Journal of Ambient Intelligence and Smart Environments 3 (2011) 83–110
IOS Press

Flexible Management of Smart Homes

Kenneth J.Turner *
Computing Science and Mathematics,
University of Stirling,
Stirling FK9 4LA,
UK

Abstract. An approach is presented for flexible management of smart homes, covering both home automation and telecare. The aim is to allow end users to manage their homes without requiring detailed technical knowledge or programming ability. This is achieved at three levels: managing home components and their interactions, stating policies for how the home system should react to events, and defining high-level goals for what the user wishes to achieve. The component architecture is based on OSGi (Open Services Gateway initiative). Policies and goals are formulated in the APPEL language (Adaptable and Programmable Policy Environment and Language), and supported by the ACCENT policy system (Advanced Component Control Enhancing Network Technologies). At run-time, high-level goals lead to selection of an optimal and conflict-free set of policies. These in turn determine how the home should react to various events. The paper closes with an evaluation of the approach from the points of view of functionality and usability.

Keywords: Component Framework, Home Automation, Goal Refinement, Open Services Gateway initiative, Policy-Based Management, Sensor Network, Telecare

1. Introduction

The goal of this work is to support flexible management of devices in the home, with applications in home automation and telecare. A framework is proposed for control over devices and services at multiple levels. The background to the work is given in areas such as home automation, telecare, component frameworks, and policy-based management.

1.1. Motivation

Home automation aims to let the user control a variety of devices around the home. Telecare aims to provide automated support to those receiving care at home. It is a contention of this paper that both aspects can be underpinned by a similar infrastructure, although the specific devices and services needed for each will vary.

Many commercial solutions exist for home automation. However, these generally operate at a fairly low level (being oriented towards controlling particular devices). Although there are standards for communicating with devices, there is little standardisation at a higher, service-oriented level. The situation for telecare is, if anything, worse. At present, telecare manufacturers usually follow their own proprietary approaches at both device and service levels. The functionality of home automation systems and telecare systems tends to be relatively fixed. Where alteration is possible, it usually requires specialist expertise and reprogramming. It can thus be difficult to achieve higher-level, flexible, user-oriented control over the home.

The motivation of this work is to make it easier to define how homes should react to user needs. This is realised at three levels:

– At component level, devices and services are integrated into a flexible component architecture that makes addition, change and removal easy. In addition, more complex services can be created from simpler devices and services. This is achieved through visual design of higher-level device services.

*E-mail: kjt@cs.stir.ac.uk

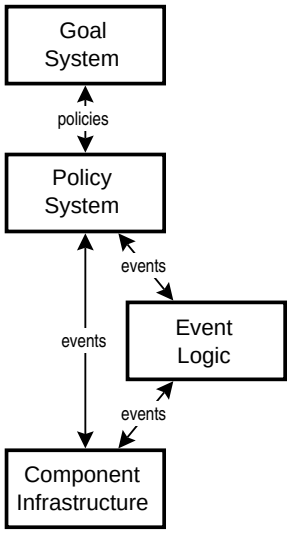

Fig. 1. System Architecture

– At policy level, rules can be defined for how the house should react to events. These policies are as device-independent and protocol-independent as possible. Policies are defined by wizards that are designed for non-technical users.
– At goal level, users can define high-level aims for how they wish their home to behave. This is the highest, user-oriented level of control.

These three different levels are targeted at different kinds of users. Although end users might define (and have been observed to define) simple device services, this level is intended more for use by the technically-minded (e.g. system integrators). Policies are intended for use by ordinary users, but mainly through simple parameterisation and extension of template policies provided in a library. It is believed that end users can make best use of goals as these are formulated in terms that users can relate to. The associated library of policies enables goals to be realised automatically without users having to deal with implementation details.

The paper gives a broad overview of the approach in the three areas of components, policies and goals. The aim is to present the key principles behind these. Although user interfaces are touched on, they are not the main focus of the work.

1.2. System Architecture

The high-level system architecture is shown in figure 1. This has the following layers:

Goal System: This is the primary level at which users are expected to interact with the system. Goals are high-level objectives such as staying comfortable in the home, remaining safe, or eating well. When external events happen, the goal system chooses an optimal set of policies to react to them.

Policy System: This is a secondary level at which users might interact with the system. However, it is expected that users will have limited involvement with policies such as simply filling in a few parameters (e.g. the number to call if the user falls). The policy system receives triggers (e.g. house temperature is low, back door is unlocked) and responds with actions dictated by the rules (e.g. turn heating on, lock back door).

Event Logic: Event logic can optionally be used to filter and manipulate triggers and actions. As an example of a synthetic trigger, a fall alert may not be reported unless the fall detector signals a possible fall and the user remains motionless. As an example of a synthetic action, a message to a user might be tried in several ways (as a text message, then as an email message).

Component Infrastructure: This includes all the home components along with the system infrastructure provided by the OSGi platform (Open Services Gateway initiative). Components may be low-level drivers (e.g. for appliance control) or higher-level functions (e.g. for weather forecasts). The infrastructure exchanges signals and messages with external devices (which may include external services).

The following high-level walk-through illustrates how all the elements of the system interwork. As an example of home automation, imagine that the main rooms have temperature sensors, and that there is control over window opening and air conditioning.

Suppose it is a warm summer's day and it is reported that the lounge temperature is 25°C. This will be sent via a wireless signal to the component that interfaces such sensors. This will then be converted into an OSGi event with associated parameters that identify the message (reading, temperature, lounge, 25).

Since temperatures will drift up and down due to external factors such as incoming sunlight, suppose that the developer has decided to intercept temperature readings and check for longer-term trends. The temperature message is then picked up by event logic that checks the temperature against values in the last ten

minutes. If this indicates a steady rise in temperature, the logic will report an over-temperature event.

The policy system continually monitors incoming events. If no policy matches a temperature reading, this will have been ignored. However, suppose several policies have over-temperature triggers. Policy 1 that opens the windows for ventilation has the effect of letting in outside air (and hence pollen), increasing the noise level from outside, and reducing household security. Policy 2 that switches on the air conditioning has the effect of increasing power consumption. Policy 3 that partly closes the curtains to shade the room may have no particular effect besides cooling the room.

These policies are then passed for consideration to the goal system. The user might have defined goal 1 for staying comfortable (measured by room temperature and noise level), goal 2 for saving energy (measured by power usage), and goal 3 for keeping healthy (measured by avoiding allergens and eating properly).

The three policies that cool the room are then considered against these goals. Opening the windows is positive room for the temperature aspect of the comfort goal, negative for the noise aspect of this goal, but barely affects the energy goal. Using air conditioning improves temperature, but seriously affects the energy goal by using a lot of power. Drawing the curtains also improves temperature, but barely affects the energy goal. An optimal choice of policies is then made according to current circumstances and goal weights. This might select the policies that open the windows and draw the curtains.

The selected policies are then returned to the policy system for execution. Conflict analysis will find their actions cannot both be performed since opening the windows will obstruct the curtains. Conflict resolution might then select opening the windows. Finally, this action is sent via the component infrastructure to the window opener. The entire sequence of steps, from temperature reading to window opening, will typically take a second or so. In other circumstances, or with changed goal weights, the outcome could be different.

1.3. Related Work

1.3.1. Home Automation

Device protocols underlie home automation. Many standards have evolved in this area, such as:

- IR (Infrared, *www.irda.org*) is used to control many kinds of domestic appliances.
- KNX (Konnex Association, *www.knx.org*) derives from earlier work on EIB (European Installation Bus). It supports a variety of media and devices. KNX is widely used in building management, including domestic applications.
- Lonworks (Echelon Corporation, *www.echelon.com*) is a set of networking standards that have been used in applications such as building management, home automation and transportation.
- UPnP (Universal Plug and Play, *www.upnp.org*) is an extension of the plug-and-play concept into the world of networked devices.
- X10 (*www.x10europe.com*) is widely used to control appliances using existing mains cabling. There is also some support for wireless control.

The work in this paper aims to be independent of particular devices and protocols. Device-level communication is essentially a solved problem. A harder question is how to abstract from device details and present a uniform interface to higher levels of control.

Many commercial packages support home automation, such as the following:

- Control4 (*www.control4.com*) offers a framework for tightly integrating third-party devices. The approach encourages third parties to make their devices compatible with Control4, and so moves the effort to the device suppliers.
- Cortexa (*www.cortexa.com*) is a sophisticated package for home automation that supports many kinds of devices. Although it can be programmed using various kinds of rule editors, these are not flexible and simple enough for users to have full control over their homes.
- Girder (*www.promixis.com*) aims to flexibly manage a variety of home devices by defining the mapping of input events to output events. This requires specialist technical knowledge and so is not appropriate for direct use by ordinary users.
- HAI (Home Automation Inc., *www.homeauto.com*) offers a home automation system that addresses control, safety, entertainment and energy management. However, this is more oriented towards installation experts than for programming by home users.
- HomeSeer (*www.homeseer.com*) is widely used to control a variety of home devices. It has particular strengths in allowing control via remote interfaces such as PCs and mobile telephones.

The work in this paper does not aim to compete with well-established commercial solutions. Rather it focuses on new techniques such as a flexible component framework and user-oriented control over the home.

Location-based services are becoming more popular. In the main, these are being designed for mobile users. However, some have been applied in office buildings through use of active badges [20] and 'sentient computing' [1]. In future, location-based services for the home may become more popular (e.g. to allow media to be played wherever the user is).

Interplay [30] aims to provide high-level control over the home using pseudo-English. This is supported by seamless device interconnection over a UPnP network. However, like other systems such as OSCAR [33], Interplay is focused on handling media.

1.3.2. Telecare

The world population is gradually ageing, with the percentage of older people (over 65) expected to rise by 2050 to 19.3% worldwide, and much higher in some countries [16]. Although the population is ageing, older people are generally healthier and more active than in previous generations. It is beneficial for older people to live independently in their own homes as long as possible. The impracticability and cost of providing sufficient care homes also makes this a necessity. This situation has been recognised by governments in many countries, where national programmes are in place to promote the use of telecare.

Telecare systems are computer-based systems that support delivery of care to the home. They can give the user advice, identify situations that may need intervention, reassure family members and informal carers, and relieve professional carers of low-level monitoring tasks. Telecare systems should be appropriate (for different stakeholder viewpoints), customisable (for specific user needs), flexible (offering a range of solutions), and adaptive (as care needs and conditions evolve). Companies involved in developing telecare solutions include Cisco, General Electric, Initial, Intel, Philips and Tunstall.

Unfortunately, most telecare solutions are proprietary and relatively fixed. To change functionality usually requires specialised technical knowledge and reprogramming. Telecare standards are also in their infancy due to the relative newness of the approach. Ongoing work on standardisation includes the following:

- Continua (the Continua Health Alliance, *www.continuaalliance.org*) is working towards standards for interoperability of healthcare systems, including those used at home (normally called telehealth). However, telecare includes important elements of social care as well.
- ETSI (the European Telecommunications Institute, e.g. Special Task Force 264, *portal.etsi.org/stfs/STF_HomePages/STF264/STF264.asp*) has been working towards telecare standards.
- TSA (the Telecare Services Association, *www.telecare.org.uk*) is working towards telecare standards, though these tend to emphasise procedural rather than technological aspects.

Because telecare is an emerging discipline, the work in this paper has the potential to have more impact. In particular, the paper's emphasis on flexible and adaptable control is particularly relevant. Since telecare devices currently follow proprietary protocols, it is necessary to manage these in a device-independent way.

1.3.3. Component Frameworks

A variety of frameworks have been developed to allow flexible combination of components. Approaches include the following:

- ADLs (Architecture Description Languages [29]) focus on how complex systems are put together out of their component parts. For example, the pipe-and-filter approach is widely used to create chains of interacting components.
- Atlas started as an academic project to support sensor-actuator networks in a plug-and-play, service-oriented manner [22], but is now available commercially. Among other applications, the Atlas middleware has been used for smart homes and healthcare.
- EasyLiving [8] is designed to support intelligent environments through dynamic interconnection of a variety of devices. This middleware supports mechanisms such as inter-system communication, location tracking for objects and people, and visual perception.
- Gaia [36] offers a component architecture based on CORBA (Common Object Request Broker Architecture). This is used to support 'active spaces' that rely heavily on contextual information including presence.
- Jini (*www.jini.org*) extends Java for distributed and federated systems. The architecture supports service components, service registration, access control, distributed events and transactions. Although a general-purpose approach, Jini has found use in the design of home systems [34].

