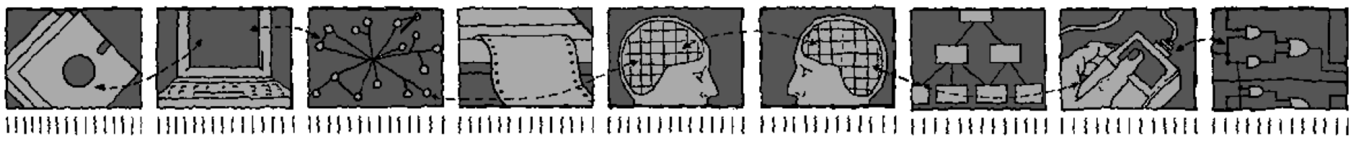
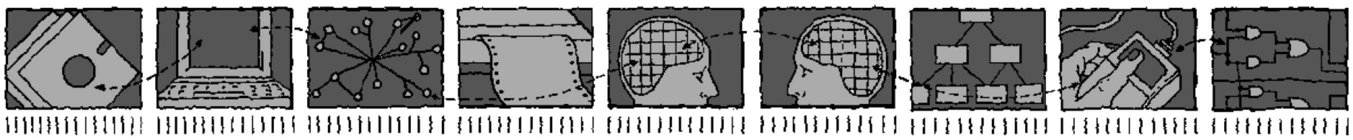


Department of Computing Science and Mathematics
University of Stirling



A System for Controlling, Monitoring and Programming the Home

Claire Maternaghan

Technical Report CSM-193

ISSN 1460-9673

September 2012

Department of Computing Science and Mathematics
University of Stirling

A System for Controlling, Monitoring and Programming the Home

Claire Maternaghan

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

Telephone +44 1786 467 421, Facsimile +44 1786 464 551
Email cma@cs.stir.ac.uk

Technical Report CSM-193

ISSN 1460-9673

September 2012

ABSTRACT

As technology becomes ever more pervasive, the challenges of home automation are increasingly apparent. Seamless home control, home monitoring *and* home programming by the end user have yet to enter the mainstream. This could be attributed to the challenge of developing a fully autonomous and extensible home system that can support devices and technologies of differing protocols and functionalities.

In order to offer programming facilities to the user, the underlying rule system must be fully independent, allowing support for current and future devices. Additional challenges arise from the need to detect and handle conflicts that may arise among user rules and yield undesirable results. Non-technical individuals typically struggle when faced with a programming task. It is therefore vital to encourage and ease the process of programming the home.

This thesis presents Homer, a home system that has been developed to support three key features of a home system: control, monitoring and programming. Homer supports any third-party hardware or software service that can expose its functionality through Java and conform to the Homer interface. Stand-alone end user interfaces can be written by developers to offer any of Homer's functionality.

Where policies (i.e. rules) for the home are concerned, Homer offers a fully independent policy system. The thesis presents a custom policy language, Homeric, that has been designed specifically for writing home rules. The Homer policy system detects overlaps and conflicts among rules using constraint satisfaction and the effect on environment variables.

The thesis also introduces the notion of perspectives to ease user interactivity. These have been integrated into Homer to accommodate the range of ways in which a user may think about different aspects and features of their home. These perspectives include location, device type, time and people-oriented points of view. Design guidelines are also discussed to aid end user programming of the home.

The work presented in this thesis demonstrates a system that supports control, monitoring and programming of the home. Developers can quickly and easily add functionality to the home through components. Conflicts can be detected amongst rules within the home. Finally, design guidelines and a prototype interface have been developed to allow both technically minded and non-technical people to program their home.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ACCENT	Advanced Component Control Enhancing Network Technologies
ACCORD	Administering Connected Co-Operative Residential Domains
ACHE	Adaptive Control of Home Environments
ADL	Architecture Description Language
API	Application Programming Interface
APPEL	Adaptable and Programmable Policy Environment and Language
BPEL	Business Process Execution Language
CAMP	Capture and Access Magnetic Poetry
CEDIA	Custom Electronic Design and Installation Association
CORBA	Common Object Request Broker Architecture
CRESS	Communication Representation Employing Systematic Specification
CTT	Concur Task Trees
ECA	Event-Condition-Action
EPSRC	Engineering and Physical Sciences Research Council
GUI	Graphical User Interface
HAI	Home Automation, Inc.
HCI	Human Computer Interaction
HTTP	Hyper-Text Transfer Protocol
HTTPS	Hyper-Text Transfer Protocol Secure
IBM	International Business Machines
ID	IDentifier
IDE	Integrated Developer Environment
INCA	Infrastructure for Capture and Access
iOS	iPhone Operating System
IP	Internet Protocol
iROS	Interactive Room Operating System
IT	Information Technology
JaCoP	Java Constraint Solver
JSON	JavaScript Object Notation
JTP	Java Theorem Prover
MATCH	Mobilising Advanced Technologies for Care at Home
OHF	Open Healthcare Framework

OSGi	Open Services Gateway initiative
PC	Personal Computer
PCOM	Pervasive COMputing
PDA	Personal Digital Assistant
PHP	PHP Hypertext Processor
PROSEN	Proactive Condition Monitoring of Sensor Networks
PVR	Personal Video Recorder
RECAP	Rigorously Evaluated Conflicts Among Policies
RFID	Radio-Frequency IDentification
RMI	Remote Method Invocation
SAT	Service Activator Toolkit
SCA	Service Component Architecture
SHA	Secure Hash Algorithm
SHAKE	Sensing Hardware Accessory for Kinaesthetic Expression
SMS	Short Messaging Service
SOA	Service-Oriented Architecture
SODA	Service-Oriented Device Architecture
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCA	Trigger, Condition and Action
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URL	Uniform Resource Locator
VCR	Video Cassette Recorder
WD	When-Do
WS	Web Service
XML	eXtensible Mark-up Language

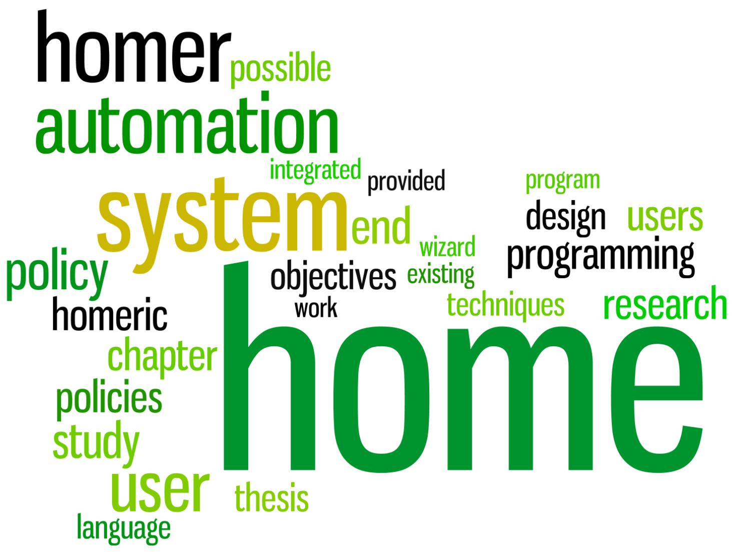
TERMINOLOGY

Case Study	Describes the different evaluations performed in Chapters 3, 4 and 5. Typically these case studies were small illustrations or modest evaluations.
Event	Is used throughout this thesis to mean a trigger, condition or action (TCA).
Policy	Within this thesis, a type of rule in the format *when* something happens or is true *do* something.
Web Server	Is used to describe the Homer HTTP API.

Part I

INTRODUCTION

1 INTRODUCTION



This chapter will introduce the context, motivations, objectives and research contributions for this thesis. Additionally, the structure of the thesis is given.

1.1 CONTEXT

The home forms the integral central core of everyday life. This is where our day starts and ends, and our homes should exist to make our lives as comfortable as possible.

Many people would like help to perform the mundane and routine home tasks, leaving them more time to enjoy life. No one can carry out banal tasks as well as a machine, and here enters home automation.

A home automation system should weave itself into everyday home life, working tirelessly behind the scenes to help and ease daily chores for the residents. Turning lights off behind you, ensuring all the doors are locked when you are asleep or at work, maintaining a comfortable household temperature when occupied. These are all examples of what a home automation system could do for you, and there are endless other possibilities.

1.2 MOTIVATION

Home automation has been something of a science fiction fantasy for decades. Never before has it been as technically feasible as it is today. The past five years alone has seen pervasive computing become a reality, with many homes now overrun with smart phones, tablet computers, laptops, web-enabled television sets, and home computers.

The time is ripe to make the home automation fantasy come true. Leading worldwide companies, such as Google and Microsoft, are spending billions of dollars to give this research the push that it requires to transform it from possibility to reality.

Currently, approaches are emerging from two polar extremes. One which focuses on a single task, such as home security systems, and another which aims to integrate available tasks into one coherent home system.

Single task solutions can offer highly sophisticated functionality for end users. However, these are ultimately limited in scope and cannot be combined with other solutions.

Home systems, on the other hand, offer users a range of functionality for their home through a unified interface. Such an example is Control4, a leading home automation system available on the market, which combines various lighting, security and media systems for the user.

There exists a strong desire for home customisability and programmability, to allow the home to be tailored to individual needs [17, 125]. Despite this desire, existing solutions available do not allow programmability by end users. Instead, home systems typically require trained professionals to install, maintain and customise the system which is both inconvenient and expensive.

A user customisable home system brings new challenges to the typical home solutions available by enabling programmability by untrained, and often non-technical, users. Research needs carried out to help ease the task of programming the home for users, from the underlying language to the end user programming interface techniques.

Through developing a flexible home automation system which can be easily setup, maintained and programmed by the end user the need for expensive system installers and pricey customisation and maintenance costs can be substantially reduced. By lowering costs, and offering a system which can easily be extended over time, home automation becomes an affordable and realistic option for many households.

1.3 OBJECTIVES

The primary objective of my research is to design and develop a home system which can be used to address the desire for flexible, yet easy to use, customisation for the home.

The work presented in this thesis aims to address the following list of objectives:

- Design a flexible system to support the vast range of existing hardware and software for the home, as well as future devices through third-party support. The system must allow a means of flexibly combining the functionality of these devices at a higher level.
- Design a language for both technical and non-technical users which can allow the combining of the home functionality in an expressive, flexible and unambiguous form.
- Design advanced offline detection mechanisms. As there are high chances of conflicts between user written rules (policies) for the home these must be detected and reported to the user at the time of writing and saving a policy.
- Offer end user programming techniques to expose the custom home language to both technical and non-technical users, since the policies must be expressible from a very high-level by an extremely wide range of users.

These objectives are reviewed in Chapter 6.

1.4 RESEARCH CONTRIBUTIONS

The following is a list of research contributions that are made throughout this thesis, as an outcome of the objectives stated in Section 1.3.

- **Custom Home Policy Language** A new custom policy language is presented within this thesis to provide a platform for customising and programming the home. The language is custom designed for the home, emerging from extensive reviews of existing approaches coupled with tailored user studies. The language extends the existing state of the art by offering novel policy language features such as the blurred distinction of triggers and conditions, optionally ordered terms, and conditional actions.
- **Novel Policy Overlap Technique** To improve the accuracy of the offline policy conflict detection and to aid the user understand any presented conflicts at definition time, a technique is presented which can detect if two or more policies overlap with one another. If a policy is considered to overlap with another, then the two policies are analysed for conflicts. If, on the other hand, the policies do not overlap, then these are not considered to conflict and therefore would not be analysed. This reduces the number of unlikely conflicts presented to the user. The policy overlap technique presented is unique within the policy domain, therefore advancing the state of the art.

- **Advancements in Policy Offline Conflict Detection** Existing approaches to conflict detection within the home are primitive and rigid. Advancements to these existing techniques are presented which improve the accuracy of conflicts detected, as well as allow users to personalise what is considered a conflict within their home. The conflict detection work presented makes use of customisable environment variables and resources to obtain a more accurate understanding of how various actions effect the home.

In summary, this thesis presents a customisable home system by allowing the user to effectively program their home. This is achieved through the design of a home policy language to enable the functionality of the various hardware and software in the home to be combined flexibly. The supporting policy engine enforces this language and offers an advanced two-stage offline conflict detection process. The first stage involves novel policy overlap detection techniques, whilst the second offers personalisable conflict detection techniques.

1.5 STRUCTURE OF THESIS

This thesis contains the following seven chapters:

- **Chapter 1 – Introduction**
  Introduces the thesis, providing a summary of context, objectives and achievements.
- **Chapter 2 – Background**
  Introduces the background to home automation and telecare, and describes the user studies performed within this research. The studies are analysed to produce three collections of requirements, addressed in Chapters 3, 4 and 5.
- **Chapter 3 – Homer: Architecture**
  Describes the underlying architecture of Homer. This includes a full review of existing home systems and service-oriented frameworks. The design and implementation details are then described, providing an explanation of how Homer was built and operates. The work presented in this chapter offers a test-bed for the policy work within Chapter 4. By researching and developing a dynamic and programmable home system, policy research can be grounded in a concrete way to allow strong research contributions to be made.
- **Chapter 4 – Homer: Policies**
  Describes the Homer policy system's ability to handle rules within the home. This involves Homeric (the Homer policy language) whose language specification and representation are provided. Details are provided of how a policy can be validated and checked if and how it overlaps with existing policies, and secondly how any policies which it may conflict with are reported. A case study is also provided, evaluating the success of Homeric with end users.
- **Chapter 5 – Homer: Programming the Home**
  Describes the research and development carried out in regard to programming the home. Design guidelines are provided, as well as the notion of perspectives. Again, a case study is given to evaluate the work presented. This chapter aims to produce a set of guidelines which express the most successful end user programming techniques for programming the home, for both technical and non technical individuals, which will emerge from existing end user programming research. This work, therefore, provides a means of evaluating the policy research from Chapter 4.
- **Chapter 6 – Conclusions**
  Concludes the work presented in this thesis by exploring the strengths and weaknesses of the work, reviewing the extent to which the research objectives have been met, and discussing possible future work directions.

2 BACKGROUND

This chapter discusses the background and requirements for the research presented in this thesis.

2.1 INTRODUCTION

The notion of "automating" tasks through the advances of technology dates back to the 1950s, with the commercialisation of computer technology [35]. Since then, despite early fears of increasing control of "electric brains" [35], we have witnessed notable impact on a wide range of industries. Telephone switchboard operators, who were at one stage ubiquitous, are now largely replaced by automated telephone exchanges and answering machines. In the manufacturing industry it is claimed that, as a result of technology, productivity rose roughly 250 percent [98]. To get a premium car out of the driveway it now takes 100 million lines of code running on dozens of microprocessors [26], automating many of the required tasks of early cars. These form just a few examples of the many industries which have adopted computing technology, enhancing and automating their products and manufacturing processes.

Technology is becoming ever more pervasive. With the falling price of technology, and the general public's increase in technical ability, interest and acceptance, the possibility of automating and enhancing our daily lives within our home is becoming more and more feasible.

There now exist a large number of academic researchers and commercial companies developing specialised hardware, software systems and user interfaces for home automation.

This chapter introduces the state of the art in home automation. User studies performed for this research project are discussed, leading to a collection of requirements to be tackled in later chapters.

2.2 HOME AUTOMATION

2.2.1 *Introduction*

The terms "home automation" and "smart homes", used interchangeably, imply some level of sophistication and intelligence in the home itself. However, this is far from the case. The term "automation" gave rise to a lot of semantic confusion in the early 1950s, when the term was first used in relation to computer technology. A definition was settled upon as:

> *Automation simply means something significantly more automatic than previously existed in that plant, industry or location.*
>
> Bright, 1958 [35]

Within the home this confirms the notion that homes themselves are not "automated" or "smart". Instead tasks can be automated through the use of a computerised system which, without such a system, is not possible.

A "smart" or "automated" home describes a home which has a number of technologies connected to some central system. The system can be used to offer monitoring and control capabilities locally or remotely. The automated aspect can be achieved by combining the individual pieces of functionality of the various installed technologies in flexible ways to result in "rules" or "policies" for the home. These rules offer a mechanism for the home system to automate tasks that were previously performed manually. Augusto documents the history of smart environments and discusses current technical challenges that researchers and developers face [7].

Within this thesis, we refer to automation as "programming". This term best describes the concept of being able to combine the functionality available in the home at a higher level, to produce custom, richer functionality. The automated home, in effect, is one which is programmable. This differs from the concept of "configurability", which is purely a concept of parametrisation. Certain variables and values of the home can be altered by the user to customise the home system. For example, setting the home to use metric units instead of imperial or to use 24-hour time. Programming, on the other hand, is allowing the end user to define control logic of the system. For example, making the heating come on when the front door opens after 6PM on Mondays and Wednesdays.

This section discusses the motivation for automated and smart homes, introduces the leading research, examples and companies working within this field, and finally closes with an analysis of the state of the art.

2.2.2 *Motivation*

On the whole, people are feeling "busier" than ever before [42]. The focus of "busyness" can be careers, families or social activities [33]. Regardless, if industries can successfully make use of technological advances to enhance and automate the production process, why can we not obtain these same benefits in our daily lives?

As technology continues to seep into our homes, so too does the desire to make use of its functionality to help streamline and automate many of our daily tasks (the user study performed in Section 2.3.1 confirms this desire).

Ultimately, with the combination of busier lives, ubiquitous technology and increased technical abilities, the real question is – why would we not want to ease and automate our daily lives through the use of technology in our homes?

2.2.3 *State of the Art*

A thorough overview of the history of home automation systems over the years can be found in [11] and [49]. This section aims to categorise and summarise the most relevant of work, with an analysis in the following section.

LIVING LABS There is currently a large amount of research into living labs which can allow experimental work to be carried out in a more realistic setting. Examples include Mozer's adaptive house [29], Georgia Tech Aware Home [61], Orange [49], eHome [69] and MIT's

House_n [54]. These labs provide a test-bed for research into, for example, networking and protocols within the home, reliability of home control systems and trialling of new user interfaces. In addition, user trials can take place whereby users are asked to effectively live in the lab for a set time. This offers a means of evaluating the research in a safe and controlled environment, with the benefits of a more realistic setting.

NETWORKS Home networks is also of popular focus [27, 47, 103], along with their management and access control issues [62, 85]. The goals are to design common protocols and standards for device communication, enhance and integrate existing protocols, increase general device security within the home and simplify the concepts for end users.

USER STUDIES User studies into social and acceptability issues surrounding some areas of home automation help researchers better understand the current problems. For example, easing network management [48, 114], diversity between households [47], and the role of the technology "guru" of the home [103, 108].

HOME SYSTEMS At the heart of such home automation research is the underlying home system itself. Currently this research typically resides in one of three main categories: living labs, telecare research and commercial solutions. The most notable are now discussed.

Control4 (www.control4.com) is a leading home automation company in both Europe and the US, and offers solutions all over the world. It currently has custom integrators and distributors in over 70 countries worldwide [28]. Control4 offers tailored home automation packages to suit individual home owners needs, at "affordable" prices. However, in reality these solutions are luxury treats for fortunate individuals, and are unlikely to be found in the average home any time soon. Although the technology costs themselves are reducing, with Control4 much of the cost is found in the design, installation, customisation and maintenance of each individual package.

Cortexa (www.cortexa.com) is an American company which offers customers control over the home. The company has integrated its system with a small range of the most popular technologies in home automation, such as Home Automation, Inc. (HAI) (mentioned below) and Insteon (www.insteon.net, a leading manufacturer of heating, lighting and home sensor control for the home). Cortexa have manufactured custom computer boxes to house the system and a 15-inch touch-screen to interact with and manage the home. Cortexa have also developed an iPhone application and a web interface. The company has tried to stand alone from others in the market, which has resulted in isolating itself. Customers are offered only the Cortexa user interfaces, and must purchase only Cortexa-compatible hardware. When other companies update or add new hardware to their collection, Cortexa has to update its own code to provide these updates or to add functionality for their users. This is unsustainable and can only lead to frustrated customers and out-of-date software.

MATCH (*Mobilising Advanced Technologies for Care at Home*, www.match-project.org.uk) is a collaborative research project between the Universities of Dundee, Edinburgh, Glasgow and Stirling. It is primarily focused on home care, with four main strands of research: monitoring and analysis of resulting home care data, speech recognition and synthesis, multimodal interaction, and finally the use of policies and ontologies within the home care field.

The MATCH home system is designed and based upon research into home care and the unique problems it brings. For example, MATCH offers the ability for multiple stakeholders to define desired actions that should take place. It can then intelligently and appropriately handle requests from these different stakeholders, calculating what should occur and when, whilst ensuring all goals and restrictions that may be set are adhered to [87]. The MATCH home system was the inspiration for the spin-off project MultiMemoHome (www.multimemohome.org), which explores effective reminder systems within home care.

