Policy Conflicts in Home Automation

Claire Maternaghan and Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK

Abstract

The nature of home automation is introduced. It is argued that end users should be able to define how the home system reacts to changing circumstances. Policies are employed as user-defined rules for how this should happen. The architecture of the Homer home automation system is briefly overviewed. The Homer policy system and the Homeric policy language it supports are explained. A technique is described for offline conflict analysis among policies (the analogue of the feature interaction problem). A substantial worked example shows how conflict detection is performed on a range of sample home policies.

Key words: Feature Interaction, Home Automation, Policy-Based Management, Policy Conflict

1 Introduction

This section introduces home automation and the desirability of end-user programming for a home system. Policy-based approaches to home management are discussed, along with relevant work on features and policies for the home.

1.1 Background

Home automation has been a goal for many years. This is coming closer to fruition with the increased number of computer-based devices around the home, coupled with greater user understanding of technology. However, many existing systems do not really deal with home *automation* (i.e. defining automated reactions by the home system). Rather, the focus tends to be on home *control* (e.g. of audio-visual

Email addresses: `clairematernaghan@gmail.com` (Claire Maternaghan), `kjt@cs.stir.ac.uk` (Kenneth J. Turner).

devices, heating and lighting). Based on a survey by the first author [6], there is user demand for being able to modify how the home system should react to changing circumstances.

Although some commercial systems allow a degree of programming, in many cases this has to be performed by the system installer. Where end user programming is possible, this requires detailed technical knowledge that is likely to be beyond the ordinary householder. What is required is a convenient, user-definable way of specifying how the home should support the residents. However, there are many challenges in creating a home automation system such as the following.

Heterogeneity: The system must be able to accommodate a wide variety of domestic appliances and devices, ranging from simple ones like lamps to complex ones like media centres. These are likely to be from many manufacturers and to conform to many standards.

Extensibility: The range of computer-controllable devices in the home is continually increasing. Examples are the recent introduction of media servers and media players. It is impracticable for the system developer to provide support for every kind or make of device. Instead, the system must be extensible to allow third parties to add support for new devices. It is also necessary to support configuration of new devices and automatic integration of these into home control and customisation.

Separability: A home automation system needs to be able to deal with devices directly and also with complete subsystems. For example, a heating system or an entertainment system is likely to be self-contained in its own right. A home system should therefore not expect to control everything directly. However, everything is managed from one location and can therefore be considered as centralised (though not all in one system).

Mobility: The user is likely to require monitoring and control of the home system from outside the home (e.g. in the office or while on holiday). Although most facilities are likely to be fixed in the home, it may be desirable to extend these to mobile devices. For example, the user might wish to stream music or video from the home to a mobile device or to a car. The boundaries of the home mostly obviously include the building and its immediate environment, but increased demands for mobility can extend these boundaries.

Customisability: Besides being able to monitor and control the home, users would like to modify how it reacts to changing circumstances [6]. Home users tend to think of this as customisation rather than programming (a term that could put off non-technical users).

Usability: Home users are unlikely to be technically knowledgeable, and therefore need easily understood ways of configuring, monitoring and customising the home system. Usability can also be subjective, so differences in user preferences must be accommodated.

This paper introduces the Homer home automation system and its associated Homeric policy language. More about these can be found in [8]. The aim has been to meet the challenges above. The approach supports heterogeneity, extensibility, separability and mobility, while the Homer policy system supports customisability and usability through user-defined policies for how the home should behave. As policies may conflict with each other, a technique is used to detect and handle these kinds of problems.

1.2 Related Work

As the focus of this article is on conflicts among home policies, see [8] for related work on home systems.

1.2.1 Features and Interactions in The Home

Many techniques have been developed for detecting feature interactions, particularly in telephony. There has also been relevant work on feature interaction in home systems:

Nakamura *et al.* model appliances within a home network [9,10]. Before a method can be invoked on an appliance, its pre- and post-conditions must be met. Failure to meet these conditions is treated as an interaction. Methods also affect a pre-defined list of 'environment' properties such as power or temperature. The approach is rather restrictive in the range of environment properties that can be considered. More seriously, it assumes that precise models can be created for all the domestic appliances. This is unrealistic as appliances are typically proprietary; they also exist in many varieties that could not all be reasonably modelled.

Wilson *et al.* follow a high-level approach that aims to detect conflicts among services [3,20]. The effects of each device on 'environment' variables are defined, and used to detect whether invoking a device action can lead to interference. Environment variables can be 'locked' according to whether they must be exclusively modified, may be shared for increases or decreases, or could be either increased or decreased. The approach was inspired by operating systems, but is rather basic for home use. In practice, whether a conflict exists can be subjective. For example, in some regions it is common to have an open fire (for visual effect) but also to have the air conditioning on (to avoid overheating). Wilson's strict interpretation would consider this to be a conflict.

In the field of feature interaction, detection techniques are usually classified as offline (static, definition-time) or online (dynamic, run-time). Offline techniques often require detailed knowledge of features, while online techniques aim to deal with the effects of features as they are executed. In telephony, online techniques are preferable as the specification of features may be proprietary and unavailable. In addition, the features involved in a call may not be known in advance since anyone on any network can call anyone else.

The situation in the home is different. Features are centralised in the home system, so the issue of feature distribution does not arise. This allows simplifications to be made in detecting interactions, though proprietary device interfaces may still be a challenge.

1.2.2 Policies and Conflicts in The Home

The notion of a feature originated in telephony to describe functionality that can be added to a called, e.g. the ability to redirect calls. Computer-interpreted policies were originally devised for applications such as managing systems, networks, quality of service and access control. More recent policy applications have included body sensor networks, business rules and call control.

In the context of the home, telephony and home automation can overlap. For example, when the phone rings at home then the TV could be muted. Also, when the user goes out then calls could be redirected to a mobile phone. Thus both features and policies can be relevant in the home. Policies resemble features, so policy conflict is the analogue of feature interaction.

The two policy approaches of greatest relevance to home automation are as follows.

ACCENT (Advanced Component Control Enhancing Network Technologies [18]) was originally developed for call control in Internet telephony. However, it was subsequently extended to manage home care [16] and so is highly relevant to this paper. ACCENT supports policies in ECA form (Event-Condition-Action). In addition ACCENT supports domain-specific ontologies, goals and their realisation through policies, conflict handling among goals and policies, and wizards for easy policy definition.