– OSGi (originally Open Services Gateway initiative, *www.osgi.org*) provides a framework within which components called bundles reside. The overall framework supports common functions such as a service registry, lifecycle management and event administration.
– Patch Panel [4] aims to solve component interconnection by interposing intermediate logic that maps between devices. This logic is expressed by state machines that map events from one device onto actions at another device.
– SCA (Service Component Architecture, *www.osoa.org*) is a service-oriented approach that allows flexible and implementation-independent 'wiring up' of components.
– SOA (Service Oriented Architecture [26]) treats components as loosely coupled services. Components are typically realised as web services.
– SODA (Service Oriented Device Architecture, *www.eclipse.org/ohf/components/soda*) is an approach to treat device interfaces in a service-oriented way. The Service Activator Toolkit and Development Kit ease the process of developing OSGi components for devices.
– Speakeasy [14] supports flexible communication among devices without *a priori* knowledge of each other – so-called recombinant computing. Two key principles are the use of well-defined and generic interfaces, coupled with the possibility of exchanging mobile code to extend these.

The work in this paper incorporates several aspects of the approaches above. OSGi is used as the framework within which services live. The approach reflects the principles of SOA. Although SCA has been evaluated as a flexible way of interconnecting home components [28], OSGi remains the main focus. As discussed in section 2.3, the architectural concept of filter is supported within the approach. A home system is likely to be tightly integrated such that the flexibility of, say, Patch Panel or Speakeasy is unnecessary. In addition, neither of these offers user programmability.

OSGi was originally conceived for use in a home environment, so it is hardly surprising that it is suitable for the author's purpose. However, several researchers have sought to extend its applicability. [25] enriches the SODA approach by dealing with data semantics. [17] focuses on self-configuration of home devices and personalisation of services offered to the user.

1.3.4. Policy-Based Management

Policies have been used in applications such as access control, network/system management, and quality of service. Policies are rules that are automatically applied when events occur. Most policy languages are in ECA form (Event, Condition, Action). Examples, drawn from a large field, include the following:

– Drools (*www.jboss.org/drools*) is an approach for enforcing business rules. This is an implementation of the Sun community standard for a Java Rules Engine (JSR 94). The rule language does not make a sharp distinction between events and conditions. Its focus is more on business logic than on system control.
– Police [13] follows a traditional policy approach. It deals particularly with management and conflict in a distributed setting. The approach avoids modality conflicts, but provides mechanisms for handling domain-specific conflicts.
– Ponder [11] is a well-known policy approach. It offers a mature methodology for handling policies in applications such as system management and sensor networks. Ponder supports policy domains, policy conflicts, and policy refinement.

For the work in this paper, the human aspects of home care and home automation tend to rule out the more technically-oriented policy approaches used in system management. As argued in [43], a different kind of policy approach is needed for 'softer' management tasks of the kind found in human-oriented systems (e.g. telecare).

[24] describes a rule-based system for smart homes. However, this is a rather heavyweight solution that expects home devices to be interconnected via an Ethernet. Although the system supports basic ECA rules, these do not seem to be for definition by end users.

Although it would not be regarded as a policy system, Gadgetware [21] aims to achieve a similar effect. Physical objects are given a digital representation as eGadgets with 'plugs' that make their capabilities available. eGadgets can then be connected via 'synapses' that allow their functionality to be chained. This can achieve some of the effects of a home policy system (e.g. turning on a desk lamp when someone sits down to read a book). However, a full policy system is capable of a much wider variety of control.

Context-aware systems (e.g. [6]) aim to make a system reactive to context, and in that sense have some affinity to policy systems. Gaia [36] aims to create 'active spaces' from physical spaces supplemented by a

context-aware infrastructure. However, context awareness is a separable aspect. Indeed as will be seen, the policy system described in this paper accepts information from an external context system in order to influence its behaviour. Any third-party system could be used to provide this information.

This paper follows the approach of ACCENT (Advanced Component Control Enhancing Network Technologies, *www.cs.stir.ac.uk/accent*). This is an approach and set of tools for managing systems through policies. Originally developed for controlling Internet telephony, ACCENT and its accompanying policy language APPEL (Adaptable and Programmable Policy Environment and Language, *www.cs.stir.ac.uk/appel*) have now been extended into new domains such as home management, sensor networks and wind farms.

1.3.5. Goal Refinement

Goal refinement has been investigated for many years. In artificial intelligence, for example, planning approaches such as STRIPS (Stanford Research Institute Problem Solver) go back about 40 years. More recent work includes the following:

- Agent systems often follow a goal-based approach. As an example, 3APL (Agent Programming Language, *www.cs.uu.nl/3apl*) defines goals and beliefs. Plans are created from predefined rules, using an action base to achieve goals.
- Goal refinement has been treated from a logic perspective by several researchers. For example, [5] uses event calculus for formal refinement of goals into system operations that achieve them.
- Requirements engineering has also made use of goal concepts. Approaches such as KAOS (originally Knowledge Acquisition in Automated Specification [44]) make use of refinement patterns to decompose goals into subgoals. [37] uses temporal logic in the refinement of goals into subgoals, and then subgoals into policies.
- A few examples exist of goal refinement in telephony. URN (User Requirements Notation [2]) allows system goals to be expressed, and to be related how the system is designed. Goals for telephony are investigated in [19], which also addresses the potential for conflicts among goals.

The work in this paper has a deliberately pragmatic philosophy. Logic-based approaches require specialised expertise, can involve lengthy computations that make them unsuitable for real-time use, and work best offline (i.e. statically).

1.3.6. User Interfaces

User interface design is a broad and intensively researched area. The following gives a brief overview with particular regard to allowing end users to program the home. [32] is a classic overview of the area.

- Programming by demonstration has been advocated as a suitable approach. For example, a CAPpella [12] allows a situation to be set up and then an appropriate response to be demonstrated. Although users like this way of defining rules, it can be tedious or infeasible to create suitable demonstrations for all possible situations or for rare events. Alfred [15] uses demonstration to capture macro-like rules, making particular use of speech interaction.
- Several projects have investigated the use of tangible programming. ACCORD [35] uses jigsaw-like pieces to assemble rules. Although the approach ensures that the rules are meaningful, the range of possible rules is very restrictive. As noted by the developers of CAMP [39], this is not a natural way for users to define how their home should react. Instead, CAMP allows end users to define their requirements using words drawn from a 'magnetic poetry' set. This requires natural language processing – a major challenge that is alleviated by restricting the concepts that can be expressed. The approach has also been validated only on capture applications (for audio/video). Media Cubes [7] have been used to define rules by placing action requests next to devices (read from the cube faces using infrared). Again, this is a device-oriented approach.
- Visual programming languages have been developed for home control. [23] describes an approach for ubiquitous computing environments that allows end users to define rules graphically. The system is, however, very constrained in what users can say. iCAP [38] permits new devices to be added through the user drawing icons for them. These icons can then be dragged onto a situation window (thus identifying the conditions of a rule) or onto an an action window (thus identifying the desired system output). A useful feature is the ability to simulate and thus predict the effect of rules. OSCAR [33] provides a visual environment for selecting and interconnecting components. However, this has so far focused almost exclusively on handling media in the home (audio, graphics, video).

Nearly all of these approaches are strongly device-oriented and rule-oriented. This has been criticised (e.g. by [39]) as not being particularly natural for end users. Instead, it has been argued that high-level goals should be used. ACHE [31] is closer in philosophy to the work reported in this paper. However, the goals supported by ACHE are very restricted (user comfort and cost) and the emphasis is on the system learning how best to meet these goals.

1.4. Structure of The Paper

Section 2 discusses the architecture and components of the home system, including how events are handled. Section 3 introduces the policy system that allows user-defined control without specialised technical knowledge or programming. Section 4 discusses how home users can define high-level goals rather than low-level policies. Section 5 evaluates the functionality and usability of the approach. Section 6 rounds off the paper with a summary of the work.

2. Component Level

The architecture and components of the home system are presented. Events are sent between 'devices' of all kinds (including services) and the policy server that handles high-level control of the house. An event transformer is used for flexible mapping between device events using visually-defined logic.

2.1. Component System Architecture

The basic architecture followed in this work is that of OSGi. Components are therefore OSGi bundles that register services (e.g. for control of devices). This service-oriented approach makes it easy for components to use other components in a loosely coupled way. The home services could, in principle, call each other directly. However, they are designed to communicate via an event bus (mediated by the OSGi Event Admin service). This further decouples components, allowing them to register only for events they are interested in.

The high-level component architecture is shown in figure 2. The home components are generically called 'devices', though this covers a variety of functions. Devices that provide inputs would conventionally be called sensors (e.g. medicine dispensers, motion detectors, video cameras). Devices that act on outputs would conventionally be called actuators (e.g. door locks, gas shut-off valves, video recorders). In a domestic setting, the term 'appliance' would also be used (e.g. CD player, microwave oven, TV). More significantly, 'devices' can also be software services (e.g. data logging, text messaging, weather forecasting).

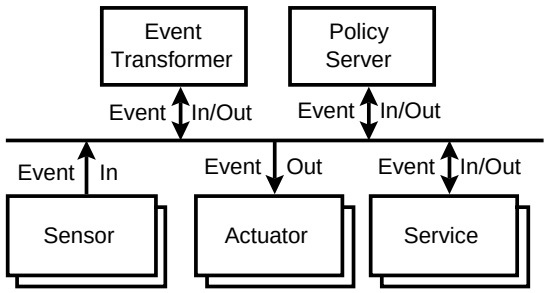

Fig. 2. Component Architecture

Two components are distinguished in the architecture: the event transformer (low-level services that transform device events, see section 2.3) and the policy server (high-level services that manage the home, see sections 3 and 4).

Device input and output events (*device_in*, *device_out*) have a uniform structure with the following fields (identified by argument number when used by the policy system):

message type (arg1): the type of device input or output (e.g. *activity*, *on*, *open*, *reading*).

entity name (arg2): an optional entity associated with a device message (e.g. *door*, *message*, *motion*). Some messages imply a unique entity (e.g. a *log* message implies a system-wide logger), so the entity name can be omitted.

entity instance (arg3): an optional instance of an entity associated with a device message (e.g. *front*, *email*, *hall*). Some entities may have only a single instance (e.g. a central heating system), so the entity instance can be omitted. Entity instances may also identify groups (e.g. all digital TVs, all upstairs windows)

message period (arg4): an optional interval or time to which an event applies. For example, a temperature input might have period *15* if it was measured during the last 15 minutes, or *21:30* if it was measured at 9.30PM. For output, the same values could be used to start a video recording in 15 minutes or at 9.30PM. The period is normally omitted, meaning 'now'.

parameter values (arg5): an optional device parameter list. For example, this might give the reading for a temperature input, or the dimming percentage for a light output. The parameter values may be omitted if not relevant to the event.

Components are interoperable at the event and service level, using common OSGi mechanisms. This requires the components to respect certain interface standards, notably the use of device event information as above. The standard OSGi service registry is somewhat limited, e.g. components can be discovered only by name (though abbreviation is possible). For use in the home, more comprehensive and semantically based techniques are needed.

A set of modular ontologies has been created for describing concepts, terms and their relationship in a home context [42]. The base ontology specifies basic concepts in the home environment such as device, service and user. Each of these is then elaborated in a core ontology that relates the basic concepts and defines attributes. Generic ontologies define the concepts and relationships needed for rich service descriptions. This allows inferences to be drawn automatically, e.g. a hall lamp is a kind of light, offers a lighting service, and is located near the front door. Protocol ontologies add the specific characteristics of protocols such as UPnP and X10. Finally, a home ontology describes networked components used in home automation and telecare.

This collection of ontologies supports semantically based discovery of components. The approach can discover what components do and also how to use them. Suppose a component needs to increase the lounge lighting level. A query for this function will cause the component registry to reason using its ontological knowledge. This might indicate use of an X10 component, identifying a particular X10 address and that this lamp can be dimmed as well as turned on or off.