OmniQare (`www.omniqare.com`) is a leading system within Europe promoting independent living, and encouraging and enabling people to live in the comfort of their own homes for longer. The system aims to keep the ageing population informed, independent and safe. OmniQare comes packaged on a purpose-built all-in-one touch-screen Personal Computer (PC) offering 6 main strands of services: safety, communication, shopping, services, contact and well-being. Various devices can be connected to the system to enhance its functionality. For example, the user could connect sensors to measure and share their health details (such as heart-rate or blood pressure). OmniQare also incorporates a small application store (similar to the Apple "app store" or Android "market place"), which offers individual services to further enhance the system, including games, radio players and specialised communication applications.

2.2.4 *Analysis*

The technical feasibility for home automation has existed for decades, so why then has it not been widely adopted? This question is posed by Torbensen [125] and Brush *et al.* [17], who both aimed to gain insight into the situation. Torbensen conducted a comprehensive analysis of the market and Brush organised structured home visits with a range of households who currently live with home automation. Both provided useful and insightful information regarding many of the barriers of current home automation solutions which are now discussed.

- **Cost** The first major problem observed was the cost. Either in terms of installation, customisation and maintenance for those with outsource setups from commercial companies, or in terms of time for those custom installs by the technically capable. In the case of commercial companies there is a large premium on the base cost of the technology which is unavoidable for those less technically capable households. This cost, however, typically buys a more coherent experience for the users and removes a lot of the maintenance and customisation issues that can arise.

  Although cost is a current issue within home automation, it is not really one which further research can aid. Overtime technology costs will fall as popularity and adoption increases.

- **Disjointed Hardware** The second major problem leads on from this notion of cost and time and focuses on the raw hardware itself. The non-outsourced homes reported how disjointed the market is and how challenging it can be to integrate the available hardware to exploit the full potential of an automated home. In many cases these highly technically capable individuals had failed to successfully combine the various sub-systems within their home due to compatibility and integration issues. Companies such as Control4 and Cortexa spend a lot of resources trying to integrate and support the wide range of home technology available on the market. This leaves those more technically capable individuals, who are unwilling to pay premium costs, stuck with disconnected sub-solutions.

  This is a two-fold problem: one of hardware and one of software. The notion of using one common protocol for device communication within the home has been a goal for decades, with the dream that all hardware within the home can interact using the same means of communication [3, 37, 119, 122]. Software research, on the other hand, has learnt to adapt to the realisation that the home is full of devices which communicate using different protocols and technologies. Much work has been carried out to explore possible hardware-software bridge layers (typically termed "glueware") [25, 64] and service oriented architectures [44, 86, 109, 111] as a means of supporting this ever-widening range of technology within the home.

- **Inflexibility** Another major frustration is the lack of customisation options available. Those who outsourced and made use of solutions offered commercially felt trapped, stuck with an inflexible system which required professional aid for every slight adjustment or alteration. On the other hand, those who had chosen the independent route were faced with the integration challenges and the complexity of customisation across the various manufacturers and technologies.

Inflexibility and rigidity in home automation solutions is a major unsolved problem, and one which industry is not keen to address (likely due to business reasons – the less the customer can do for themselves, the more they must pay the company). At the technical level there are multiple problems to solve: firstly a platform is required which can allow communication with a vast range of hardware within the home; secondly some means of exposing the functionality available in the home; thirdly, a means of combining this functionality in a flexible way; and finally providing a way that the user can perform the customisation by combining the functionality available in their home. Within academia, research has focused on the general hardware and architecture aspects [44, 86, 109, 111] mentioned in the first two problems. However, the latter problems of combining the functionality within the system and by the user is lacking.

- **User Interfaces** In addition to the lack of customisation made available to the user, the user interfaces are typically extremely complicated and overloaded. This is apparent when looking at the interfaces of Crestron, Control4, Cortexa and many of the other leading home automation systems available [75].

  Research has explored many avenues for possible user interaction within the home, for example through gesture [52, 120], remote control [97], voice [40, 41, 145, 146], and web [30, 99]. However, less research has been conducted to explore how best to expose end user programming functionality to the user. Currently, there exist early explorations into the end user programming techniques of demonstration [36], natural language [40, 126], tangible [107] and visual [117].

- **Security** A final concern to the participants regarding their home automation solutions was that of security. This often came down to personal preference of the end user, with some preferring to not expose any functionality of their home externally whilst others were happy to, for example, control their front door and access security cameras remotely. Such personal preference varied regardless of technical competence.

  Security should be of high importance for any home automation solution, and as such has been researched for decades [6, 12, 51]. However, more research needs carried out to help simplify and ease the stigma and issues of security for the end user.

2.2.5 *Summary*

Currently, there exist home systems which provide monitoring and control capabilities for users. However, only a handful of companies additionally offer automation aspects, confirming the abuse of the phrases 'home automation' and 'smart home'. For those systems that do offer automation, most are programmed by the developer at installation time. The few systems remaining, which actually allow users to program their own home, are overly complicated and crude. This leaves customers limited, and therefore frustrated.

Another major problem with existing home systems is the lack of functionality. Many solutions support only a limited subset of available hardware, therefore each system limits the user to predefined manufacturers and restricted capabilities. This same limitation can be found in the user interface to the home, where users typically can only make use of one company's interface.

Control4 is, by far, the most advanced and flexible solution currently available. It allows manufacturers to integrate their technology with the Control4 system, both for underlying home functionality and user interactions. The main drawback with Control4, however, is not only the premium cost, but the lack of customisation available for the user. All automation is written by developers for each installation and is therefore costly and inconvenient for the user to tweak or customise their home's behaviour.

To summarise, the notion of a "smart home" is proven technically possible by the existing range of systems. Sadly, however, there is no one solution which allows the user freedom to use any desired technology and user interface, or the flexibility to program their home.

2.3 USER STUDIES

Three user studies were carried out over the duration of this research project and are discussed in this section. Requirements gathered from the studies are documented in Section 2.4. The

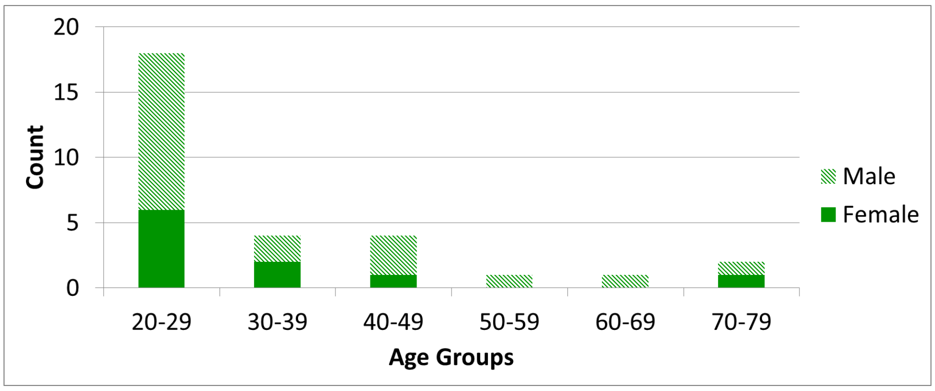


Figure 2.1: Age and Gender Distribution for User Study 1: What People Want

first user study explored desired functionality of a smart home. The second, more in-depth, user study built upon the findings of the first, and examined user preferences for general control and methods of interaction with a hypothetical smart home. The final study provided feedback regarding design decisions about the Homer policy language, to verify the suitability of the chosen approach.

2.3.1 *User Study 1: What People Want*

This preliminary user study was carried out in June 2009. It aimed to investigate general desired functionality that technology could provide within a home. This was achieved by asking individuals the following question:

"Is there anything you'd like your house to be able to do for you?"

This question was intended to return open-ended feedback, without constraints and guidelines. This format encouraged imaginative and creative thinking and maximised the variety of responses received. A small number of interviews were carried out, as well as sending targeted emails, inviting people to participate by answering the question. These people included family members, work colleagues and employees from a couple of local IT companies. Of these potential respondents, thirty responded. The range of age and gender of the respondents is shown in Figure 2.1.

Of the responses received there were some general similarities both amongst and across age ranges. Within the younger age brackets (20 – 29 and 30 – 39) there was much more desire for functionality that would improve the ease and efficiency of busy everyday lives. Examples include:

- **Everyday Tasks**: numerous respondents expressed the wish for automation or help with everyday tasks within the home. Recurring examples included watering plants, maintaining food stocks and feeding pets.
- **Reminders**: there was a clear desire for integrated reminders and relevant notifications in people's everyday lives. These ranged widely, including grocery reminders when passing a shop, home insurance up for renewal, car running low on petrol, reporting breaking news, needing a card for a partner's birthday and dental appointments. One respondent concluded her list of desired reminders with "you get the idea - reassurance for the absent-minded and faintly neurotic".
- **Flexible Appliance Control**: continuing the idea of rushed daily lives, many people described how they would like to streamline their days with more automated features and advanced means of appliance control. Some examples include switching on the coffee machine from bed, turning on the oven on the way home from work, automatically locking the front door as you leave, opening curtains when it is time to rise, and running bath water to desired parameters.

Respondents of a younger demographic also expressed interest in general energy efficiency, monitoring and management. Such examples include having the heating only on in occupied rooms, remotely switching on and off appliances, making use of cheaper energy tariffs at night and automatically reporting energy usage data to the energy supplier (to improve billing accuracy). Generally this demographic showed a lot of interest in energy, with one respondent saying upfront that they would like: "all the technology to be as energy-efficient as possible."

As the age bracket increased, so did the desire for advanced heating control: one respondent summed this up by writing "why are central-heating timers so utterly stupid?!". Respondents described how they would like to have control of their heating remotely, as well as being dependent on their physical location (both internally in occupied rooms, and externally based on proximity to the home). A more advanced response described the desire for the environment within their house to be kept "comfortable", defining comfortable as temperature, humidity, lighting, ventilation and sound level.

Due to the limited sample size of the older age bracket, detailed conclusions cannot be drawn. However some general themes were identified, including the desire for information about local events, special offers of relevance (such as local cheapest priced goods), health issues (incorporating appointment scheduling and receiving updates on local flu vaccinations) and important news.

The only feature that was desired by all age brackets was general peace of mind features. Examples include security alerts when away from the home and notifications of any appliances left on or windows or doors unlocked when leaving the home or going to bed.

In addition to the main themes there were some other interesting responses. These included the following categories, with examples:

- **Media**: multi-room playback, global control, intelligent music choice based on mood and people present.
- **Lighting**: lighting modes, intelligent control, dimming to act as a reminder.
- **Security**: status reports (particularly when on holiday), contact in case of anything untoward happening.
- **Food**: keep the house stocked with food normally bought, suggest recipes based on ingredients within the home, reminder of what groceries are needed when near a supermarket.
- **Social**: digital whiteboard for shared family communications, local events, new film releases.
- **Reminders**: shopping, appointments, bills, TV programs.
- **Remote Access and Control**: ability to call/SMS/email the home, status reports, query appliance states.

The full results from this study can be made available by the author on request.

This study produced interesting results which helped guide the initial design of the Homer system. A simple observation is that, even from a limited set of individuals questioned, people can want very different things from a potential home system. In order to design and produce a home system that can satisfy this wide range of requests it is key that the system be modular, extensible and flexible. Requirements gathered from this study are discussed further in Section 2.4.

2.3.2 *User Study 2: How People Want To Control the Home*

Having observed the desire for smarter homes from user study 1, the author then required an understanding of how people would like to control and interact with their home. This study was a far more detailed analysis, with 150 respondents taking part. A summary of the key details of this study are discussed within this subsection. The full description and analysis is published in [79].

The study was a short (5–10 minute) online questionnaire designed to collect a range of demographic data, information about technical abilities and opinions, and finally information about interactions with a hypothetical home system. This information included the preferred modalities, devices and locations for interactions with the home system.

The survey took place over two weeks in April 2010. Exponential non-discriminative snowball sampling [1] was used to distribute the survey to a wide audience. This resulted in 150 completed surveys by individuals from a diverse demographic background and with a wide range of technical abilities.

The results produced a range of quantitative and qualitative data that was analysed thoroughly to evaluate hypotheses and to gain a clear understanding of exactly how people would like to control the home. The respondents demonstrated a very strong desire to control the home when offered "the perfect system". PCs, laptops, tablets and smart phones were of similar high preference as possible devices to control the home, with the exception of a games console which proved to be very unpopular. As for the means of control, respondents much preferred the notion of controlling their home through touch or remote control, rather than voice or gesture based. Home security was also of concern, confirming the findings from the previous study. There was very little mentioned about security issues concerning the home system itself, which is somewhat surprising: the respondents had few trust or safety issues with a potential home system.

The key findings from this study include the desire for flexibility, programming and general freedom over control and customisation for any potential home system. A flexible home system should be able to be controlled through a range of interfaces and devices, using different modes of interaction. A home system should be programmable by the user if desired. Finally, the system should offer the freedom to be controlled from anywhere.

The findings from this user study aided the design and support of end-user interfaces for the Homer system (discussed further in Chapter 5). Acknowledging the general desire for choice and freedom when controlling the home, it became important to offer a flexible and extensible means of control. The requirements from this study are described in more detail in Section 2.4.

2.3.3 *User Study 3: Can People Program Their Homes*

Having explored what potential users would want to be able to do with a home system, and how they would like to control it, a final study was carried out to explore if users would be able to program their home. A small paper-based questionnaire session was carried out with local maths and computer science postgraduate research students. The questionnaire aimed to discover if technically minded people can formulate policies, allowing the author to consider the format, wording and general understanding of the policy approach. This information was then used to help guide the design of the Homer policy language.

The study was designed to take 50 minutes in total. This included a five minute introductory discussion to describe the high-level notion of programming a home, three example simple policies, and what was involved in the tasks set in the questionnaire. Although the purpose of the study was described by introducing the concept of programming the home, very little direction and background was provided about policies and how to formulate them. This was to simulate a more realistic situation where end users of a home system would, most likely, have no or little prior knowledge or experience of policies.

General information was collected about the participants, including their gender and familiarity with policies. This opportunity was used to obtain information about general interest in home systems from a more technically minded group of people. The participants were asked if they would like a smart home, how they would like it controlled (under their control or by the home system) and if they had ever looked into home automation. The main focus of the questionnaire involved three policy based tasks for the participant to complete:

1. The first question asked the participant to formulate a policy from a given subset of example terms (Trigger, Condition and Actions). An example of each type of term is shown in Figure 2.2. Thirty-five sample terms were given which included terms about doors, windows, lights, fans, temperature, SMS, email, boiler, motion detection and weather.

1 A technique used to obtain access to a wider range of people, by asking respondents to pass the study to others and similarly asking those individuals to pass it on, and so on [24].

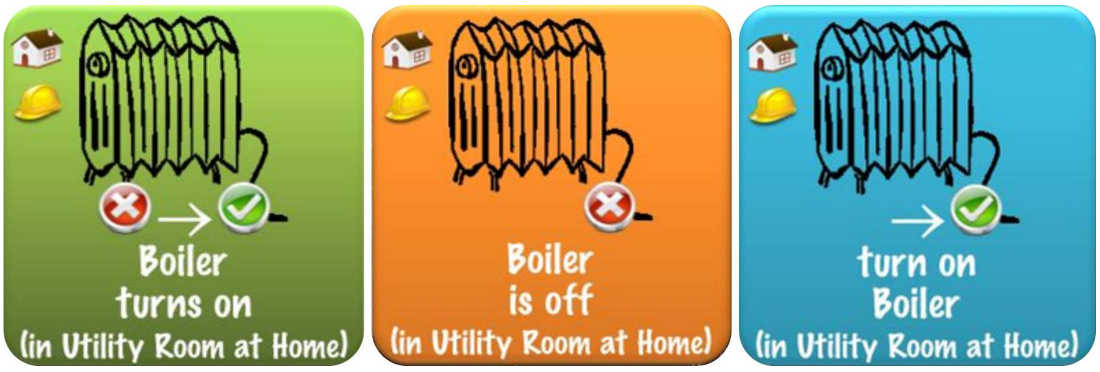

Boiler
turns on
(in Utility Room at Home)
Boiler
is off
(in Utility Room at Home)
turn on
Boiler
(in Utility Room at Home)

Figure 2.2: Example Trigger, Condition and Action in User Study 3

2. The participants were asked to write example terms that had not already been provided on the example sheet.

3. Finally, sample policies were requested that would help automate or enhance the participant's daily life. It was made clear that there was no correct way to write a policy, and encouraged the participant to join the terms to formulate policies in ways that made sense to them.

The user study took place in October 2010, where twelve postgraduate research students from the University of Stirling came together for the hour of the study. There was a mix of gender and policy experience, as shown in Figure 2.3, with expert policy familiarity being rare. This, however, is not an issue within this user study as the less exposure and experience the participants had to policies the more similar they are to a potential home system users.

The questionnaire forms were collected, collated and analysed to reveal some interesting results. Most policies that were written, for both question 1 and 3, were valid. Only three participants wrote an invalid policy for question 1, reducing to only one respondent in question three. This is greatly reassuring in that, although these participants were of a technical and intellectual background, they could formulate valid policies free-form, without guidance, restriction of choice, or feedback from a user interface. The three most notable results are discussed in more depth in turn, including the lack of distinction between triggers and conditions, perspectives of terms and the trend for simplicity.

For the policies written in question 1 and 3, 75% of participants wrote at least one policy that had no trigger and was composed entirely of conditions and actions. In many cases, conditions were used when the nature of the trigger alternative was intended. For example, one participant wrote "when time is 8:00pm then feed cat". In this policy the intention is that when the time becomes 8pm, the cat should be fed, which is a trigger, rather than a condition.

The example terms that were given to the participants were colour coded (as shown in Figure 2.2): green for triggers, orange for conditions and blue for actions. Even with this visual distinction, along with the natural language, the difference between triggers and conditions was not apparent to most of the participants.

A second discovery this user study revealed is that different individuals chose to refer to terms from different perspectives. These perspectives were:

- **Location**: the location of the device or service. Some examples include "hall lamp is on", "bedroom door opens" and "house receives SMS".

- **Time**: based upon some notion of time. Some examples include "when dusk", "time is later than 10pm" and "morning alarm goes off".

- **Personal**: referring to a device or service from first person perspective. Some examples include "check my temperature", "when I get up" and "my fridge is empty".

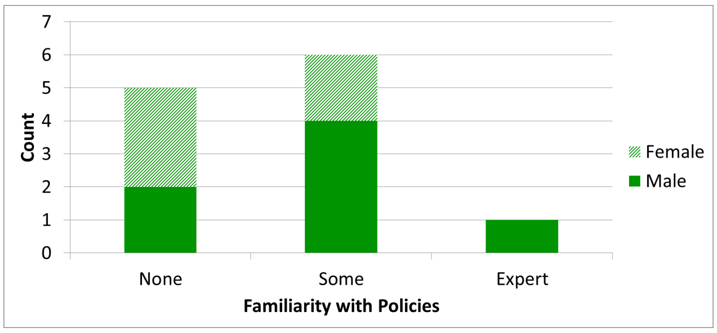


Figure 2.3: Gender and Familiarity with Policies for User Study 3

- **Device**: referring to the device or service directly. Some examples include "phone receives call", "shower is on" and "main door is open".

On average participants made use of two perspectives throughout the policies they wrote for question 3, and the most popular of these was the device perspective. The personal perspective produced an interesting problem: the term "send SMS" could be interpreted as both a trigger and an action. From the personal perspective "send SMS" is a trigger, meaning that when "[I] send SMS" the policy should then react and do something. However, from a device perspective "send SMS" is meant as a command, written in a policy as an action that should be carried out.

A final observation made from the results of this study is that, on the whole, participants wrote relatively simple policies. Despite the participants' technical backgrounds, policies had an average of only three terms. (There was only one policy far longer than the average length, with eight terms in a single policy). By far the most common formats for policies was the standard "*when trigger – if condition – then action* ", and the blurred trigger version "*when condition and condition – then action* ".

Three participants wrote more sophisticated policies, including ordered lists of terms, optional terms and some examples of conditions amongst the action list. This was interesting, as these participants were able to express logic within one policy where other participants required two. For example, logic expressed using two policies: "*when* I arrive home from work *and* temperature > 24°C *do* open window" and "*when* I arrive home from work *and* temperature < 24°C *do* turn on heating", was represented in one policy as: "*when* I arrive home from work *do if* temperature > 24°C *then* open window *else* turn on heating".

Various discussions took place among participants as the questionnaire was carried out, the most relevant of which are discussed below.

There was a debate about the wording difference of the start of a policy, where some people said they would prefer to use "if" rather than "when", which was used in the example policies given at the beginning of the study. However, interestingly they led themselves to the correct conclusion that actually there is very little difference, and both would be completely acceptable.

The participants also noticed the lack of any form of *not* within given terms and examples, with some participants wanting to make use of such logic. Other participants were actively against the use of *not*, pointing out that all terms typically had an opposite. So instead of saying *not* "boiler is on", you could simply just use the appropriate term "boiler is off". Those in favour of *not* were unable to produce examples that would require the operator, and as a result all policies written by the participants within the questionnaire did not involve any *not*s.