However, ACCENT is intentionally a rather generalised approach. Its applicability to many domains is a strength, but is also a weakness in that it is not well tailored for particular domains (such as home automation). ACCENT is therefore not as convenient for home users as it should be. The approach is designed for a distributed setting such as a telephone network, and is more complex than needed for the home (where nearly everything is centralised).

ACCENT also does not support certain key capabilities that have been identified as important for home system rules [6]. For example, home users do not readily distinguish events and conditions so these should be blurred. As an example, most users would consider 'the front door opens' (trigger) and 'the front door is open' (condition) to be equivalent when writing a policy. It has also been found that users like to define conditional actions in policies (e.g. 'when the front door opens, if the house is occupied then play music else report an intruder').

Ponder (and its derivative Ponder 2, *www.ponder2.net*) is a well-known policy language that is often used for system management, but has now been extended for applications such as Body Sensor Networks [2]. Like ACCENT, Ponder supports policies in ECA form. Additional features include policy domains, support of obligation and prohibition policies, and integration with goal refinement.

Ponder has not, however, been designed for use in a home setting. It is also more oriented toward systems programmers, and is therefore less suitable for ordinary users. Like ACCENT, Ponder lacks important features that were found to be desirable for home use. Experiments by the authors with Ponder showed that it would not be a good match to the needs of managing the home.

Policies are the analogue of telephony features, so policy conflict is the analogue of feature interaction. For the approaches discussed above, policy conflict for ACCENT has been investigated for telephony [4,12] and for home care [19]. ACCENT supports offline detection of conflict-prone policies [1] and also online detection [17]. Policy conflict for Ponder has been investigated for distributed system management [5,11].

1.2.3 Constraint Satisfaction

Constraint satisfaction originated with early work on Artificial Intelligence [14]. The aim is to find values that satisfy a set of constraints. Sometimes the existence of a solution is a sufficient answer, but the ‘best’ solution may also be required. A wide range of constraint solvers is available. JaCoP (Java Constraint Programming, *www.jacop.eu*) was of particular interest to the authors because it can be integrated easily with Homer (which also uses Java). As will be seen, JaCoP is used as the basis for detecting policy overlaps.

1.3 Paper Outline

Section 2 gives a high-level overview of the Homer system. The policy server and its policy language are also introduced. Section 3 discusses how conflicts are handled in three stages: detecting overlaps among policies, detecting conflicts among their actions, and dealing with detected conflicts. An extended example of conflict analysis is given. Section 4 summarises the paper and points to future work.

2 The Homer Home Automation System

This section introduces the Homer home automation system and the support it offers for policy-based management through the Homeric language.

2.1 Homer Philosophy

Homer aims to support control, monitoring *and* customisation of a home system. It is designed to be open, allowing easy addition of new device types by third party developers. User-friendly definition of rules is supported, using interfaces provided by Homer or by third parties. The policy language reflects the requirements expressed by users for home management [6].

An important aspect is allowing policies to be expressed in multiple ways. Many other approaches enforce a device-oriented view which (as argued in [13]) can be unnatural for the user. Instead, Homer is designed to allow policies to be written from multiple perspectives: device (e.g. how signals should be handled), location (e.g. what should happen in a room), time (e.g. what should happen on a weekday) and people (e.g. how an individual should be supported). For example, similar policies with different perspectives might say ‘when the bedroom sensor reports movement’, ‘when the bedroom becomes unoccupied’, ‘when it is 7:30AM’ and ‘when I get up’. This gives users the freedom to formulate policies as they choose. It has been confirmed that users like and can make use of this flexibility [7].

The challenges of home automation outlined in section 1.1 motivated the design of Homer as follows:

Heterogeneity: Homer needs to support a wide range of appliances and devices. In comparison, traditional telephony has a relatively limited variety of components and is usually under the control of one design authority (the network operator).

Homer needs to support a rich (and growing) variety of events and actions, and also a wide range of conflicts that arise from these. In comparison, telephony feature interaction involves a limited range of well-defined events and actions. As a result, many of the techniques for feature interaction in telephony do not map well to home automation.

Extensibility: Traditional telephony does not have to support addition of third-party devices. For home automation, Homer was designed to be extensible by separating the core system from component-specific configuration, control and customisation. The Homer framework is device-independent, with variations embedded in the components. Homer already supports many devices, but can be (and has been) extended by third-party developers for new kinds of devices.

Separability: Since Homer components are self-describing, it is easy for Homer to manage not only simple devices but also complex ones (that may in fact be self-contained subsystems).

Mobility: Homer supports user mobility by allowing the home to be managed through mobile devices like phones and tablets (as well as remote web-based access). Homer itself does not support other forms of mobility such as streaming media to a mobile device or a car. However, these are likely to be offered by other services that Homer can simply use without having to duplicate the capability.

Customisability: Requirements for the Homer policy language were drawn from a survey of user needs [6]. Examples of specific needs for home automation were noted above. Many policy approaches (e.g. ACCENT and Ponder) do not meet these requirements. Many policy languages are also designed for technically oriented users (e.g. for system management, access control or quality of service). These and other factors mean that standard policy languages are not especially suitable for home use.

Customisation also includes how policy conflict is handled. Policy conflict (or feature interaction) in home automation can be subtle and subjective. For example, turning the heating on and opening a window might or might not be considered to be an interaction. Switching on two power-hungry appliances might be considered undesirable if it exceeds electricity consumption limits. Homer is designed to deal with these kinds of issues. In contrast, telephony feature interactions tend to be much more clear-cut, e.g. accept call vs. block call, or forward call vs. reject call. As a result, feature interaction techniques developed for telephony have limitations when applied to home automation.

Usability: Unlike many technical systems, home systems must be usable by ordinary people. This requires the system to be easily conceptualised and to have convenient user interfaces. These requirements are reflected in the design of Homer and also in its interfaces. For example, Homer supports monitoring, control and customisation using desktop, web, phone and tablet interfaces.