2.2. Home Components

A variety of components have been developed for use in home automation and telecare. Examples in various categories are as follows:

appliances: wired and wireless equipment in the home. Examples of appliances controlled via the mains are: fan, heating, light, oven. Examples of appliances controlled via infrared are: air conditioning, camera, DVD and TV.

communication: services for communication. Examples of services are: email, message display, SMS (Short Message Service), and speech input/output (using code from the University of Edinburgh).

environment: wireless devices used for environment information. Supported sensors include those from Oregon Scientific (*www.oregonscientific.com*): humidity, temperature and forecast.

interfaces: user-friendly interfaces (not the traditional keyboard, mouse, monitor). Touch screens are used for information display and for easy user interaction (e.g. appointment reminders, house control, managing media albums). Various 'Internet buddies' are used as they appeal to non-technical users: the i-Buddy 'angel' (*www.unioncreations.com*), the Nabaztag 'rabbit' (*www.nabaztag.com*), and the Tux Droid 'penguin' (*www.ksyoh.com*). The WiiMote (*www.nintendo.com*) is used for gestures, light patterns and sound. The SHAKE (Sensing Hardware Accessory for Kinaesthetic Expression, *www.dcs.gla.ac.uk/research/shake*) can communicate using gestures and tactile output (using code from the University of Glasgow).

security: wireless devices used for general security. Sensors supported include those from Tunstall (*www.tunstallhealth.com*) and Visonic (*www.visonic.com*): alarm (pendant, wrist), movement detector, pressure mat (bed, chair, floor), reed switch (cupboard, door, window), RFID reader (active badge), and other detectors (flood, gas, smoke). Actuators supported include those from Tunstall: curtain controller (open, shut) and door controller (lock, unlock).

telecare: wireless devices used specifically in telecare. Sensors supported include those from Tunstall: enuresis detectors (bed wetting), epilepsy detectors, fall detectors, and medicine dispensers (which report if medicine is not taken).

2.3. Event Transformation

In normal operation, input devices cause events that trigger the policy server. This results in actions that are sent to devices via output events. A home automation example might be: 'when the front door is opened, turn on the hall and lounge lights, play the user's favourite music, and activate climate control'. A telecare example might be: 'when the user is late in taking medicine, issue a spoken reminder in the relevant room; if the user still does not take the medicine, alert a neighbour by text message'.

However, greater flexibility is desirable in handling device events. The event transformer can filter and modify events before they are seen by the policy server. Policies can be written to use the raw device events or the ones created by the event transformer. Possible patterns for event transformation are as follows. (It is also possible for the triggering event to lead to nothing, meaning that the event should be ignored.)

in → in: an input event is mapped to input events. This is normally called sensor (data) fusion, the idea being that raw input from several sensors can be combined to produce higher-level, more meaningful events. For example, a more accurate prediction of falls might be obtained by combining fall detector data with movement detector data. If it is reported that the user has fallen and there is no movement within the next minute, a fall alert can be generated. This allows for synthetic input events, e.g. a macro trigger that reports someone has entered the house. This might build on raw sensor inputs that the front door was opened and there is movement in the porch.

out → out: an output event is mapped to output events. This allows for synthetic actions, e.g. a macro action for contacting someone. This might first trying calling the user's mobile phone. If the call is not answered within 10 seconds, a text message is sent.

in → out: an input event is mapped to output events. This supports low-level, device-oriented services that do not require policies (as opposed to the high-level, user-oriented services supported by the policy server). For example, if the lounge is entered, the system should turn on the light if it is dark and set the room temperature.

out → in: an output event is mapped to input events. This allows the policy server to trigger the execution of further policies. Suppose the lounge light is turned on under policy control because it has become dark outside. The action to turn on the light can become a trigger that the lounge is now brighter. Other policies might react to this by increasing TV brightness and closing the curtains.

Event transformations could be coded in a conventional programming language. However, this would negate the goal of making it possible to change system functionality without detailed technical knowledge and reprogramming. Event transformations are therefore described in a visual design language (a simple form of programming). This is an adaptation of CRESS (Communication Representation Employing Systematic Specification, *www.cs.stir.ac.uk/~kjt/research/cress.html*).

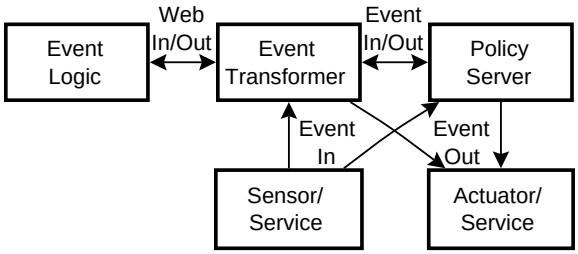

Fig. 3. Event Transformer Role

CRESS is a graphical notation and toolset for designing service flows (e.g. for grid, voice or web services). A root diagram describes a basic service. This may be extended by feature diagrams that automatically add capabilities to the basic service. In a home context, the diagrams describe event logic. A CRESS compiler automatically converts these diagrams into BPEL (Business Process Execution Language [3]) and deploys them into a BPEL engine (ActiveBPEL, *www.activebpel.org*).

Home components have an OSGi event interface, whereas BPEL processes have a web service interface. The event transformer therefore maps bidirectionally between OSGi events and web service calls. As a more precise description of the system architecture in figure 2, figure 3 shows the relationship among the various components. Here, the event logic consists of BPEL processes created from CRESS diagrams.

This approach also offers a beneficial capability: devices can be handled by any external web service. This allows remote entities (e.g. a mobile phone or a PC) to control the home, and to receive information about significant home events (e.g. a low-temperature alarm or an intruder alert). Exposing home control to external entities is, of course, a security risk so authentication is used for web services.

For space reasons, only two small examples of the approach will be given; see [**?**] for a more extensive set of examples. A CRESS diagram contains numbered activities in ellipses. These are linked by arcs that can be governed by value conditions or event conditions. CRESS offers a complete methodology for service creation, including automated specification, verification, validation, implementation and performance analysis.

Figure 4 shows the logic for the in → out example mentioned above. Node 1 shows reception of a

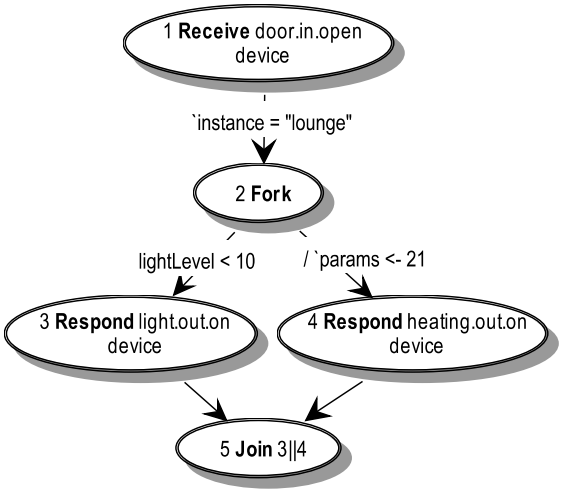

Fig. 4. Lounge Control

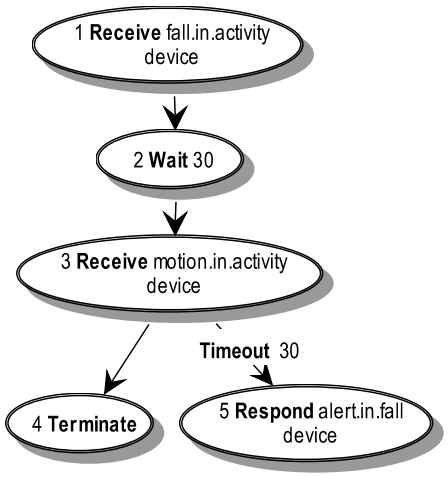

Fig. 5. Fall Detection

door open message: *Receive* means message input, *door.in.open* is in the form *device.direction.operation*, and *device* is a message variable. Such a variable carries the instance, period and parameter fields described for events in section 2.1. The fields of this message can be used in later activities. As this is common in diagrams, a shorthand notation is provided: *`instance*, for example, is short for *device.instance*.

If the door message is from the lounge (arc from node 1 to 2), two paths are followed in parallel (node 2). On one path, if the light level is less than 10 lux (arc from node 2 to 3) then the lounge light is turned on (node 3). On the other path, the device parameter is set to 21°C (arc from node 2 to 4) then the lounge heating is turned on with this setting. Both paths then join (node 5). Sophisticated control can be exerted over concurrency by specifying which activities must succeed. '3 || 4' means that either or both of the prior paths must be successfully executed.

Figure 5 shows the logic for the in → in example mentioned above. Node 1 receives a fall detector message. After 30 seconds (node 2), a motion input is then awaited (node 3). If this occurs, it is assumed that the user has not had a serious fall so the logic terminates (node 4). If there is no input after 30 seconds (arc from node 3 to 5), a fall alert event is generated.

3. Policies

The policy system allows user-defined control of the home without requiring specialised technical knowledge or programming. The structure of a regular policy is explained. Resolution policies are also introduced as a means of handling conflicts among policy actions.

3.1. Policy System Architecture

The policy system is called ACCENT (Advanced Component Control Enhancing Network Technologies, *www.cs.stir.ac.uk/accent*). This supports a policy language called APPEL (Adaptable and Programmable Policy Environment and Language, *www.cs.stir.ac.uk/appel*). The foundational work on ACCENT for policies in Internet telephony is described in [43].

The policy system architecture is shown in figure 6. From its beginnings in Internet telephony, the system has been extended in a number of ways to support home control: extensions to the policy language, support of goals, and detection of policy conflicts. Although the internal structure of the policy system is complex, the user sees the outer dashed box in figure 6 as a single entity.

For historical reasons, and because the system mostly handles policies, a number of components are labelled 'policy'. However, apart from regular policies, the system also deals with goals, prototype policies (that achieve goals), resolution policies (to handle conflicts), and policy variables (used in goals and policies). System elements have the following functions:

managed system: the home system under control.

policy wizard: a user-friendly interface for defining and editing goals and policies. A web-based wizard allows policies to be reviewed and edited using stylised natural language. However, other wizards have been developed to make policies

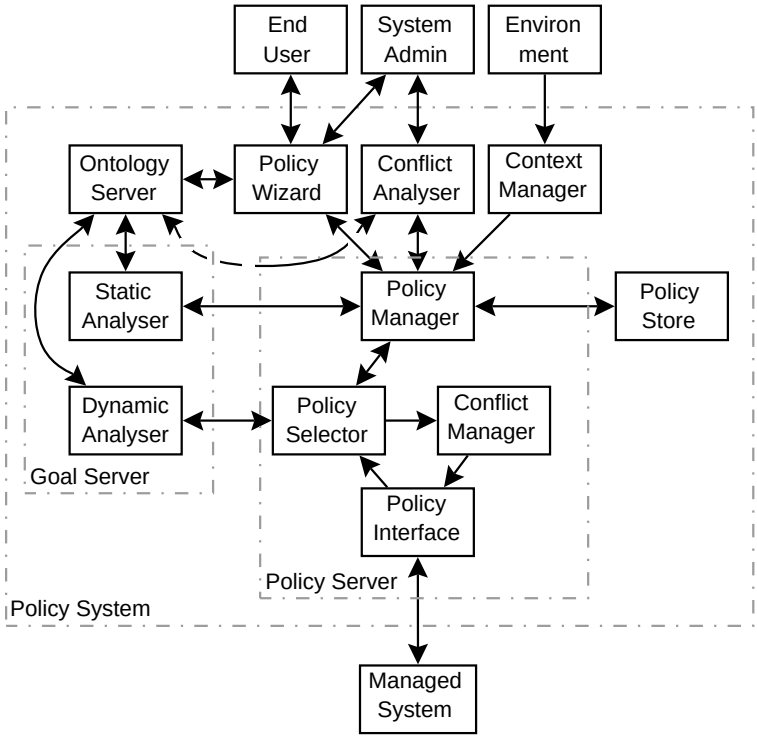

Fig. 6. Policy System Architecture

easier to use by non-technical people. One is voice-based (VoiceXML [45]), another uses digital pen and paper (Anoto) to define policies through simple forms.

context manager: an interface for providing additional information about the managed system (e.g. user diary or household configuration).

policy server: the heart of the policy system. The policy manager is the interface to the policy store, isolating the rest of the system from the particular choice of database. It receives new or updated goals and policies from the policy wizard, and also contextual information from the context manager. When goals or prototype policies are modified, the static analyser is notified. This may result in changes to the generated policies. The policy manager is also asked to query the policy store when event triggers are received. These arrive from the managed system and are passed to the policy selector. This chooses relevant policies (i.e. those associated with this trigger and whose conditions are met). If any triggered policies are derived from goals, the dynamic analyser is notified. This produces an optimal set of policies that are submitted to the conflict manager. Conflicts among policy actions are detected and resolved. Finally, an optimal and compatible set of actions is sent to the managed system.

policy store: an XML database that stores information about goals and policies.

conflict analyser: a tool to analyse policies offline for conflict-prone interactions [10].

ontology server: a generic interface to ontology information about each application domain [9]. A domain-specific ontology is used by the policy wizard to define valid goals and policies. An ontology is also used by the offline conflict analyser and by the goal system.

goal server: the heart of the goal system. The static analyser is invoked when goals or prototype policies are added, modified or deleted (see section 4.3). The dynamic analyser is invoked when selecting goal-derived policies (see section 4.4).