A final point of discussion revolved around policies that could take place when the home was empty. One participant asked if policies were only for when within the home, of if they could write policies for when they are on holiday. This question inspired others, with the realisation that the home could be automated from afar.

This user study revealed some crucial information that helped define the Homer policy language. Realising that the participants could not differentiate between triggers and conditions led to the decision to blur the distinction between triggers and conditions. Also, by observing that, when given the freedom, participants referred to terms from different perspectives, it showed that the user should not be forced to refer to the devices and services in a predefined way. Instead, they should be able to freely jump between perspectives as a way of accessing devices and services within their home. Another lesson learnt was that the most common policies written were actually relatively simple in terms of language requirements. Although a few participants chose to write more complex policies, they were indeed outliers – even amongst a technically minded group of people. The message here is that:

> *"Simple things should be simple and complex things should be possible."*
>
> Alan Kay[2] [35]

Of the more sophisticated policy formats, ordered and optional events should be possible. A final decision learnt from this user study was that a *not* operator was most likely unnecessary. If a technically minded set of participants could not think of a requirement for it then it would just become a language feature that added more complexity than was really required. The policy requirements are discussed in greater detail in Section 2.4 and Chapter 4.

Many valuable lessons were learnt from these user studies. They all helped to guide the design decisions made for the various aspects of Homer, as well as extend – and suppress – the requirements for the research.

2.4 REQUIREMENTS FOR A HOME SYSTEM

From the three user studies many valuable lessons were learnt and observations made about what people want from a home system, how they would like to control it, and finally if they are able to program it. A list of requirements for a home system was drawn from the user studies.

2.4.1 *System*

Observing the wide range of desired home functionality and automation, it is clear that a home system must be extremely modular and extensible. As new devices and services become available to the consumer market it is crucial that the home system can support them (given an open API for developers). This allows the home system to offer a wide, and ever-growing, array of functionality for the user. However, not only should the home system support as many devices and services as possible, it should also allow these individual pieces of functionality to be combined in flexible, configurable and dynamic ways. With the combination of up-to-date devices and services and a means of combining their functionality, a fully dynamic and configurable home system can be offered. This would meet the extremely wide list of desired home functionality described by users in all three trials. Chapter 3 expands upon these requirements in relation to home systems, and explores existing home architectures to assess if such a system currently exists.

2.4.2 *User Interaction*

A home system clearly must be designed to be simple and easy to use for a wide range of possible end users. The second user study, described in Section 2.3.2, explored the preferred modes and devices for interacting with a home system. The results from this study showed that, on the whole, people liked the idea of interacting with their homes through touch – whether using a touch phone, tablet, wall panel, or any other touch input device. However, people were not against the idea of interacting with their home through other means such as voice, remote controls or gesture. Similarly, respondents showed interest in making use of different devices for interacting with the home. These devices ranged from mobile phones to televisions. For these reasons it would be advantageous for a home system to be controllable through a range of different interfaces and devices. This would maximise the ease of integration and acceptability of a potential system within a home.

2 Alan Curtis Kay (born in 1940) is an American computer scientist known for his early pioneering work on object-oriented programming and windowing graphical user interface design.

The second user study also showed that people were keen to have control over their home remotely, which the other studies also confirmed. Therefore a second requirement of user interaction with a home system is to ensure that the system can be controlled away from the home, ideally using the same applications and interfaces. For example, an iPad application used within the home to control devices and services should behave identically within *and away* from the home.

The third user study demonstrated that people typically referred to the devices and services within the home from different perspectives. This must be taken into consideration in the design of home system interfaces. Forcing the user to think about the home in one particular way will increase the difficulty and frustration for the user, perhaps making the system seem unnatural to the less patient or less technically minded. So, any user interface for the home system must support access to devices and services through the four perspectives described in Section 2.3.3 (personal, device, time and location).

The user interaction requirements are all discussed in further detail in Chapter 5.

2.4.3 *Policies*

The third user study revealed interesting observations about how people write policies for the home. Four critical requirements for a policy language can be taken from the study.

Firstly, the difference between triggers and conditions is too subtle. Many of the participants in the third study wrote policies that contained no trigger, as well as using conditions when they wanted trigger behaviour. It is vital that policies can handle interchanging use of triggers and conditions and behave as the user would expect.

The ability to negate terms was decided as unnecessary by the participants of study 3. If twelve technically minded individuals were unable to require this operator then it can be assumed that the average home resident will also be unable. For this reason, a policy language designed for the home does not need to support negation of terms.

The justification for not supporting negation of terms also extends to prohibition policies, where the user could specify something that they did not want to happen. For example, "never turn on the heating when the front door is open". Such policies introduce ambiguity to the policy system. What should the policy system do if a user tells the heating to turn on, at the same point as someone walks through the front door? Such ambiguous behaviour is undesirable for a home system, as users must always feel in control of their home.

A final requirement for a home policy language is to support optional and ordered term combinators, as well as allow conditionals within the action list. These features were used by some of the participants in the study, and therefore shows that on occasion the possibility of more advanced features would be desirable.

Each of these policy language requirements is discussed in further depth in Chapter 4.

The three studies described in Section 2.3 have all contributed invaluably to the requirements gathering for the three strands to this work: the home system, user interaction and the policy system. These strands are discussed in further detail in their respective chapters.

2.5 CONCLUSIONS

This chapter introduced home automation, explaining the motivations and exploring the state of the art. It was shown that with the increase in pervasive computing there is demand for home automation. Existing home systems are limited, revealing a gap in both industrial development and research.

Three user studies were performed during the research. These were discussed in turn, describing the methodologies and results for each. The first study involved 30 participants, who helped paint a picture of the general desires people have for technology within their homes. This led onto the second study, involving 150 participants, which provided a deeper understanding of the desires expressed in the first study. The preferred modes, means and tools for interacting with the home through some home system helped influence key research and design decisions as seen in coming chapters. The third user study was much narrower, involving only twelve participants and focusing on purely the language aspects of programming the home.

These studies produced crucial requirements for home systems, including their functionality and user interactivity. All requirements grouped naturally into the three main research areas of this thesis: the home system (architecture), home customisation (policies), and home interactions (end user programming). It was observed that a home system should be extensible and modular, to ensure that third-party developers can add support for additional hardware and software services. The policy system should support a language which allows the distinction between triggers and conditions to be blurred, actions that can be conditional, and triggers and conditions which can be combined, optional or ordered. It was also discovered that there would be little requirement for negation of triggers and conditions, nor the possibility to handle prohibition policies. Finally, the user studies showed that it would be optimal to provide a range of different interface modalities and styles, and to ensure that a wide range of ages and technical abilities were catered for. For interacting with and programming the home, it was observed that many people think about their home and the devices within it from different perspectives, and this should be respected in the interface design.

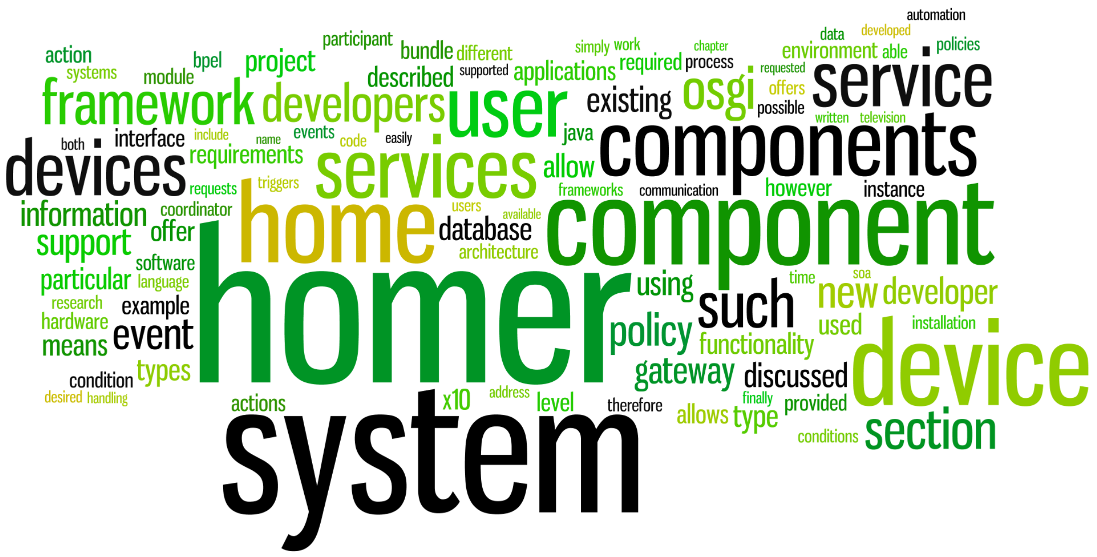
Five open problems emerged after exploring the state of the art of home automation. Within this thesis, it was decided to address the problem of how to offer flexible customisation of the home.

The following chapters now take forward the knowledge of existing home automation systems and research, coupled with user requirements for the three key aspects of a home system, aiming to answer the research question: How can we offer flexible customisation of the home?

Part II

MY CONTRIBUTION

3 HOMER: ARCHITECTURE



This chapter explores what is required for the core aspect of a home system and introduces Homer, a home automation system developed to satisfy these requirements. This work provides a test-bed for the forthcoming policy work in Chapter 4.

3.1 INTRODUCTION

Within a home environment it is crucial that any system behaves both reliably and consistently to gain the trust of the user. A home system should be able to support a wide range of different devices and services, both existing and future. The dream of a common protocol for all devices within our daily lives is just that. In reality there are many different communication means, protocols and APIs, and the chances of these ever sharing some universal standard is remote. Instead, any home system must be able to support the wide and ever growing range of devices or, as many companies do, support at least a subset of devices.

The core framework for the home system is one of the most vital parts of the whole setup. Any system which is not reliable, robust and adaptable has increased chances of being rejected by users, as well as potential third-party developers. Most home automation companies make their own custom framework which then makes it difficult to allow or encourage other developers to integrate. This, therefore, leaves that company responsible for integrating other technologies and protocols in order to maximise their capabilities and attractiveness to the market.

This chapter first introduces the background of service-oriented computing. The requirements for a home system are discussed, followed by an exploration of the state of the art. The architecture of Homer is then described in detail, as are the components and services. The Homer system has been fully implemented except where stated.

3.2 BACKGROUND

As computing becomes more pervasive, so too does software. This naturally leads to the desire of software reuse and integration, developed locally or by third-party developers. A widely adopted approach is for software to be decomposed into useful components which each offer a useful *service*. These services can then be combined to produce higher level software which is more robust and maintainable. This concept is known as Service-Oriented Computing and is realised through a Service-Oriented Architecture (SOA) [115]. This describes principles in software design which encourage code reuse, modularity, distribution, interoperability and standards-compliance.

This section introduces the buzzwords within the area of service-oriented computing, followed by how Service-Oriented Architecture (SOA) can help the problems faced with home automation systems.

3.2.1 *Conceptual Approaches*

There exist three main conceptual approaches used within service-oriented computing, each of which is described below.

SOA (*Service-Oriented Architecture*) is a concept for designing software in a modular way. Code is split into natural components, or tasks, which offer particular services. These loosely-coupled blocks of code are relatively independent from each other. Individual SOA blocks can be associated with one another through service orchestration (or 'choreography') [5, 76, 137]. This process allows applications to be developed utilising a set of services. With such a system in place, it is very easy to replace or add new services, providing a highly flexible and dynamic architecture.

SCA (*Service Component Architecture*, `www.osoa.org`) is a collection of standards describing a component-based framework that complies with the principles of SOA. The idea behind SCA is that component implementations should be separable from how they are interconnected. It should be possible to develop components in a range of languages. The framework should allow these components to be 'wired up' in a variety of configurations, without having to worry about the component implementations. Furthermore, reconfiguration should be easy – perhaps because of a new component implementation, or because new requirements need a different configuration.

SODA (*Service-Oriented Device Architecture*, `www.eclipse.org/ohf/components/soda`) is a service-based programming model designed to standardise and simplify the interaction of devices with service-oriented systems. The goal is to allow easy interaction with sensors and actuators for developers. A development kit and Service Activator Toolkit (SAT) have been created. The development kit, which is an Open Services Gateway initiative (OSGi) component, is for interfacing with hardware devices through Java. The SAT eases the process of building and working with service-oriented OSGi components by simplifying the registration and discovery of services.

3.2.2 *Within the Home*

The home is full of individual devices and self-contained automated systems (such as home theatre setups, security systems and lighting) that typically cannot communicate with one another. The principles of SOA lend themselves to the home automation domain, providing tools and frameworks for supporting the combination and programming of various individual devices, technologies and systems in the home.

With the rise in knowledge and quality of SOA in the late nineties, service frameworks have become increasingly popular. The most common and successful of these are discussed in Section 3.4.

3.3 REQUIREMENTS

Requirements for a home system were gathered from user studies presented in Section 2.3. These studies looked into what users may want from a potential home system, and how they would wish to interact with such a system. These requirements were summarised in Section 2.4 and are expanded below with respect to architectural issues.

Devices The following features are required for the support of devices and software services within the home:

- **Extensible** The system should be extensible to allow new/existing/future devices and services to be added and supported by the system.
- **Modular** The system should be modular to ensure device support is self-contained, therefore allowing easy installation (and uninstallation) of device support without affecting the system.
- **Developer Friendly** The system should allow independent developers to write and install support for different devices and services quickly.

System The following features are required for the Homer system:

- **Embedded Policy Support** The system should offer policy support as a means of managing and orchestrating functionality from devices and services in a flexible manner.
- **Dynamic** The system should be dynamic, with no dependencies on the devices, to ensure all internal features (such as policy support) continue to work as new devices are added to the system, are reconfigured or are removed.
- **Configurable** The system should be easily or automatically configurable in ways convenient for the user, and should minimise any decisions or hard-coded settings at code level. This also supports the design principles recommended by Davidoff *et al.* [34] for developing end-user programming systems within a smart home environment.

These were taken as the requirements for the home system to be developed. Primarily, the resulting system must offer a test-bed for the policy work undertaken in Chapter 4. Existing home frameworks, as well as software architecture technologies, were explored to ensure the correct tools and methods were used. These are described in Section 3.4.

3.4 STATE OF THE ART

The following state of the art discusses existing home frameworks and possible framework technologies. The relevant possibilities are analysed, then final conclusions are drawn to decide how best to meet the requirements of Section 3.3.

3.4.1 *Existing Solutions*

Firstly existing home frameworks are explored then more generic architecture technologies are discussed, to ensure a broad scope of possible architectures is considered.

3.4.1.1 *Ubiquitous Frameworks*

There are numerous systems, mostly from academics, which aim to deal with the wide array of different devices and protocols in our technological world. These projects have often not dealt with the home environment, so only a few of the most relevant are discussed briefly here.

Aura [118] was an ambitious research project carried out between 2000 and 2004 at Carnegie Mellon University. The goal was to increase an individual's productivity by implementing a system which supports the notion of each individual having an 'aura' of computing and information services around them at all times, regardless of location. The project team believed that a major source of user distraction is the management of computing resources as situations and environments evolve and change. The Aura implementation involves a central "task manager" which processes task requests and filters them to the appropriate application, on the appropriate device, through the appropriate communication channel, all dependent on the current environment and context of the user and the desired task.

Interactive Workspaces [56] is a research project from Stanford University exploring the technologies and solutions for integrated multi-person and multi-device collaborative workspaces. The project aims to ease and enhance the meeting space, where resources can easily be shared and exploited to their full potential. The Interactive Workspaces project utilises the Interactive Room Operating System (iROS); a custom built framework using Java. iROS mostly comprises a central event handler and data store. Particular device types are wrapped within separate "applications", and all communication between the applications must go via the event handler. This ensures each application has no dependencies on other applications, and allows for a more robust and easily maintained system.

As can be seen from these ubiquitous environment frameworks, a common technique for handling communication among possible devices is through some central intermediary. This eliminates the dependency of devices on one another, and allows all communication and control to be coordinated at a higher level. Therefore, by using some central intermediary it is possible to increase overall system modularity, scalability and configurability. However, the above solutions do not offer integration support for new devices written by developers or the ability to manage or program the device functionality at a higher level.

3.4.1.2 *Home Frameworks*

Having looked at existing generic ubiquitous computing frameworks, major frameworks tailored for the home are discussed below. Firstly systems and projects from academia will be introduced, then the more polished commercial projects.

ACADEMIC

Atlas [64] started as an academic project in Florida supporting sensor-actuator networks in a plug-and-play, service-oriented manner. Atlas can now be purchased through its retailer Pervasa (`www.pervasa.com`). Atlas M2M Middleware, as it is now known, provides a framework that can automate the sensor-to-service (hardware-to-software) conversion.

Atlas is primarily aimed at developers, who can make use of specialised Atlas hardware modules to connect and interface to devices, and then program them through the Atlas middleware platform. Atlas uses OSGi (a service-oriented Java framework, discussed in more depth in Section 3.4.1.3), designing each hardware module to be a separate service which can be installed into the middleware instance with no dependencies on other hardware. This service-oriented design therefore increases modularity and extensibility. In the developers' words "Atlas offers the magic of plug-and-play to the widest array of sensors and devices". However, the main limitation of Atlas is the limited number of hardware modules available, and the lack of ability to add support for third-party devices.

Gaia [109] was an academic research project from 2000-2005 which aimed to tackle many of the same problems as this research project, using the same philosophies. Although Gaia is now out-of-date and unsupported, it is of interest due to the strong crossover and similarities discussed below.

The researchers of the Gaia project believed that "people's living spaces" (including homes, offices, cars and airports) should be programmable and interactive, and always remain in the control of the user. Gaia was developed as middleware between people and their technological environment, offering a programmable and customisable system. Gaia and Homer share the common belief that all device functionality should be made available through one system and then programmed at a higher level to fully exploit the environment (though focused primarily on the home within Homer). This results in one system which can meet the many different needs of users.

Gaia emerged from years of research on reflective middleware and meta-operating systems, particularly focused on ubiquitous computing and the idea of hiding device and operating system heterogeneity, adapting dynamically to changes in the environment. There are three main parts to Gaia: the kernel (manager of the various devices connected to the system, where CORBA (*Common Object Request Broker Architecture*, `www.corba.org`) is utilised for all communication); the application framework (implemented in Java and offering functionality to register, manage, and control Gaia applications); and finally the applications themselves (higher level chunks of functionality, such as a music player, calendar reminder or presentation viewer). The applications are written using LuaOrb [25], a high-level scripting language, to program and configure the functionality of the various devices of the system. All internal communication is carried out using events to decouple the system and enable a more dynamic and flexible architecture.

There were many researchers and projects involved in development of Gaia resulting in a very sophisticated framework for programming an environment. However, there is little information regarding dynamically supporting new devices and services by third-party developers. Instead, it appears, meta-information about devices and services is required within Gaia for it to understand and make use of their functionality. This is highly restrictive, as it means support for new devices, technology and services remains in the hands of the project developers. It is a near impossible task to keep up with the fast-paced changing nature of the technology world.

MATCH (*Mobilising Advanced Technologies for Care at Home*, `www.match-project.org.uk`) is introduced in Section 2.2.3. The philosophies and core system are discussed in more detail here.

MATCH emerged from a research project which was primarily focused on addressing key problems within telecare. This included the complex scenario surrounding care management for the residents; trying to harmonise the various goals and desires of the care givers, friends, family and the resident themselves. To address this challenge policies and goals were used to allow desired rules to be expressed for the resident by the various parties. The core policy system and supporting language is known as Advanced Component Control Enhancing Network Technologies (ACCENT), and is discussed in detail in Section 4.4.

MATCH, from its foundation up, focuses on easing and minimising user influence at runtime. This has resulted in a system which aims to make decisions behind the scenes and therefore has a tight integration between function and behaviour. This is arguably appropriate within the telecare situation, due to the advantages of easing the burden of the system upon the user and having to handle multiple requirements from the various stakeholders. However, in terms of home automation it is highly desirable to ensure that all behaviour is fully controlled and understood by the user. The system should purely act as a proxy between the user and the desired function.

Due to this major difference in philosophy, MATCH was unable to provide a test-bed to demonstrate the policy work described in Chapter 4. The core MATCH system is explored and discussed below to observe any successful architectural decisions.

Major implementation work was carried out within the project, resulting in an advanced home system (described in further detail by McBryan [88]). It was developed using OSGi, with the notion of different devices and software services (such as email, weather forecasts and calendar events) each being their own component. Components request bindings to other components if required (for example the speakers component may require the speech synthesis component). All communications go through a central "event broker" to reduce unnecessary coupling where possible. MATCH also offers an embedded policy service, allowing a higher means of combining the functionality from the devices and services.

Many positive design decisions can be learnt from the MATCH home system, namely the use of OSGi, components and policies. However the actual implementation is too closely focused on home care problems. This resulted in a different philosophical approach which involves decision-making processes taking place at code level. This is considered disadvantageous within the home, as all events that take place should do so from command (such as through a policy or directly through some user interface) to minimise potential user confusion.