In a home context, all policies are under the control of the central home system and are known to it. This means that the information required to detect conflicts is available when a policy is defined. It is preferable to report problems when policies are defined rather than when they are executed. This gives the user the opportunity to modify problematic policies (whether new or existing). It is still possible for run-time conflicts to occur (e.g. due to competition for a resource), but these are relatively rare. In any case, a resident would probably not wish to be interrupted by execution-time problems (and may not even be around to handle them). For these reasons, offline conflict detection is preferable.

2.2 Homer Architecture

Homer builds on and extends OSGi (‘Open Services Gateway initiative’, *www.osgi.org*). OSGi is a dynamic modular system for Java that supports components called bundles. These have a simple interface which Homer enriches to allow many kinds of components (and devices) to be integrated. Bundles can be started, stopped and updated while the platform is running. Homer components make extensive use of the OSGi Event Admin service that allows asynchronous communication via an event bus. The major Homer elements shown in figure 1 are as follows:

Database: A lightweight relational database (H2, *www.h2database.com*) stores

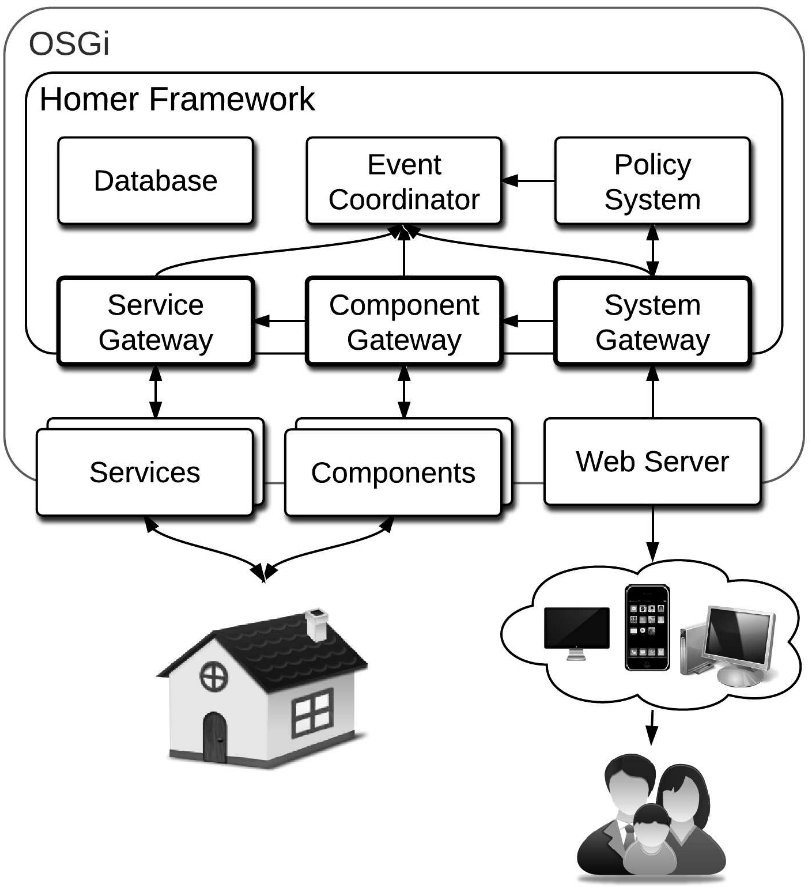

Fig. 1. Overall Homer Architecture

component and configuration data. Only the Homer framework has direct access to the database, partly for security and partly to isolate components from database details.

Event Coordinator: This posts events relating to policy triggers, conditions and actions, and allows components to register for events of interest. In particular, components can listen for classes of events (e.g. for a room or device type).

Policy System: This supports the storage and execution of user-defined rules for the home system. When policies are defined, they are checked for validity and conflict with other policies.

Service Gateway, Services: This gateway deals with communication between the Homer framework and services. A Homer service supports common code (e.g. for logging or communication) on behalf of a number of components. By treating this kind of functionality as a separate service rather than a component, it is possible to maintain the independence of components.

Component Gateway, Components: This gateway deals with communication between the Homer framework and component bundles. This includes dealing with component configuration, triggers, conditions and actions. A component offers self-contained functionality that typically deals with a class of device (e.g. X10 mains-controlled appliances or Visonic wireless sensors).

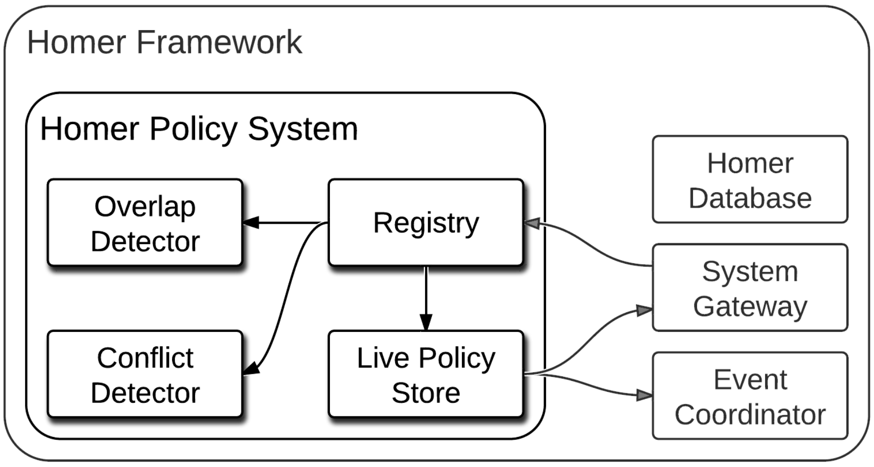

Fig. 2. Policy System Architecture

System Gateway, Web Server: This gateway deals with communication between the Homer framework and external entities via the Web Server. Configuration is also handled by the system gateway. The Web Server offers an HTTP-based interface for external monitoring and control. External interfaces exchange data with Homer in JSON format (JavaScript Object Notation, *www.json.org*).

2.3 Policy System

The policy system is independent of particular components and so is applicable to a wide variety of devices. This is possible because components *declare* their triggers, conditions and actions to the policy system. This makes the core system independent of devices since components inform the policy system of what they handle. Components also have the freedom to use absolute measures (e.g. temperature in °C) or subjective measures (e.g. ‘chilly’, ‘comfortable’, ‘hot’). The policies that can be written by a user automatically reflect what components declare about themselves. The policy language is therefore fully extensible.