3.2. Regular Policies

Regular policies define how the home should react to events. Policies are stored internally as XML. As an example, figure 7 shows a policy that turns on the hall light with a dim level 60% when there is motion there between 10PM and 8AM.

```
<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 id="Activate hall light on night-time movement"
   owner="ken@stirling.org"
   applies_to="@house4.stirling.org"
   enabled="true" changed="2009-12-03T15:12:43">
    <preference>should</preference>
    <policy_rule>
      <trigger arg1="activity" arg2="motion" arg3="hall">
        device_in(arg1,arg2,arg3)
      </trigger>
      <condition>
        <parameter>time</parameter>
        <operator>in</operator>
        <value>22:00:00..08:00:00</value>
      </condition>
      <action arg1="dim" arg2="light" arg3="hall"
              arg5="60">
        device_out(arg1,arg2,arg3,,arg5)
      </action>
    </policy_rule>
  </policy>
</policy_document>
```

Fig. 7. Hall Light Policy in XML

A policy document can contain one or more policies (among other things), though usually just one policy. The policy language is defined by an XML schema, so some 'red tape' is required to identify this.

The attributes of a policy include the following:

id: a descriptive identity for the policy.

owner: the person who defined the policy.

applies_to: the optional domain to which the policy applies. By default, policies apply only to their owners. However, it is possible to specify a domain of applicability. This capability is useful when defining policies for all occupants of a house or for a group of houses (e.g. all those operated by *stirling.org*). It also allows someone other than the householder to define policies (e.g. a doctor, social worker or warden might define policies on behalf of the householder).

enabled: whether a policy is enabled .

changed: when the policy was changed (in XML date-time format).

The content of a policy consists of:

preference: an optional expression of how strongly a policy should apply. This information is used in the case of conflicts (e.g. a policy with a *must* preference might be given priority).

policy_rule: a rule for the policy. Although policies normally define only one rule, it is possible to have several rules (e.g. tried in succession until a relevant one is found).

trigger: an event trigger for the policy. For home policies, the triggers are usually *device_in* events that carry the arguments described in section 2.1. To allow proper XML validation of properties, arguments are named symbolically as *arg1*, etc. and are given values as attributes (e.g. message type *activity*, entity name *motion*, entity instance *hall* here). In general, several triggers may be combined with *and* and *or*.

condition: an optional pre-condition for execution of the policy. This usually has the form of a parameter (*time* here), an operator (*in* here, meaning in a range), and a value (*22:00:00..08:00:00* here, meaning 10PM to 8AM). In general, several conditions may be combined with *and*, *or* and *not*.

action: the action to be performed by the policy. For home policies, the actions are usually *device_out* events that carry the arguments described in section 2.1. The fifth argument in this case is the dim percentage for the hall light (*60* here). In general, several actions may be combined with *and*, *or* and *else* (and in other subtle ways).

The home user could hardly be expected to write XML policies such as the one just discussed. Although tangible interfaces as described in section 1.3.6 are attractive to users, they are quite restrictive in what can be stated. Policies are potentially more flexible and expressive, and therefore lend themselves to different styles of interface. In fact, the system described in this paper is agnostic regarding the way policies are formulated. The policy system has a general interface for storing and retrieving policies; these may be defined using any convenient interface style.

For the work in this paper, 'wizards' make it possible to write policies in a user-friendly manner. Figure 8 shows a web-based wizard that allows policies to be created using stylised natural language. Two important advantages of a web-based interface are familiarity to many users and the ability to enable or modify policies from any location (e.g. the office or when on holiday). Each element of a policy can be clicked to edit it. Hovering over a policy element brings up a tool tip that indicates what it is. Certain structural elements can be combined (e.g. multiple triggers); the '···' symbol can be clicked to achieve this.

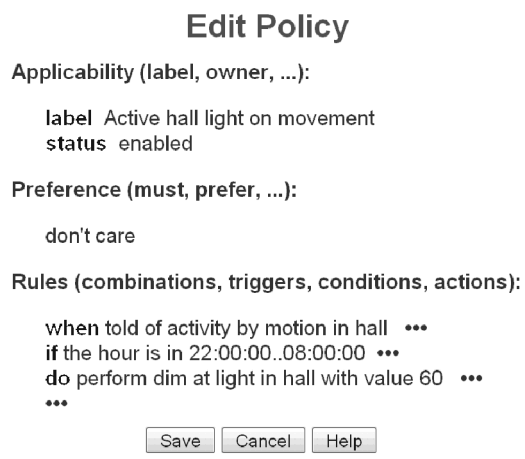

Fig. 8. Hall Light Policy using Web-Based Wizard

As a further example of a policy wizard, the Apple iPad has been used to create a more sophisticated and appealing interface [27]. Figure 9 shows a new policy being defined: when there is movement in the hall, then the front door opens and closes, the hall light is turned on for security. This wizard supports the notion of perspectives: the same policy can be defined or viewed from the point of view of locations, devices, individual people, time or groups (of people, such as family members or home assistants). Users can thus formulate policies in ways meaningful to them instead, for example, of having to think in device-oriented terms.

Although householders can define their own policies from scratch, a variety of mechanisms make policy definition easier:

- The policy system comes with a set of predefined template policies. There is currently a library of about 100 home policies. These cover common situations (e.g. what to do during holidays or if the user falls). The user can then pick a suitable template and fill in a few details (e.g. the dates of a holiday or a person to call).
- Householders need not define their own policies at all. Other people such as relatives, carers or wardens can define policies on behalf of individuals or groups (e.g. in sheltered housing).
- Other forms of policy wizard can make defining policies very easy. For example the digital pen and paper wizard allows policies to be defined just by ticking boxes on a form.

The policy system allows a wide variety of policies to be defined for both home automation and telecare. These policies cover aspects such as appliance control (e.g. energy usage), communication (e.g. how to be contacted), comfort (e.g. room temperatures), entertainment (e.g. favourite programmes), modalities (e.g. use of speech), reminders (e.g. appointments), security (e.g. intruder detection), system aspects (e.g. access control), and telecare (e.g. medication alerts).

Fig. 9. Hall Light Policy using iPad-Based Wizard

3.3. Resolution Policies

The approach to static (definition time) handling of policy conflicts is an extension of earlier work on Internet telephony [10]. The approach to dynamic (run time) handling policy conflicts is also based on the Internet telephony work [40].

Conflicts are almost inevitable with policies. Typically, conflicts arise because the policies of different people are inconsistent. For home automation, the householder may wish to reduce heating levels at night, but a family member may wish better heating if they are not in bed. For telecare, a resident of sheltered housing may wish to watch TV at any hour, but the warden may have a house rule that TVs are switched off after 11PM. Even the policies defined by one individual might be inconsistent, especially if the user de-

fines many policies over time. For example, the user might have a policy of saving energy but also wish the house to be warm in winter.

Resolution policies are defined to detect and resolve conflicts among policy actions. These are specialised policies that a system designer or administrator rather than a householder is expected to define. A library of resolution policies has been created for dealing with likely conflicts, so little work or no may be needed to handle these.

Resolution policies are normally created by a wizard rather than writing XML. The structure of resolution policies resembles that of regular ones. However, the triggers of a resolution policy are the actions of a regular policy. The parameters of these actions are bound to variables in a resolution policy, and are then used in the conditions and actions of the resolution. A resolution policy may have regular actions, but may also use generic actions to choose among conflicting policies.

Conflict detection is activated in the policy server once triggered policies and their actions have been identified. At this point, the policy server considers the set of proposed actions and filters them for conflict. Although actions are considered pairwise, the procedure guarantees a unique set of non-conflicting actions that are then sent to the home system for execution.

As an example of a resolution policy, the following is triggered when two device output actions are proposed. Rather than giving the somewhat indigestible XML, its representation by the web-based wizard is:

when
  told of device output for message1,entity1,instance1,param1
and
  told of device output for message2,entity2,instance2,param2
if
  message1,entity1,instance1 is equal to
  message2,entity2,instance2
and
  param1 is not equal to param2
do apply the stronger policy

There may be conflict if the message type, entity name and entity instance are the same in both cases (e.g. both actions wish to dim the hall light). If the parameter values are different then there is indeed conflict (e.g. the dim levels are different). In such a case, the policy with the stronger preference is followed (e.g. *must* is given priority over *should*). This is an example of a generic action; other generic actions include choosing the more recent policy or one defined in a higher-level domain (e.g. all houses operated by *stirling.org*, as opposed to one particular house). A resolution may also use regular actions. For example, the householder might be told of a volume level conflict and be asked to decide what to do.

Note that conflict detection can be quite subtle. Suppose that two policies wish to send different text messages to the same person. Superficially this is the same kind of conflict as above, but in fact both actions should almost certainly allowed (e.g. one is reporting that the house is too cold, while the other is reporting that a favourite TV programme is starting soon). In practice, what this means is that resolution policies may need to be written to handle specific situations (dimming a light, turning an appliance on or off, sending a message, etc.).

For home systems, many conflicts like this are obvious (e.g. trying to turn heating on and off at the same time). The RECAP tool (Rigorously Evaluated Conflicts Among Policies [10]) has been adapted to statically detect conflicts among home policies. It also automatically generates outline resolution policies .

4. Goals

Users can define high-level goals rather than low-level policies. Predefined prototype policies are introduced as the basic ingredients for achieving goals. These prototype policies are used both statically (when goals are defined) and dynamically (when goals have to be optimised in response to an event).

4.1. Goals

Compared to having to re-program a home system, policies offer an easier way of modifying how the system should react to circumstances. However, policies are still relatively technical. As such, end users are more likely to use and adapt policies from the predefined library than to define new policies from scratch. Goals are hence supported as a more abstract and user-oriented way of defining how the home should behave.

Policies could, in principle, exist for all the sensors and actuators in the home. In a fully automated home, there could easily be 50 managed devices. With multiple occupants of a home and multiple stakeholders in telecare, this might mean up to 100 policies for how the home should react. As it happens, technical scalability of the system is not an issue (having been run with hundreds of policies). However it could be challenging for home users to manage such a large set of policies. Goals help here because they are higher-level and

therefore much less numerous (perhaps 10 per home). Each goal can activate many policies.

The approach to goal definition and refinement is an extension of earlier work on Internet telephony [41]. The OGRE tool (Optimising Goal Refinement Engine) performs static and dynamic analysis of goals, resulting in execution of the most appropriate policies.

A high-level goal is a user objective for the home such as making it comfortable or complying with medication. It can be easier for users to identify their goals (e.g. 'I wish to be secure') than it is for them define a comprehensive set of policies (e.g. 'alert me if I leave the house and a window is open', 'inform a neighbour if the house is entered while I am on holiday').

Goals are defined in terms of subgoals, e.g. comfort might include aspects such as lighting levels, ambient noise levels, and room temperatures. These subgoals are called goal measures, and are the means of assessing how well a goal is achieved. Most approaches to goal refinement take a logic-based approach (see section 1.3.5). However, the author considers this to be impracticable on account of its technical difficulty and run-time inefficiency. Instead, the approach here takes a numerical approach. This allows (sub)goals to be given appropriate weights, allows goal refinement to take current circumstances into account, and allows goals to be achieved as far as possible (and not necessarily fully in some absolute sense).