COMMERCIAL

Amigo [4] is an advanced, commercial, open-source, middleware platform for dynamically integrating heterogeneous services and devices within the networked home environment. The service-oriented system is split into three sections: the base middleware, the intelligent user services, and the programming and deployment framework. The programming section allows programmers to create "Amigo aware" services in Java (using OSGi) or .NET (Microsoft's .NET framework, `www.microsoft.com/NET`). Supporting the two main runtime environments for developing services allows the vast majority of programmers to be able to easily create components for their system. The core Amigo technology is web services, following the WS-Addressing, WS-Discovery and WS-Eventing standards to increase compatibility.

Amigo handles every software layer needed to create and control a smart home system, ranging from the communication protocols to the user interfaces. Developers are allowed to write applications and component software as they please. Unfortunately, Amigo does not have any in-built means of combining functionality in the form of policies. This could be achieved through an external policy server, however I hypothesise that only basic functionality would be possible. More advanced tasks, such as conflict detection, would require internal system knowledge and adjustments.

Control4 (`www.control4.com`) was introduced in Section 2.2.3, and can be considered a leading home automation company. Their business model is extremely simplistic: they provide a framework and allow anyone to develop 'drivers' to allow their products to interoperate with Control4 (currently 6500 third-party devices have been made compatible [28]). By forcing the other companies to write their own drivers in order for their devices to be compatible means that Control4 does not need to manage and maintain updates or alterations other companies may make.

Control4 also offers companies the opportunity to embed the 'Control4 operating system' into other devices, integrating their platform tightly with Control4 (using highly functional 'two-way drivers') and producing applications which users can buy and run through their Control4 user interfaces. The latter requires developers to pay a membership fee which entities them to create (using Adobe Flash ActionScript and PHP) and sell applications through the store.

As sophisticated and highly polished as Control4 is, there is no means to program the system at a high-level. This results in each home installation having to be customised by developers and system installers for that particular installation. Any changes that may be desired in the home setup must be requested and handled by the company rather than the customer. This is both costly and inconvenient for users.

iQare (`www.omniqare.com`) is a software platform made in the Netherlands by OmniQare (introduced in Section 2.2.3). Although designed for telecare and older users, the framework and approach is applicable across a broader domain. The iQare framework is designed to be run on a purpose-built touch-screen PC, and any user interactions must take place through this.

There exist over 50 applications for the device, although developers can build their own to be integrated using a custom eXtensible Mark-up Language (XML) protocol. OmniQare have simplified the notion of interacting with various different devices and services by offering one standard means of supporting them through purpose-built applications. However, this results in a system with an interface that acts purely as a gateway to single-purpose applications. So far there is no means of integrating or managing these applications, or services, on a larger scale. This means that the overall system cannot be programmed to combine the various pieces of functionality of the individual applications

At this stage of the literature review it can be seen that there has not been a solution which meets all the given requirements outlined in Section 3.3. An interesting observation is that each system comes close, yet none is able to offer a test-bed for the policy work in Chapter 4, nor meet the requirements of supporting new devices from developers as well as offering the ability to program the devices at a higher level.

3.4.1.3 *Service-Oriented Frameworks*

Having learnt design successes from various existing solutions, and observed the lack of an existing system which would support the requirements, frameworks for writing a custom home system were explored. The nature of the proposed home system within this research lent itself to service orchestration, where individual devices can be represented as services that can be combined at a higher level. For this reason service frameworks were explored and discussed below.

Jini (www.jini.org) is a Java-based, service-oriented architecture for distributed systems. This could allow components to be located within the home, while accessing common services such as weather forecasting on a central server. However, the full benefits of Jini, primarily its distributed nature, are not applicable for my research as typically a home system and its connected devices are all local within the home. For this reason Jini was discounted, along with other distributed solutions such as Fabric3 (www.fabric3.org) and Paremus (www.paremus.com).

OSGi (*Open Services Gateway initiative*, www.osgi.org) is a popular open standard by the OSGi Alliance, defining a modular software framework for Java which follows the SOA paradigm. It is designed to provide developers with a service-oriented, component-based environment, offering standardised ways to manage the software lifecycle.

Numerous existing home systems and projects are implemented using OSGi, including Amigo, Atlas, and MATCH. It is rumoured that OSGi was originally drafted to address the unique problems that home automation produces [100], offering a fully dynamic component based framework which supports the notion of different developers and companies writing components (termed "bundles" within OSGi) that can simply be installed into the one framework. OSGi also offers an event messaging service to support communication between components and the home system without introducing dependencies. OSGi is highly suited to the problems presented in this chapter, and was therefore seriously considered as a possible home framework (conclusions drawn in Section 3.4.2).

Stepstone (stepstone.projects.openhealthtools.org) is a project that began between IBM and the University of Florida. It focused on research for the Gator Technology Smart Home (www.gatortechintegration.com). The project now involves Eclipse and OHF (*Open Healthcare Framework*, www.eclipse.org/ohf), which is a project within Eclipse designed to improve the process of creating healthcare systems by providing extensive tools and frameworks. Existing and emerging standards are followed to lower integration barriers and encourage interoperable open source infrastructures. Stepstone demonstrates how a service-based programming model can be used to build healthcare solutions using embedded technology, service-oriented architecture and open standards. Unfortunately, Stepstone only formed in early 2009 (during the architecture research, design and development work within my project) and as such was too immature to be considered. However, Stepstone in reality is a very poorly supported open-source project and as such has remained rather primitive and immature, and has not lived up to the expectations and promises of its sales pitch.

3.4.1.4 *Component Glueware*

Having explored home frameworks, this section looks into possible ways of combining components. These glueware technologies could work alongside, or instead of, the aforementioned potential home framework.

Acme (www.cs.cmu.edu/~acme) is an Architecture Description Language (ADL) system. ADLs are used to represent systems in a high-level abstract way, which is both human and machine readable. Acme is simple and light weight. It can be used as a common interchange format, or as a foundation for developing new architecture design and analysis tools. Unfortunately, there is still no universal agreement on what ADLs should represent, and the existing tools, such as Acme, are very academically oriented with little commercial support. Such an unclear high-level description language provided little concrete support for my research.

BPEL (*Business Process Execution Language*, [43]) is an XML-based standard for creating a 'business process' flow from discrete services. It can be used as a component glue framework, allowing processes composed of the components to be defined. ActiveBPEL [39] is an open source Java implementation of the BPEL engine. In addition to ActiveBPEL, a visual BPEL tool could be used to ease the process of writing BPEL. The main four available are: ActiveVOS Designer [39], Oracle BPEL Process Manager [101], CRESS (introduced later in this section) and the Eclipse BPEL Project (`www.eclipse.org/bpel`). Potential use of BPEL is discussed with CRESS later in this section.

Concur Task Trees CTT is a notation proposed by Mori and Parernó [92]. Relationships and possible tasks are represented by a task model in a CTT as an inverted tree. This logic can be applied to the concept of representing home services as a tree of services, where each service is made up of child services. However, although CTT may be appropriate for producing high-level services, I consider this technique too restrictive for combining events to represent policies and for little benefit.

CRESS (*Communication Representation Employing Systematic Specification* [132]) was developed by Turner at the University of Stirling. It is a visual programming language oriented towards linking components through arbitrary program logic (internally expressed in BPEL). Using Turner's system would allow the BPEL input-output (Receive, Reply and Invoke) to be mapped to Java objects using ActiveBPEL's custom invoke handlers. BPEL could then be used along with a Java-based architecture as, or in addition to, the method of connecting components. CRESS allows tasks to have loops, branches, fault handlers, concurrency, data structures and void nodes, which allows a complex service glueware to be created. The use of CRESS in relation to BPEL is discussed at the end of this section.

BPEL and CRESS are a potential component glueware option for within the home. This would involve using BPEL as a means of combining home components and services, and potentially CRESS as a visual designer. Turner has indeed used BPEL for this purpose [130, 131]. However, the BPEL language itself is bloated (using XML to represent a process flow between components), and adds complexity and rigidity to potential solutions. Although CRESS hides this complexity and offers a much cleaner user interface, it requires a technical user to translate a policy into the CRESS format. This process should ideally be carried out automatically when a policy is written using any one of Homer's user interfaces (described in Chapter 5). Therefore a systems interface to BPEL would need to be written to allow the Homer policy server to generate the component process in BPEL when a new policy is saved. Therefore, although BPEL and CRESS could be used, a lot of adaptation and alterations would be required for little benefit.

PCOM (*Pervasive COMputing* [9]) is a lightweight component system for pervasive computing. Resulting applications consist of a tree of components, where the functionality of a component is the sum of its children. Each component has a contract defining its dependencies, which allows for self-configuration and no manual setup. The contracts use the concept of Signals and Slots [140]. Weis *et al.* [141] used PCOM to create a software development solution supporting developers, customisers and users. This is achieved using a graphics toolkit for customisers and self-configuration algorithms to ease development. The model is based upon desktop applications where developers can create extensible applications and components, customisers use these to develop custom applications, and finally users configure these applications to their individual needs by adjusting predefined settings. PCOM, which forms the lower-level part of their architecture, allows developers to create components which "automatically orchestrate themselves". So any user could download new components, "PCOM enabled", and they would automatically be integrated into the environment. The middleware software, Nexel, is a graphical programming language aimed at "hobby programmers" to allow the production of PCOM components through a drag-and-drop style interface. PCOM is interesting, but the solution is designed for technical people and focuses on the creation of applications rather than rules and policies.

Tuscany (`tuscany.apache.org`) is an Apache open-source implementation of SCA. Tuscany offers a means of combining components written in different languages, including BPEL, C++, Java, JavaScript, Spring, and various scripting languages. This has many advantages, such as the possibility of different developers being able to write components for the home in their preferred language, or language most suited to the hardware or software service. Components are combined using custom XML, but graphical tools are available to ease this process.

Due to Tuscany's potential appropriateness within the home, a prototype home system was developed to allow a more educated judgement. Tuscany was still in its infancy during implementation, as was the Eclipse Integrated Developer Environment (IDE) plug-in which offered a graphical tool for combining components. Nonetheless a prototype was developed – for more information regarding the prototype system see [81]. A small set of sample Java components (a clock, simulated boiler and email/SMS sender/receiver) were written to allow testing of Tuscany's ability to combine their functionalities. Unfortunately, Tuscany provided no support for service discovery or management, and in practice brought little advantage when used with local Java-based components. A Remote Method Invocation (RMI) component was written to interface to a Nabaztag rabbit (`www.nabaztag.com`) to test Tuscany's behaviour with non-local components. This turned out to be extremely awkward and cumbersome to both setup and utilise.

The main attraction of Tuscany is its ability to integrate with other technologies. However, within my project the assumption is that the majority of the components are both local and in Java, so not enough use would be made of this major advantage. Even if it were, it is challenging to interface Tuscany with components of different languages. With respect to this observation, as well as Tuscany having no support for service discovery and management, and its immature state, it was decided that it would not make a suitable home system within this research project.

3.4.1.5 *Device Communication*

Finally, the last section of this state of the art looks at two possible means of communicating with services at a more automated level.

Speakeasy [38] is a set of common interaction patterns presented by Edwards *et al.* It provides a solution to the low-level problem of communication between hardware components. These patterns allow rich interactions between computational entities with little previous knowledge of each other. Speakeasy uses Java bytecode and Jini multicast/unicast discovery protocols. However, the solution focuses on connecting devices for immediate feedback in the environment; there would be little knowledge of exactly what functionality the device could support and offer. This, therefore, means it could not be used to meet the requirements for a home system.

Zeroconf (`www.zeroconf.org`) is a service discovery concept pushing for "zero configuration". Bonjour, formally Rendezvous, is Apple's implementation of Zeroconf, and currently the most popular of implementations. Others include Avahi and Microsoft's "wireless zero configuration". Within local networks Zeroconf locates devices such as printers and computers, and the services that they offer, using TCP/IP. Zeroconf services need to be programmed at the application level. The Zeroconf protocol is not strongly supported within typical home devices, so unfortunately it could not be used as a sole means of detecting and communicating with devices within the home.

3.4.2 *Conclusions*

There were two primary aspects to consider when choosing the home framework: firstly, the flexibility of combining the functionality of these devices at a higher level and, secondly, its ability to support devices written by third-party developers. As was described in the literature review, there is no suitable existing home framework which meets this criteria. However, various lessons were learned: components should be treated as services for the home system, components should not have dependencies on one another and to encourage a dynamic modular system an event broker can be used for passing messages between components and the home system.

Different technologies for developing a home system were then explored. This included service-oriented frameworks, component glueware and device communications. Of these, OSGi was considered the most suitable framework, offering a mature means of supporting a component structure and message passing through an in-built event messaging service. The possibility of combining these components through Tuscany was considered, but dismissed due to its immature state and over-complexity. Finally, it was judged that there was no existing automated means of detecting and handling devices, and instead this would need to be done manually by corresponding components.

A custom framework was needed to offer a truly modular, extensible and programmable home system which meets the requirements outlined in Section 3.3. This framework should be built using OSGi and its internal event messaging service. The resulting system, named Homer, is described in the sections that follow.

3.5 ARCHITECTURE

A high-level architectural diagram for Homer is shown in Figure 3.1. Each part of the diagram is explained throughout the thesis. Firstly, the internal workings of the Homer Framework are described within this section. Secondly, the components and services which form the link between the home and the internal system are discussed in Sections 3.6 and 3.7 respectively. The Policy System, which combines the functionality of the various components at a high level, is discussed in Chapter 4. Finally, the Web Server, (simply an HTTP API for Homer) that allows end-user applications to be developed for interaction with the home system, is described in Chapter 5.

All elements within the Homer framework can access the database API directly, can register listeners with the event coordinator, and can be notified of events taking place. The only public developer interfaces available for communicating with the internal Homer framework are the three gateways. Similarly, the only external entries to the Homer system are through a service, a component or an HTTP request.

3.5.1 *OSGi*

The Knopflerfish implementation of OSGi was chosen as the tool to build the Homer home system, due to both the success of existing projects in using this implementation and also to the local knowledge and expertise available.

OSGi offers a service-oriented architecture, as described in Section 3.4.1.3. Each service is implemented as an OSGi bundle. Each bundle can be installed, uninstalled, started and stopped – all while the OSGi framework is running. This allows users to install new components without restarting their machine, and offers greater flexibility to Homer, component developers and system maintainers. As an example of this flexibility is if something went wrong internally with a particular piece of hardware, or the user was no longer using a certain device type, the system (or system maintainer) could easily restart or uninstall the relevant bundle at runtime without affecting the home system.

OSGi also offers a flexible event service which allows events to be both sent and received globally within the OSGi system. This is discussed in more detail in Section 3.5.3.

3.5.2 *Database*

The Homer database stores all information and data regarding the components and the user installation. Figure 3.2 shows a diagrammatic high-level overview of the database and the relationship between the various entities.

Notionally, the database can be split into two main aspects: the component data and the user data. The component data consists of all information provided by Homer components, describing device types that they offer as well as any supported triggers, conditions or actions of that device type. On the user side, all such data exists per home installation and is fully customisable and definable by the home user or system installer. This user data consists of all information regarding locations, devices within the home (including configuration details, such as the module address in the case of X10), any environment factors that interest the user, how the actions of the various devices affect these factors, and finally any policies that have been defined.

The Homer database is implemented using the Java embedded SQL H2 Database Engine. This can be installed anywhere (local or remote) for access by Homer. Only the Homer Framework itself has access to the database: no external access by components, services, or any other aspect of the OSGi system is permitted. In a typical home installation it is envisioned that the database would be stored locally, on the same machine as the home system is running

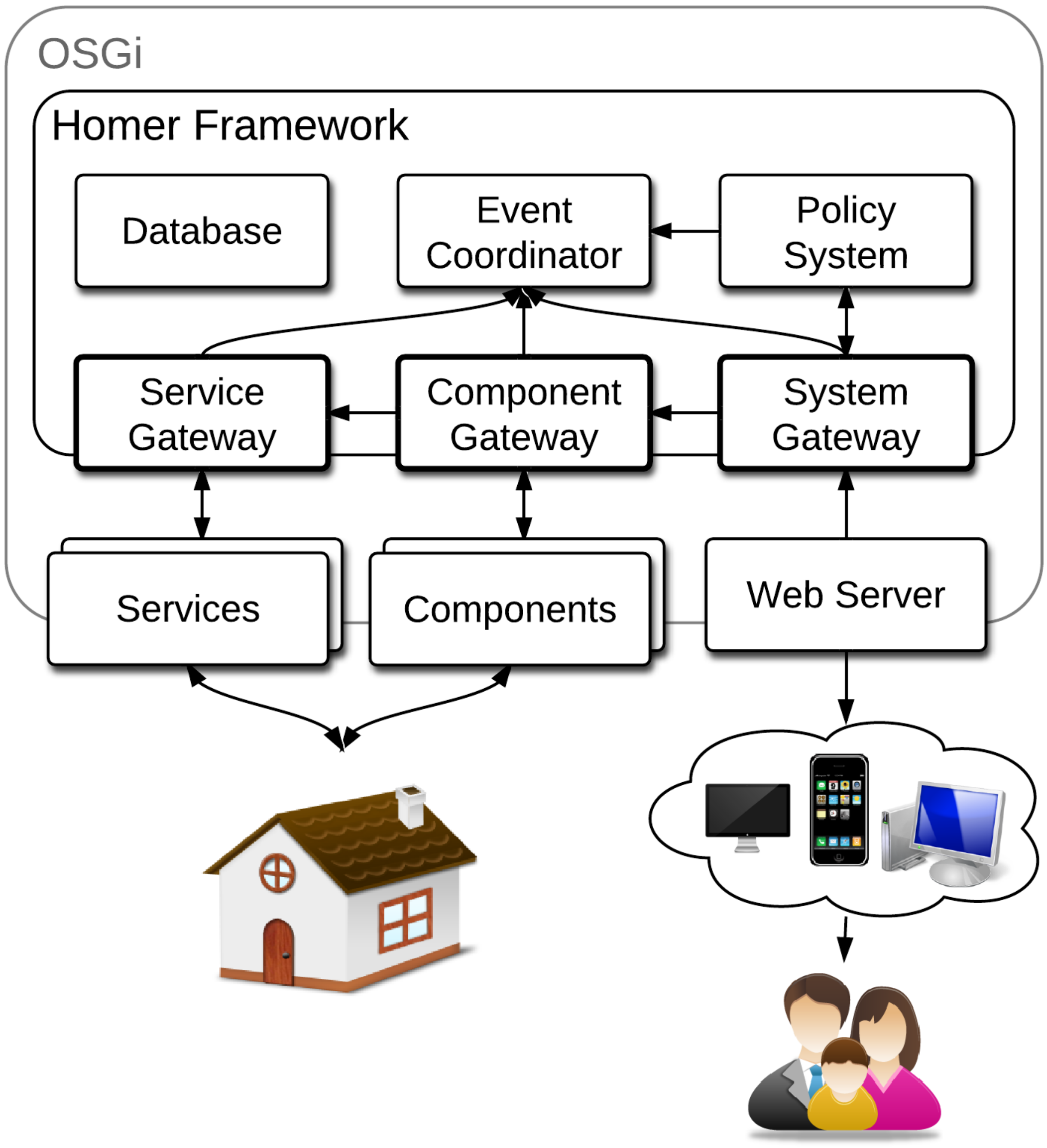


Figure 3.1: Homer Architecture

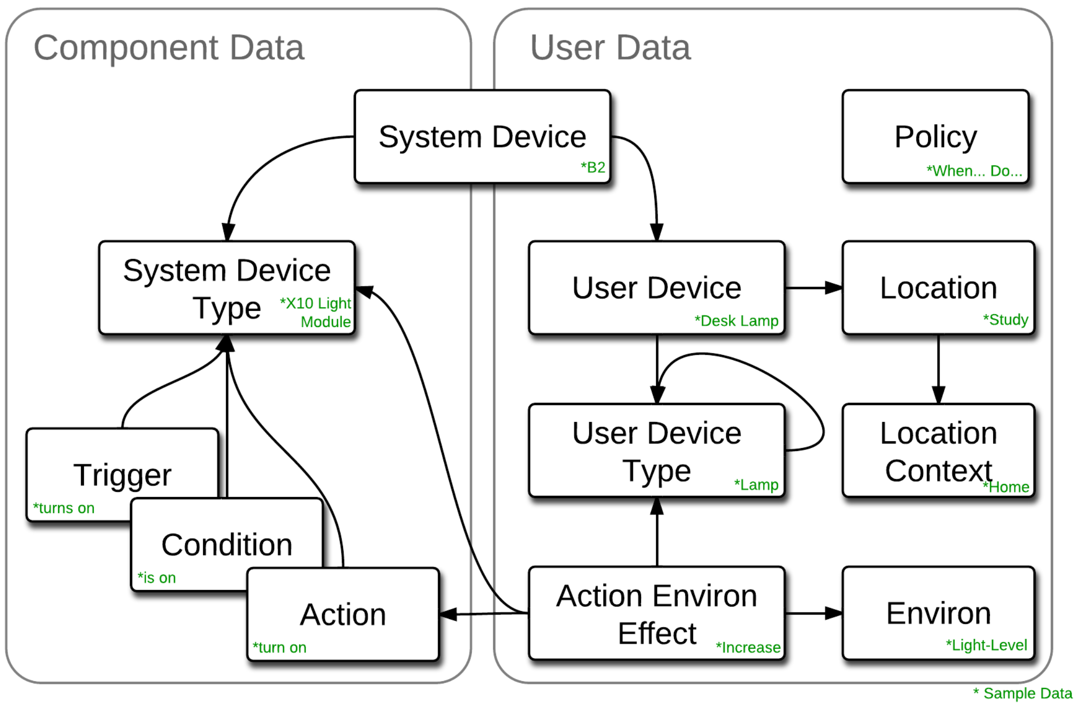


Figure 3.2: Homer Database Organisation

on. This would minimise security risks, and offer the fastest access option. H2 offers many strong security features which can be used to ensure that all data within the Homer Database is protected. Such features include user password authentication (which uses cryptography methods SHA-256 and salt), use of SSL/TLS for remote connections, and encryption of database files.

