even if the same variable is involved in both cases).

This resolution explicitly deals with the equal and unequal cases. Note that no preferences are taken into account as an assignment conflict is considered to be a fundamental problem.

```
<resolution id="Variable set-set conflict"
 owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk" enabled="true"
 changed="2007-12-13T20:40:00">
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1="variable0"
       arg2="variable2">set_variable(arg1,arg2)</trigger>
      <trigger arg1="variable1"
       arg2="variable3">set_variable(arg1,arg2)</trigger>
    </triggers>
    <conditions>
      <and/>
      <condition>
        <parameter>variable0</parameter>
        <operator>eq</operator>
        <parameter>variable1</parameter>
      </condition>
      <condition>
        <parameter>variable2</parameter>
        <operator>ne</operator>
        <parameter>variable3</parameter>
      </condition>
    </conditions>
    <actions>
```

```
      <else/>
      <action arg1="Incompatible assignment to :variable0">log_event(arg1)</action>
      <actions>
        <and/>
        <action arg1=":variable0" arg2=":variable2">
          set_variable(arg1,arg2)
        </action>
        <action arg1=":variable1" arg2=":variable3">
          set_variable(arg1,arg2)
        </action>
      </actions>
    </actions>
  </policy_rule>
</resolution>
```

Similar resolutions could be defined for *set_variable* vs. *unset_variable*, and *unset_variable* vs. *unset_variable*.

3.5 Prototype Policies for Call Control

An XML wrapper is required for a call control prototype policy in the following form:

```
<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_document>
```

Because a prototype policy is instantiated as a regular policy, the relevant schema is that of the latter.

As an example of a prototype policy, the following aims to limit call duration. The call length is determined by a parameter (*$duration*) that is optimised during goal refinement. The policy has three sequential rules that are tried in order until the one with a matching trigger is found. When a call is connected, a *call_timer* is started. When the call is disconnected, this timer is stopped. If the timer expires, the call is forcibly disconnected.

```
<prototype owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
 effect="call_duration = $duration" id="Parameterised call duration"
 enabled="true" changed="2009-01-03T17:18:00">
  <policy_rules>
    <sequential/>
    <policy_rule>
      <trigger>connect</trigger>
      <action arg1="call_timer" arg2="$duration">
        start_timer(arg1,arg2)
      </action>
    </policy_rule>
    <policy_rules>
      <sequential/>
      <policy_rule>
        <trigger>disconnect</trigger>
        <action arg1="call_timer">stop_timer(arg1)</action>
      </policy_rule>
      <policy_rule>
        <trigger arg1="call_timer">timer_expiry(arg1)</trigger>
        <action>disconnect</action>
      </policy_rule>
    </policy_rules>
  </policy_rules>
</prototype>
```

The abstract effect of this policy on controlled variables is to limit the call duration to the optimal value of the *$duration* parameter.

3.6 Goals for Call Control

An XML wrapper is required for a call control goal in the following form:

```
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel.xsd">
  ...
</policy_document>
```

As an example of a goal for call control, the following applies only when the day is Monday to Friday:

```
<goal owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
 id="Maximise multimedia use" enabled="true" changed="2009-01-03T17:18:00">
  <policy_rule>
    <condition>
      <parameter>day</parameter>
      <operator>in</operator>
      <value>1..5</value>
    </condition>
    <action arg1="multimedia_use">maximise(arg1)</action>
  </policy_rule>
</goal>
```

This goal aims to maximise *multimedia_use*. This is a goal measure that is defined separately in terms of various controlled variables:

$$0.03 \times \text{call\_bandwidth} \times \text{call\_quality}$$

Chapter 4

Sensor Networks

4.1 Regular Policies for Sensor Networks

4.1.1 Introduction

This section describes regular policies for the domain of sensor networks. It discusses the triggers, conditions and actions used to specialise the language for this field. The policy server employs a terminology mapping between information in the underlying sensor network and information in the policy system.

For sensor networks, regular policies are extended with domain-specific triggers, conditions and actions. The relationship among these is shown in figure 4.1. Core triggers (section 2.6), condition parameters (section 2.7) and actions (section 2.8) are also applicable. The combination rules are as follows:

- Triggers may be combined with *and* or *or*. Internal triggers (marked † in figure 3.1) may be combined with any other triggers. At most one external trigger (no † in figure 4.1) may appear underneath an *and*.
- Triggers establish environment variables that are used as condition parameters. If a combination of triggers is provided, *and* establishes the union of the environment variables, while *or* establishes their intersection.
- Triggers allow certain actions to occur. If a combination of triggers is provided, *and* allows the union of the actions, while *or* allows their intersection.

Trigger	**Parameters Established**	**Actions Permitted**
internal†		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 trigger that may be combined with any other

Figure 4.1: Relationship between Sensor Network Triggers, Conditions and Actions

4.1.2 Common Parameters and Environment Variables

Regular policies for sensor networks use a common set of parameters within triggers, conditions and actions. These parameters are outlined below together with examples of their use.

message_type: This identifies the type of trigger or action. The value will vary depending on the particular *entity_name*. Example trigger types might be *generator_overspeed*, *minimum_temperature* and *operator_input*. Example action names might be *set_rule* and *set_threshold* for a sensor node, and *restart_agent* or *stop_agents* for a policy agent. This parameter is mandatory.

entity_name: This identifies an external entity that may have policies associated with it. Example entity names might be *operator_console*, *policy_agent*, *sensor_node* and *turbine*. An entity name may be unnecessary if it is implied by the message type, e.g. operator input or output implies use of the default console. If this parameter is omitted, its default is an empty string.

entity_instance: This identifies a particular instance of *entity_name*. For example, a *turbine* instance might be turbine number *B43*, while a *policy_agent* instance might be *temperature_anomaly_agent*. It is possible not all entities will have instances, such as an operator console. If this parameter is omitted, its default is an empty string.

message_period: The time period to which a trigger or action refers. The period is 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). Due to transmission and processing delays, this time period is unlikely to be respected exactly. An absolute time reference spans a maximum of 24 hours. Suppose the time is currently 12:00:00. For a trigger, 11:00:00 means 'since 11AM today' and 13:00:00 means 'since 1PM yesterday'. For an action, 11:00:00 means 'at 11AM tomorrow' and 13:00:00 means 'at 1PM today'. If this parameter is omitted, its default is 'just occurred' for a trigger and 'now' for an action.

parameter_values: This is an open-ended string that may be a single value like 'configuration success' or a comma-separated list of values like '14,20,28'. It may also include sub-strings in square brackets, such as '40,B34,[wind_speed,15,alert_operator]'. If this parameter is omitted, its default is an empty list of values.

As far as the policy server is concerned, these parameters are simply uninterpreted strings whose meaning is given only by the external system. The configuration, addressing and parameters of sensor networks are outside the scope of the policy system.

The policy system establishes environment variables on the occurrence of each trigger. The parameters above double as environment variables. This is a flexible approach, allowing each parameter to appear explicitly in a trigger or action, as well as being defined implicitly as an environment variable. Environment variables may also be used to output parameter values literally in an action, such as inserting the *entity_instance* of a turbine in a warning message to an operator console. The example policies in sections 4.2.2 and 4.2.3 demonstrate the use of environment variables.