A goal measure is a formula over relevant system variables. Some variables are held per user (or entity), while others are shared across the system. System variables fall into the following categories:

uncontrolled variables: not managed by the policy system, typically being environmental factors (e.g. energy cost, outside temperature).

controlled variables: managed by the policy system (e.g. monitoring interval, room humidity).

derived variables: defined in terms of (un)controlled variables (e.g. cost of a dishwasher cycle, disc storage space for a recording).

Some variables have a natural measurement (e.g. temperature in degrees, additive intake in grams). Other variables have to be placed on a numerical scaling that ranks them (e.g. security, chill risk). System variables depend on the application domain. They are therefore defined in the domain ontology, and made available by the ontology server (see section 3.1).

Syntactically, a goal is a simplified form of policy. There is no trigger because goals always apply. A goal may have a (compound) condition that uses general information like time of day or an environment value. Unlike a policy, a goal has a single action of the form *maximise(measure)* or *minimise(measure)*. Positive goals aim to maximise their associated measure, while negative goals aim to minimise their measure. The web-based policy wizard formulates goals in stylised natural language, with conveniences such as drop-down boxes for system variables and sliders for weights.

Suppose the goal is to minimise household disturbance at night. This is expressed as follows using the policy wizard (and internally converted to XML):

if the hour is in 23:00:00..07:00:00
do minimise household disturbance

The measure of household disturbance then needs to be broken down into factors such as the ambient noise level and other residents being active at night. These are uncontrolled system variables that are used to define the measure of disturbance such as:

$1.5 \times \text{threshold}(\text{noise\_level}, 70) + 0.5 \times \text{night\_activity}$

In general, a goal measure is an arbitrary formula over system variables. In practice, it is usually defined as a linear weighted sum. In fact the weights are automatically inferred from typical values of the variables, so defining a goal simply requires specifying the relevant variables. The *threshold* function is used when a factor should be counted only over a certain level (here, 70 dB). The *ideal* function sets a planned value for some factor (e.g. 21°C for a room temperature).

In both home automation and telecare, there are often multiple stakeholders with their own goals. As examples, the following cover goals for home automation as well as telecare:

- doctor: minimise allergen exposure (e.g. food additives, pollen)
- doctor: maximise medication compliance (e.g. taking the correct dose of medicine on time)
- family: maximise user activity (e.g. avoiding sleeping too long or watching too much TV)
- householder: maximise household security (e.g. detecting intruders, keeping doors and windows locked)
- householder: minimise home discomfort (e.g. ambient noise level, room temperature)
- social worker: maximise social contact (e.g. going out, phoning friends)
- warden: minimise household disturbance (e.g. noise, night-time activity)

There are normally multiple goals. It is therefore necessary to combine their individual measures into an overall evaluation function. Goal refinement into policies then becomes an optimisation problem: choose the set of prototype policies that maximises the contribution to the overall evaluation function.

In general the evaluation function is an arbitrary formula over goal measures, but is usually a linear weighted sum. The weights in this function determine the relative priority of different (and possibly conflicting) goals. They therefore have to be determined by the user. It has been found that goal achievement does not critically depend on the choice of weights [41]: they can be varied over a wide range with minimal impact on policy selection.

4.2. Prototypes

Goals are not achievable directly as they are high-level. Instead they are realised through sets of policies. Regular policies can contribute to goals. However, the need to manage goal-related policies leads to defining special prototype policies as the building blocks for achieving goals. These are very similar to regular policies, but are considered separately by the goal system. In fact, the policy system normally uses both regular and prototype policies.

Defining prototype policies is a specialised task that a system designer or administrator rather a householder is expected to undertake. A library of prototype policies has been created to support typical home goals. Only a subset of the prototype library is likely to be used to meet a particular set of goals.

Prototypes define an effect that specifies how they modify one or more system variables, and thus how they contribute to goal measures. At definition time, this identifies the relationships between goals and prototypes. At run time, this is used to determine the set of policies that optimally satisfy the goals. Prototypes are also allowed to have parameters that are optimised at run time by the goal system.

The effect of a prototype is an abstraction of the actions it can perform. More specifically, an effect is defined in the same terms as a goal measure. As an example, the following prototype aims to cool the house naturally. Suppose a temperature reading is received, the indoor temperature is above 30°C, and the outdoor temperature is below 25°C. The central heating and air conditioning are then turned off (in case they are already operating), and the upstairs windows are opened for one hour. The web-based policy wizard allows this to be formulated as follows:

when there is a temperature reading
if
  the indoor temperature is above 30
and
  the outdoor temperature is below 25
do
  turn the central heating off
and
  turn the air conditioning off
and
  open the upstairs windows for one hour

So far, this is a regular policy. As a prototype, it also needs to define the abstract effect of this policy. An individual effect names a system variable, an operator, and an expression (e.g. *indoor_temperature -= 4)*. The basic operators are '=' (to set a variable), '+=' (to increase it) or '-=' (to reduce it). These are the operators that have been found most useful, though others would be possible. Sometimes a prototype has an effect that is exclusive of other prototypes. For example, it may be undesirable to have multiple prototypes reducing the indoor temperature at the same time. There are therefore special exclusive forms of the effect operators: '+~' and '-~'. During optimisation, only one such prototype can appear at a time in combination with other prototypes.

Returning to the prototype above, this has the effect of cooling the house (*indoor_temperature -= 4*), reducing home security (windows are now open, *home_security -= 3*), increasing pollen levels in the house (windows are now open, *home_pollen += 1.5)*, and improving home air quality (due to exchange with outside air, *air_quality += 1*). These kinds of effects require expert judgment, and so are formulated as part of the predefined prototype library.

4.3. Static Analysis

Static analysis is performed when a goal or prototype is created, modified or deleted. Whether a prototype contributes to a goal is determined by comparing the prototype effects with how the goal measure is defined (i.e. which system variables it uses). A prototype is considered to contribute to a goal if it affects one or more system variables involved in the goal measure. A prototype effect may modify an arbitrarily complex measure. The sense of the effect is therefore not known until run time, when it may worsen or improve the evaluation of including this prototype.

For each prototype, the goals it contributes to are identified. The prototype is then instantiated as a regu-

lar policy, but specially identified by the list of goals it contributes to. The rest of the policy system treats this generated policy just like any other. The policy server therefore applies the policy at run time exactly as normal. When a prototype is instantiated as a policy, it retains any parameters for later use in dynamic analysis.

4.4. Dynamic Analysis

Dynamic analysis is performed when an event trigger selects policies that derive from goals. These policies are then sent to the dynamic analyser; any regular policies that were selected are later combined with the optimised policies derived from goals.

The dynamic analyser now chooses the subset of goal-related policies that optimise the overall goal evaluation function. Because this is evaluated using the current values of system variables, the most appropriate set of policies can be selected for the current circumstances. It also means that the selection of policies can vary as the system evolves over time. This dynamic approach is much more flexible than the static, logic-based techniques used in most other approaches. Besides choosing an optimal set of policies, the dynamic analyser also optimises the parameters of prototypes (e.g. a monitoring interval or room temperature).

Because of the way the policy system works (choosing only policies relevant to the current trigger and system state), goal refinement normally has to optimise only a modest set of policies. As a result, the optimisation can be performed efficiently (typically in less than a second). The policy system works only with significant events (e.g. someone has entered a room) and not with very frequent events (e.g. every time someone moves). The overhead of the goal/policy system is thus not a problem.

The optimal selection of policies (and parameters) is now returned by the dynamic analyser to the policy server. There, they are combined with other policies that were already selected but did not derive from goals. The actions of the entire policy set are then analysed for conflicts (see section 3.3). The end result is an optimal, conflict-free set of actions to be performed by the home system.

5. Evaluation

This section considers the three levels of the approach (components, policies, goals) from the points of view of functionality and usability.

5.1. Deployment

The entire system is Java-based (apart from third-party operating system drivers) and has been used on Microsoft Windows XP with Sun JDK 1.6. The system uses standard mechanisms defined by OSGi version 4.2, as implemented by Knopflerfish version 2.3 (*www.knopflerfish.org*). The web interfaces are supported by JSPs (Java Server Pages) using Tomcat 5.5. Overall, this has proved to be a stable platform for development, though some problems with the web services interface to Knopflerfish (AXIS) had to be corrected.

Scalability of the system has several facets:

- Since each class of device requires its own component bundle, the system has to support many such bundles. This is not an issue for OSGi, which is quite capable of supporting hundreds of bundles. The only potential issue is run-time memory requirements, but Knopflerfish has been observed to run happily with about 50 mbytes.
- The policy system may also have to cope with many policies. These are stored in an XML database with an efficient indexing mechanism, so use of hundreds of policies is not an issue. At run time, a trigger has to be matched against these – initially to select those with matching triggers, and then to select those whose conditions are met. Indexing allows efficient search of the database. Policy selection and execution (including conflict handling) typically takes around a second. Optimal selection of policies by the goal system typically adds a further second (measured in the case of 15 simultaneous goals). A couple of seconds in processing is barely noticeable to the user as there are no hard real time constraints.
- Policies are intended for higher-level triggers that are significant events (e.g. someone has entered a room) rather than for lower-level events (e.g. the movement sensor just recorded motion). As a result, triggers are relatively infrequent (a few per minute). The load on the system in handling policies is thus relatively low.
- Where a large number of policies or goals might be problematic is from the point of view of comprehension and management. Although users can write and adapt policies, this is a fairly small-scale activity. Most policies are part of the prototype library used to support goals. In practice it has been found that 5 to 10 goals are sufficient to capture the key requirements of users. These

goals also tend to be fairly static and thus do not need much evolution. The only exception is telecare where the end user's condition and abilities deteriorate over time, and hence need adjustment to goals (or policies) on a timescale of months.

Following development of the system described in this paper, it was deployed in two computing labs where it has been operational for 18 months. The system has been used for lab practicals by computing students. Three student projects have also used the system as the basis for new developments, involving 14 person-months of effort in total. After lab evaluation, the system was deployed in two homes where it has been operational for a combined total of 9 months.

The aim of lab deployment was to check correct functioning of the system. The later deployment in user homes was to check aspects such as robustness and the practicality of remote system management. During home deployment, the developers took responsibility for defining policies and goals in conjunction with the end users. Future trials will be more open-ended, including an expectation that users will assume more responsibility for configuring how the home should behave.

Management is a potential problem with home installation as the homes may be remote from the developers. It is also not reasonable to require frequent home visits to correct technical problems. The system has therefore been installed with a broadband connection. Since the system is web-based, it can be managed remotely without requiring on-site access. Goals, policies and device services can be defined remotely as well as locally.

The main issue with remote access is addressing the home PC over a standard broadband connection. Although some Internet Service Providers can allocate a fixed IP address to a broadband router, Dynamic DNS (Domain Name System) is a more practical solution that allows a dynamically allocated IP address to be discovered. For security, remote web access to the home uses HTTPS and standard authentication.

The home system must run unattended, so all system components maintain logs that can be interrogated remotely. Since the system is still experimental, it supplements and does not replace normal functions of the home (for home automation) or normal care services (for telecare). This aspect has been strongly emphasised to the partner social service organisations. For now, alerts generated by the system are not sent to a call centre. As a decision by the end user, alerts can be sent to nominated individuals (family or friends).

Informed consent from end users has been obtained through documents describing what is involved, followed up by interviews. An important issue has been clarifying what data is collected by the system, and taking adequate precautions as to confidentiality. In fact, the data collected is inherently anonymous and essentially concerns only the technical operation of the system (specifically system logs). This is used by the developers to monitor correct operation of the system, and is not exported from the home.

Installation in the home is relatively straightforward. Wireless devices (e.g. Oregon Scientific, Tunstall, Visonic) are easy to install, though care has to be taken to do so unobtrusively and without damaging decoration. The wireless devices used by the system have a typical battery life of three years. The only wired devices are mains-connected (e.g. IRTrans, HomePlug, PlugWise, X10) and so do not require additional wiring. Infrared control requires on-site work by the installer to learn the codes used by the householder's appliances.