Figure 2 shows an expansion of the policy system appearing in figure 1. The major elements of the policy system are as follows:

Registry: This deals with addition, deletion and editing of policies. It invokes functions in other elements to validate a policy and to apply an offline conflict detection algorithm.

Live Policy Store: This handles the storage and execution of currently enabled policies.

Overlap Detector: This checks a new or edited policy against existing policies. Apart from examining the validity of a policy it also checks if a policy might be simultaneously enabled along with an existing one. If so, they are considered to overlap.

Conflict Detector: This checks whether overlapping policies have actions that conflict with each other. As will be seen later, conflict detection relies on information that users provide about 'environment' effects of particular actions.

2.4 Policy Language

The structure of Homeric, the Homer policy language, is defined by the following grammar. Following a common grammar convention, non-terminal symbols are plain identifiers while terminal symbols are given in quotes. The operator '?' means optional, '*' means zero or more, and '|' separates alternatives.

The first part of a policy is a **when** clause that determines when it is triggered. A simple event is an external trigger or a condition on values. Events can be combined with **and**, **or** and **then** (for an event sequence). A time limit can be imposed on a group of events to allow the definition of higher-level events. For example, the macro event 'user returns home' might be defined as 'garage door opens **then** car enters garage **then** garage door closes **within** five minutes'.

policy : ″when″ event ″do″ execution ″.″ ;
event : simple_event | ″(″ compound_event ″)″ ;
simple_event : trigger | condition ;
compound_event : event (timed_event | or_event) ;
timed_event : ((″then″ | ″and″) event)* (″within″ duration)? ;
or_event : (″or″ event)* ;

The second part of a policy is a **do** clause that states what it does. Simple actions may be combined with **and**. Actions may also be made conditional. For example, when a policy is triggered by movement its action might be: '**if** the house is occupied **do** play music **else do** report an intruder'.

execution : simple_execution | ″(″ compound_execution ″)″ ;
simple_execution : action ;
compound_execution : and_execution | conditional_execution ;
and_execution : execution (″and″ execution)* ;
conditional_execution : ″if″ action_condition ″do″ execution (″else″ ″do″ execution)? ;
action_condition : condition | ″(″ (and_condition | or_condition) ″)″ ;
and_condition : condition (″and″ condition)* ;
or_condition : condition (″or″ condition)* ;

A trigger indicates the user-defined device causing it, the trigger name and optional parameters. An example trigger could be: hall sensor, temperature reading, value 20°C. Conditions and actions are similarly defined. Rather than using fixed strings as identifiers, Homer maps user-defined names to internal identifiers. This allows names to be later changed or to be rendered in different languages. For example the original name 'TV' might later become 'lounge TV' when a second TV is added, or might be termed 'la télé' by a French speaker.

trigger : user_device_id trigger_id (parameter)* ;
condition : user_device_id condition_id (parameter)* ;
action : user_device_id action_id (parameter)* ;
duration : /* unsigned positive integer */ ;
parameter : /* uninterpreted character string, e.g. ″12″, ″Alice″ */ ;
***_id** : /* uninterpreted unique character string, e.g. 984657651468 */ ;

For extensibility, the values above are intentionally not mandated by the policy language. Rather they are defined by components. For example, a door component might declare its triggers as ‘has opened’ and ‘has closed’, its conditions as ‘is open’ and ‘is closed’, and its actions as ‘open’ and ‘close’. Although triggers and conditions are distinguished internally, they can be used interchangeably in a policy.

2.5 Policy Examples

Example Homeric policies appear in figure 3. These are revisited in section 3.4 when a worked example is given. As explained later, there is a deliberate problem with the event clause of policy 5. Also note that the messages in policies 7 and 8 are treated as equivalent since letter case is unimportant in strings.

3 Home Policy Conflict Handling

This section describes the approach to offline handling of conflicts among home policies. An extended worked example explains the technique. Although conflict detection looks at only pairs of policies (a new policy and an existing one), the detection algorithm is commutative (if A conflicts with B then B conflicts with A) and associative (if A conflicts with B-and-C, then A-and-B also conflict with C). This means that a pairwise algorithm is guaranteed to detect conflicts among two *or more* policies (the analogue of the three-way feature interaction question).

3.1 Overlap Detection

3.1.1 Philosophy

Homer aims to be independent of particular devices and the user’s language. This means that only components, and not the policy system, embody device-specific information. This is why components declare their triggers, conditions and actions to Homer, along with methods to evaluate these.

Homeric policies naturally fall into two parts: **when** (event) and **do** (action). When

Id	**Policy**
1	**when** time is earlier than 8:30PM **do** turn on hall lamp
2	**when** time is 7PM **do** turn off hall lamp
3	**when** time is between 8PM and 10PM **do** open window
4	**when** washing machine turns off **do** turn on dehumidifier
5	**when** (front door is open **and** front door is closed) **do** turn on washing machine
6	**when** (front door is open **or** front door is closed) **do** turn on washing machine
7	**when** SMS received from Alice saying ‘On The Way Home’ **do if** indoor temperature is below 18°C **do** turn on gas central heating
8	**when** SMS received from Alice saying ‘on the way home’ **do** turn on gas oven **and** **if** indoor temperature is above 24°C **do** open window **else if** indoor temperature is below 15°C **do** turn on gas central heating
9	**when** (day is a weekday **and** time is 6:30AM) **or** (day is a weekend **and** time is 8:30AM) **do** turn on gas central heating
10	**when** ((front door opens **or** back door opens) **and** time is 5PM or later **and** lounge lamp is off) **do** turn on lounge lamp
11	**when** (day is a weekday **and** time is 7:30AM) **do** turn on immerser **and** open curtains
12	**when** time is 9:45PM **do** turn off TV
13	**when** (TV turns off **then** lounge lamp turns off) **do** turn on bedside lamp **and** close curtains
14	**when** SMS received from Alice saying ‘Start washing machine’ **do** turn on washing machine
15	**when** (day is Sunday **and** time is 11AM) **do** turn on washing machine
16	**when** (day is Sunday **then** washing machine turns off) **do** turn on dehumidifier
17	**when** (day is a weekday **and** time is 7:30AM) **do** turn on air conditioning
18	**when** (time is 8AM **or** time is 5PM) **do** send SMS to Alice saying ‘Feed cat’
19	**when** (indoor temperature is above 25°C **and** time is 2PM) **do** turn on lawn sprinkler
20	**when** ((curtain closes **then** bedside lamp turns on) **and** time is after 10PM) **do** turn on burglar alarm

Fig. 3. Sample Policies

a new policy is defined (or an existing policy is edited), it is checked for overlap with existing policies. That is, it is checked whether a new policy and existing policies can be simultaneously triggered (i.e. their event clauses can both hold at the same time).