4.1.3 Triggers

Regular policies for sensor networks have triggers as follows. (The somewhat lengthy definition simply says that a mandatory *arg1* is followed by optional trailing arguments.) These are added to the regular policy triggers in section 2.13.1.

```
<xsd:simpleType name="trigger_domain">
  <xsd:restriction base="trigger_domain">
    <xsd:enumeration value="device_in(arg1,arg2,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_in(arg1,arg2,arg3,arg4)"/>
    <xsd:enumeration value="device_in(arg1,arg2,arg3)"/>
    <xsd:enumeration value="device_in(arg1,arg2)"/>
    <xsd:enumeration value="device_in(arg1)"/>
    <xsd:enumeration value="device_in(arg1,,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_in(arg1,,arg3,,arg5)"/>
    <xsd:enumeration value="device_in(arg1,,arg3,arg4)"/>
    <xsd:enumeration value="device_in(arg1,,arg3)"/>
    <xsd:enumeration value="device_in(arg1,,,arg4,arg5)"/>
    <xsd:enumeration value="device_in(arg1,,,arg4)"/>
    <xsd:enumeration value="device_in(arg1,,,,arg5)"/>
    <xsd:enumeration value="device_in(arg1,arg2,,arg4,arg5)"/>
    <xsd:enumeration value="device_in(arg1,arg2,,arg4)"/>
```

```
    <xsd:enumeration value="device_in(arg1,arg2,,,arg5)"/>
    <xsd:enumeration value="device_in(arg1,arg2,arg3,,arg5)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy triggers for sensor networks are structured as follows:

device_in(type,entity,instance,period,parameters)**:** This arises from any external device. Only the *type* is mandatory; section 4.1.2 explains these values.

Trigger arguments are simply uninterpreted strings. Once more definite examples in this new domain have been determined, trigger arguments may be made more constrained.

4.1.4 Conditions

Regular policies for sensor networks have condition parameters as follows:

```
<xsd:simpleType name="parameter_domain">
  <xsd:restriction base="parameter_domain">
    <xsd:enumeration value="entity_instance"/>
    <xsd:enumeration value="entity_name"/>
    <xsd:enumeration value="message_period"/>
    <xsd:enumeration value="message_type"/>
    <xsd:enumeration value="parameter_values"/>
  </xsd:restriction>
</xsd:simpleType>
```

A detailed explanation of each parameter is given in section 4.1.2. The operators permitted for each parameter are listed in figure 4.2; the use of other operators will yield a result, but this may not be useful. Some special rules for values are as follows:

- For the *identifier* category, *in* and *out* are used with a comma-separated list of values to mean 'among' and 'not among'. That is, the parameter is checked for presence or absence in a list of values.
- For the *value* category, *in* and *out* are used with a comma-separated list of values to mean 'among' and 'not among'. That is, the parameter is checked for presence or absence in a list of values.

  In the case of date, day or time, a value may also be a range (e.g. 09:00:00..12:00:00 or 2005-02-03..2005-02-10).

  In the case of *parameter_values*, a comma-separated list is given in brackets. A list index is given in brackets, e.g. *parameter_values[2]* refers to the third element in the list. For convenience with single values, omitting the index implies an index of 0 (so *parameter_values* means *parameter_values[0]*).

The semantics of the condition operator depend on the types of its operands. Suppose the operator is *gt* is used with parameter operand *parameter_values[2]* and value operand '12'. If the parameter operand is '20' (i.e. is in numerical format), the condition will be treated as numerical comparison. If the parameter operand is 'on' (i.e. a string), the condition will be treated as string comparison.

Category	**Parameter**	**Operator**
identifier	*entity_name*, *entity_instance*, *message_type*	*eq*, *ne*, *in*, *out*
value	*message_period*, *parameter_values*	*eq*, *ne*, *in*, *out*, *gt*, *ge*, *lt*, *le*

Figure 4.2: Identifier and Value Parameter Operators

4.1.5 Actions

Regular policies for sensor networks have actions as follows. (The somewhat lengthy definition simply says that a mandatory *arg1* is followed by optional trailing arguments.) These are added to the regular policy actions in section 2.8. Actions are requests to the underlying communications layer. If the policy *preference* is *must_not*, an action argument can be omitted to mean any argument value.

```
<xsd:simpleType name="action_domain">
  <xsd:restriction base="action_domain">
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3)"/>
    <xsd:enumeration value="device_out(arg1,arg2)"/>
    <xsd:enumeration value="device_out(arg1)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,arg3)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,,arg5)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy actions for sensor networks are structured as follows:

device_out(type,entity,instance,period,parameters)**:** This controls or configures an entity external to the policy system. Only the *type* is mandatory; section 4.1.2 explains these values.

Action arguments are simply uninterpreted strings. Once more definite examples in this new domain have been determined, action arguments may be made more constrained.

4.2 Example Regular Policies for Sensor Networks

The following sensor network policies illustrate the capabilities of the language. They should provide insight into the policy language for real examples in this domain. Examples include policies to configure the sensor network and to control aspects of an agent-based diagnostic system. Policies interact with the sensor network and also with a human operator via a console interface.

Each example policy is introduced briefly by its natural language meaning. This introduction also draws attention to some details of the policy language where appropriate.

An XML wrapper is required for a sensor network policy in the following 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_document>
```

4.2.1 High Wind Alert

The following policy applies to domain *3.wind_farm.com*, i.e. wind farm 3. It alerts the operator if the average speed from any anemometer is reported to be above 20m/s for at least one hour. No entity name or instance is given in the action because operator messages are for the default console. In general there may be several trigger parameters, but just a single value (the wind speed) is used here. Note that this policy deals with a single wind speed report. To deal with multiple reports, the trigger history would need to be checked.

```
<policy owner="admin" applies_to="@3.wind_farm.com"
 id="High wind alert" enabled="true" changed="2007-09-25T10:20:59">
  <policy_rule>
    <trigger arg1="average_speed" arg2="anemometer">
      device_in(arg1,arg2)
```

```
    </trigger>
    <conditions>
      <and/>
      <condition>
        <parameter>message_period</parameter>
        <operator>ge</operator>
        <value>60</value>
      </condition>
      <condition>
        <parameter>parameter_values</parameter>
        <operator>gt</operator>
        <value>20</value>
      </condition>
    </conditions>
    <action arg1="operator_output" arg5="Persistent high wind speed">
      device_out(arg1,,,,arg5)
    </action>
  </policy_rule>
</policy>
```

4.2.2 Low Battery Alert

The following policy applies to the *myers_hill.org* domain. It deals with a low battery warning from any sensor node, logging the event and sending a message to 07779-432-020. Note the use of environment variables in the actions.

```
<policy owner="admin" applies_to="@myers_hill.org"
 id="Low battery alert" enabled="true" changed="2009-05-01T22:20:59">
  <policy_rule>
    <trigger arg1="low_battery" arg2="sensor_node">
      device_in(arg1,arg2)
    </trigger>
    <actions>
      <and/>
      <action arg1="Low battery at sensor :entity_instance">
        log_event(arg1)
      </action>
      <action arg1="sms:07779-432-020"
              arg2="Low battery warning from :entity_name/:entity_instance">
        send_message(arg1,arg2)
      </action>
    </actions>
  </policy_rule>
</policy>
```

4.2.3 Sensor Wake-up

The following policy assumes that a sensor causes a 'hello' event when it goes online. This information is reported to operator console 3. The sensor is then configured to immediately store a rule with parameters 'wind_speed,15,alert_operator'.

```
<policy owner="admin" applies_to="@15.wind_farm"
 id="Sensor wake-up" enabled="true" changed="2009-05-01T22:20:59">
  <policy_rule>
    <trigger arg1="sensor_hello">device_in(arg1)</trigger>
    <actions>
      <and/>
      <action arg1="operator_output" arg2="operator_console" arg3="3"
              arg5="Sensor :entity_instance is now online">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
```

```
      <action arg1="set_rule" arg2=":entity_name" arg3=":entity_instance"
              arg5="[wind_speed,15,alert_operator]">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
    </actions>
  </policy_rule>
</policy>
```

4.2.4 Reset All Agents

The following policy resets all agents in the external agent system at 9AM every Monday. Note that there is no explicit trigger, only an implicit time trigger.