Where possible, the home PC is placed in an unobtrusive place. It requires nearby power, a direct or WiFi broadband connection, and wireless connection to devices. The householder may not wish to define services, policies and goals locally; others can then do this remotely. In such a case, the PC can be 'headless' (i.e. only the PC box, without keyboard or mouse). This is still possible (and desirable) even if local definition is required, as a web-based interface is supported via an Archos 5 media tablet (using WiFi).

5.2. Component Level

5.2.1. Functionality Evaluation

The component framework is broadly comparable to that developed by other projects. However, the approach described in this paper is unusually flexible. Components are self-describing in that they identify the triggers, conditions and actions that they handle. This makes it easy to add (or remove) components without having to change the rest of the system (notably the policy server). The value of this has been demonstrated as new components have been added (e.g. Oregon Scientific and Tux Droid support).

The main technical issue with components has been occasionally unreliable support for X10. When the X10 bundle is (re)started it sometimes fails to communicate with the attached X10 computer module. This is due to a combined weakness of the Sun Java serial port

driver and the Prolific USB chip driver for the computer module. The X10 bundle therefore has additional code to monitor for lockup and force its restart. Fortunately this problem happens only if the X10 bundle is quickly stopped and restarted, so it is a rare occurrence. However, it does point to the need for an improved Java serial port driver (which has not been updated on Microsoft Windows since 1998).

5.2.2. Usability Evaluation

The author considered several approaches to evaluating the usability of creating device services with CRESS: 'think aloud' analysis, heuristic evaluation with use cases, cognitive walkthroughs with user profiles and tasks, theoretical analysis of 'cognitive dimensions' [18], and empirical evaluation using quantitative/qualitative analysis. Given that CRESS is intended to be a practical tool, empirical evaluation seemed the most appropriate method. At this stage, only a preliminary evaluation has been conducted.

A mixed empirical evaluation has been conducted to check the following hypothesis: someone with experience of software development, with 45 minutes of training on the approach and the CRESS diagram editor, can define device services consisting of at most four activities, with 80% accuracy, in at most 15 minutes per service. This hypothesis reflected the author's aspiration that the approach be easy to learn and easy to use (at least, for relatively straightforward tasks). The numerical measures were based on the author's experience of teaching programming to students.

The author recruited five software developers without previous experience of the device services approach: two female, three male, average age 25 (range 22 to 31). The participants were given written instructions to follow in their own time, without training or advice from the author. A copy of the CRESS diagram editor (called CHIVE) was provided for local installation, along with a 'palette' of typical symbols used in constructing device services.

The instructions began with a three-page explanation of the approach and the CRESS editor, including three diagrams that the participants were asked to study and then to reproduce themselves using the diagram editor. 45 minutes was suggested as appropriate for this phase, though no time limit was imposed.

In the next part of the instructions, the participants were given five specific tasks to perform. Each task required a device service diagram to be drawn (somewhat different from the examples), based on a natural language description. The participants were asked to record how long tasks took, and to save their diagrams on completion (or after 15 minutes if a task was not completed). The participants were then asked to rate five statements about the approach on a five-point Likert scale. They were also given the opportunity to provide a free-form qualitative evaluation of the exercise.

All collected information was submitted by email to the author. Task times and questionnaire answers were collected and analysed. The only metric that needed to be defined was accuracy. The author scored participant attempts at service diagrams, comparing these against previously created sample solutions. Each possible element was given one mark (e.g. number, name, activity and parameters for a diagram node). This resulted in a percentage score for the accuracy of each diagram.

The participants spent an average of 34 minutes (range 10 to 60) on the familiarisation phase. This compares favourably with the author's expectation of 45 minutes. The shortest period (10 minutes) may reflect this participant's preference for learning by doing rather than extended prior study.

The following describes the five services to be drawn as diagrams, the number of diagram elements in the author's solutions, and participant completion time and accuracy. These exercises were designed to demonstrate understanding of key features of the approach (e.g. use of device input/output, device messages, timers, concurrency, guards and termination).

Exercise 1: *when the front door opens, switch on the porch light*: 15 diagram elements, average time 5.6 minutes (range 3 to 10), average accuracy 85% (range 73 to 100).

Exercise 2: *when the front door opens between 12AM and 6AM, say 'it is night, go back to bed', and turn on the stair and bedroom lights*: 35 diagram elements, average time 8.0 minutes (range 4 to 15), average accuracy 82% (range 47 to 91).

Exercise 3: *when the front door opens and there is movement at the front within 20 seconds, report that the house is occupied*: 19 diagram elements, average time 6.8 minutes (range 3 to 10), average accuracy 89% (range 68 to 100).

Exercise 4: *when the heating is turned on and the outside temperature is above 20, report an energy alert*: 11 diagram elements, average time 3.8 minutes (range 1 to 5), average accuracy 100.0% (range 100 to 100).

Exercise 5: *when requested to secure the house, simultaneously lock the front and back doors*: 25 diagram elements, average time 4.4 minutes (range 2 to 8), average accuracy 87% (range 71 to 100).

Overall, participants completed tasks in an average of 5.7 minutes, with an average accuracy of 88% (compared to the author's hypothesis of 15 minutes and 80%). The author expected exercises 2, 3 and 5 to be the most challenging. Indeed exercises 2 and 3 took the participants longer, but exercise 5 (involving parallelism) was completed quickly. Surprisingly, most participants chose to use parallelism in exercise 2 (where the author's solution was sequential). An extra, anticipated difficulty with exercise 2 involved the use of speech output and device parameters (which had intentionally not been illustrated in the familiarisation examples). For the most inaccurate diagram (exercise 2, 47% correctness), the participant commented that it was unclear how to create compound conditions or how to use device parameters. As a result, this diagram stood out as being rather incomplete.

The commonest errors in diagrams were omitting a node number (which two participants reasonably argued should be irrelevant or automatically generated), omitting an activity name (which one participant reasonably argued should be automatically inferred), and simple syntax errors (such as using '/' rather than '\' before an assignment).

The participants were asked to rate five statements about the approach on a scale from 1 (strongly disagree) to 5 (strongly agree). These were designed to elicit qualitative information about the usability and comprehensibility of the approach for device services.

Statement 1: *I was able to create the service diagrams without too much difficulty*: average score 3.8 (range 3 to 4).

Statement 2: *I found it fairly straightforward to translate the English descriptions into diagrams*: average score 3.2 (range 1 to 4).

Statement 3: *I found it fairly straightforward to create and edit diagrams using* CHIVE: average score 3.6 (range 3 to 4).

Statement 4: *I think the approach would be usable by people with experience of software development*: average score 4.0 (range 3 to 5).

Statement 5: *I think that the approach could be useful in practice for controlling devices in the home*: average score 3.2 (range 2 to 5).

The rating of statement 1 suggests that the approach is usable by the planned type of user, though the diagram editor needs technical improvements as noted elsewhere in this section. The author had expected statement 2 to be least agreed with, since significant mental effort is required to translate a natural language requirement into any formal representation (including programming languages). Like statement 1, the scoring of statement 3 offers encouragement – though improvements to the diagram editor are desirable. The evaluation of statement 4 suggests that the author has correctly targeted an appropriate class of users. Based on the accompanying free-form comments, the lack of a more positive response to statement 5 appears to reflect the need for improvements in the diagram editor rather than doubt over the general approach.

Given the short time that the participants spent in familiarisation (average 34 minutes), their performance impressed the author. Although the limited number of participants does not allow statistically valid conclusions, the results of the preliminary evaluation are encouraging and favour the author's hypothesis.

Note that this was a knowingly demanding evaluation in the following ways:

- The participants were given only a short written briefing and not an extended technical manual or training course. They had no opportunity for classroom instruction or one-to-one advice before undertaking the formal evaluation. This was deliberate by the author, to see how readily the approach could be used with minimal instruction.
- Participants were asked to create diagrams without any way of machine-checking for errors. The full CRESS toolset (as opposed to the diagram editor) does, of course, check for syntactic and static semantic correctness. Indeed, all the syntax errors in the participant diagrams would have been readily identified and corrected in this way. Not providing the full CRESS toolset was again a deliberate decision by the author, in order to discover the extent to which the approach exhibited syntactic idiosyncrasies that would trip up novices.

An unexpected technical problem with the editor arose because, on first use, the editor may be unable to save a diagram. Some participants were therefore unable to save diagrams in native format, and instead saved them as image files. This turned out to be a small technical fault that the author has now corrected.

Several participants complained that text editing was very awkward. This problem was already well-known to the author (and in fact was mentioned in the written briefing to participants). The root cause of this is the *RTFEditorKit* (and related classes) in the Sun Java distribution. This has weaknesses that make text editing rather unpleasant in some circumstances. This

class has not been updated since its original appearance some years ago. The author has sought in vain for alternative implementations or to correct the problem himself. Until this issue is addressed, it would not be wise to deploy the CRESS editor more widely.

In their free-form comments, the participants also provided valuable feedback on how the approach could be improved. In some cases, the observations arose from the shortness of the written briefing, e.g. it was not mentioned that the editor indicated page boundaries with gray lines, the syntax of assignments and conditions was only briefly illustrated, and the explanation of device events was inadequate. These points can readily be addressed through more extended training notes. Concrete suggestions that the author will consider include automatic node numbering, automatic inference of activity names, and use of a toolbox with typical device service symbols.

5.3. Policy Level

5.3.1. Functionality Evaluation

The policy system (policy server, policy wizard, conflict analyser, context manager, ontology server) had previously been developed for use in Internet telephony, where it had been in use for five years and was therefore stable. For use in the home, the policy language was extended with features to make it particularly suitable for this application (e.g. extended use of time and expressions, generic device support). The policy language was also specialised for home applications, with a new home ontology and a new schema for defining home policies. The major technical change was to modify the existing code for use as an OSGi bundle, which also required use of components through the OSGi event interface.

The policy system is broadly comparable to that developed by other projects. However, the approach differs in a number of important respects. Unlike other policy systems that are designed for technical applications (e.g. access control, quality of service, system management), use in the home requires more human-oriented support. For example, the policy wizards were designed for ordinary users and do not require programming (unlike most policy approaches). Furthermore, an effort has been made to make the policy wizards multi-lingual; they currently support policy definition using English, French and German.

Detection and handling of policy conflicts are externalised through separate resolution policies that are also defined by a policy wizard. Conflict handling is flexible, and can make use of preferences and domains (hierarchical authority levels). Although Ponder [11] has similar capabilities, the ACCENT approach was designed for 'softer' management tasks such as those required in the home.

Although the policy server is fully automatic (given appropriate policy definitions), it can sometimes appear too automated to the user. A basic simulator allows the user to evaluate the consequences of defined policies. This can be important if policies evolve over time, leaving 'dead wood' behind that interferes with newer policies. It is planned to extend simulation so that the consequences of policies can be checked in a friendlier manner. This will include an explanation feature reminiscent of expert systems. This will allow the user to ask for an explanation of why something happened in the home (e.g. the heating was turned off due to the requirement for saving energy). It may also prove desirable to support fuzzy policies that deal with necessarily inexact information in the real world (e.g. it will *probably* be a cold night, so extra heating *may* be required). Support is provided for static (definition time) and dynamic (run time) detection and resolution of policy conflicts. However, the former is designed for use by someone technically-minded. It is intended to develop a less technical approach that will warn users of conflicts as policies are defined with a wizard.

5.3.2. Usability Evaluation

Of the various policy wizards, the web-based one is the most developed and was therefore evaluated. Broadly the same approach was followed as for the device service editor (section 5.2.2), but modified for the class of participants involved. At this stage, only a preliminary evaluation has been conducted.

A mixed empirical evaluation was conducted to check the following hypothesis: someone with basic Internet experience (web browsing), with 60 minutes of training on the approach and the policy wizard, can define home control policies consisting of at most four clauses, with 90% accuracy, in at most 10 minutes per policy. At this stage, only a preliminary evaluation has been conducted. This hypothesis reflected the author's aspiration that the approach be understandable and usable by ordinary users (at least, for relatively straightforward tasks). The numerical measures were based on the author's assessment of what would be acceptable and feasible for end users.