All desired data requests for the database are abstracted into a general Homer Database interface, allowing any data storing implementation to be supported.

3.5.3 *Event Coordinator*

The Homer Event Coordinator is responsible for posting events regarding triggers, conditions and actions within Homer, and allowing other Homer entities to register listeners for such events.

OSGi provides an event messaging service which allows events to be broadcast throughout the entire OSGi framework. Any bundle (component) within OSGi can choose to listen for events and (more importantly) for event patterns.

Any Homer Event which the Event Coordinator posts, via OSGi, uses a Homer-specific event name (such as "uk.ac.stir.cs.homer") and contains identifiers for:

- User device and user device type
- Location and location context
- System device and system device type
- Event (trigger, condition or action).

With all of this information present in each post, it makes it possible for any interested party to listen to particular events. For example, it is possible to listen to all Homer events by simply registering a listener with OSGi anywhere within the framework for all events with the chosen Homer event name. On the other hand, specific filters could be applied to restrict the events. For example, one might specify only events that take place in a particular room, all actions requested of a particular device type, or all events which involve a particular device.

To listen for an event, convenience methods have been provided within the Event Coordinator to conceal the custom OSGi event structure, and instead make use of the default Java listeners. This allows the internal Homer modules to register their Java listener with the Event Coordinator, providing any desired filters. Internally, the Homer Event Coordinator registers this listener with OSGi, specifying the filters. If an event is broadcast within OSGi that passes the given filters, the listener will automatically be notified of this event.

Utilising this event mechanism reduces inter-system dependencies, allows knowledge of triggers occurring, conditions and action requests to be obtained anywhere within OSGi, and finally allows such events to be filtered to ensure interested parties are notified of only relevant events.

3.5.4 *System Gateway*

The System Gateway is used for all communication between external Homer entities and the Homer framework. This typically involves requests from the user via the Web Server, however the Policy System also makes use of some functions offered by the System Gateway. A complete list of functions includes:

- *Setup and Configuration*:
  These features are achieved by communicating directly with the Homer Database (through its 'public' interface).
  - Add, edit or remove
    - device instances
    - device types
    - locations and location contexts
    - environs and information regarding their effects.
  - Rename
    - trigger, condition and action names
    - system device type names.
- *Report New Hardware* that a component has detected and reported to the Component Gateway. Any information is passed to the Webserver for any interested user applications. This allows these applications to notify or ease the setup of any new hardware by knowing some key configuration details such as identification codes (as with Plugwise and Visonic Sensors).
- *Device Control*, whereby an action to be performed on a particular device instance is requested by either a user or a policy. This is achieved by communicating directly with the Homer Event Coordinator which broadcasts a request that such an action is carried out.
- *Information Retrieval*:
  - Ask if a condition is true or false with a particular device instance.
  - Request any combination of information from the database, optionally restricted by any criteria (such as the location of a particular desk lamp, all devices within a particular location, or all locations within a particular location context which contain devices with actions). Such requests are handled by communicating with the Homer Database.
- *Policy Configurations*:
  - Save a new policy (optionally requesting any conflicts to be checked).
  - Edit, delete, enable or disable an existing policy.
  - Retrieve a list of all existing policies, optionally restricted by any criteria (such as all enabled policies, all policies involving something in the living room, or all policies which are a month old). All policy requests are handled by communicating with the Policy System directly. If any changes are to be saved to the database, the Policy System will handle them. This encourages the principle that only the Policy System should manipulate the policies in the database.

3.5.5 *Component Gateway*

The Component Gateway is used for all communication between Homer components and the Homer framework. It is in charge of the following functionality:

- *Registering components* (and their associated triggers, conditions and actions) with Homer, and ensuring this information is always up-to-date. This is carried out by communicating directly with any registered Homer component and the Homer Database.
- *Reporting triggers* that occur, *responding to condition queries* and *handling action requests*. These requests are handled by communicating directly with the relevant Homer component and the Event Coordinator. All triggers are reported by the component, then the Component Gateway forwards this information to the Event Coordinator (which in turn broadcasts this information as an event, so all interested parties, such as the Policy System, can be notified). In the opposite direction, the Component Gateway registers listeners with the Event Coordinator for condition and action requests. If such an event is broadcast, the Component Gateway detects this and handles it appropriately. In the case of an action request, the appropriate component is simply requested to perform the desired action. In the case of a condition, the component is asked if such a condition is true or false. The Component Gateway then contacts the Event Coordinator with the given result.
- *Reporting Detection of New Hardware* that any Homer component detects and reports. Any configuration information that the component can report, along with the system device type, is passed to the System Gateway. The System Gateway then sends it to the Webserver for any interested user applications.
- *Registering new instances of devices*, and editing or removing existing instances, with the relevant component. Such requests are instigated by the user, therefore entering Homer via the System Gateway. For convenience, the System Gateway can communicate directly with the Component Gateway to make such requests. The System Gateway, however, remains responsible for contacting the database for this request.
- *Exposing services* by offering a method for components to access the Service Gateway.

3.5.6 *Service Gateway*

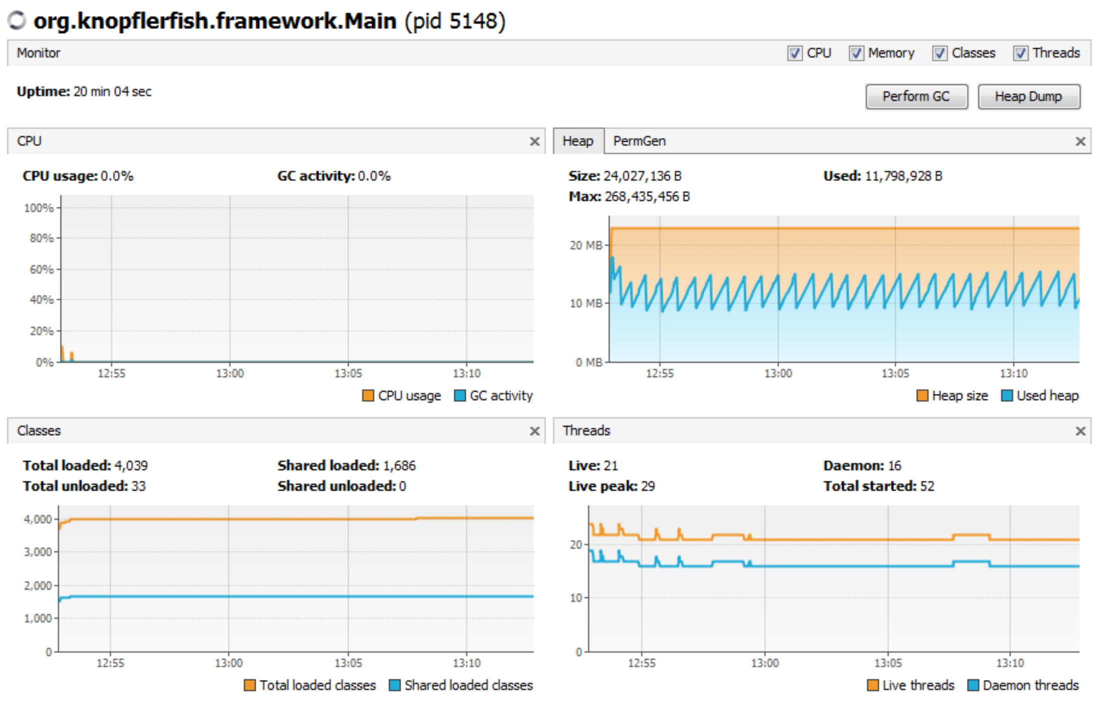
The Service Gateway manages and provides Homer services to developers. It is responsible for maintaining the list of current services available within the framework, creating, managing and closing instances when required, and handling access permissions. Anyone can contact the Service Gateway and request a desired service by name. If such a service is available at runtime, and the requester has the correct permissions, an instance of this service is returned. More information regarding Homer Services can be found in Section 3.7.

3.5.7 *Runtime Requirements*

The Homer framework is built upon OSGi, therefore any operating system that can support Java could technically support Homer. Figure 3.3 shows a profile of Homer starting and running, with typical household activity, for 20 minutes with eight components on a Windows 7 32-bit Operating System, with an Intel Core 2 Quad CPU 2.66GHz and 3GB of RAM. As can be seen, Homer used at most 20MB of memory and barely made use of the CPU after a 15% initial use at start up. Homer, with eight major components, requires approximately 20MB in memory. With such small requirements, it is believed that Homer could run on far less powerful machines.

Homer would be ideally suited for a small media PC style machine in the home, positioned relatively centrally to maximise connectivity options and wireless ranges. The core Homer system does not require Internet Access to function, but some components or services may (such as Weather, email and Twitter), as well as remote user interfaces as these communicate with Homer via HTTP.

Android supports Knopflerfish [124], so it would be possible to run Homer on the more powerful mobile phones and tablet computers which run Android, such as the HTC One X, the Samsung Nexus, or the Acer Transformer. However, this would mean that there would be limited hardware support due to lack of connectivity options. A mobile phone also implies


org.knopflerfish.framework.Main (pid 5148)

Monitor [✓ CPU] [✓ Memory] [✓ Classes] [✓ Threads]

Uptime: 20 min 04 sec

CPU usage: 0.0% **GC activity:** 0.0%

Size: 24,027,136 B **Used:** 11,798,928 B
Max: 268,435,456 B

Total loaded: 4,039 **Shared loaded:** 1,686
Total unloaded: 33 **Shared unloaded:** 0

Live: 21 **Daemon:** 16
Live peak: 29 **Total started:** 52

Figure 3.3: Homer Running Profile

that it would be taken with the person when they leave the home, which would be undesirable for a home system as it should be based permanently in the home to be able to send and receive local requests, such as listening for triggers or requesting actions to take place.

3.6 COMPONENTS

As previously discussed, the use of components allows for independent units of code to interact with particular devices within the home. If a device is to be supported by the home system, the component can be installed and therefore its functionality can be exposed and made usable by Homer. This is a popular concept, and one that has been around for decades:

> *"He will consult a catalogue offering routines in varying degrees of precision, robustness, time-space performance, and generality. He will be confident that each routine in the family is of high quality - reliable and efficient. [...] He will expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, he should be able safely to regard components as black boxes."*
>
> M. D. McIlroy, 1969 [89]

This section discusses the use of components within the Homer system, firstly from the developer's perspective, then from the user's. Finally, the currently supported components of Homer are described.

3.6.1 *Developer Perception*

3.6.1.1 *SetUp*

In order for a component to be installed within Homer it must be an OSGi bundle. This requires installation of the Knopflerfish OSGi framework. The bundle must import the Homer Framework bundle, and can optionally make use of any Homer services (such as the logger, described in Section 3.7). The bundle should be self-contained. Any dependencies on third-party libraries should be contained within the bundle itself. Knopflerfish provides a means of storing system properties, which should be used for any pure developer properties.

3.6.1.2 *Registration*

When a bundle is installed within OSGi it is simply registered within the OSGi runtime framework, ready for use. Once the bundle is "started" (configured to do so automatically after installation) OSGi calls the start-up code within the bundle. Here is where the developer should notify Homer of new components, achieved simply by registering their Homer Components with the Component Gateway.

A Homer Component is simply a Java class which implements a Homer Component interface, provided within the Homer Framework bundle. This class ensures that specific methods are provided by the new Homer Component. These methods are:

- Getting all supported system device types. This allows Homer to obtain a list of system device types offered by any component. When the bundle registers its components with Homer, each component is taken in turn and the following steps take place (shown pictorially in Figure 3.4):
  - System devices types are requested and handled, as discussed in Section 3.6.1.3.
  - If the component offers triggers, conditions or actions, these are requested and handled, as discussed in Section 3.6.1.5.
- Registering, editing or deleting an existing system device. This is discussed in more detail in Section 3.6.1.4.

3.6.1.3 *System Device Types*

Each bundle can offer multiple Homer Components, which in turn can each offer any number of system device types. A system device type correlates directly to the hardware (or individual software service). Examples include email, door sensor, X10 lamp module, and infrared controller. Each system device type offers its own collection of triggers, conditions and/or actions, as discussed in Section 3.6.1.5.

A system device type is described with a unique identifier (for more information regarding how Homer handles identifiers, see [78]), and an easily understood name (such as "X10 Appliance Module" – the name can be changed by the user easily when configuring the home). Finally, any information that would be required when creating an instance of the device is described. For example, in the case of an X10 appliance module, the address of the module (such as "A1" or "B12") would be required when defining a new device which makes use of this device type. Therefore, the X10 component will state its requirement is a 2-3 character string with label "Address".

An example scenario can be illustrated in the design of an OSGi bundle for support of the Visonic sensors. Visonic offer a range of wireless devices and sensors. This could result in a Visonic bundle, offering three components: a magnetic contact, a button and an environment sensor. In turn, these three components would offer individual system device types. The magnetic contact could offer a door and a window sensor device type. The button could offer a pendant alarm, wrist alarm and wall alarm device types. Finally, the environment sensor component could offer temperature and humidity device types. However, this is unrestricted and each developer can design components entirely as they wish. For example, it is perfectly acceptable to have had just one Visonic component offering the entire collection of devices, instead of splitting this across multiple components.

3.6.1.4 *System Devices*

A system device is an instance of a system device type. For example, an instance of the X10 lamp module would be a particular module with a particular address (such as B3). These instances are defined by the end user when setting up and customising the home.

A walk-through is now provided to illustrate system devices and the component role when instances are added, edited and removed:

1. A user buys a new X10 module to extend the current collection, and physically installs it for a lamp within his home.
2. The user then tells Homer about this new device, describing its information and stating that it uses an X10 module (a particular system device type currently installed in Homer). At this point the user is required to provide the address of the new module.

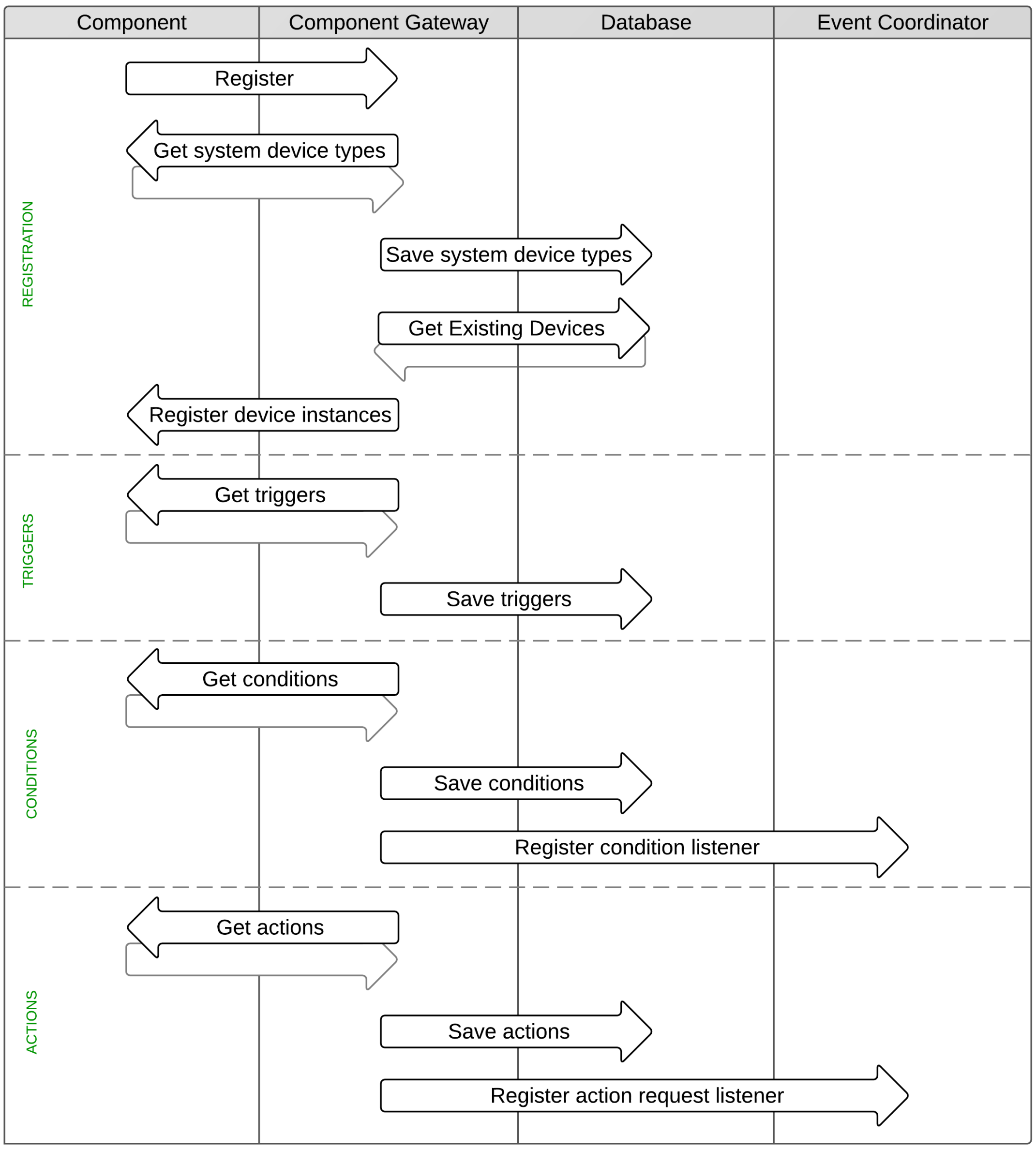


Figure 3.4: The Sequence of Events on Registration of a Component

3. Homer saves this information in the database.

4. All components of the chosen system device types are notified of the new instance. They are provided with the system device type, (newly assigned) system device identifiers, and any parameter values. In this scenario, only the x10 lamp module system device type is associated with the new installation. It is given the new system device identifier for the lamp, and the address of the module (such as b3).

5. The x10 component should store this address along with the system device identifier at runtime. This will allow any future action requests to turn on or off this new device by simply providing the component with the system device identifier and the desired action identifier. The x10 address is already known.

- If and when the Homer system is restarted in the future, each component will be notified about its existing devices in the same way at start-up.

- If the user changes the address of the newly installed device, they can simply edit the already provided address through a user interface to Homer, in exactly the same way as they would edit any other property of the device (such as its name, location or icon). Homer stores any changes made in the database and notifies the x10 component of any changes.

- If the user decides they no longer wish to make use of their new x10 module they should edit the properties of the lamp to represent which hardware it is now using (or simply delete the lamp if it is no longer to be part of Homer). Again, any changes are saved to Homers database and the x10 component is notified of the removal of the x10 module.

There are multiple reasons to have both system and user devices/device types. Primarily, the separation allows developers to work at a lower level without needing to worry about users. In turn, this allows users to setup their home system using any desired vocabulary and customisations without worrying about the rigidity and complications of the underlying hardware. It also allows for user devices within the home to be of one user device type (such as television), but making use of multiple system device types (such as x10 and infrared). Conversely, one system device type (such as x10) can be the hardware supporting multiple different user device types (such as a kettle, television and fan). User devices and their types are discussed from the user perspective in Section 3.6.2.

3.6.1.5 *Triggers, Conditions and Actions*

Triggers, conditions and actions (termed here as TCAs) must be specified within the component. The developer must specify if any of its system device types offer TCAs by implementing the relevant Homer interface for each. Each interface defines a method to get the list of respective triggers, conditions or actions. The condition and action interfaces also define methods to check a particular condition and request a particular action respectively.

The registration of TCAs, the posting of triggers, and the handling of condition and action requests are discussed in turn below.

REGISTRATION The process of registering TCAs is illustrated in Figure 3.4. Each step takes place if the component specifies it offers the respective TCA by implementing the corresponding interface. If any interface is implemented, the required getter method will also be implemented and therefore allow Homer to obtain a list of any offered TCAs.

All TCAs are described with the following information:

- Unique identifier.

- Friendly description, such as "opens" or "turn on".

- System Device Type identifier with which it is associated.