```
<policy owner="admin" applies_to="@blacklaw.scotland.uk"
 id="Reset all agents" enabled="true" changed="2009-05-01T22:20:59">
  <policy_rule>
   <conditions>
     <and/>
     <condition>
       <parameter>day</parameter>
       <operator>eq</operator>
       <value>1</value>
     </condition>
     <condition>
       <parameter>time</parameter>
       <operator>eq</operator>
       <value>09:00:00</value>
     </condition>
   </conditions>
   <action arg1="reset_agents" arg2="agent_system">
     device_out(arg1,arg2)
   </action>
  </policy_rule>
</policy>
```

4.2.5 Retrain Power Agent

The following policy is triggered by input from the default operator console, asking that the power curve agent be retrained. It asks the external agent system to retrain this agent in ten minutes using value 30.

```
<policy owner="admin" applies_to="@air.co.uk"
 id="Retrain power agent" enabled="true" changed="2009-05-01T22:20:59">
  <policy_rule>
    <trigger arg1="operator_input" arg5="retrain_power_curve">
      device_in(arg1,,,,arg5)
    </trigger>
    <action arg1="retrain_agent" arg2="agent_system" arg3="power_curve"
            arg4="10" arg5="30">
      device_out(arg1,arg2,arg3,arg4,arg5)
    </action>
  </policy_rule>
</policy>
```

4.3 Resolution Policies for Sensor Networks

4.3.1 Introduction

This section is specific to resolution policies for sensor networks. It discusses the triggers, conditions and actions that are used in this domain.

Resolution policies are extended with domain-specific triggers, conditions and actions. The relationship among these is much simpler than for regular policies. Any action may be associated with a trigger. The actual parameters of these actions must be literal values, or the values of *variable0* to *variable9* (if a trigger has bound them).

In general, conflicting situations may arise due to the *preference* attributes of policies being opposite, or due to the effects of policy actions clashing. As described in section 4.1.5, policies for sensor networks have only one domain-specific action. Resolutions for this domain therefore deal with modality (preference) conflicts and parameter conflicts for this action.

4.3.2 Triggers

Resolution policies for sensor networks have triggers as follows. These are added to the resolution policy triggers in section 2.14.1 and to the core triggers in section 2.6. Since resolutions are triggered by actions, resolution policy triggers are identical to regular policy actions for sensor networks (see section 4.1.5).

```
<xsd:simpleType name="trigger_domain">
  <xsd:restriction base="trigger_domain">
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3)"/>
    <xsd:enumeration value="device_out(arg1,arg2)"/>
    <xsd:enumeration value="device_out(arg1)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,arg3)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,,arg5)"/>
  </xsd:restriction>
</xsd:simpleType>
```

4.3.3 Conditions

Resolution policies for sensor networks have the same condition parameters as resolution policies in section 2.14.2.

4.3.4 Actions

Resolution policies for sensor networks have actions as follows. These are added to the resolution policy actions in section 2.14.3 and to the core actions in section 2.8.

```
<xsd:simpleType name="action_domain">
  <xsd:restriction base="action_domain">
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3)"/>
    <xsd:enumeration value="device_out(arg1,arg2)"/>
    <xsd:enumeration value="device_out(arg1)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,arg3)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4,arg5)"/>
```

```
    <xsd:enumeration value="device_out(arg1,arg2,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,,arg5)"/>
  </xsd:restriction>
</xsd:simpleType>
```

These actions are those of regular policies for sensor networks in section 4.1.5. A number of goal-related or domain-specific resolution actions will be considered in future, based on further policy attributes such as cost, efficiency or time. These properties could help determine the action which best meets some goal. Once more definite examples in this new domain have been determined, generic resolution actions may be extended.

4.4 Example Resolution Policies for Sensor Networks

The following sensor network conflict policies demonstrate the capabilities of the language. They should provide an insight into the use of the policy language for real examples of handling conflicts in this domain.

Each example is introduced briefly by its natural language meaning. This introduction also draws attention to some details of the policy language where appropriate.

An XML wrapper is required for a sensor network conflict policy in the following 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_document>
```

A resolution policy typically contains two triggers, each of which binds explicit *variable* values and implicit *preference* values. In general, there are four cases to be considered by a resolution depending on whether the variables are equal/unequal and whether the preferences are similar/opposite. This can often be simplified. In some cases it is sufficient to check for the equal/similar combination; the other combinations are handled by default. In other cases, the equal/opposite and unequal/similar combinations may be handled. In unusual cases, all four combinations may need individual treatment. Since the **else** operator can be used with only two alternative actions, multiple resolutions may therefore be required for the same trigger combination.

There are other variations on the basic case. For example, each trigger of a resolution policy may have up to five parameters, and more than two triggers may be present in a resolution policy.

Policy conflicts in sensor networks can occur between the parameters of *device_out* actions. Examples of generic and specific resolutions for this type of conflict are given in sections 4.4.1 and 4.4.2. Conflicts may also occur indirectly when actions cannot be performed simultaneously, due to limitations of the underlying hardware. Section 4.4.3 gives an example of this type of conflict.

4.4.1 Action Parameter Conflict – Generic Resolution

The following resolution policy checks whether the action type, entity, instance and parameters are identical for both actions. If the preferences are opposite, the action with the stronger preference is selected. To avoid a tedious list of conditions on each individual action parameter, they are compared as a single string (the *variable* values being interpolated into this).

Virtually any *device_out* action may conflict with itself if its arguments are similar and the preferences of each policy are opposite. The following example detects this conflict, and resolves it by choosing the stronger of the two preferences.

This resolution explicitly deals with only the equal/opposite case. The equal/similar and unequal cases are handled by default.

```
<resolution owner="admin" applies_to="@turbine.org"
 id="Action parameter conflict - generic resolution" enabled="true"
 changed="2009-05-01T22:20:59">
  <policy_rule>
    <triggers>
      <and/>
```

```
      <trigger arg1=″variable0″ arg2=″variable2″ arg3=″variable4″
               arg5=″variable8″>
        device_out(arg1,arg2,arg3,,arg5)
      </trigger>
      <trigger arg1=″variable1″ arg2=″variable3″ arg3=″variable5″
               arg5=″variable9″>
        device_out(arg1,arg2,arg3,,arg5)
      </trigger>
    </triggers>
    <conditions>
      <and/>
      <condition>
        <value>′:variable0,:variable2,:variable4,:variable8′</value>
        <operator>eq</operator>
        <value>′:variable1,:variable3,variable5,:variable9′</value>
      </condition>
      <condition>
        <parameter>preference0</parameter>
        <operator>out</operator>
        <parameter>preference1</parameter>
      </condition>
    </conditions>
    <action>apply_stronger</action>
  </policy_rule>
</resolution>
```

The same instance and period are important. If the periods for applying the actions is different, then they may not conflict. Even if both actions concern the same entity, there may be no conflict if the instance differs.

In the example above, the policy with the stronger preference was selected. Other generic actions could include applying the weaker policy, the newer/older policy (based on definition date), or the policy belonging to the inferior/superior domain.