The author recruited five ordinary users without previous experience of the approach and without a professional computing background: three female, two male,

average age 55 (range 40 to 70, with three retired). The (former) occupations of participants were personal assistant, secretary, teacher, manager, salesman. Each participant was initially given a verbal briefing (supplemented by a written note), explaining the nature of the evaluation. The author then visited each participant with a laptop running the policy wizard. This included a set of policy templates typical of those required by householders. The author explained and demonstrated the approach face-to-face. Participants were shown how to create three sample policies, and were given the opportunity to define these policies themselves (with the author on hand to provide advice).

The participants were then given five specific tasks to perform. Each task required a home control policy to be defined, based on a natural language description. The tasks were somewhat different from the earlier examples, but could be completed by taking a template policy and filling in the missing parts. This is typical of how the policy wizard is expected to be used in practice. It requires the user to search the template library for an outline policy that could be adapted. Since template policies are largely complete, the number of elements requiring to be filled in is relatively small.

During this second phase, participants were asked to save what they had if a task had not been completed within 10 minutes. The author was present during this phase to record task times and observations, but stated that he would not be able to offer help; reassurance was provided to participants that they should not be concerned if they could not complete a task.

The participants were then asked to answer rate five statements about the approach on a five-point Likert scale. They were also given the opportunity to provide verbal comments on the approach.

Task times and questionnaire answers were analysed by the author. The only metric that needed to be defined was accuracy. Participant attempts at policies were scored and compared against sample solutions previously created by the author. Each possible element was given one mark (e.g. condition parameter, operator and value for a policy). This resulted in a percentage score for the accuracy of each policy.

An average of 41 minutes (range 35 to 46) was spent introducing participants to the approach and allowing them to become familiar with the policy wizard. The familiarisation also allowed time for general discussion about the nature of home automation. The participant familiarisation time compares favourably with the author's expectation of 60 minutes.

The following describes the five home control policies, the number of elements in the author's solutions (though only new elements to be created were scored), and participant completion time and accuracy. These exercises were designed to demonstrate understanding of key features of the approach (e.g. use of triggers, conditions, time and actions).

Exercise 1: *when the front door opens, switch on the porch light*: 8 policy elements, average time 1.6 minutes (range 1 to 3), average accuracy 86% (range 83 to 100).

Exercise 2: *when the front door opens between 12AM and 6AM, say 'it is night, go back to bed', and turn on the stair and bedroom lights*: 16 policy elements, average time 2.8 minutes (range 2 to 5), average accuracy 95% (range 75 to 100).

Exercise 3: *when the medicine dispenser reports the user is late in taking medication, send a message to 456789 to report 'Medicine has not been taken'*: 8 policy elements, average time 1.8 minutes (range 1 to 2), average accuracy 95% (range 75 to 100).

Exercise 4: *never switch on the TV between 10PM and 8AM* : 8 policy elements, average time 4.4 minutes (range 2 to 7), average accuracy 90% (range 50 to 100).

Exercise 5: *when it is Wednesday and 9PM, use the DVD to record channel 3 for 1 hour*: 11 policy elements, average time 5.6 minutes (range 3 to 10), average accuracy 85% (range 83 to 92).

Overall, participants completed tasks in an average of 3.2 minutes, with an average accuracy of 90% (compared to the author's hypothesis of 10 minutes and 90%). The author expected exercises 4 and 5 to be the most challenging, and indeed these took longer.

An unexpected difficulty arose with exercise 4. When searching for an appropriate template policy, several participants first chose the template entitled 'TV recording' rather than the intended one of 'No night time appliance'. This was understandable as exercise 4 does indeed ask for a policy about TV use. On realising this was an inappropriate template, most participants then went back and found the more suitable one. However, one participant decide to stick with the selected template and had extra work to modify it as required (explaining why this was the longest attempt at this exercise). Underlying this difficulty is a key point: the templates need to be clearly described so that users can quickly find an appropriate one. As the

number of templates grows, this could become more difficult. An obvious solution that is being considered is a search facility that allows users to find relevant templates easily.

Exercise 5 exhibited extra challenges in that two action parameters had to be inserted (to define the channel to be recorded and the recording duration). These were intentionally not described during familiarisation, as the author wished to see whether participants had grasped the idea of filling in all aspects of a policy. For the most part the participants coped well with this. However, one participant decided to create this policy from scratch (which is possible, but had not been explained during familiarisation). As a result, this individual reached the 10-minute time limit and had to stop with an incomplete policy.

A common error in policies was defining what the participant thought was reasonable and not what the written exercise called for (e.g. giving a family member's phone number for reporting lateness in medication rather than the specified number). Although this was considered to be a partial error when scoring results, it is acknowledged that the exercises posed artificial situations. Another common error was in naming household objects (e.g. TV vs. television, back door vs. side door). In practice, the policy system is used with a configuration that reflects the user's actual house. This can include synonyms (e.g. main bedroom, master bedroom, double bedroom). However as suggested by one participant, it would be helpful if drop-down lists (populated from the configuration) could be used to give better guidance when defining policies. This improvement was already planned.

The participants were asked to rate five statements about the approach on a scale from 1 (strongly disagree) to 5 (strongly agree). These were designed to elicit qualitative information about the usability and comprehensibility of policies for home control.

Statement 1: *I was able to create the policies without too much difficulty*: average score 3.8 (range 2 to 5).

Statement 2: *I found it fairly straightforward to translate the English descriptions into policies*: average score 3.6 (range 2 to 5).

Statement 3: *I found it fairly straightforward to create policies using the wizard*: average score 3.8 (range 2 to 5).

Statement 4: *I think the approach would be usable by people with basic Internet experience*: average score 3.8 (range 2 to 5).

Statement 5: *I think that the approach could be useful in practice for controlling devices in the home*: average score 4.0 (range 2 to 5).

In fact, these ratings were more positive than the author had expected. Until this evaluation, the author could not be sure if ordinary users could formulate rules for managing their homes. Given the short time that the participants spent in familiarisation (average 41 minutes) and their limited computing experience, their performance impressed the author. Although the limited number of participants does not allow statistically valid conclusions, the results of the preliminary evaluation are encouraging and favour the author's hypothesis. Three of the participants expressed enthusiasm for the approach, and would be keen to use it in practice. They also expected that they would become more adventurous with experience, defining more sophisticated or more complex rules.

During each evaluation, and formally at the end of it, the author noted comments and questions by the participants. A few participants observed that the terminology and style of the interface tended to be technical, e.g. 'rule' would have been preferred instead of 'policy', 'predefined rule' instead of 'template', 'back' instead of 'cancel'. One participant also requested greater flexibility in entering dates and times, e.g. 'Monday' instead of 'Day 1', and '10.30' instead of '10:30'. Another participant commented that editing policies requires a mental model of going up and down a tree (e.g. choosing part of a policy to edit, going into the detail of this, then returning to the main policy). This was felt to require a more technical way of thinking, though it was accepted that a similar kind of behaviour is common during web browsing (e.g. clicking on a link and then returning to the original page).

At the start of each evaluation, the browser font size was adjusted to a comfortable value for the participant. An oddity that emerged is that JavaScript alerts (used to report incorrect parameter formats or ranges) still appear in a default font size – too small for some participants. This is a well-known limitation that affects many browsers (including FireFox 3 and Internet Explorer 8 as used in the evaluations). At best, there are complex and awkward work-arounds for this that require alerts to be reported in a completely different fashion. The larger font selected by several participants caused a small difficulty in that it was then necessary to scroll down to the Save/Cancel/Help buttons. Placing these at the top of pages would ease this problem.

For the most part, the structure of policies was clear to the participants. Two participants queried whether the policy names had any significance; in fact they do not, being merely identifying strings. All participants gave meaningful names to their policies. A syntactic idiosyncrasy of the policy wizard emerged during the evaluations. The operator must be 'is' when a single condition value is required (e.g. 'the hour is 09:00'). However when multiple values are permitted for a list or range, the operator must be 'is in' (e.g. 'the hour is in 09:00,12:00,17:00'). This caused considerable confusion on exercise 4 for one participant, who could not see why the formulation of a time condition was being rejected. This accounted for the single low score (50%) of all the participant attempts. This deficiency is an obvious and readily rectified problem.

One participant requested a more visual form of wizard in which household objects could be selected by clicking on a floor plan. In fact, just such an interface is currently under development. The same individual also wondered if rules might be too rigid, and might need to be relaxed on some occasions. This is a fair observation, and is addressed to the extent that the policy system can always be overridden – either by temporarily disabling a rule (a simple mouse click on the main menu) or by manually overriding the actions of the policy system.

5.4. Goal Level

5.4.1. Functionality Evaluation

The goal system is a relatively new extension to the policy system. Also originating in control of Internet telephony, its adaptation for home use was relatively straightforward. The main work was in defining a new home ontology.

Compared to other (logic-based) approaches to goal refinement, the goal system is unusual in treating refinement as an optimisation problem. This has a number of advantages. Goal refinement is inherently dynamic and can thus take current circumstances into account when determining the most appropriate set of policies. The numerical approach also means that goals are achieved on a sliding scale, unlike other approaches that require complete goal fulfilment.

Goals are currently defined using an extended version of the web-based policy wizard. Although this uses stylised natural language, a new project is under way to create a friendlier way of defining goals. Goal weights are defined by the user (graphically). It may be desirable for the goal system to learn weights automatically (perhaps in the style of ACHE [31]). User feedback or demonstration could be used to indicate that a certain goal should be changed in importance.

5.4.2. Usability Evaluation

At the present stage of development, evaluation of the goal-oriented approach requires a more theoretical and conceptual analysis than an empirical one. Significant questions include whether users do (or can) think in terms of goals for their household, whether they can articulate goals in the way described in this paper, whether the goal wizard is usable by ordinary householders, and whether users will understand the connection between goals and how their house reacts to circumstances (as dictated by the underlying policies). As the goal system evolves, these aspects will be the subject of a future usability study.

6. Conclusion

It has been explained that the aim of this work is to support flexible management of smart homes, both in home automation and in telecare. As far as possible, management of the home is made possible without requiring specialised technical knowledge or programming skills. This is achieved at three levels: components, policies and goals.

At component level, the architecture is essentially that of OSGi and makes use of the Event Admin service to decouple the home components. Flexible handling of device events is made possible through separate event logic. This also allows external control of the home through web services.

At policy level, rules can be defined that react to home events and lead to home actions. Special resolution policies are used to detect and resolve conflicts among policy actions. Although the underlying policy representation uses an XML schema, wizards make it easy for users to define policies using interfaces such as stylised natural language and digital forms. Other conveniences include a policy template library.

At goal level, users can define high-level objectives for what they wish to achieve. These goals are refined at run time into an optimal set of policies that best achieve the combination of goals. The basic ingredients to achieve goals are predefined prototype policies. These are like regular policies but define their effects on goal measures. Goal refinement is treated as an optimisation problem that uses numeric goal measures and an overall evaluation function.

An evaluation has been given of functionality and usability: for the whole system, for visual design of device services, for creation of policies, and (briefly) for definition of goals. This gives some evidence that the approach is dependable and usable, though specific improvements have also been identified.

In fact the whole approach is much more general than just for managing smart homes; its use for Internet telephony, sensor networks and wind farms has been mentioned. The versatility of the system is a strength, but it also means that it is not specialised for any one domain. Several of the improvements suggested by the evaluation (e.g. visual interaction via a house plan) are appropriate for home management but would not be so appropriate for other applications. The multilingual approach also requires the use of stylised natural language; full natural language processing would have been a completely different undertaking. As a result, the author has steered a middle course between generality/reusability on the one hand, and specificity/inextensibility on the other hand. However, there is scope for more domain-specific interfaces in each kind of application.

Future work will focus on more usable interfaces and more thorough trials. The digital pen and iPad policy wizards in section 3.2 are seen as the most promising interfaces to develop further. The evaluations so far have been limited in scope and scale. Future trials will be more open-ended in nature, allowing users to play a larger role in defining goals and policies. It will be particularly interesting to see the extent to which users are willing to evolve these as their requirements change and their understanding of the system grows.

Acknowledgements

The author thanks his colleagues at the University of Stirling for their contributions to this work. Stephan Reiff-Marganiec undertook the original design of the policy language and policy system. Lynne Blair implemented policy conflict handling. Feng Wang adapted the policy server for OSGi and telecare, and worked on several home components. Gavin A. Campbell implemented goal refinement. Claire Maternaghan contributed to the component framework and library.