- Any parameters should be specified using the same techniques as that for describing additional required information when adding a new system device. Each parameter that is required for the trigger, condition or action must be described. For example, dimming an X10 lamp module to a specific level would require one parameter which is to be a percent. By describing any required parameters, user interfaces will be able to gather the parameter values from the user when they are describing their desired TCA.

Condition descriptions should state any trigger or actions which result in the condition in question becoming true. For example, the "is open" condition would become true if the "opens"/"open" trigger/action takes place. This is useful information to allow Homer to better understand the relationship between TCAs.

In the case of conditions and actions, the Component Gateway registers listeners with the Homer Event Coordinator. The listeners will be notified when Homer has been requested to verify a condition or perform an action respectively.

POSTING TRIGGERS When a trigger takes place, the bundle is responsible for notifying Homer. This is achieved very simply by calling a method provided by the Component Gateway to report a trigger occurred. The only information required is the system device and trigger identifiers, and any associated parameters. For example, *system device* temperature sensor[1] *trigger* rises above[1] *parameter* 28°C.

When the Component Gateway is contacted, it passes on the information directly to the Homer Event Coordinator where it is then broadcast as an event within OSGi. The event handling is described in more detail in Section 3.5.3.

HANDLING CONDITIONS If the Component Gateway is notified about a desired condition validation, it contacts the appropriate component directly, providing the details to allow the component to calculate the answer, and then responds with a simple true or false. The details include, again, the system device and condition identifiers, as well as any parameters. The system device type identifier is also provided for completeness. With this information, the component is able to verify if that particular system device meets the given condition.

Once a condition result is calculated, the Component Gateway contacts the Event Coordinator with this. The Event Coordinator can then broadcast this result for the requester (along with any other interested parties) to hear.

HANDLING ACTIONS Actions are handled in a very similar way to conditions. If the Component Gateway is notified about a requested action to be performed, it contacts the appropriate component, passing on the identifiers and parameters. The responsibility then lies with the component to perform the requested action.

3.6.2 *User Perception*

Unlike the developer, whose perspective revolves around the hardware, the user's view of the home is at a higher level. As an example, the user will most likely want a "Television" device type, with instances such as "Kitchen Television" or "Main Television". On a hardware (system) level this could correlate to multiple different hardware (system device) types. For example, the Kitchen Television may make use of an X10 Appliance Module for turning the television on and off, whereas the Main Television could make use of both X10 and an infrared controller, allowing both the turn on and turn off features of X10 as well as the more fine-grained control provided by the infrared controller.

The notion of user device types and their instances is discussed below.

3.6.2.1 *User Device Types*

The user (or system installer) can define any number of user device types with no restrictions. A user device type is simply described by a name (such as Television) and an icon. A unique identifier is provided automatically.

1 Replaced identifiers with their associated name for readability.

3.6.2.2 *User Device*

A user device is an instance of a user device type. Any number of user devices can be defined for the home. They are given a name (such as desk lamp, main television, outside temperature), their associated user device type, their location, and an icon. When describing this information, the user must also specify which system device types it utilises, and must provide any information these system device types require. For example, the desk lamp may use an X10 lamp module with address A2, the main television may use an X10 appliance module with address B3 and an infrared controller for a Sony KDL-9500D, and the outside temperature may use a Visonic temperature sensor with address 7D5A8. A default state can also be specified for the user device by choosing a condition from any conditions offered by the system device types in question. For example, the desk lamp and television could have default states "is off" (provided by X10 permitted conditions). The outside temperature would most likely not be given a default state, since it is such a variable measurement.

If the chosen system device types contain any actions, the user is asked to describe if and how they affect the environment. For example, turning on the desk lamp will increase the light level. This is discussed in further detail within Chapter 4.

When the user saves a new device, Homer creates a new system device instance for each system device type supported by the new user device. Each system device is associated with the user device in the database. Each component in charge of a supported system device type is notified of the new system device added to Homer (this is described in Section 3.6.1.4). This decouples the hardware data (such as X10 address) from the higher level information about the device (such as location and icon). The advantages of this strategy are also described in Section 3.6.1.4.

3.6.3 *Existing Homer Components*

Device support has been created for use in both home automation and telecare. Examples of possible Homer components are provided below. Although most of these are fully implemented, some (e.g. for Tunstall equipment or the Wii) are at an early prototype stage.

- Appliance Control: Appliances that are controlled via the mains, including lighting, fans and televisions can be controlled through X10 or Plugwise. Additionally, appliances that are controlled via infrared, including televisions, audio-visual systems and DVD recorders, can be controlled by Homer using IRTrans (`www.irtrans.com`).

- Communication: Communication services, including email, SMS, Twitter, message display on a digital photo frame, and speech input/output (using code from the University of Edinburgh) can be offered.

- Energy Consumption: Energy usage can be monitored per appliance using Plugwise sensors. This allows Homer to react to how much energy is being used and can help reduce energy consumption. For example, clothes washing might be delayed until other energy demands are lower.

- Environment: Oregon Scientific sensors (`www.oregonscientific.com`) are used for humidity and temperature. The Google Weather API is used to obtain the current weather or a forecast for chosen locations.

- Home Automation: Sensors from companies such as Tunstall (`www.tunstallhealth.com`) and Visonic (`www.visonic.com`) include movement detectors, pressure mats and reed switches (cupboard, door, window). Future support will include curtain/blind controllers, garage door controllers, and remote door locking and unlocking.

- Telecare: Telecare sensors from Tunstall and Visonic include alarms (pendant, wrist), hazard detectors (flood, gas, smoke), medicine dispensers, and pressure mats (bed, chair). Specialised sensors include detectors for enuresis, epileptic seizures and falls.

- User Interfaces: Various 'Internet buddies' are supported as they appeal to less technical users. Examples of these are 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`) can be used to communicate using gestures and tactile output. Using code from the University of Glasgow, similar functionality found with a WiiMote is also available from the SHAKE (*Sensing Hardware Accessory for Kinaesthetic Expression*, `www.dcs.gla.ac.uk/research/shake`). Screen-based user interfaces, such as through an iPhone or website, are discussed in Chapter 5.

3.7 SERVICES

Homer Services are designed to handle common tasks and functionality used by component developers. This saves duplication of logic within different components (or dependencies between them), eases the development process, and encourages separation of logical units of code. This section introduces the concept of OSGi services and how Homer handles them. Next, the section looks at how the developer can write and make use of Homer services. Finally, existing Homer Services are described.

3.7.1 *OSGi Services*

Within OSGi, services can be registered and obtained from any bundle. A service can be any Java interface, with an implementation that is registered with the OSGi framework. Once registered, anyone else within the framework can request the service for their own use.

Unfortunately the design of OSGi services is very limited. In reality all that OSGi provides is a mechanism for obtaining an instance of a class which was instantiated elsewhere. OSGi service handling has the following limitations:

- Services are looked-up by using the class name of the service, rather than a more friendly alternative. There is no global way of handling services of the same nature, such as loggers.
- The same instance is passed around, where in some cases it may be desirable to have a new instance of the service for each usage.
- A service would be accessible by any developer, as there is no central means of controlling access to the services.

It was decided that a custom Homer Service Gateway would be used in favour of OSGi services due to these limitations.

3.7.2 *Homer Services*

Similar to the design and installation of Homer Components, developers create dedicated bundles which offer individual services that implement the Homer Service interface. Services are registered with Homer through the Service Gateway (discussed in Section 3.5.6).

A Homer Service can be public or private, running as a single or multiple instance. Public services are available to all Homer developers, whereas private services are written and used only by the core Homer framework. If single instance, only one instance of the service will exist at runtime. This is useful for services like a database. Multiple instance services, on the contrary, are services where each service user should have their own instance.

Homer Services are managed by the Service Gateway, which was discussed in Section 3.5.6.

3.7.3 *Developer Perception*

3.7.3.1 *Writing a Service*

If a developer has a useful service to offer, or makes use of a particular piece of functionality across multiple bundles, it can be wrapped as a Homer Service for themselves and others to make use of.

To create a Homer Service, the developer should make a new OSGi bundle which will represent it. A class should be written which implements the Homer Service interface. This interface ensures that required properties are supplied and methods are supported. These include if the service should be single or multiple instance, whether access should be public or private, and deal with instantiation and service lifecycle maintenance (such as reaction to starting and stopping the framework). Once written, the service can be registered with the

Homer Service Gateway by calling the appropriate registration method and providing the class and a short user-friendly descriptive name.

At runtime, when the service bundle is installed into the framework and started, the new service will be registered with Homer and made available to all those with the correct permissions.

3.7.3.2 *Using a Service*

To make use of an existing Homer Service its given name should be used. Firstly, the component developer gets hold of the Service Gateway through the Component Gateway. Then the developer can query the Service Gateway, requesting the service of interest. If the service is available at runtime, an instance of the service is returned for the component's use.

If a service has been requested but does not exist, possible future work could be undertaken to allow Homer to make attempts to locate and install the necessary service as required. Currently, the Service Gateway simply returns a null response and the component developer must handle this.

3.7.4 *Existing Homer Services*

Homer has a few services that have been found useful. These are:

- Logging: A logger is provided to offer the standard features of any logger library. This provides consistency amongst all loggings, controlled singular output of log messages, and saves developers having to include a logger within each bundle.
- Serial Port Communication: This service offers a means for hardware component developers to connect to a specified serial port and handle communications with the device.
- Email Receiver: This service allows developers to register an email account and to be notified when new email is received.

3.8 CONCLUSIONS

This chapter presented Homer, a Home Automation system which addresses the full range of requirements that emerged from the user studies described in Chapter 2 and, most importantly, provided a system to base the policy server work upon described in Chapter 4.

A service-oriented framework was developed which offers a plug-and-play style architecture to allow third-party developers to write components and services for the home. This ensures that Homer's functionality offered to end users is not reliant solely on the developers of Homer, but rather on anyone who decides to add support for a particular product or service.

Homer offers a fully embedded and device-independent policy server (discussed in Chapter 4) which can make use of Homer components and the services they support autonomously. This ensures that the components can be programmed at a high level, without dependencies or native code within Homer.

A web server (discussed in Chapter 5) is also offered, which exposes all the functionality of Homer through a web API. This allows third-party developers to write user interfaces for any device and any purpose.

With the core Homer system in place research contributions can be made through the policy engine work, including a custom policy language for the home and unique overlap and conflict detection techniques. These are described in Chapter 4.

4 HOMER: POLICIES

This chapter explores the nature of policies within the home and the journey taken to design, develop and evaluate a policy system built for Homer.

4.1 INTRODUCTION

Currently most home automation companies combine devices and services within the home at code level. Typically when a home system is installed, the occupiers are asked how they would like their home to behave and then the developers code this logic at the system level. When users want to change how the home behaves they may have to pay the developer to amend the code.

User needs within a home change over time, so it makes sense that the user can easily and quickly change how the home behaves. Unfortunately this view is currently not shared with many of the home automation companies in today's market (an overview of the limited support that some companies do give for programming the home was discussed in Chapter 2, and in more depth in Chapter 5).

A home system with a library of components could have some form of logic to combine them. All components offer information about the environment, events that occur, and/or can influence the environment in some way. This logic lends itself to policies, which are often represented in Event-Condition-Action (ECA) form. An example could be: "when freezing weather is forecast, if the user is on holiday, then switch the central heating on for an hour in the morning and evening". A telecare example could be: "when the user is late in rising, if it is not the weekend, then alert a relative to this by text message". This concept of integrating home devices and services using policies allows the home system to be managed at a higher-level. With a user interface in place, the user can then "program their home" (see Chapter 5).

This chapter first discusses the background of policies. The requirements for the policy system are then introduced, leading on to an exploration of existing work in this field. The policy language (named "Homeric") and server developed for Homer are then presented, along with the methods used to handle conflicts. Finally, the work of this chapter is evaluated in a case study.

4.2 BACKGROUND

The term 'policy' is centuries old, with the general definition:

"A set of ideas or a plan of what to do in particular situations that has been agreed officially by a group of people, a business organization, a government or a political party."

Cambridge English Dictionary, `dictionary.cambridge.org`

Expanded slightly: a policy is a description of desired behaviour under specified circumstances. A policy is typically described by a set of statements outlining the specific intentions, regulations and actions, as well as other meta information such as the purpose of the policy, who or what the policy applies to or affects, and the time-scale or date for when the policy applies. It should be noted that a policy and a law differ as a law requires or prohibits actions, whereas a policy aims to guide actions toward a desired goal. An in-depth exploration into the history of policies can be found in work by Campbell [21].

Within the computing field a policy is a high-level statement specifying a list of desired actions to occur when certain events occur and conditions are met. The concept of policies allows desired functionality and rules to be written for a given system separately from the underlying code. This allows for the behaviour of systems to be handled, configured and reconfigured without the need to reprogram.

A computing system typically receives inputs, can verify different conditions, and can perform actions. The system features do not tend to vary once provided, however the runtime combination of these features can. Hence, it makes sense to separate the features from combination logic.

During the past decade policies in computing have been applied to a wide array of fields. In the early days policies were commonly applied to networking and to the handling of security and user authentication (a history of policies can be found in [16]). Nowadays policies have many additional applications, including call control (e.g. [14]), business rules (e.g. [55]), access control (e.g. [72]), user preferences (e.g. [110]), sensor data collection (e.g. [22]) and power management (e.g. [116]).

Within the context of the home, there is a wide range of devices and services that can be supported by the home system. For example, a lamp can be turned on and off, a door opens and closes, the temperature can be queried, and an SMS can be sent and received. These individual pieces of functionality may be combined in endless ways to create very different effects. The combination of these pieces of functionality should be dynamic, reconfigurable and scalable. This can be achieved through the use of polices, allowing the various triggers, conditions and actions offered by the system to be combined.

4.3 REQUIREMENTS

Given the user study in Section 2.3.3 exploring how technically minded individuals might wish to write policies for the home, many requirements can be inferred for the Homer policy system and language.

The language for Homer must be flexible, supporting a range of policies that could be written for the home. As the user interface for writing policies for Homer is separate (discussed in Chapter 5) from the underlying language, it is possible for differing user interfaces to be designed and supported by Homer. Therefore a more sophisticated language can be offered by the policy server, allowing Homer user interfaces to decide which of the language features to offer.

Language Features The following language features for the Homer policy system were derived from the user study presented in Section 2.3.3:

- **When-Do Format** Policies should be expressed in a *when* – *do* format, rather than in the *when trigger* – *if condition* – *then action* format.
- **Composite Terms** It should be possible to combine triggers and conditions with *and/or* (but avoiding *not*).

- **Ordered Terms** Support for ordered triggers and conditions is desirable, using an appropriate operator such as *then*.
- **Blurred Triggers and Conditions** The distinction between triggers and conditions should be blurred.
- **Conditional Actions** The ability to qualify actions with conditions is desirable.

Server Features The following features are required for the Homer policy server itself:

- **Scalable Number of Policies** Based upon what users would be able to state about their home, the policy server should be able to handle 125 policies[1] as a minimum.
- **Local Policies** All policies for a home should be stored locally, within the home system, for efficiency, offline conflict handling, and security reasons.
- **Offline Conflict Handling** is crucial within a home system, as multiple policies will be written perhaps by different people and perhaps with large time frames between them. Therefore, there is high chance that conflicting policies will be written for the home. Detecting these conflicts when the user tries to save the newly written policy is desirable, as opposed to runtime analysis where the system must make attempts to resolve conflicts as they arise. This could confuse the residents of the home if events do not occur when they are expected.

This section has stated the requirements for the Homer policy system. Existing policy solutions are explored in Section 4.4 and each is analysed against this set of requirements. Conclusions are drawn at the end of the Section, where an appropriate policy approach is discussed given the requirements for Homer. The Chapter then introduces, discusses and evaluates the chosen solution and its features.

4.4 STATE OF THE ART

The following state of the art discusses possible enforcement, representation and conflict handling of policies, focusing only on the most relevant work. Existing work is analysed, and final conclusions are drawn to decide which policy solutions will work best for Homer.

4.4.1 *Existing Work*

The following section discusses existing work in policy enforcement, representation and conflict handling.

4.4.1.1 *Policy Enforcement*

Leading existing policy enforcement work is by Sloman and Turner, with their systems Ponder and ACCENT respectively. Each are discussed in turn.

ACCENT (*Advanced Component Control Enhancing Network Technologies*, `www.cs.stir.ac.uk/accent`) was a research project funded by the Engineering and Physical Sciences Research Council (EPSRC) between September 2001 and March 2005. It resulted in a call control language which could be used to define user policies and find conflicts amongst them. ACCENT therefore allowed users to manage and configure call preferences.

There are three layers comprising the ACCENT system: a communications layer to communicate with the managed system, a policy server layer to store and deploy the policies, and a user interface layer to allow policies to be written by end users. Complementing the policy server is a goal server that allows the user to define high-level objectives. There are various policy wizards which allow non-technical users to define policies. Conflicts among the policies are automatically detected and resolved (for example, the user wishes the house to be warm, but also wishes to save energy).

1 As a rough calculation: an average of 5 rooms with roughly 5 appliances or devices within and 5 policies controlling such devices results in 125 policies.

The underlying language which represents the goals and policies is APPEL (*Adaptable and Programmable Policy Environment and Language*, www.cs.stir.ac.uk/appel), discussed in more detail in the following section.

ACCENT has been adapted for use in other domains, including sensor networks in the PROSEN project (*Proactive Condition Monitoring of Sensor Networks*, www.cs.stir.ac.uk/prosen) for wind turbine management and, more relevantly, telecare in the MATCH project (*Mobilising Advanced Technologies for Care at Home*, www.match-project.org.uk).

ACCENT is of significant relevance to this work. This research has been conducted at Stirling, has been used within the home, and offers sophisticated policy tools. Due to the relevance of this work, a more detailed analysis of how to use ACCENT within Homer was carried out.

The main advantages of the ACCENT system are local expertise, ample documentation and appearing to meet the requirements for the Homer policy system. APPEL has some interesting features which could be desirable within a home, yet have not been discussed in the requirements of Homer. These features include:

- **Timed Policies** which can allow the policies to have timers within, such as "*when* the garage door is opened, set a timer for two minutes, *when* this expires close the door". To fit into Homer it would be more desirable to have a timer component, which would allow policies of the form: "*when* the garage door is opened *do* wait two minutes *then* close the door".

- **Policy Variables** which can increases flexibility when writing policies. Two examples include: "*when* the user has not risen by 10AM increment the count of such occasions, *when* this count reaches 5 within a week, text a relative" and "*when* I receive an email from a friend *do* display the email content on my television".

- **Runtime Conflict Detection *and* User-Defined Resolution** can be extremely useful within a home environment, as conflicts will undoubtedly take place. It is also crucial, from the Homer philosophy, that the user is able to define how the home should behave if such conflicts do arise. This means that the home system remains unintelligent, simply following commands specified by the user.

However, there were some critical drawbacks in using ACCENT within Homer:

- **Poor Identifiers** ACCENT is very closely tied to the application domain it is being used for. Policies embody the particular devices and the TCAs associated with them. The given name for a particular device instance, such as "front door", is used as the unique identifier for the instance within ACCENT. This is also the case for device types (such as "door") and actions (such as "opens"). Using the user-given name as a unique identifier has major drawbacks, including the most obvious notion of wanting to change the name of a given instance or supporting different natural languages.

- **Philosophical Differences** As much as it is advantageous to develop solutions that can be applied to different problem domains, it also has drawbacks. ACCENT was originally designed for telephony, which requires decisions to be made live on behalf of the user (such as when choosing which policy to execute based on "policy preferences" or which actions to carry out to meet various "goals" of the system). For telecare this was also the case. A resident's home system would make decisions on behalf of carers, family and friends. It would be rare for the residents themselves to decide how their home should behave. This is perfectly acceptable for the application domains of ACCENT so far. However, within the home automation domain the author believes it is vital for the residents to always feel in control of their home at all times. The system should never make independent decisions on behalf of the resident as this will cause confusion and reduce trust. For this reason many of ACCENT's philosophies do not fit with Homer.

- **Policy Language** The underlying policy language, APPEL, is very intertwined with the policy engine and is incapable of supporting a different language. As discussed in the next section, APPEL does not meet the language requirements for Homer.

Due to this list of limitations, and the fact that it does not meet the given requirements in Section 4.3, this project was not considered further as a possible policy system for Homer.

Ponder [32] (and its derivative Ponder2, www.ponder2.net) is a policy system that also incorporates a self-contained, general-purpose, stand-alone management system. It is an ongoing research project from Imperial College, London.

Ponder supports an awareness of events, allowing external Java-based applications to communicate with Ponder through events alone. Ponder has been designed to offer a simple, extensible and scalable means of supporting policies within a Java system. PonderTalk [133], a high-level, object-oriented language based on Smalltalk, is used to control and configure Ponder. A number of research projects have made use of Ponder, including body-area networks of sensors and actuators [60], unmanned autonomous vehicles [134], and large web service-based infrastructures [112]. Ponder is even available on Android.

The supported policy language offered by Ponder is discussed in the following section, though was found to not support the desired language features for Homer.