4.4.2 Action Parameter Conflict – Specific Resolution

The following resolution policy checks whether the action type, entity, instance and first parameter are identical for both actions. If so, an operator is alerted to the conflict. Note the use of '[0]' to select the first parameter.

```
<resolution owner=″admin″ applies_to=″@5.windy.us″
 id=″Action parameter conflict - specific resolution″ enabled=″true″
 changed=″2009-05-01T22:20:59″>
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1=″variable0″ arg2=″variable2″ arg3=″variable4″
               arg5=″variable8″>
        device_out(arg1,arg2,arg3,arg4,arg5)
      </trigger>
      <trigger arg1=″variable1″ arg2=″variable3″ arg3=″variable5″
               arg5=″variable9″>
        device_out(arg1,arg2,arg3,arg4,arg5)
      </trigger>
    </triggers>
    <condition>
      <value>′:variable0,:variable2,:variable4,:variable8[0]′</value>
      <operator>eq</operator>
      <value>′:variable1,:variable3,:variable5,,:variable9[0]′</value>
    </condition>
    <action arg1=″operator_output″
            arg5=″Conflicting values for entity :variable2 instance :variable4″>
      device_out(arg1,,,,arg5)
    </action>
```

```
    </policy_rule>
  </resolution>
```

Other resolutions could be defined such as a generic action, logging an event, or choosing one value.

4.4.3 Resource Conflict – Generic Resolution

The following resolution policy checks if one action is *get_log* and the other is *sensor_check*. If the action entity, instance and period are identical for both actions, the action with the stronger preference is applied.

This conflict is an example of two actions that cannot be performed simultaneously due to limitations of the underlying sensor network hardware. For example, there may be constraints on processing power, memory, electrical power and link bandwidth. Although not detectable using analysis of policies alone, resolutions could be written to handle potential conflicts with some knowledge of the abstract resources need to perform actions.

```
<resolution owner="admin" applies_to="@3.wind_farm"
 id="Resource conflict - generic resolution" enabled="true"
 changed="2009-05-01T22:20:59">
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1="variable0" arg2="variable2" arg3="variable4"
               arg4="variable6">
        device_out(arg1,arg2,arg3,arg4)
      </trigger>
      <trigger arg1="variable1" arg2="variable3" arg3="variable5"
               arg4="variable7">
        device_out(arg1,arg2,arg3,arg4)
      </trigger>
    </triggers>
    <conditions>
      <and/>
      <conditions>
        <or/>
        <conditions>
          <and/>
          <condition>
            <value>variable0</value>
            <operator>eq</operator>
            <value>'get_log'</value>
          </condition>
          <condition>
            <value>variable1</value>
            <operator>eq</operator>
            <value>'sensor_check'</value>
          </condition>
        </conditions>
        <conditions>
          <and/>
          <condition>
            <value>variable0</value>
            <operator>eq</operator>
            <value>'sensor_check'</value>
          </condition>
          <condition>
            <value>variable1</value>
            <operator>eq</operator>
            <value>'get_log'</value>
          </condition>
        </conditions>
      </conditions>
      <condition>
```

```
        <value>':variable2,:variable4,:variable6'</value>
        <operator>eq</operator>
        <value>':variable3,:variable5,:variable7'</value>
      </condition>
    </conditions>
    <action>apply_newer</action>
  </policy_rule>
</resolution>
```

4.5 Prototype Policies for Sensor Networks

An XML wrapper is required for a call control prototype policy in the following form:

```
<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_document>
```

Because a prototype policy is instantiated as a regular policy, the relevant schema is that of the latter.

As an example of a prototype policy, the following aims to deal with a higher than normal frequency of vibration in a turbine blade. If this exceeds 20Hz, the turbine yaw angle is set to zero (i.e. the turbine is pointed directly into the wind) and the yaw brake is turned on (so that the turbine head cannot rotate about a vertical axis).

```
<prototype owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
 effect="turbine_yaw_angle = 0, turbine_yaw_status = 1"
 id="Cancel yaw on high blade vibration" enabled="true"
 changed="2009-03-25T21:20:59">
  <policy_rule>
    <trigger arg1="blade_vibration_frequency" arg2="turbine">
      device_in(arg1,arg2)
    </trigger>
    <condition>
      <parameter>message_values</parameter>
      <operator>ge</operator>
      <value>20</value>
    </condition>
    <actions>
      <and/>
      <action arg1="set_parameter" arg2="turbine" arg5="[yaw_angle,0]">
        device_out(arg1,arg2,,,arg5)
      </action>
      <action arg1="set_parameter" arg2="turbine" arg5="[yaw_brake,on]">
        device_out(arg1,arg2,,,arg5)
      </action>
    </actions>
  </policy_rule>
</prototype>
```

The abstract effect of this policy on controlled variables is to set the yaw angle to 0 and the yaw brake status to 1 (i.e. on).

4.6 Goals for Sensor Networks

An XML wrapper is required for a sensor network goal in the following form:

```
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel.xsd">
  ...
</policy_document>
```

As an example of a goal for sensor networks, the following always applies:

```
<goal owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
 id="Minimise component damage" enabled="true" changed="2009-03-25T22:44:59">
  <policy_rule>
    <action arg1="turbine_damage">minimise(arg1)</action>
  </policy_rule>
</goal>
```

This goal aims to minimise *turbine_damage*. This is a goal measure that is defined separately in terms of various controlled variables:

```
0.4×(
    +turbine_blade_amplitude +turbine_blade_pitch
    +turbine_blade_frequency +turbine_cable_twists
    +turbine_nacelle_amplitude +turbine_nacelle_frequency
    +turbine_rotor_speed
    -turbine_brake_lining -turbine_rotor_status
    -turbine_yaw_angle -turbine_yaw_status
)
```

Chapter 5

Home Care

5.1 Regular Policies for Home Care

5.1.1 Introduction

This section describes regular policies for the domain of home care. It discusses the triggers, conditions and actions used to specialise the language for this field.

For home care, regular policies are extended with domain-specific triggers, conditions and actions. The relationship among these is shown in figure 5.1. Core triggers (section 2.6), condition parameters (section 2.7) and actions (section 2.8) are also applicable. The combination rules are as follows:

- Internal triggers (marked † in figure 5.1) may be combined with any other triggers. Currently in home care at most one external trigger (no † in figure 5.1) is allowed.

Trigger	Parameters Established	Actions Permitted
internal†		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 trigger that may be combined with any other

Figure 5.1: Relationship between Home Care Triggers, Conditions and Actions

5.1.2 Common Parameters and Environment Variables

Regular policy conditions can use the following parameters: *date*, *day* and *time* (section 2.13.2). Regular policies for home care can also use a common set of parameters within triggers, conditions and actions. These parameters are outlined below together with examples of their use.

message_type: This identifies the type of trigger or action. The value will vary depending on the particular *entity_name*. Example trigger types might be *door_status*, *window_status* and *room_movement*. Example action names might be *off* and *on* for a home appliance. This parameter is mandatory.

entity_name: This identifies an external entity that may have policies associated with it. Example entity names might be *logging_service*, *door* and *window*. An entity name may be unnecessary if it is implied by the message type. If this parameter is omitted, its default is an empty string.

entity_instance: This identifies a particular instance of an *entity_name*. For example, an instance of entity *door* might be the identifier *front*, while an instance of a *window* might be *bedroom_1*. It is possible not all entities will have instances, such as a logging service for sensor data. If this parameter is omitted, its default is an empty string.

message_period: The time period to which a trigger or action refers. The period is 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). Due to transmission and processing delays, this time period is unlikely to be respected exactly. An absolute time reference spans a maximum of 24 hours. Suppose the time is currently 12:00:00. For a trigger, 11:00:00 means 'since 11AM today' and 13:00:00 means 'since 1PM yesterday'. For an action, 11:00:00 means 'at 11AM tomorrow' and 13:00:00 means 'at 1PM today'. If this parameter is omitted, its default is 'just occurred' for a trigger and 'now' for an action.

parameter_values: This is an open-ended string that may be a single value like 'log success' or a comma-separated list of values like '[20,30]'. To specify the parameters for an action, each value in the list may be name-value pair, such as *period=01:00:00*. The parameter list may also include sub-strings in brackets, such as *[motion,01/12/2007,12/12/2007],40*. If this parameter is omitted, its default is an empty list of values.