The author also thanks his colleagues on the MATCH project (*www.match-project.org.uk*) for their collaboration on home care technologies. At the University of Glasgow, Phil Gray and Marilyn McGee-Lennon provided advice on evaluating usability of the system discussed in this paper. Tony McBryan provided code for interfacing to the SHAKE. At the University of Edinburgh, Steve Renals and Maria Wolters arranged a licence to use the Cerevoice speech system, and developed appropriate 'voices' for the MATCH project.

The author is grateful to the anonymous referees for their thoughtful comments on a draft of the paper.

References

[1] M. Addlesee, R. Curwen, S. Hodges, J. Newman, A. W. P. Steggels, A. Ward, and A. Hooper. Implementing a sentient computing system. *IEEE Computer*, 34(8):50–56, Aug. 2001.

[2] D. Amyot. Goal-oriented requirement language (GRL) and its applications. In *Proc. 31st Int. Conf. on Software Engineering*. IEEE Computer Society, Los Alamitos, California, USA, May 2009.

[3] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Lie, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language*. Version 2.0. Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, Apr. 2007.

[4] R. Ballagas, A. Szybalski, and A. Fox. Enabling control-flow interoperability in ubicomp environments. In S. K. Das and M. Kumar, editors, *Proc. 2nd Conf. on Pervasive Computing and Communications*, pages 241–252. IEEE Computer Society, Los Alamitos, California, USA, Mar. 2004.

[5] A. K. Bandara, E. C. Lupu, J. D. Moffett, and A. Russo. A goal-based approach to policy refinement. In *Proc. Workshop on Policies for Distributed Systems and Networks*, pages 229–239. IEEE Computer Society, Los Alamitos, California, USA, 2004.

[6] J. E. Bardram. The Java context awareness framework – A service infrastructure and programming framework for context-aware applications. In H. W. Gellersen, R. Want, and A. Schmidt, editors, *Proc. 3rd Int. Conf. on Pervasive Computing*, number 3468 in Lecture Notes in Computer Science, pages 98–115. Springer, Berlin, Germany, May 2004.

[7] A. F. Blackwell and R. Hague. AutoHAN: An architecture for programming the home. In *Proc. Symp. on Human Centric Computing Languages and Environments*, pages 150–157. ACM Press, New York, USA, Sept. 2001.

[8] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. Easyliving: Technologies for intelligent environments. In P. J. Thomas and H.-W. Gellersen, editors, *Proc. 4th Int. Symp. on Handheld and Ubiquitous Computing*, number 1927 in Lecture Notes in Computer Science, pages 12–29. Springer, Berlin, Germany, Sept. 2000.

[9] G. A. Campbell and K. J. Turner. Ontologies to support call control policies. In N. Meghanathan, D. Collange, and Y. Takasaki, editors, *Proc. 3rd Advanced Int. Conf. on Telecommunications*, pages 5.1–5.6. IEEE Computer Society, Los Alamitos, California, USA, May 2007.

[10] G. A. Campbell and K. J. Turner. Policy conflict filtering for call control. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 83–98. IOS Press, Amsterdam, Netherlands, May 2008.

[11] N. Damianou, E. C. Lupu, and M. Sloman. The Ponder policy specification language. In *Policy Workshop 2001*, number 1995 in Lecture Notes in Computer Science. Springer, Berlin, Germany, Jan. 2001.

[12] A. K. Dey, R. Hamid, C. Beckmann, I. Li, and D. Hsu. A CAPpella: Programming by demonstration of context-aware applications. In *Proc. Conf. on Human Factors in Computing Systems*, pages 33–40. ACM Press, New York, USA, Apr. 2004.

[13] T. Dursun and B. Örencik. Police: A novel policy framework. In *Proc. ISCIS 2003*, number 2869 in Lecture Notes in Computer Science, pages 819–827. Springer, Berlin, Germany, 2003.

[14] W. K. Edwards, M. W. Newman, J. Sedivy, T. Smith, and S. Izadi. Recombinant computing and the Speakeasy approach. In I. F. Akyildiz, J. Y.-B. Lin, R. Jain, V. Bharghavan, and A. T. Campbell, editors, *Proc. 8th Int. Conf. on Mobile Computing and Networking*, pages 279–286. ACM Press, New York, USA, Sept. 2002.

[15] K. Gajos, H. Fox, and H. Shrobe. End user empowerment in human centered pervasive computing. In F. Mattern and M. Naghshineh, editors, *Proc. 1st Int. Conf. on Pervasive Computing*, number 2414 in Lecture Notes in Computer Science, pages 134–140. Springer, Berlin, Germany, Aug. 2002.

[16] L. A. Gavrilov and P. Heuveline. Aging of population. In P. Demeny and G. McNicoll, editors, *The Encyclopedia of Population*, pages 27–50. MacMillan, London, UK, Jan. 2003.

[17] P. Gouvas, T. Bouras, and G. Mentzas. An OSGi-based semantic service-oriented device architecture. In R. Meersman, Z. Tari, and P. Herrero, editors, *Proc. On the Move to Meaningful Internet Systems*, number 4806 in Lecture Notes in Computer Science, pages 773–782. Springer, Berlin, Germany, Nov. 2007.

[18] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Visual Languages and Computing*, 7(2):131–174, June 1996.

[19] N. D. Griffeth and H. Velthuijsen. Negotiations in telecommunications systems. Technical Report R2401, Bellcore, Morristown, New Jersey, USA, June 1992.

[20] A. Harter and A. Hooper. A distributed location system for the active office. *IEEE Network*, 8(1):62–70, Jan. 1994.

[21] A. Kameas, I. Mavrommati, and P. Markopoulos. Computing in tangible: Using artifacts as components of ambient intelligence environments. In G. Riva, F. Vatalaro, F. Davide, and M. Alcañiz, editors, *Ambient Intelligence: The Evolution of Technology, Communication and Cognition*, pages 121–142. IOS Press, Amsterdam, Netherlands, Jan. 2005.

[22] J. Kind, R. Bose, H.-I. Yang, S. Pickles, and A. Helal. Atlas: A service-oriented sensor platform. In *Proc. Workshop on Practical Issues in Building Sensor Network Applications*. Institution of Electrical and Electronic Engineers Press, New York, USA, Nov. 2006.

[23] M. Knoll, T. Weis, A. Ulbrich, and A. Brändle. Scripting your home. In *Proc. Symp. on Human Centric Computing Languages and Environments*, number 3987 in Lecture Notes in Computer Science, pages 274–288. Springer, Berlin, Germany, May 2006.

[24] C. Leong, A. R. Ramli, and T. Perumal. A rule-based framework for heterogeneous subsystems management in smart home environment. *IEEE Transactions on Consumer Electronics*, 55(3):1208–1213, Aug. 2009.

[25] J. E. López de Vergara, V. A. Villagrá, C. Fadón, J. M. González, J. A. Lozanoc, and M. Álvarez-Campana. An autonomic approach to offer services in OSGi-based home gateways. *Computer Communications*, 31(13):3049–3058, Aug. 2008.

[26] B. Margolis and J. L. Sharpe. *SOA for The Business Developer*. MC Press, Woodland, Texas, USA, 2007.

[27] C. Maternaghan. The homer home automation system. Technical Report CSM-187, Department of Computing Science and Mathematics, University of Stirling, UK, Dec. 2010.

[28] C. Maternaghan and K. J. Turner. A component framework for telecare and home automation. In S. Balandin, M. Matuszewksi, J. Ott, and G. Chan, editors, *Proc. 7th Conf. on Computer Communications and Networking*, pages N4.1–N4.5. IEEE Computer Society, Los Alamitos, California, USA, Jan. 2010.

[29] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proc. 6th. European Software Engineering Conference/Proc. 5th. Symposium on the Foundations of Software Engineering*, pages 60–76, Zurich, Switzerland, Sept. 1997.

[30] A. Messer, A. Kunjithapatham, M. Sheshagiri, H. Song, P. Kumar, P. Nguyen, and K. H. Yi. Interplay: A middleware for seamless device integration and task orchestration in a networked home. In E. Gregori and A. Hurson, editors, *Proc. 4th Conf. on Pervasive Computing and Communications*, pages 307–316. IEEE Computer Society, Los Alamitos, California, USA, Mar. 2006.

[31] M. C. Mozer. The neural network house: An environment that adapts to its inhabitants. In M. Coen, editor, *Proc. AAAI Symp. on Intelligent Environments*, pages 110–114. AAAI Press, Mar. 1998.

[32] B. A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Boston, USA, 1993.

[33] M. W. Newman, A. Elliott, and T. F. Smith. Providing an integrated user experience of networked media, devices, and services through end-user composition. In *Proc. Symp. on Human Centric Computing Languages and Environments*, number 5013 in Lecture Notes in Computer Science, pages 213–227. Springer, Berlin, Germany, May 2008.

[34] P. Rigole, T. Holvoet, and Y. Berbers. Using Jini to integrate home automation in a distributed software system. In J. Plaice, P. G. Kropf, P. Schulthess, and J. Slonim, editors, *Distributed Communities on The Web*, number 2468 in Lecture Notes in Computer Science, pages 185–232. Springer, Berlin, Germany, Apr. 2002.

[35] T. Rodden, A. Crabtree, T. Hemmings, B. K. J. Humble, K.-P. Åkesson, and P. Hansson. Configuring the ubiquitous home. In *Proc. 6th Int. Conf. on The Design of Cooperative Systems*, pages 215–230. IOS Press, Amsterdam, Netherlands, May 2004.

[36] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure for active spaces. *Pervasive Computing*, 1(4):74–83, Oct. 2001.

[37] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, and A. L. Lafuente. Using linear temporal model checking for goal-oriented policy refinement frameworks. In *Proc. Workshop on Policies for Distributed Systems and Networks*, pages 181–190. IEEE Computer Society, Los Alamitos, California, USA, 2005.

[38] T. Sohn and A. K. Dey. iCAP: An informal tool for interactive prototyping of context-aware applications. In *Proc. Int. Conf. on Human Factors in Computing Systems*, pages 974–975. ACM Press, New York, USA, Apr. 2003.

[39] K. N. Truong, E. M. Huang, and G. D. Abowd. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In N. Davies, E. Mynatt, and I. Siio, editors, *Proc. Ubiquitous Computing*, number 3205 in Lecture Notes in Computer Science, pages 143–160. Springer, Berlin, Germany, Sept. 2004.

[40] K. J. Turner. Device services for the home. In K. Drira, A. H. Kacem, and M. Jmaiel, editors, *Proc. 10th Int. Conf. on New Technologies for Distributed Systems*, pages 41–48. Institution of Electrical and Electronic Engineers Press, New York, USA, May 2010.

[41] K. J. Turner and L. Blair. Policies and conflicts in call control. *Computer Networks*, 51(2):496–514, Feb. 2007.

[42] K. J. Turner and G. A. Campbell. Goals and conflicts in telephony. In M. Nakamura and S. Reiff-Marganiec, editors, *Proc. 10th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 3–18. IOS Press, Amsterdam, Netherlands, June 2009.

[43] K. J. Turner, L. S. Docherty, F. Wang, and G. A. Campbell. Managing home care networks. In R. Bestak, L. George, V. S. Zaborovsky, and C. Dini, editors, *Proc. 9th Int. Conf. on Networks*, pages 354–359. IEEE Computer Society, Los Alamitos, California, USA, Mar. 2009.

[44] K. J. Turner, S. Reiff-Marganiec, L. Blair, J. Pang, T. Gray, P. Perry, and J. Ireland. Policy support for call control. *Computer Standards and Interfaces*, 28(6):635–649, June 2006.

[45] A. van Lamsweerde and E. Letier. From object orientation to goal orientation: A paradigm shift for requirements engineering. In *Proc. Radical Innovations of Software and Systems Engineering in The Future*, number 2941 in Lecture Notes in Computer Science, pages 153–166, Berlin, Germany, Mar. 2003. Springer.

[46] VoiceXML Forum. *Voice eXtensible Markup Language*. VoiceXML Version 2.0. VoiceXML Forum, Piscataway, New Jersey, USA, Jan. 2003.