Ponder offers a sophisticated set of tools for defining, compiling and managing a set of policies. It also supports domains and embedded conflict handling and refinement. However, Ponder can not easily interact with the managed system, and policy specification is considered an offline activity. This means that Ponder is difficult to install, run and experiment with.

Ponder has not been designed for use in a home setting, and suffers the same list of drawbacks as discussed for ACCENT. Turner discusses Ponder's shortcomings in [104]. For such reasons Ponder was also discounted as a possible policy system for Homer.

4.4.1.2 *Policy Representation*

Three leading policy representations and languages are discussed in turn.

APPEL (*Adaptable and Programmable Policy Environment and Language* [129]) is the underlying language which represents the goals and policies in ACCENT (discussed in the previous section). It describes the policies using an XML-based grammar. APPEL supports two policy types: regular policies in the ECA format, and resolution policies which allow administrators to customise how conflicts among regular policies are handled.

Each policy can be given a preference ("must", "should", "prefer", "prefer not", "should not", "must not") to allow users to state how strongly they would like the policy to be considered when selected for execution.

ACCENT policies must be in the format: "when *triggers* (optional: if *conditions*) do *actions*". Triggers and conditions may be joined with *and* and *or* operators, and actions combined with just the *and* operator.

With respect to the requirements for Homer, outlined in Section 4.3, there are a number of language features that ACCENT does not support. These include:

- having policies in the When-Do (WD) format instead of an ECA format
- allowing conditions to be treated as triggers, whereby they can be interspersed amongst the list of triggers
- allowing ordered terms of triggers and conditions
- allowing conditional actions.

Due to ACCENT not supporting the required language features outlined in Section 4.3, ACCENT is not a valid policy language solution for Homer.

Formal Logic can be used to represent policies in a strict, unambiguous form. It can be difficult to formulate such policies, however they are simpler to interpret by computers. Example projects which make use of formal logic to represent policies include KAoS [136], Rei [57] and work carried out by Owen *et al.* [102]. These research concepts are relatively similar, so for that reason only one is discussed.

Description logic is used by Owen *et al.* [102] for representing policies within pervasive computing, primarily focused in the office environment. The approach is very user centric, wanting to simplify the notion of policies and find a language flexible enough to support the wide array of policy formats that non-technical users wish to express.

The policy language designed and used by Owen is in the form of a rule which contains one precondition and one postcondition. Examples of policies that Owen designed the language for include:

- "Send me an email when my printer goes offline."
- "Print colour documents to the colour printer."
- "If a document is sent to printer x and it is offline, send to printer y instead."

The language supports the when-do format and can blur the distinction between triggers and conditions by treating all triggers as explicit conditions. However, it is designed to be simplistic, aimed at non-technical end users. The language, as a result, is rather primitive and does not support optional terms, ordered terms, multiple actions or conditional actions. Examples of policies that Owen could not support (purposely aimed at the office environment) include:

- "Send me an email when my printer goes offline or runs out of ink."
- "If printer x runs out of ink email me and email computer administration."
- "If a document is sent to printer x then printer x runs out of ink, send me an email."

Due to the failure of meeting the Homer requirements outlined in Section 4.3, as well as the uncertainty of success, difficulty to reproduce, and concerns over the expressibility of policies, it was decided that formal logic would not be best suited for representing policies with Homer.

Ponder2 Ponder2 supports ECA policies (termed obligation policies within Ponder), but cannot support *when – do* policies. The language is very restricted: it is limited to simple *and* and *or* combinations of the same term type (so it would not be possible to write "*when trigger or condition*"), requires conditions to be handled in the traditional way (instead of combined with triggers within the same *when* clause), cannot support ordered terms within the *when* clause, and finally cannot support conditional actions. Example policies that could not be represented within Ponder2, therefore not meeting the requirements for a Homer policy system, are:

- "*when* the front door is open *do* turn on the hall light."
- "*when* the front door is open *or* the back door opens *do* turn on the hall light."
- "*when* the front door opens *then* the front door closes *do* turn off the hall light."
- "*when* the front door is open *then* the back door is open *do if* it is windy outside *do* close the front door."

Ponder2 is not flexible enough to represent the range of policies that are required for Homer, therefore it was discounted as a possible policy solution for Homer.

4.4.1.3 *Policy Conflict Handling*

Within a home environment it is inevitable that policy conflicts will arise. A conflict is when actions take place within the home causing undesirable or unexpected outcomes. There are typically multiple residents within the home who will be writing, amending and removing policies over time as their requirements change. It is therefore highly desirable to prevent as many conflicts as possible.

Work on policy conflict handling, traditionally termed feature interactions, originated from telecommunications services [105]. Here, various call features would interact at runtime and cause undesirable and sometimes unpredictable behaviour for users. An exhaustive look at existing techniques to help detect potential call feature conflicts are described and compared by Calder [20] and Keck [59].

Within telecommunications services feature interactions, research exists which aims to filter features which do not conflict. This helps to reduce the cost of conflict analysis as the process would be performed on a subset of all features. Novel approaches are presented in [63], [67] and [95], which aim to filter features prior to conflict analysis. The techniques cannot be applied readily to policy systems in general, due to extreme differences both philosophically and technically. However, the broader concept of filtering has not yet been applied to the general policy field. In the case of Homer with offline policy conflict detection, the number of existing policies any given policy should be analysed against could be dramatically reduced by examining the when clause of the policies to observe if they could happen at the same time. This is of benefit as it reduces the number of potential conflicts reported to the user.

Within a policy system there are two main approaches for handling conflicts. One approach makes attempts to detect and handle conflicts at runtime (online), whilst the other works statically at definition time (offline) [77].

ONLINE Runtime conflict detection is extremely useful in distributed policy systems, where policies may interact with other policies from a different policy system. This is a frequent scenario within telephony, where the caller's policies may interact for the first time with a receiver's policies. There, possible conflicts must be handled at runtime. ACCENT, discussed in Section 4.4, is a policy system designed for telephony which handles such runtime conflicts.

OFFLINE Offline, or static, conflict handling is a useful technique to prevent or prepare for conflicts in advance. It can be used to aid prevention of policy conflicts by notifying the user of any potential conflicts at the time of writing a policy, or to ask the user how any such conflicts should be handled in advance of them ever occurring.

Existing solutions for conflict handling within the policy domain are now discussed.

ACCENT provides a solution for handling conflicts across both local and distributed policy systems [15] at runtime. The distributed approach purposely avoids the use of a centralised resource, instead opting to send policies (including conflict resolution policies) to the server (in telephony: the callee). The server collates all the policies, and formulates a list of requested actions. The resolution process attempts to find and resolve any conflicts by making use of conflict resolution policies. The local solution works in a similar manner, by collating all the actions requested to happen when a particular trigger occurs, and again handling any conflicts among these.

ACCENT, as well as handling online detection, can also handle offline policy conflict detection. This is achieved using a separate tool called RECAP (*Rigorously Evaluated Conflicts Among Policies* [23]), which mostly focuses on telephony, though it could also be applied to the home. RECAP makes use of ontologies to obtain meta-data about action effects, such as technical (e.g. bandwidth) and social (e.g. privacy) aspects. Policies can therefore be analysed to look for any which will alter the same effect, and flag these as conflicting pairs. A simple user interface allows an end user to confirm or deny any detected conflicting pairs, and optionally write resolution policies which describe what the policy server should do if such a conflict were to arise at runtime.

The offline conflict detection algorithms used within ACCENT originated from telephony, and have never been applied to home automation or telecare. Resolution policies are an attractive feature, however the metadata describing the effects of actions require specialised technical expertise as they are defined by developers and cannot be altered or customised at a higher-level.

KAoS [135] is a policy system developed by researchers at the University of West Florida. It supports offline policy conflict detection and resolution using a version of Stanford's Java Theorem Prover (JTP). When searching for conflicts, all policies are sorted according to user-defined criteria, then analysed for any potential issues (such as undesired authorisation and obligation combinations). The resolution process involves determining the lowest priority policy from a conflicting policy pair, and allowing their custom policy harmonisation algorithm to modify this policy logic to the minimum degree necessary to resolve the conflict, producing zero, one or more replacement policies. Their resolution process, therefore, involves no user input.

KAoS is, like Homer, designed to ease policy management for end users. However, KAoS differs greatly in its approach as it handles policy conflicts behind the scenes and alters policies without user approval or consent. Within Homer, this would be considered disadvantageous as the user would write a policy that may well be altered, or affect existing policies, without the user knowing.

Wilson worked on a solution for handling runtime conflicts within home automation [143]. The home has many services and devices which are made by different manufacturers and developers, so there are plenty of conflicts that could arise within the home. Their solution draws from the concept of file locking, such that when a file is in use it becomes locked so that no other user can make changes. Similarly, if any policy is interacting with a particular device or service within the home then that device or service becomes "locked", meaning no other policy or user can change it.

Wilson, as well as Nakamura *et al.* [94], adapted the work of previous researchers within the feature interaction domain. Typically this work was applied within the call control area, so Wilson and Nakamura adapted and extended the existing research for the home environment. Nakamura took a mathematical, object-oriented, low-level approach whereas Wilson worked at a more flexible higher-level.

Both Wilson and Nakamura model the effect of actions upon the home in a relatively rigid and primitive manor, in order to gauge what actions should be permitted at runtime. With respect to Homer, and the requirements outlined in Section 4.3, this is not a desirable solution as the home would not behave consistently and predictably for the user. Offline conflict detection is far preferred for Homer, however the general approach of making use of meta-information about the effects of actions upon the environment is be explored further in Section 4.7.

4.4.2 *Conclusions*

This section discussed existing work in policy enforcement, representation and conflict handling.

It has been shown that related approaches for policy enforcement did not fit the requirements for managing the home. ACCENT and Ponder2 could conceivably have been adapted for Homer, however neither offered a rich and flexible enough policy language that met the requirements for Homer. It was decided that a custom-built policy system would be written. This was designed to handle all the requirements (discussed in Section 4.3) cleanly and efficiently, and to be integrated with Homer as a separate module. Policy enforcement within Homer is described in Section 4.6.

Existing policy representations in leading policy systems were discussed, showing that many of the desired language features for the home could not be represented. These features include a when-do format, ordered and optional terms, blurred distinction between triggers and conditions, and finally conditional actions. These language features emerged from user studies discussed in Section 2.3.3 and, within this thesis, are deemed valuable for a language designed to allow both technical and non-technical individuals to program their home. For these reasons a new, custom language will be designed for the home. This is described in Section 4.5.

Existing conflict detection techniques emerged from the call control domain, and as such are typically designed to work at runtime with a pre-set collection of meta-information about the affect of actions. An approach is required that makes use of the advances made by Wilson and Nakamura, primarily the nature of comparing environment effects to determine conflicts, and the idea of describing the effect of a device and its functions on the surrounding environment. An improved solution should extend their work by offering more sophisticated and user-customisable handling of conflicts to handle the challenges described in Section 4.7.2.1. In

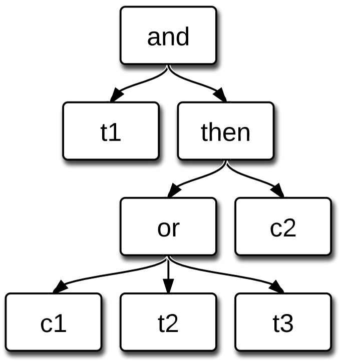


Figure 4.1: Sample When Clause Tree

addition to the conflict detection work, inspiration should be drawn from the notion of filtering potential conflicting call features and applied within the policy field. Both filtering and conflict handling techniques for Homer are described in Section 4.7.

4.5 POLICY LANGUAGE

The requirements for the language are stated in Section 4.3. This section describes the language designed for Homer ("Homeric") based upon these requirements.

4.5.1 *Format*

4.5.1.1 *When Clause*

A single *when* clause introduces a list of triggers and conditions. These express the events that must occur, and circumstances that must be met, for a policy to fire. The language blurs the distinction between triggers and conditions, allowing the user to mix and match triggers and conditions freely within the *when* clause. This results in a simple but flexible language.

Each term within the *when* clause must be combined using an *and*, *or* or *then* operator. This allows for terms to be required, optional or ordered. The *when* clause has a tree structure which allows for precedence of terms to be stated unambiguously. Any leaf of a tree must be a trigger or condition and any parent node must be an operator. Figure 4.1 demonstrates this visually, and represents the clause: "*when* t1 *and* ((c1 *or* t2 *or* t3) *then* c2)".

Due to Homer's support of ordered triggers and conditions, there was a need for a time limit that these terms must occur within. This is called a duration limit. Duration limits determine how closely in time events must occur in the when clause. As an example, consider a policy that detects night wandering: "*when* the resident gets out of bed at night *then* opens the front door". A time limit of ten minutes might be appropriate for this. Without such a limit, the user getting up earlier than usual, then later checking if the milk has arrived could be misconstrued as night wandering. Any groups of terms that are joined using the *and* or *then* operator can specify a duration limit. If no duration limit is specified then the default duration limit for a policy is 60 seconds. This can be seen in the language specification (in Section 4.5.2) with further details in Section 4.6.3.

4.5.1.2 *Do Clause*

The *do* clause is simply one or more actions that are combined using *and* operators.

The only additional complexity is the support of conditional actions. Conditions are supported within the do part of the policy as it was found that users often wish to impose conditions on actions. A typical policy might be: "*when* I get home from work *do* play my favourite music *and if* it is dark outside *do* turn on the hall light". Anywhere amongst the list of actions there can be a condition, with an associated list of actions to be carried out if the condition is met, and optionally a list of actions if the condition is not met.

Figures 4.2 and 4.3 visually demonstrate two sample *do* clause trees for Homer. Figure 4.2 reads: "*do* a1 *and* (*if* c1 *do* a2)". Figure 4.3 reads: "*do* a1 *and* (*if* (c1 or c2) *do* a2 *else do* a3)".

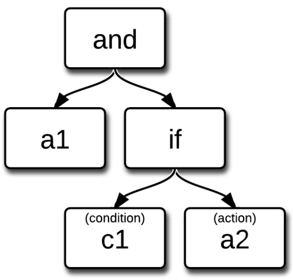


Figure 4.2: Sample Do Clause Tree - Simple

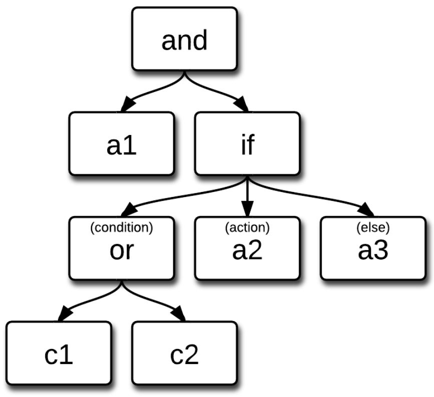


Figure 4.3: Sample Do Clause Tree - Complex

4.5.2 *Language*

The policy language, defined using Antlr (www.antlr.org), for Homer is:

```
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)* ;


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)* ;


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,
                               DFAS8FD62FAD9DF00033498D */ ;
```

4.5.3 *Representation*

An individual policy is stored and expressed using JSON (*JavaScript Object Notation*, www.json.org). JSON is a text-based, language independent data-interchange format. It is a lightweight and easier to parse alternative to XML.

A policy must specify a unique identifier, a name, the author, the dates it was created and last edited (in UNIX time), if the policy is enabled, and finally the *when* and *do* clauses. A skeleton policy JSON representation looks like:

```
"policy": {
    "id": "123", // id of the policy, set by the Homer Database
    "name": "Summer Heating", // user chosen identifier name of the policy
    "author": "Alice", // author of the policy
    "dateCreated": 1326985200, // date created in UNIX time
    "dateLastEdited": 1326985200, // date last edited in UNIX time
    "enabled": true, // boolean: true if enabled, false if not
    "when": {...}, // JSON object of when clause
    "do": {...} // JSON object of do clause
}
```

The *when* and *do* clauses of a Homer policy represent the tree structure of the policy. Any parent node must be an operator (e.g. *and*, *or*, *then*, *if*) which states how its array of children are combined. A child is either an operator node, or an event node. An event node is a leaf,

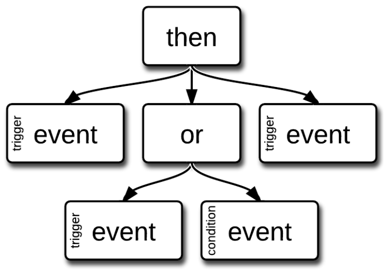


Figure 4.4: When Clause for JSON Example

which must provide the user device ID, event ID, the type, and any parameters to describe the exact trigger, condition or action specified by the user. This is discussed again in Section 4.6.3, and in more detail in Chapter 3. As an example, take the following policy:

> *when* office door opens
> > *then* (movement is detected in hall *or* front door mat is being stood on)
> > *then* office door closes
>
> *do* turn on heating
> > *and if* light level is below 50% *do* turn on the hall light.

The *when* clause is relatively simple, with three terms combined using *then*, with the middle term an option of two terms combined using an *or*. The tree shown in Figure 4.4 shows the hierarchical representation of this policy, which within JSON would look like:

```
{ "then": [
    { "event": {
        "userdeviceid": "12", // office door
        "eventid": "185DDD121346C5A207A4", // opens
        "type": "TRIGGER" }
    },
    { "or": [
        { "event": {
            "userdeviceid": "25", // hall movement detector
            "eventid": "1F960252B3EF9E51B25", // movement detected
            "type": "TRIGGER" }
        },
        { "event": {
            "userdeviceid": "6", // front door mat
            "eventid": "E8DF87ACB3EF9E51D76", // is being stood on
            "type": "CONDITION" }
        }
    ]},
    { "event": {
        "userdeviceid": "12", // office door
        "eventid": "76E4D7FE437D04A8B323E", // closes
        "type": "TRIGGER" }
    }
]}
```

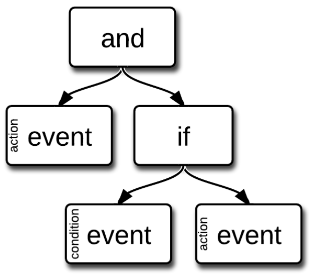


Figure 4.5: Do Clause for JSON Example

A diagrammatic tree and the JSON code representation are now given for the *do* clause from the example policy (”*do* turn on heating and *if* light level is below 50% *do* turn on the hall light.”), just as was given for the *when* clause. The tree is shown in Figure 4.5, and the JSON is as follows:

```
{ "and": [
    { "event": {
        "userdeviceid": "16", // heating
        "eventid": "9D9E9DC9A9D9E9F658D56E87123", // turn on
        "type": "ACTION" }
    },
    { "if": {
        "condition": [
            { "event": {
                "userdeviceid": "19", // light level
                "eventid": "D2E1F5D7E89A912A5E", // below
                "type": "CONDITION",
                "parameters": ["50"] } // parameters always in array for consistency
            }
        ],
        "action": [
            { "event": {
                "userdeviceid": "2", // hall light
                "eventid": "C7D8E65A1A1133AD45E6", // turn on
                "type": "ACTION" }
            }
        ]}
    }
]}
```

The Homer technical report [78] contains further information and further examples of Homer policies expressed in JSON.

4.5.4 *Applicability*

Homeric is a language that is custom designed for the both technical and non-technical people to be able to program their home. The language has emerged from user requirements, and focused on providing a flexible and advanced language for technical users, as well as being able to offer a simplistic language which less technical users can work with. Homeric offers advanced language features as requested by very technically capable people, and due to the range of operators and notion of precedence highly sophisticated logic can be represented in one policy. At the same time, the design of interchangeable triggers and conditions in favour of the traditional *when* <trigger> *if* <conditions> *then* <actions> has hugely simplified the policy language for less technical users [80]. This approach and goal is different than existing policy work, which typically tries to design a hybrid solution. This results in a language that is too restrictive for technically capable individuals, and too complicated for those less technical.

The Homer policy language, although designed for the home, could be used in other domains. When a policy language is required to combine triggers, conditions and actions,

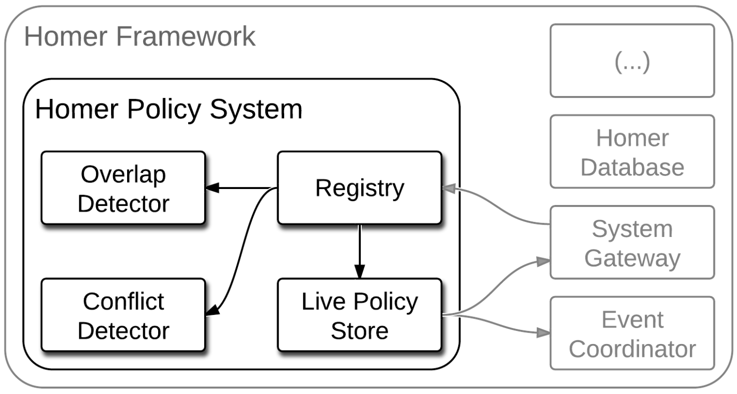


Figure 4.6: Homer Policy System

regardless of domain, Homeric could be used to offer a highly flexible and sophisticated means of combing the terms. Homeric has been tested within the telecare field, but not directly with any other field. Nonetheless, it is believed that Homeric could be readily applicable to other policy-ready fields.