A philosophical question is how literally to treat ‘at the same time’. There are clear cases where events could be simultaneously true (e.g. ‘the kettle is boiling’ vs. ‘the kettle is over 90°C’). There are clear cases of the opposite (e.g. ‘it is 9AM’ vs. ‘it is the afternoon’). However, there are also less clear-cut possibilities (e.g. ‘the front door opens’ vs. ‘the fire alarm goes off’).

A permissive view would consider independent events to be capable of happening at the same time. However, this could lead to the user being alerted to many possible conflicts that a commonsense view would exclude. In these circumstances, the user would be likely to ignore any conflict warnings. Homer therefore takes a pragmatic view and aims to report only plausible overlaps.

The approach identifies all common cases of overlap (e.g. ‘the front door opens’ vs. ‘the front door opens’). However, as noted earlier, Homeric deliberately blurs triggers and conditions. For example, ‘the front door opens’ and ‘the front door is open’ would be treated as equivalent by most users. However, the first of these is technically a trigger while the second is technically a condition. Component support of triggers and conditions deals with this situation, so it would be concluded that there is an overlap despite the different linguistic forms. Components also handle comparison of parameter values, e.g. numbers or strings.

Overlap detection aims to discover whether some combination of triggers and conditions can cause the event clauses of two policies to hold. This is treated as a constraint satisfaction problem: can the constraints imposed by the event clauses be simultaneously satisfied? If so, the policies can be triggered at the same time.

However, there are additional complications due to the richness of event clauses. For sequences to overlap, their common elements must be able to occur in the same order. Consider ‘the front door opens **then** someone enters **then** the front door closes’ vs. ‘the doorbell rings **then** the front door opens **then** the front door closes’. These are considered to overlap as their common events occur in the same order. The same type of event may also be repeated within a clause (e.g. ‘the front door opens **then** the TV is switched on **then** the front door opens’). In fact these repetitions refer to different instances of the same type of event. Care is therefore necessary to distinguish these, but also to relate them properly to the corresponding events in a second policy.

Consider two policies with event clauses ‘the front door opens **then** the front door closes’ and ‘the front door closes **then** the front door opens’. It might seem obvious that these do not overlap. However, now add a time constraint to both: ‘**within** 1 minute’. If the front door opens, closes, then opens again within a minute, then both policies could be simultaneously triggered. This is a somewhat pathological case, so again Homer takes a commonsense view and does not consider these to overlap.

Policy events can have parameters (e.g. ‘a message arrives from *Alice* saying *turn the heating on*’, ‘the *lounge* humidity is above *70%*’). Some parameter values belong to an enumerated set (e.g. on, off, dim) while others are drawn from an unlimited set (e.g. temperature). Overlap detection therefore needs to take parameters fully into account. For extensibility, parameters comparison is carried out by individual components and not by the policy system.

The technique for overlap detection also has an unexpected benefit. The event clause of a policy can be considered in isolation to find triggers and conditions that make the event clause hold. This allows an event clause to be checked for validity. If a policy cannot be triggered, the user is asked to correct this. As a simple example, consider the problematic event clause: ‘the day is Tuesday **and** it is the weekend’.

3.1.2 Overlap Detection through Constraint Satisfaction

Policy event clauses can become complex, with multiple sub-clauses and operators. It is thus non-trivial to decide whether two policies overlap. The event clauses are therefore mapped into constraints and analysed by a constraint satisfaction solver. As noted in section 1.2.3, the authors use JaCoP as an efficient, Java-based solution that can readily be integrated with Homer.

Each term in an event clause is translated into its JaCoP equivalent. An unparameterised event corresponds to a simple JaCoP variable. A parameterised event corresponds to a JaCoP variable with a range of values. Since JaCoP solves only integer programming problems, parameters must be mapped to integers. The values of an enumerated type are simply mapped to 0, 1, 2, etc. This includes string values since Homer is aware of all the strings used in policies (e.g. the SMS message ‘turn the heating on’ might be mapped to 3).

For continuous values a more complex mapping is required. To allow efficient searches for a solution, JaCoP requires variables to have a modest range of integer values (a few thousand). Real numbers are therefore treated as fixed-point values with limited precision. These are turned into integers, e.g. 23.1 is mapped to 231 (allowing for one decimal place).

JaCoP requires the minimum value of variables to be defined; for simplicity, this is always treated as 0 by the policy system. Minima depend on the application domain and the device, so there is no sensible default. Negative parameters must therefore be mapped with an offset. For indoor temperature, 0°C is a plausible minimum. However, the minimum outdoor temperature might be more plausibly -20°C. This would be treated as 0, so 19.2°C would be mapped to 392 (i.e. 19.2 plus offset of

20.0, allowing for one decimal place).

The operators **and**, **or** and **then** are translated into calls of (polyadic) Java methods that impose the relevant conditions on their parameters. The case of **then** requires special treatment. The time at which an event occurs needs to be recorded in the corresponding JaCoP variable. The absolute time of an event is unimportant, so events in a sequence are considered to occur at time index 0 for the first event, 1 for the second event, etc. A JaCoP variable therefore has two fields: the actual value of the variable, and the time index at which this value was established.