As far as the policy server is concerned, these parameters are simply uninterpreted strings whose meaning is given only by the external system. The following are concrete examples of what might be written:

X10: X10 (*http://www.x10community.com*) is a standard for controlling appliances connected to the electrical mains. X10 is mostly used for output (turning appliances on and off) though it can also be used to set an intermediate value (e.g. to dim a light). Some X10 input devices also exist (e.g. movement sensors). Parameters for an X10 device are as follows: *message_type* is 'on' or 'off'; *entity_name* is 'X10'; *entity_instance* is '<house code><unit code>' (e.g. 'B3'); *parameter_values* may carry an integer value (e.g. a dimming level 0..31 for devices that support this).

UPnP: UPnP (Universal Plug and Play, *http://www.upnp.org*) is a standard for controlling devices connected via a home network. UPnP is not (yet) common, but is intended for things like audio-visual devices (e.g. DVD players, TVs, surveillance cameras). Parameters for a UPnP device are as follows: *message_type* is a string meaningful to the device (e.g. 'play', 'change_channel' or 'view'); *entity_name* is UPnP; *entity_instance* is a user-friendly identifier for the device (e.g. 'DVD', 'TV', 'exterior_camera'); *parameter_values* may be set to a comma-separated list of values meaningful to that device in the form '*key*=*value*' (e.g. 'track=5', 'channel=BBC1', 'pan=20').

For a particular home, there may be other classes of devices such as 'Jini' (a distributed device architecture), 'Tunstall'(telecare devices) or 'Visonic' (wireless devices). The configuration, addressing and parameters of home care devices are outside the scope of the policy system.

The policy system establishes environment variables on the occurrence of each trigger. The parameters above double as environment variables. This is a flexible approach, allowing each parameter to appear explicitly in a trigger or action, as well as being defined implicitly as an environment variable. Environment variables may also be used to output parameter values literally in an action, such as inserting the *entity_instance* of a sensor in a warning message to care centre staff. The example policies in sections 5.2.2 and 5.2.3 demonstrate the use of environment variables.

5.1.3 Triggers

Regular policies for home care have triggers as follows. (The somewhat lengthy definition simply says that a mandatory *arg1* is followed by optional trailing arguments.) These are added to the regular policy triggers in section 2.13.1.

```
<xsd:simpleType name="trigger_domain">
  <xsd:restriction base="trigger_domain">
    <xsd:enumeration value="device_in(arg1,arg2,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_in(arg1,arg2,arg3,arg4)"/>
    <xsd:enumeration value="device_in(arg1,arg2,arg3)"/>
    <xsd:enumeration value="device_in(arg1,arg2)"/>
    <xsd:enumeration value="device_in(arg1)"/>
    <xsd:enumeration value="device_in(arg1,,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_in(arg1,,arg3,,arg5)"/>
    <xsd:enumeration value="device_in(arg1,,arg3,arg4)"/>
    <xsd:enumeration value="device_in(arg1,,arg3)"/>
    <xsd:enumeration value="device_in(arg1,,,arg4,arg5)"/>
    <xsd:enumeration value="device_in(arg1,,,arg4)"/>
    <xsd:enumeration value="device_in(arg1,,,,arg5)"/>
    <xsd:enumeration value="device_in(arg1,arg2,,arg4,arg5)"/>
    <xsd:enumeration value="device_in(arg1,arg2,,arg4)"/>
    <xsd:enumeration value="device_in(arg1,arg2,,,arg5)"/>
    <xsd:enumeration value="device_in(arg1,arg2,arg3,,arg5)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy triggers for home care are structured as follows:

device_in(type,entity,instance,period,parameters)**:** This arises from any external device. Only the *type* is mandatory; section 5.1.2 explains these values.

5.1.4 Conditions

Regular policies for home care have condition parameters as follows:

```
<xsd:simpleType name="parameter_domain">
  <xsd:restriction base="parameter_domain">
    <xsd:enumeration value="entity_instance"/>
    <xsd:enumeration value="entity_name"/>
    <xsd:enumeration value="message_type"/>
    <xsd:enumeration value="parameter_values"/>
    <xsd:enumeration value="message_period"/>
  </xsd:restriction>
</xsd:simpleType>
```

A detailed explanation of each parameter is given in section 5.1.2. The operators permitted for each parameter are listed in figure 5.2; the use of other operators will yield a result, but this may not be useful. Some special rules for values are as follows:

- For the *identifier* category, *in* and *out* are used with a comma-separated list of values to mean 'among' and 'not among'. That is, the parameter is checked for presence or absence in a list of values.
- For the *value* category, *in* and *out* are used with a comma-separated list of values to mean 'among' and 'not among'. That is, the parameter is checked for presence or absence in a list of values.

  In the case of date, day or time, a value may also be a range (e.g. 09:00:00..12:00:00 or 2005-02-03..2005-02-10).

  In the case of *parameter_values*, a comma-separated list is given in brackets. A list index is given in brackets, e.g. *parameter_values[2]* refers to the third element in the list. For convenience with single values, omitting the index implies an index of 0 (so *parameter_values* means *parameter_values[0]*).

The semantics of the condition operator depend on the types of its operands. Suppose the operator is *gt* is used with parameter operand *parameter_values[2]* and value operand '12'. If the parameter operand is '20' (i.e. is in numerical format), the condition will be treated as numerical comparison. If the parameter operand is 'on' (i.e. a string), the condition will be treated as string comparison.

5.1.5 Actions

Regular policies for home care have actions as follows. (The somewhat lengthy definition simply says that a mandatory *arg1* is followed by optional trailing arguments.) These are added to the regular policy actions in section 2.8. Actions are requests to the underlying communications layer. If the policy *preference* is *must_not*, an action argument can be omitted to mean any argument value.

```
<xsd:simpleType name="action_domain">
  <xsd:restriction base="action_domain">
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3)"/>
    <xsd:enumeration value="device_out(arg1,arg2)"/>
    <xsd:enumeration value="device_out(arg1)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,arg3)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,,arg5)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular policy actions for home care are structured as follows:

device_out(type,entity,instance,period,parameters)**:** This controls or configures an entity external to the policy system. Only the *type* is mandatory; section 5.1.2 explains these values.

5.2 Example Regular Policies for Home Care

The following home care policies illustrate the capabilities of the language. They should provide insight into the policy language for real examples in this domain. They demonstrate the possibilities of supporting old people in their everyday life through ICT (Information and Communications Technologies) to promote their independent living. Examples include policies to support users in medical care, home assistance, safety, communication and entertainment. Home care systems make use of situations detected by 'sensors' (including software components such as a reminder service), and interact with the user by controlling 'actuators' (including software components such as a speech synthesiser).

Each example policy is introduced briefly by its natural language meaning. This introduction also draws attention to some details of the policy language where appropriate.

An XML wrapper is required for a home care policy in the following 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_home.xsd">
  ...
</policy_document>
```

Category	Parameter	Operator
identifier	*entity_name*, *entity_instance*, *message_type*	*eq*, *ne*, *in*, *out*
value	*message_period*, *parameter_values*	*eq*, *ne*, *in*, *out*, *gt*, *ge*, *lt*, *le*

Figure 5.2: Common Parameter Operators

5.2.1 Night Wandering Reminder

Older people, especially those with dementia, are prone to waking in the middle of the night and leaving the house. The following policy advises the user to return to bed using synthesised speech (say, of a family member). It applies to domain *house3.match-project.org.uk*, i.e. house 3 in the trial area of the MATCH project.

When the front door is opened, if the time is 11PM–7AM and the main bed is unoccupied, the message 'It is night time – go back to bed' is synthesised on the hallway loudspeaker. To avoid having to hard-code the policy for the entity instance corresponding to the front door, the policy variable *front* is used instead. Similarly, *main_bed* is a policy variable that identifies the main bed occupancy sensor. These will be defined separately for the house (whether manually through the policy wizard or automatically through a configuration system). The trigger input carries a single value 'open'. The message period is omitted for both the trigger (i.e. 'trigger just occurred') and the action (i.e. 'perform action now').