JaCoP has the notion of a 'store' that holds all the constraints to be satisfied. The constraints constructed from each event clause are imposed on the store, then JaCoP is requested to look for solutions. For policy purposes it is sufficient to find one solution in order to determine overlap; JaCoP reports the variable values for this solution. If subsequent conflict detection discovers contradictory actions, the variable values allow a concrete example to be given to the user of when conflict can occur. If JaCoP cannot find any values to satisfy the constraints then no overlap (and therefore no conflict) exists. As a concrete example, consider two policies with the following event clauses:

when day is a weekday **and**
  (movement is detected in drive **then** Alice opens garage door **then**
    movement is detected in garage **then** garage door closes **then**
    front door opens)

when (movement is detected in drive **then** someone opens garage door **then**
    movement is detected in garage **then** garage door closes **then**
    front door opens) **and**
  house temperature is at least 20°C

The result from JaCoP is that the constraints can be satisfied by the following variable assignments (with time indexes shown in parentheses): day is Monday (0), movement in drive (0), Alice opens garage door (1), someone opens garage door (1), movement in garage (2), garage door closes (3), front door opens (4), house temperature is 20°C (0). In the case of a timed/parameterised event, JaCoP reports the earliest time/lowest value that satisfies the constraints.

3.2 Conflict Detection

3.2.1 Philosophy

Since the actions of a policy self-conflict, it is necessary to check the actions of each policy in isolation. For pairs of policies whose event clauses overlap, it is also necessary to check whether their actions conflict. The work on action conflict was inspired by the work of Nakamura [9,10] and Wilson [3,20], though it is an extension of their ideas. If conditional actions are present, their conditions are considered part of the event clause for the purposes of overlap detection (see section 3.1.2).

In fact, action conflict can be quite subtle and even subjective. Suppose one policy would open a window while another would turn on the heating. It could be argued that these have contrary effects (cooling, heating) and are in conflict. However, the user might consider both actions desirable (fresh air, keeping warm). Closing the curtains and turning on a light might also appear to be contradictory (reducing light, increasing light). Yet the corresponding effects (negative, positive) should be considered acceptable.

As a result, it is not possible to automatically decide whether a conflict is significant or not. Since the nature of policy conflict can be subjective, the approach aims to identify conflict-*prone* policies. Since determining whether conflict exists often requires human judgment, cases of potential conflict are reported to the user for a decision.

Even if both effects are in the same direction, they may be considered undesirable if their cumulative effect is excessive. In some parts of the world there are limits on how much power a house should consume. Turning on both the washing machine (e.g. 3kW) and the air conditioning (e.g. 5kW) might cause this limit to be exceeded. As an extra complication, some effects do not add linearly. Suppose the washing machine increases background noise by 30dB while the air conditioning increases it by 25dB. The net noise power increases not by 30dB + 25dB but actually by 31.1dB because decibels are on a logarithmic scale.

Actions are assessed for conflict by their effect on ‘environment’ values. In some cases these could be considered as variables (e.g. light level, humidity) and in other cases as resources (e.g. power, water). The generic term ‘environ’ is used for something that an action can affect. Environs are not defined by the policy system. Rather, for generality, environs and effects on these are defined by users when configuring their home. For example, a user may state that turning on the washing machine increases power consumption by 3kW, water consumption by 25l, and noise level by 30dB. Environs can also be abstract concepts such as security or comfort. Homer currently expects the user to define environs for each device class. There are no default environs, but this will be supported in future (e.g power consumption would be an effect for all electrical appliances).

An environ is classified according to how effects on it are treated: *minimise* (effects on the environ should be minimised), *maximise* (effects on the environ should be maximised), *neutral* (effects should be both minimising or maximising), and *ignore* (the kind of effect is unimportant).

Example environs are listed in figure 4. Note that environs are *user-defined* and not intrinsic to Homer; this allows conflict detection to be independent of devices,

Environ	**Treatment**
audio	neutral
gas	minimise
light	neutral
humidity	minimise
noise	minimise
power	minimise
security	maximise
temperature	neutral
water	ignore

Fig. 4. Example Environs and Their User-Defined Treatment

application domain and user language. These environs are revisited in section 3.4 when a worked example is given.

3.2.2 Conflict Detection through Environmental Effects

The action clauses of overlapping policies are analysed for how their terms affect the environs. The conflict analysis leads to one of the following results for a pair of actions: *none* (e.g. ‘send email’, ‘turn on lamp’), *same* (e.g. ‘turn on lamp’, ‘turn on lamp’), *opposing* (e.g. ‘turn on lamp’, ‘turn off lamp’), and *possible* (e.g. ‘open window’, ‘turn on heating’). The analysis helps users to assess whether a conflict really exists or can be ignored as unimportant. Users will typically treat *opposing* as needing attention. The outcome *same* is likely to be regarded as harmless unless it unnecessarily repeats an expensive action (like sending a large text message by phone twice).

The analysis depends on whether the actions affect the same device (figure 5) or different devices (figure 6). For the same device, analysis also depends on whether the actions are the same (e.g. both ‘turn on’), are opposing (e.g. ‘turn on’ and ‘turn off’), or are otherwise different (e.g. ‘turn on’ and ‘dim’).

Whether an action is parameterised or not may affect the outcome. Unparameterised actions are plain ones like ‘call emergency services’ or ‘sound the alarm’. Examples of parameterised actions are ‘send a message to *Alice* about *a possible intruder*’ and ‘set the *lounge* temperature to *20°C*’. Finally, the sense in which an action affects an environ is also taken into account (‘+’ increasing it, ‘-’ decreasing it).

If an action affects multiple environs, the analysis is undertaken for each environ. It is possible for some of the effects to be conflict-free, but for others to suggest conflict. The user is therefore informed of cases where at least one environ suggests

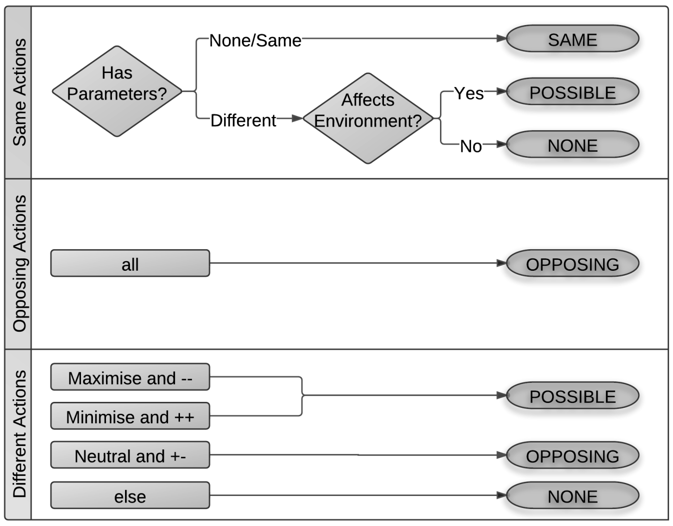

Fig. 5. Conflict Analysis for Two Actions on The Same Device

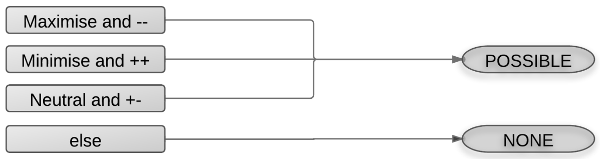

Fig. 6. Conflict Analysis for Two Actions on Different Devices

conflict.

As an example, suppose one action wishes to turn a floodlight on while another wishes to dim it. Both actions increase the light level, but light is neutral in figure 4. In figure 5, different actions, neutral and '++' lead to a 'none' judgment for conflict. However, both actions also increase power consumption (which could be significant for a floodlight). Power is to be minimised according to figure 4. In figure 5, different actions, minimise and '++' lead to a 'possible' judgment for conflict. Overall, there is possible conflict due to power (though not due to light). This will be reported to the user for a decision as to whether the conflict is significant.

3.3 Conflict Resolution

Because of the subjective nature of conflicts, the user is allowed to decide what is important and what should be ignored.

When a new policy is defined, the user may be informed of existing policies that it could conflict with. The user is also told of the events that lead to overlap and the reasons for actions conflicting. At this point the user can edit the new or existing policy, or can disable or delete the existing policy (which might have been superseded). The user can also save the new policy on the basis that the conflict is unlikely or will not have significant consequences.

The user can tell Homer that specific conflicts among the devices in the new and existing policies should be ignored (e.g. a power conflict between a radio and a lamp is insignificant). Alternatively the user can choose to ignore classes of conflict (e.g. light level conflicts involving lamps should be ignored as they are unimportant). This avoids the user being asked repeatedly in future how to resolve such conflicts.

3.4 Worked Example

The following example illustrates conflict analysis using the example policies in figure 2.5 and the environ definitions in figure 4.

Figure 7 shows what potential conflicts are reported among the policies; possible conflicts *a* to *i* are explained in figure 8. It is assumed that the policies are defined one-by-one in the order 1 to 20. The table columns from left to right show what happens as new policies are defined: the column height grows as more policies are added to the old policies. As an example, the column numbered 17 shows what happens when policy 17 is added to existing policies 1 to 16: possible conflict *i* is detected between policies 17 and 11.

Initially there are no policies, so defining policy 1 does not require conflict analysis. When policy 2 is added, it is reported that this conflicts with policy 1 (case *a*); one of the policies needs corrected. When policies 3 and 4 are added, no conflicts are detected. When policy 5 is defined, the user is asked to correct it because its event clause is invalid: ‘front door is open **and** front door is closed’ cannot be satisfied. As policies 6 to 20 are added, eight further potential conflicts are reported and dealt with by the user.

Case *b* is interesting because the new policy (8) has actions that self-conflict; there is also conflict with an existing policy (7). Similarly case *e* exhibits self-conflict in the new policy (13) and with an existing policy (10).

The possible conflicts are explained in figure 8. This shows the policy overlap condition, the analysis outcome, and the likely user reaction to each reported case. As will be seen, there is significant subjectivity in whether a possible conflict should be treated as genuine. This is why it is important for Homer to consult the user (and to record the user’s decision for future similar cases).

Following user reactions to possible conflicts, the end result is a set of acceptable policies. As conflict detection has been performed at definition time, the policies should execute without interference (in the user’s judgement). The only exception would be if policies conflicted at run-time due to competition for resources. However, this is unlikely as resource conflict should have been determined through the

New																				
1	**2**	**3**	**4**	**5?**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**Old**
	a																			**1**
									c											**2**
											d									**3**
															g					**4**
																				5
																				6
							b													**7**
													f							**8**
																				9
												e								**10**
																i				**11**
																				12
																				13
																				14
															h					**15**
																				16
																				17
																				18
																				19
																				20

Fig. 7. Analysis Results for Example Policies and Environs

analysis of environ effects (e.g. large demands for power or gas).

From a purely technical perspective, none of the conflicts in this example is particularly surprising. However, from the perspective of a home user the results are nonetheless useful. Conflicts that may be obvious to a developer may not be anticipated by a non-technical user. Over time, policies defined for the home can accumulate ‘dead wood’: older policies that are no longer relevant and conflict with new policies. The user may also not think of subtler conflicts such as the tumble drier interfering with the air conditioning keeping the humidity down. If several residents are allowed to define policies, conflict can also arise due to differing viewpoints. For example, one resident might like to keep all rooms at a temperature of 18°C but another might prefer to have public rooms warmer. These are all examples where the conflicts reported by Homer are useful to the user.

Case	Overlap Condition	Action Analysis	Likely Reaction
a	7PM	*opposing*: hall lamp on **and** off (opposite actions)	change policy
b	not applicable (self-conflict)	*possible*: gas oven on **and** gas central heating on (may exceed gas use limit)	change policy if gas use important
	homeward SMS **and** 0°C	*possible*: gas oven on **and** gas central heating on (may exceed gas use limit)	change policy if gas use important
		same: gas central heating on twice (duplicate actions)	ignore as harmless
c	front door opens **and** 7PM **and** lounge lamp off	*possible*: hall lamp off **and** lounge lamp on (opposite light effects)	ignore as harmless
d	9:45PM	*possible*: window open **and** TV off (both decreasing security)	change policy if security important
e	not applicable (self-conflict)	*possible*: bedside lamp on **and** curtains closed (opposite light effects)	ignore as harmless
	front door **and** 5PM **and** (TV off **then** lounge lamp off)	*possible*: lounge lamp on **and** curtains closed (opposite light effects)	ignore as harmless
f	washing machine SMS **and** homeward SMS **and** 0°C	*possible*: washing machine on **and** air conditioning on (may exceed power use limit)	change policy if power use important
g	Sunday **then** washing machine off	*same*: dehumidifier on twice (duplicate actions)	ignore as harmless
h	(Sunday **then** washing machine off) **and** 11AM	*possible*: dehumidifier on **and** washing machine on (may exceed power use or noise limit)	change policy if power use/noise important
i	Monday **and** 7:30AM	*possible*: immerser on **and** air conditioning on (may exceed power use limit)	change policy if power use important

Fig. 8. Possible Conflicts and Likely User Reactions

4 Conclusion

It has been argued that policies offer a practical way of allowing end users to specify how a home automation system should react to changing circumstances. The Homer system has been briefly introduced, along with the policy system that supports this. The Homeric policy language has been described, as well as the approach to offline conflict detection. A substantial worked example has shown how the system deals with conflicts as policies are defined. The subsequent execution of policies should be conflict-free.