```
<policy owner="admin" applies_to="@house3.match-project.org.uk"
 id="Night wandering reminder" enabled="true"
 changed="2009-05-01T22:12:00">
  <policy_rule>
    <trigger arg1="status" arg2="door" arg3=":front" arg5="open">
      device_in(arg1,arg2,arg3,,arg5)
    </trigger>
    <conditions>
      <and/>
      <condition>
        <parameter>time</parameter>
        <operator>in</operator>
        <value>23:00:00..07:00:00</value>
      </condition>
      <condition>
        <parameter>:main_bed</parameter>
        <operator>eq</operator>
        <value>'unoccupied'</value>
      </condition>
    </conditions>
    <action arg1="speak" arg2="loudspeaker"
            arg3=":hallway" arg5="It is night time - go back to bed">
      device_out(arg1,arg2,arg3,,arg5)
    </action>
  </policy_rule>
</policy>
```

5.2.2 Water Heating Control

The following policy deals with control of domestic water heating. Water heating is initiated at night to make use of off-peak electricity. Since the need for hot water is limited, this policy requests water heating only twice per week. At 11PM every day, the current day number is checked. On Wednesdays (day 3) and Saturdays (day 6), water heating is turned on (policy variable *immerser*) and an email message is sent to the householder (tom@house4.stirling.com) to report this. On other days, water heating is turned off (it may already have been off) and the householder is notified.

There is no explicit trigger in this example. Instead, a time trigger is inferred from the time condition. The water heater is presumed to be controlled by the X10 commands *activate* and *deactivate*.

```
<policy owner="admin" applies_to="@house4.stirling.com"
 id="Water heating control" enabled="true"
 changed="2009-05-01T22:12:00">
  <policy_rules>
    <sequential/>
    <policy_rule>
      <conditions>
        <and/>
         <condition>
```

```
            <parameter>time</parameter>
            <operator>eq</operator>
            <value>23:00:00</value>
          </condition>
          <condition>
            <parameter>day % 3</parameter>
            <operator>eq</operator>
            <value>0</value>
          </condition>
        </conditions>
        <actions>
          <and/>
          <action arg1="activate" arg2="X10" arg3=":immerser">
            device_out(arg1,arg2,arg3)
          </action>
          <action arg1="tom@house4.stirling.com"
                  arg2="Immerser turned on at :time :date">
            send_message(arg1,arg2)
          </action>
        </actions>
      </policy_rule>
      <policy_rule>
        <conditions>
          <and/>
          <condition>
            <parameter>time</parameter>
            <operator>eq</operator>
            <value>23:00:00</value>
          </condition>
          <condition>
            <parameter>day % 3</parameter>
            <operator>ne</operator>
            <value>0</value>
          </condition>
        </conditions>
        <actions>
          <and/>
          <action arg1="deactivate" arg2="X10" arg3=":immerser">
            device_out(arg1,arg2,arg3)
          </action>
          <action arg1="tom@house4.stirling.com"
                  arg2="Immerser turned off at :time :date">
            send_message(arg1,arg2)
          </action>
        </actions>
      </policy_rule>
    </policy_rules>
  </policy>
```

5.2.3 Log Activities

Logging everyday activity can be valuable to determine long-term trends. However, the nature of the logging should vary according to the circumstances and preferences of the individual. The following policy receives a logging request that specifies two lists of sensors: those whose logging should be activated (trigger *parameter_values[0]*, action *arg5*), and those whose logging should be deactivated (trigger *parameter_values[1]*, action *arg5*). These values are obtained implicitly from the trigger through the *parameter_values* environment variable.

```
<policy owner="admin" applies_to="@raploch.org.uk"
 id="Log activities" enabled="true"
 changed="2009-05-01T22:12:00">
  <policy_rule>
```

```
    <trigger arg1="logging_request">device_in(arg1)</trigger>
    <actions>
      <and/>
      <action arg1="activate" arg2="logger" arg5=":parameter_values[0]">
        device_out(arg1,arg2,,,arg5)
      </action>
      <action arg1="deactivate" arg2="logger" arg5=":parameter_values[1]">
        device_out(arg1,arg2,,,arg5)
      </action>
    </actions>
  </policy_rule>
</policy>
```

5.2.4 Night Security Alarm

The following policy activates an alarm and a camera (policy variables *alarm* and *camera*) for 20 minutes if there is movement in the hallway or lounge (policy variables *hallway* and *lounge*) between midnight and 6AM. Both the alarm and the camera are controlled via UPnP.

```
<policy owner="feng@stirling.org.uk"
 applies_to="@house4.cornton.stirling.org.uk"
 id="Night security alarm" enabled="true" changed="2009-05-01T22:12:00">
  <policy_rule>
    <triggers>
      <or/>
      <trigger arg1="movement" arg2="motion_sensor" arg3=":hallway">
        device_in(arg1,arg2,arg3)
      </trigger>
      <trigger arg1="movement" arg2="motion_sensor" arg3=":lounge">
        device_in(arg1,arg2,arg3)
      </trigger>
    </triggers>
    <condition>
      <parameter>time</parameter>
      <operator>in</operator>
      <value>00:00:00..06:00:00</value>
    </condition>
    <actions>
      <and/>
      <action arg1="activate" arg2="UPnP" arg3=":alarm" arg5="period=20">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
      <action arg1="activate" arg2="UPnP" arg3=":camera" arg5="period=20">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
    </actions>
  </policy_rule>
</policy>
```

5.2.5 Record TV Programme

Home care policies can assist the user in social communication and entertainment. It is presumed that a TV scheduler service can request that a programme be recorded. The first parameter of the request (trigger *parameter_values[0]*, action *arg4*) specifies the delay before the recording should begin. The second parameter of the request (trigger *parameter_values[1]*, action *arg5*) specifies the length of the recording. Recording is performed on a video cassette recorder (policy variable *VCR*).

```
<policy owner="gavin@cornton.com" applies_to="@flat2.cornton.com"
 id="Record TV programme" enabled="true"
 changed="2009-05-01T22:12:00">
```

```
<policy_rule>
  <trigger arg1="record_request" arg2="tv_scheduler">
    device_in(arg1,arg2)
  </trigger>
  <action arg1="activate" arg2="UPnP" arg3=":VCR"
          arg4=":parameter_values[0]" arg5="period=:parameter_values[1]">
    device_out(arg1,arg2,arg3,arg4,arg5)
  </action>
</policy_rule>
</policy>
```

5.3 Resolution Policies for Home Care

5.3.1 Introduction

This section is specific to resolution policies for home care. It discusses the triggers, conditions and actions that are used in this domain.

Resolution policies are extended with domain-specific triggers, conditions and actions. The relationship among these is much simpler than for regular policies. Any action may be associated with a trigger. The actual parameters of these actions must be literal values, or the values of *variable0* to *variable9* (if a trigger has bound them).

In general, conflicting situations may arise due to the *preference* attributes of policies being opposite, or due to the effects of policy actions clashing. As described in section 5.1.5, policies for home care have only one domain-specific action. Resolutions for this domain therefore deal with modality (preference) conflicts and parameter conflicts for this action.

5.3.2 Triggers

Resolution policies for home care have triggers as follows. These are added to the resolution policy triggers in section 2.14.1 and to the core triggers in section 2.6. Since resolutions are triggered by actions, resolution policy triggers are identical to regular policy actions for home care (see section 5.1.5).