The approach is independent of the application domain, the devices and the user's language. The key to achieving this is that components describe to Homer the devices they support. Although Homer has been designed and tested for home automation, the generality of the approach means that it should be applicable to other domains as well. However, one limiting factor may be that conflict analysis assumes a centralised system and hence complete knowledge of all policies. In a distributed setting (e.g. telephony) this would not hold. Alternative approaches to policy conflict such as [17] could therefore be needed.

Homer allows control, monitoring *and* programming of the home. The system architecture offers easy extension for new devices, automatically integrating these with the policy system. Homer also supports a wide variety of external user interfaces [8]. Homeric reflects the requirements expressed by users for home management policies [6]. An extensive evaluation has been carried out into the usability of Homeric policies [7]. This has demonstrated that ordinary users are able to grasp the policy-based approach, and can successfully write policies to control their homes. In particular, the use of perspectives has proven to be successful. Techniques and tool support have also been developed for handling conflicts among policies as they are defined.

The Homer system has been evaluated in a laboratory setting. Homer has also been used by independent developers to create components for new devices (the Microsoft Kinect and various multimedia interfaces). As the next major step, it is planned to deploy Homer into realistic home settings so that its usability and dependability can be assessed.

Various technical developments are planned for the future. Although Homer supports policy perspectives, these have not been fully developed in the current interfaces. Work is ongoing to extend the current iPad and iPhone applications, and also to create more comprehensive support using Android. Although offline conflict analysis is handled, online analysis and distributed conflict handling could also be desirable. It is planned to add the capability to explain policy conflicts in the same kind of way that expert systems explain their recommendations.

Other extensions include enhancing the basic support for sensor and actuator fusion, so a more comprehensive solution will be investigated (using macro events and actions, or along the lines of [15]). It would be desirable to support fuzzy policies so that the system can deal with uncertain information (e.g. 'it will *probably* be cold tonight' and 'the user *may* have fallen'). A library of pre-defined policies would be a useful future addition. For convenience in defining environs, likely defaults for known device types will be added.

Acknowledgements

Claire Maternaghan was financially supported by the Scottish Informatics and Computer Science Alliance and the University of Stirling.

References

[1] 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.

[2] S. Keoh, K. Twidle, N. Pryce, E. Lupu, A. S. Filho, N. Dulay, M. Sloman, S. Heeps, S. Strowes, and J. Sventek. Policy-based management for body-sensor networks. In *Proc. 4th Int. Workshop on Wearable and Implantable Body Sensor Networks*, pages 92–98, Aachen, Germany, Mar. 2007.

[3] M. Kolberg, E. H. Magill, and M. E. Wilson. Compatibility issues between services supporting networked appliances. *IEEE Communications Magazine*, 41:136–147, Nov. 2003.

[4] A. F. Layouni, L. Logrippo, and K. J. Turner. Conflict detection in call control using first-order logic model checking. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 66–82. IOS Press, Amsterdam, Netherlands, May 2008.

[5] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. on Software Engineering*, 25(6):852–869, Nov. 1999.

[6] C. Maternaghan. How do people want to control their home? Technical Report CSM-185, Computing Science and Mathematics, University of Stirling, UK, Dec. 2010.

[7] C. Maternaghan. Can people program their home? Technical Report CSM-191, Computing Science and Mathematics, University of Stirling, UK, Apr. 2012.

[8] C. Maternaghan and K. J. Turner. Pervasive computing for home automation and telecare. In S. I. A. Shah, M. Ilyas, and H. T. Mouftah, editors, *Pervasive Communications Handbook*, pages 17.1–17.25. CRC Press, Boca Raton, Florida, USA, Nov. 2011.

[9] M. Nakamura, H. Igaki, and K. Matsumoto. Feature interactions in integrated services of networked home appliances. In S. Reiff-Marganiec and M. D. Ryan, editors, *Proc. 8th Int. Conf. on Feature Interactions in Telecommunications and Software Systems*, pages 236–251. IOS Press, Amsterdam, Netherlands, June 2005.

[10] M. Nakamura, H. Igaki, Y. Yoshimura, and K. Ikegami. Considering online feature interaction detection and resolution for integrated services in home network system. In M. Nakamura and S. Reiff-Marganiec, editors, *Proc. 10th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 191–206. IOS Press, Amsterdam, Netherlands, June 2009.

[11] G. Russello, C. Dong, and N. Dulay. Authorisation and conflict resolution for hierarchical domains. In *Workshop on Policies for Distributed Systems and Networks*. Institution of Electrical and Electronic Engineers Press, New York, USA, 2007.

[12] M. H. ter Breek, S. Gnesi, C. Montangero, and L. Semini. Detecting policy conflicts by model checking UML state machines. In M. Nakamura and S. Reiff-Marganiec, editors, *Proc. 10th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 59–74. IOS Press, Amsterdam, Netherlands, June 2009.

[13] 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.

[14] E. P. K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.

[15] 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. IEEE Computer Society, Los Alamitos, California, USA, May 2010.

[16] K. J. Turner. Flexible management of smart homes. *Ambient Intelligence and Smart Environments*, 3(2):83–110, May 2011.

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

[18] 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.

[19] F. Wang and K. J. Turner. Policy conflicts in home care systems. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 61–76. IMAG Laboratory, University of Grenoble, France, Sept. 2007.

[20] M. Wilson, M. Kolberg, and E. H. Magill. Considering side effects in service interactions in home automation – An online approach. In L. du Bousquet and J.-L. Richier, editors, *Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 172–187. IOS Press, Amsterdam, Netherlands, May 2008.