```
<xsd:simpleType name="trigger_domain">
  <xsd:restriction base="trigger_domain">
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3)"/>
    <xsd:enumeration value="device_out(arg1,arg2)"/>
    <xsd:enumeration value="device_out(arg1)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,arg3)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,,arg5)"/>
  </xsd:restriction>
</xsd:simpleType>
```

5.3.3 Conditions

Resolution policies for home care have the same condition parameters as resolution policies in section 2.14.2.

5.3.4 Actions

Resolution policies for home care have actions as follows. These are added to the resolution policy actions in section 2.14.3 and to the core actions in section 2.8. These actions are those of regular policies for home care in section 5.1.5.

```
<xsd:simpleType name="action_domain">
  <xsd:restriction base="action_domain">
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3)"/>
    <xsd:enumeration value="device_out(arg1,arg2)"/>
    <xsd:enumeration value="device_out(arg1)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,arg3,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,arg3)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,arg4)"/>
    <xsd:enumeration value="device_out(arg1,arg2,,,arg5)"/>
    <xsd:enumeration value="device_out(arg1,arg2,arg3,,arg5)"/>
  </xsd:restriction>
</xsd:simpleType>
```

Further issues to investigate in home care conflicts include the following:

Conflicts of Multiple Stakeholders: In home care, multiple stakeholders are involved such as care centre staff, medical professionals, social workers, relatives, and the end users themselves. These will belong to different groups. The conflicts of multiple stakeholders need to be addressed in the home care systems. However, it is not yet clear whether the detection and resolution of these conflicts should be handled inside the policy system or not. If conflicts need to be addressed inside the policy system, detecting policy conflicts among multiple stakeholders is not much different from detecting other types of conflicts. However, resolving policy conflicts of multiple stakeholders remains an issue as there is no hierarchy among them.

Dependency among Situations: Raw sensor data can be used to infer high-level situations that trigger policies. Different high-level situations may be inferred from the same sensor data (e.g. the door being reported open might indicate normal or forced entry). How to detect and resolve conflicts of policies with interdependent situations needs to be investigated.

Conflicts among Time-Based Policies: Unlike triggers from sensors, time-based triggers are handled implicitly by the policy server or explicitly in policies. How to detect conflicts of policies with time-based triggers will also be considered in the future.

Addressing these issues requires a better understanding of home care systems. Once more definite examples in this new domain have been determined, generic resolution actions may be extended and more examples of resolution policies may be given.

5.4 Example Resolution Policies for Home Care

The following home care conflict policies demonstrate the capabilities of the language. They should provide an insight into the use of the policy language for real examples of handling conflicts in this domain.

Each example is introduced briefly by its natural language meaning. This introduction also draws attention to some details of the policy language where appropriate.

An XML wrapper is required for a home care policy in the following 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_home.xsd″>
  ...
</policy_document>
```

A resolution policy typically contains two triggers, each of which binds explicit *variable* values and implicit *preference* values. In general, there are four cases to be considered by a resolution depending on whether the variables are equal/unequal and whether the preferences are similar/opposite. This can often be simplified. In some cases it is sufficient to check for the equal/similar combination; the other combinations are handled by default. In other cases, the equal/opposite and unequal/similar combinations may be handled. In unusual cases, all four combinations may need individual treatment. Since the **else** operator can be used with only two alternative actions, multiple resolutions may therefore be required for the same trigger combination.

There are other variations on the basic case. For example, each trigger of a resolution policy may have up to five parameters, and more than two triggers may be present in a resolution policy. Policy conflicts in home care can be modality conflicts. They can also be one form of goal conflict, which occurs between the parameters of *device_out* actions. An example of modality conflict is given in section 5.4.1. Examples of generic and specific resolutions for conflicts among *device_out* parameters are given in sections 5.4.2 and 5.4.3. For generic resolution, the outcome is chosen from the set of conflicting actions. For specific resolution, any regular policy action can be specified as the outcome, and is not limited to the set of conflicting actions.

5.4.1 Messaging Conflict – Generic Resolution

The following example assumes a user has several policies regarding display of messages. In one policy, the user asks that messages be displayed on the TV. In another policy, the user states that this should not happen at inconvenient times (e.g. when family members are likely to be watching TV). Specifying these situations in several policies instead of one provides more flexibility, but can lead to conflict.

The following resolution policy checks for the same action (e.g. *display_message*), the same device (e.g. TV), and the same delay before displaying a message (e.g. 30 minutes). If the entity, instance and period are identical for both actions, but the preferences are opposite, the action with the stronger preference is applied. Other generic actions could include applying the weaker policy, the newer/older policy (based on definition date), or the policy belonging to the inferior/superior domain.

The same instance and period are important. Even if both actions concern the same entity, there may be no conflict if the instance differs. If the periods for applying the actions are different, then they *may* not conflict. This policy is limited in that it allows both messages to be displayed even if the periods differ by a small amount.

To avoid a tedious list of conditions on each individual action parameter, they are compared as a single list.

```
<resolution owner=″admin″ applies_to=″@sheltered.pendreich.gov.uk″
 id=″Messaging conflict - generic resolution″ enabled=″true″
 changed=″2009-05-01T22:12:00″>
  <policy_rule>
    <triggers>
      <and/>
      <trigger arg1=″variable0″ arg2=″variable2″ arg3=″variable4″
               arg4=″variable6″>
        device_out(arg1,arg2,arg3,arg4,arg5)
      </trigger>
      <trigger arg1=″variable1″ arg2= ″variable2″ arg3=″variable5″
               arg4=″variable7″>
        device_out(arg1,arg2,arg3,arg4,arg5)
      </trigger>
    </triggers>
    <conditions>
      <and/>
      <condition>
        <value>′:variable0,:variable2,:variable4′</value>
        <operator>eq</operator>
        <value>′:variable1,:variable3,:variable5′</value>
      </condition>
      <condition>
```

```
            <parameter>preference0</parameter>
            <operator>out</operator>
            <value>:preference1</value>
          </condition>
        </conditions>
        <action>apply_stronger</action>
      </policy_rule>
    </resolution>
```

5.4.2 Action Parameter Conflict – Generic Resolution

Home care systems can make use of speech synthesis to interact with the user. A speech synthesiser must not speak two different sentences at the same time. For two *speak* actions, the following resolution policy checks whether the action type, entity, instance and period are identical but the parameters (i.e. speech content) differ. If the *preference* values are different, the action with the stronger preference is selected.

The following resolution policy deals with only the case of unequal parameters and different preferences. Other cases will be handled by default, or may be defined in a more complex resolution policy.

```
<resolution owner="julia@stir.ac.uk" applies_to="@chalet9.campus.stir.ac.uk"
  id="Action parameter conflict - generic resolution" enabled="true"
  changed="2009-05-01T22:12:00">
  <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/>
      <conditions>
        <and/>
        <condition>
          <value>':variable0,:variable2,:variable4,:variable6'</value>
          <operator>eq</operator>
          <value>':variable1,:variable3,:variable5,:variable7'</value>
        </condition>
        <condition>
          <value>:variable8</value>
          <operator>ne</operator>
          <value>:variable9</value>
        </condition>
      </conditions>
      <condition>
        <parameter>preference0</parameter>
        <operator>ne</operator>
        <parameter>preference1</parameter>
      </condition>
    </conditions>
    <action>apply_stronger</action>
  </policy_rule>
</resolution>
```

5.4.3 Action Parameter Conflict – Specific Resolution

Like the example in section 5.4.1, the following deals with display of messages on a TV. It is assumed that only one message can be displayed at a time. There will be conflict if two actions try to display different messages at the same time.

The following resolution policy checks whether the two message requests are identical in terms of action type, entity and instance, and message period. If they are, but the message content differs, the resolution is to call the *schedule_display* service to manage the message display. This is provided with two message parameters: the message and the preference for displaying it (which affects the priority of the display).

```
<resolution owner="warden@glasgow.org" applies_to="@maryhill.glasgow.org"
  id="Action parameter conflict - specific resolution" enabled="true"
  changed="2009-05-01T22:12:00">
  <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(display_text,TV,arg3,arg4,arg5)
      </trigger>
    </triggers>
    <conditions>
      <and/>
      <condition>
        <value>':variable0,:variable2,:variable4,:variable6'</value>
        <operator>eq</operator>
        <value>':variable1,:variable3,:variable5,:variable7'</value>
      </condition>
      <condition>
        <value>:variable8</value>
        <operator>ne</operator>
        <value>:variable9</value>
      </condition>
    </conditions>
    <actions>
      <and/>
      <action arg1="schedule_display" arg2=":variable2" arg3=":variable4"
              arg4=":variable6" arg5=":variable8,:preference0">
        device_out(arg1,arg2,arg3,arg4,arg5)
      </action>
      <action arg1="schedule_display" arg2=":variable3" arg3=":variable5"
              arg4=":variable7" arg5=":variable9,:preference1">
        device_out(arg1,arg2,arg3,arg4,arg5)
      </action>
    </actions>
  </policy_rule>
</resolution>
```

5.5 Prototype Policies for Home Care

An XML wrapper is required for a home care prototype policy in the following form:

```
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel_regular_home.xsd">
  ...
</policy_document>
```

Because a prototype policy is instantiated as a regular policy, the relevant schema is that of the latter.

As an example of a prototype policy, the following aims to cool the household naturally. Suppose the indoor temperature exceeds 30°C and the outdoor temperature is below 25°C. The central heating and the air conditioning are turned off (in case they are on). Selected windows are then opened for one hour to give natural cooling. To avoid having to give specific addresses for these devices, they are identified indirectly through policy variables that are set to the actual addresses.

```
<prototype owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
  effect="home_indoor -= 4, home_security -= 3, home_pollen += 1.5, home_air += 1"
  id="Cool house naturally" enabled="true"
  changed="2009-01-16T17:45:00">
  <policy_rule>
    <trigger arg1="temperature_in">device_in(arg1)</trigger>
    <conditions>
      <and/>
      <condition>
        <parameter>:indoor_temperature</parameter>
        <operator>gt</operator>
        <value>30</value>
      </condition>
      <condition>
        <parameter>:outdoor_temperature</parameter>
        <operator>lt</operator>
        <value>25</value>
      </condition>
    </conditions>
    <actions>
      <and/>
      <action arg1="off" arg2="X10" arg3=":central_heating">
        device_out(arg1,arg2,arg3)
      </action>
      <actions>
        <and/>
        <action arg1="off" arg2="X10" arg3=":air_conditioning">
          device_out(arg1,arg2,arg3,,arg5)
        </action>
        <action arg1="open" arg2="X10" arg3=":windows" arg5="01:00:00">
          device_out(arg1,arg2,arg3,,arg5)
        </action>
      </actions>
    </actions>
  </policy_rule>
</prototype>
```

The abstract effect of this policy on controlled variables is to decrease the indoor temperature by 4°C, to decrease security by 3 , to increase home pollen levels by 1.5, and to improve home air quality by 1 (the latter three factors on a scale from 0 to 10).

5.6 Goals for Home Care

An XML wrapper is required for a home care goal in the following form:

```
<policy_document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://www.cs.stir.ac.uk/schemas/appel_regular_home.xsd">
  ...
</policy_document>
```

As an example of a goal for home care, the following applies between 11PM and 7AM:

```
<goal owner="admin@cs.stir.ac.uk" applies_to="@cs.stir.ac.uk"
 id="Minimise housing disturbance" enabled="true"
 changed="2009-01-16T17:45:00">
```

```
    <policy_rule>
      <condition>
        <parameter>time</parameter>
        <operator>in</operator>
        <value>23:00:00..07:00:00</value>
      </condition>
      <action arg1="housing_disturbance">minimise(arg1)</action>
    </policy_rule>
  </goal>
```

This goal aims to minimise *housing_disturbance*. This is a goal measure that is defined separately in terms of various controlled variables:

1.0×threshold(home_volume,70) + 1.0×home_night

Chapter 6

Conclusion

The concepts, syntax and semantics of APPEL have been seen. Regular policies deal with management of the system under control. Resolution policies deal with conflicts among the actions of regular policies. Prototype policies are special templates that are used to realise goals. Goals are supported by determining and instantiating the prototype polices that optimise goal measures. To allow for multiple goals, measures are combined into an overall evaluation function. Policy variables are named values that can be substituted into policies.

APPEL comprises a core language, plus extensions for regular and resolution policies in each domain. The language is defined through a collection of XML schemas. Ontologies supplement the schemas by specifying generic and domain-specific terms.

Although originally designed for call control, APPEL is intended to be a general-purpose language. Use in new domains is achieved by adding extension schemas and ontologies. The generality of the approach has been demonstrated through applications in call control/Internet telephony, sensor network/wind farm management, and home care/telecare management. It is therefore believed that APPEL is of general applicability. Further developments are anticipated in new domains.

The core language is believed to be stable, though some improvements can be expected in the area of goals and goal refinement. The language and its application are well supported by a set of advanced tools such as the policy server, policy wizard, conflict filter, and ontology server.

References

[1] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A system to specify and manage multi-policy access control models. In J. Bret Michael, Jorge Lobo, and Naranker Dulay, editors, *Proc. 3rd Int. Workshop on Policies for Distributed Systems and Networks*, pages 116–127. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

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

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

[4] Gavin A. Campbell. Overview of policy-based management using POPPET. Technical Report CSM-168, Department of Computing Science and Mathematics, University of Stirling, UK, June 2006.

[5] N. Damianou, Narankar Dulay, Emil C. Lupu, and Morris Sloman. Ponder: A language specifying security and management policies for distributed systems. Technical Report 2000/1, Imperial College, London, UK, 2000.

[6] Christos Efstratiou, Adrian Friday, Nigel Davies, and Keith Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In J. Bret Michael, Jorge Lobo, and Naranker Dulay, editors, *Proc. 3rd Int. Workshop on Policies for Distributed Systems and Networks*, pages 13–24. IEEE Computer Society, Los Alamitos, California, USA, June 2002.

[7] Stephan Reiff-Marganiec and Kenneth J. Turner. Use of logic to describe enhanced communications services. In Doron A. Peled and Moshe Y. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV)*, number 2529 in Lecture Notes in Computer Science, pages 130–145. Springer, Berlin, Germany, November 2002.

[8] Stephan Reiff-Marganiec and Kenneth J. Turner. A policy architecture for enhancing and controlling features. In Daniel Amyot and Luigi Logrippo, editors, *Proc. 7th Feature Interactions in Telecommunications and Software Systems*, pages 239–246. IOS Press, Amsterdam, Netherlands, June 2003.

[9] Stephan Reiff-Marganiec, Kenneth J. Turner, and Feng Wang. The ACCENT policy server. Technical Report CSM-164, Department of Computing Science and Mathematics, University of Stirling, UK, April 2009.

[10] Kenneth J. Turner and Gavin A. Campbell. The ACCENT policy wizard. Technical Report CSM-166, Department of Computing Science and Mathematics, University of Stirling, UK, April 2009.

[11] Kenneth J. Turner, Stephan Reiff-Marganiec, Lynne Blair, Jianxiong Pang, Tom Gray, Peter Perry, and Joe Ireland. Policy support for call control. *Computer Standards and Interfaces*, 28(6):635–649, June 2006.