Whilst Homeric offers novel language features – including blurred distinction between triggers and conditions, ordered events and conditional actions – it lacks some features of more mature policy languages. These features were discussed in detail in Section 4.4 and include timed policies, policy variables and prohibition policies. However, these features are entirely possible within Homeric and are purely an implementation task.

Homeric has been introduced and described within this section. The language is a novel contribution to research, offering a fully tailored language for the home which is designed to support the flexible nature of home policies desired by both technical and non-technical individuals.

Novel features of the language, which extend upon the work by Owen [102], Sloman [32] and Turner [129], include the WD format for policies, the blurring of distinction between triggers and conditions, support for ordered terms, and finally allow actions to be conditional.

4.6 POLICY SYSTEM

Homer policies define how the home should react when particular events occur. The Homer policy system manages and enforces all policies written. The policy system architecture, implementation and enforcement are discussed in this section.

4.6.1 *Architecture*

The overall Homer architecture diagram was given in Figure 3.1. This shows the policy server in relation to the Homer framework components. Figure 4.6 introduces a more detailed diagram of the Homer policy system.

There are four parts which comprise the policy server. Firstly, the **Registry**, which handles saving new policies, editing and deletion of existing ones, and enabling and disabling a policy. The registry is discussed in Section 4.6.2. The **Live Policy Handler** for enforcement of enabled policies is discussed in Section 4.6.3. The **Overlap Detector**, which is used to detect if a policy's *when* clause is valid, and if a policy overlaps with a given set of policies; this is discussed in Section 4.7.1. Finally, the **Conflict Detector** detects conflicts between a policy and a given set of policies and is discussed in Section 4.7.2.

The policy server is embedded within the Homer architecture. This allows the policy server to make use of the internal event coordinator (discussed in Section 3.5.3) to listen for triggers from components, and the System Gateway (discussed in Section 3.5.6) to validate conditions and request actions to be performed. There is also direct access to the Homer Database (discussed in Section 3.5.2) for the saving of policies, and for accessing any data required about terms and devices.

4.6.2 *Registry: Policy Management*

The Registry exposes the functionality of the policy server to Homer. This includes the ability to add new policies to Homer, and to edit and delete existing policies. The Registry also offers the ability to enable and disable existing policies.

When new policies are added to the policy server various tasks are performed:

1. Validation
   (a) Firstly, the policy server validates the JSON policy representation to ensure that the policy is correctly written, conforms to Homeric, and all event identifiers (trigger, condition, action and user device IDs) exist and are correct.
   (b) Secondly, the policy server checks that the *when* clause is valid. It achieves this by making use of the Overlap Detector. A policy is considered valid when it is possible for the policy to be fired. An example invalid policy would be "*when* the front door is open *and* the front door is closed". It is not possible for the front door to be both open and closed at the same time, therefore this policy is invalid. The Overlap Detector simply examines the policy for possible overlaps. If no overlaps are found, the policy is deemed invalid. Technical details of the Overlap Detector can be found in Section 4.7.1.
2. Conflict Handling (discussed in more depth in Section 4.7)
   (a) The first stage of handling conflicts is to detect overlap between the new policy and existing policies.
   (b) The second stage is to detect if there are any conflicting actions that will be carried out by the new policy and any overlapping existing policies.
3. Storage and Activation - only if any conflicts have been handled or are to be ignored (as stated by the user).
4. The policy server saves the policy into the Homer database. The information is obtained from the JSON representation of the policy which was passed to the policy server.
5. Once the policy has been saved into the database it is passed to the Live Policy Handler to be enforced. This is discussed in Section 4.6.3.
6. If the policy was set to be enabled in the original policy description, the Live Policy Handler is told to enable the policy. This is also discussed in Section 4.6.3.

Editing a policy simply involves updating the information in the database, removing the old policy from the Live Policy Handler and adding the newly updated version. Deleting a policy involves removing the policy from both the database and Live Policy Handler.

4.6.3 *Live Policy Handler: Policy Enforcement*

The Live Policy Handler is used to enforce the set of Homer Policies which are stored in the database. A policy may be enabled or disabled. There are two parts involved when enforcing policies. Firstly, the Live Policy Handler must listen for the relevant triggers and conditions taking place within the system and know when a policy's *when* clause has been met. Once this happens, the *do* part of the policy must be executed. These two stages are discussed within this subsection.

Due to the modular nature of Homer, it is vital that the policy server has no dependencies on the components themselves. Instead, the policy server registers a listener with the Homer Event Coordinator for any trigger or condition that is of interest. Similarly, it can directly request actions to be performed through the Homer Event Coordinator. If an event occurs, the Homer Event Coordinator will broadcast this information to all listening parties (the technical details of the Homer Event Coordinator can be found in Chapter 3, Section 3.5.3). This is a powerful feature of the Homer policy system as it only needs to be informed of the type of event (trigger, condition or action), the event ID (which uniquely identifies the event, such as "opens", "send", "turn on"), the device ID (which uniquely identifies the device in question, such as "front door", "bedside lamp", "Stirling weather") and any parameters required for the event (provided as a list of string values, such as "sunny", "10", "Alice is home from work now.").

4.6.3.1 *When Clause*

The *when* clause of the policy determines when a policy's *do* clause should be executed. In order to know when the requirements of a *when* clause have been met the Live Policy Store must listen for component events occurring.

CONDITION HANDLING Traditionally, policies of the ECA format involve listening for triggers occurring, before checking all conditions are true, then finally requesting all actions be carried out. However, Homeric uses the WD policy format and blurs the distinction between triggers and conditions for the user. As an example, conventionally "light turns on" is a trigger, or an event, which occurs at a particular moment in time. On the other hand "light is on" is a condition which at any point in time can be evaluated as true or false. Within the user studies carried out in this research (described in Section 2.3.3), it was shown that users would typically use conditions when the behaviour of the trigger was actually intended. Therefore the solution presented is to allow users to make use of either triggers or conditions without distinguishing between these.

Within Homer, a trigger is handled in the traditional sense. The policy server registers a listener with the Homer Event Coordinator to be notified when the trigger occurs. Conditions are handled in two ways: as a trigger or as a condition. So it is possible to evaluate any condition through the Homer Event Coordinator. It is also possible to register a listener for conditions, to be notified when the condition becomes true. Revisiting the previous example, "light is on" evaluates to true at any point the light is on. If the light is off, then at the point in time it becomes on, the condition would be met: a notification would be broadcast, notifying all listeners.

Example policies to clarify this concept are given below. Firstly, consider a policy with only one condition:

> *when* bedside light is on *do* turn off heating.

The policy server will load this policy and register a listener for the one and only term in the *when* clause, which happens to be a condition. This policy will fire when the bedside light turns on, so this condition becomes true.

A second example which includes both a trigger and a condition:

> *when* bedside light is on *and* bedroom window opens *do* turn off heating.

The policy server will load this policy and register listeners for both the condition ("bedside light is on") and the trigger ("bedroom window opens"). The policy will fire when both of these events occur or become true within the default policy duration (typically 60 seconds).

Therefore, if the bedside light is turned on, the condition is met as the light is now on, the policy server will attempt to evaluate all conditions within the policy. There are no other conditions to evaluate, so it starts a timer (discussed in further detail in the forthcoming durations paragraph) where all other terms within the policy must occur or become true. If the bedroom window is opened within this time, the policy will fire. If it is not then the policy is reset and the timer is cancelled.

If, on the other hand, the bedroom window is opened first, the policy server will again attempt to evaluate all conditions. There is one condition term unevaluated, so if at this point in time the bedside lamp is on (perhaps it has been on for some time before the window opened, hence the policy not firing when the lamp turned on) the policy will fire. If it is not, then a timer is started. If the lamp turns on within this time the policy will fire, else the policy will reset and the timer will be cancelled.

OPERATOR HANDLING There are three supported operators within the *when* clauses of Homeric: *and*, *or* and *then*. As described in Section 4.5.1.1, a Homer *when* clause is a tree structure, where each parent is one of the three supported operators.

A parent node is evaluated as true when its children are evaluated as true. For an *and* operator all children must be true. For the *or* operator any one of the children must be true. For a *then* operator each child must become true in turn.

To start, when a policy is loaded all of the children of any *and* or *or* terms are told to register listeners for becoming true. These children report to the parent when this is the case. The *then* term tells only its first child to register a listener and report back.

A leaf node is either a trigger or a condition. If a parent of a leaf node requests it to register a listener, the Homer Event Coordinator is contacted and a listener is registered for the given leaf node details (user device, event type, event and parameters).

When a trigger occurs or a condition becomes true the leaf node notifies its parent node. When any parent node is notified of its child becoming true, the parent stops listening for that event and then re-evaluates itself.

Re-evaluation of any node involves determining if that node is true at that point in time. This requires different processes for the different types of nodes:

- **Trigger** A trigger nodes describes an event which can occur at a point in time, therefore trigger nodes are false until the event takes place.
- **Condition** A condition node, on the other hand, contacts the Homer Event Coordinator requesting to know if its condition evaluates to true or false at that point in time.
- **And** For an *and* node, this involves a check to see if all its children are now true, or if it is still waiting for some children.
- **Or** For the *or* node, once any child is true the node itself becomes true.
- **Then** For a *then* node, checks must be made to see if the children are true, in order. As soon as a child reports false, the *then* node is false and all checks stop. When a child of a *then* node becomes true the listener for that node is cancelled, and replaced with a new listener for the next node in the sequence. When the last node in the sequence becomes true the *then* node evaluate to true.

When a node evaluates to true the node then reports this to its parent. When the root node reports it is true, the policy fires.

DURATIONS A policy composed of *and* and *then* nodes must have a duration, which is simply a time limit in which all nodes must evaluate to true. There is a system default value, typically 60 seconds, which can be customised to suit the users preference. It is possible that any *and* or *then* node has been given a particular duration to use, rather than the default. If no duration is given in an *and* or *then* node specification, the default value is used.

Every *and* and *then* node within a *when* clause has its own timer. This timer begins when one child becomes true, and lasts for the specified time (which, if not set, uses the default time instead). If not all of the node's children become true within the time period, the node resets itself.

As an example of policy durations take the following, purposely strange, policy:

> *when* the front door opens *and* (the temperature falls below 5°C *and* ice forms on the windows *within* 10 minutes) *within* 1 minute *do* [. . .].

This *when* clause (also shown in Figure 4.7) appears odd on first inspection, as the inner term has a longer specified duration than the outer. This can be easily handled by the Homer policy system using the methods already described. The two main scenarios that could occur are:

- If either of the sub-terms of the *and* node become true (the temperature falls below 5°C or ice forms on the windows) then the 10 minute timer will start. Both these terms must take place within this 10 minutes, or the node will reset. If they both do occur, then the parent *and* term will be notified and will start the 1 minute timer. The front door must open within this minute for the policy to execute. If it does not the root *and* node (and therefore policy) is reset.

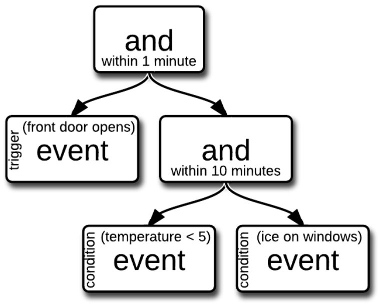


Figure 4.7: When Clause for Durations Example

- Alternatively, the door could open first. This would cause the root node *and* term to start a timer for 1 minute. In this case, the temperature must fall below 5°c and the ice must form on the windows within this time for the policy to fire. If not, again the root node will reset.

ENABLING A policy can be disabled or enabled within the Live Policy Store. This simply toggles if trigger or condition nodes are listening to events from the Homer Event Coordinator. Disabling a policy removes any listeners. Enabling the policy causes the necessary terms to request listeners.

4.6.3.2 *Do Clause*

The *do* clause specifies a list of actions to be carried out if the *when* clause is satisfied.

CONDITIONAL ACTIONS An action, or group of actions, may be preceded with a conditional clause. This clause must contain conditions, which are combined using *and* and *or* operators. Any conditional action can optionally include an else statement, which can similarly also contain a list of actions or other conditional actions.

A *do* clause, like the *when* clause, is represented as a tree. If a *do* clause simply lists actions, then the root node is an *and* operator, and all the leaf nodes are the actions. This is illustrated in Figure 4.8.

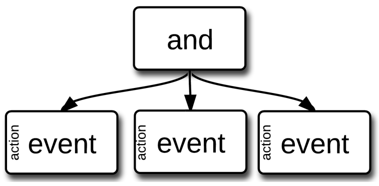


Figure 4.8: Sample *do* Clause Tree with Only Actions

If there is an *if* node the tree will contain an *if* parent node, with either two or three children: two children if there is no *else* clause, and three if there is. These two scenarios are illustrated in Figures 4.2 and 4.3.

When a policy is executed, all non-conditional actions are requested to occur. If there exists an *if* within the *do* node then the following nodes are handled:

- **Condition** Firstly, all conditions are evaluated in turn. The condition part of the *if* is either a single condition, or multiple ones joined with an *and* or *or* operator. If the node evaluates to true the actions are performed, otherwise (if an else exists) the else actions are performed.

- **Action** If the condition was evaluated to true, the action tree is requested to execute. The action tree can contain the same mix of actions and *if* statements as the root node, and is handled in exactly the same way.

- **Else Action** [Optional] If the condition was evaluated to false these action(s) will be executed.

PERFORMING ACTIONS The Homer policy system handles actions by making use of the Homer Event Coordinator. The policy system provides the Homer Event Coordinator with the action details (user device ID, event ID and parameters), and the Homer Event Coordinator handles distributing the requests to the relevant components (as described in Chapter 3).

4.6.4 *Domain and Language Independence*

Due to the clean separation of system information and identifiers, the Homer policy system could be used with other domains. Such an example could include telephony, where the same database schema would be populated with telephony devices and terms. A policy could then be written which made use of this information. An example telephony policy could be:

> *when* home phone receives a call *and* time is between 8AM and 5PM *and* day is weekday
> *do* forward call to work phone.

Pulling out the underlying data from this policy, there exists:

- Four "devices": "home phone", "time", "day" and "work phone"
- One trigger: <phone> "receives a call"
- Two conditions: <time> "is between" and <day> "is"
- One action: "forward call to" <phone>
- Three parameters: "0800", "1700" and "weekday"

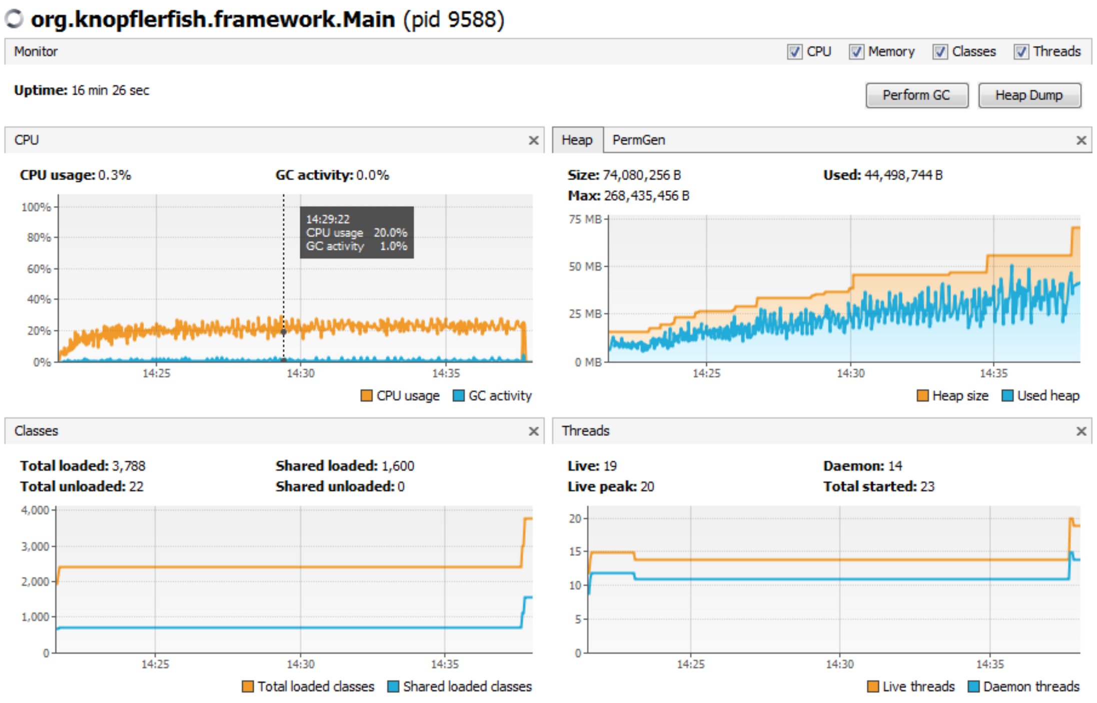
Within the database the devices, triggers, conditions and actions would have all been assigned a unique identifier. This means that through a Homer policy editor, the user could express a policy using the natural language names for each term taken from the database, then save them to Homer using their IDs instead. This would result in the above policy looking more like:

> *when* [TRIGGER:1,A] *and* [CONDITION:2,A"0800","1700"] *and* [CONDITION:3,A,"weekday"]
> *do* [ACTION:4,A].

This policy is completely domain independent. By knowing the event and its type, the device and any parameters, the policy server is able to successfully enforce policies. By knowing these IDs, listening for triggers, querying conditions and requesting actions can all be achieved using the Homer Event Coordinator. This has the added benefit that Homer is completely natural language independent. By simply altering the natural language names, or providing a language set within the database, any Homer user interface will reflect the chosen language without affecting IDs, Homer or the policy server.

4.6.5 *Scalability*

To test the scalability of the Homer policy system, 2000 policies were added to Homer one after another (with a 0.1 second delay between). Each one was validated, tested for overlapping policies, then checked for conflicts between overlapping policies. The profile of this process is shown in Figure 4.9, running on Windows 7 32-bit Operating System, with an Intel Core 2 Quad CPU 2.66GHz and 3GB of RAM. As can be seen, Homer's use of memory increased from around 15MB at the start, to around 40MB after all 2000 policies were added. CPU usage was approximately 20% whilst a policy was analysed and added. This dropped to approximately 0.3% once all policies had been added. The length of time taken to add each policy very gradually increased over time, but this was barely noticeable. Since all policies exist in working memory, the time to execute a policy is negligible as quite simply the individual policy will be notified through a listener when a relevant event occurs, and in turn requests an action to occur by directly contacting the Homer Event Coordinator.


org.knopflerfish.framework.Main (pid 9588)

Uptime: 16 min 26 sec

CPU usage: 0.3% GC activity: 0.0%

14:29:22 CPU usage 20.0% GC activity 1.0%

Size: 74,080,256 B Used: 44,498,744 B
Max: 268,435,456 B

Total loaded: 3,788 Shared loaded: 1,600
Total unloaded: 22 Shared unloaded: 0

Live: 19 Daemon: 14
Live peak: 20 Total started: 23

Figure 4.9: Homer Running Profile Whilst Adding 2000 Policies

4.7 CONFLICT HANDLING

To ensure the user is always in control of the home system, conflicts could be handled either at runtime using resolution policies written in advance, or statically at the time of writing a policy. The ideal scenario would be to incorporate both, producing a hybrid solution. However, within the scope of this research it was decided to focus primarily on one solution.

Both offline and online conflict detection and resolution are extremely useful, and arguably essential, features within a policy system. For Homer, the philosophy of the user always being in control of the home (and therefore home system) meant that automatic handling of conflicts at runtime was disadvantageous. Firstly, at runtime there is no appropriate, non-invasive, reliable or efficient way to ask the user how they would like to handle the conflict. Secondly, at runtime the home system would not behave how the user would expect, either due to the system attempting to handle a conflict or, in the case of multiple users, the conflict being handled by one user without involvement of another. Finally, it may not be possible, or appropriate, to delay the execution of time-critical or security-critical policies.

The author believes that ideally any potential conflicts with a policy should be reported to users when they are planning and writing them. This has multiple advantages, including the new policy being fresh in the user's mind, and having the user's attention and time. It also provides the opportunity to maintain and organise the collection of existing policies.

For these reasons, it was decided to add support for offline conflict detection and resolution to Homer. When the user attempts to save a policy (new or edited) it should be analysed against the existing set of policies to find any potential conflicts. This problem is two-fold. Firstly, a potentially conflicting policy must also be executable when the new policy fires (the policies *overlap*) and secondly, if the policies do overlap, the list of actions for both policies must be analysed to see if there are any conflicting effects on the home (*conflict detection*).

The following sections describe the methods used to detect if policies *overlap* or their actions *conflict* with one another, and the *resolution* handling techniques employed.

4.7.1 *Overlap Detection*

The notion of filtering potential conflicts prior to conflict analysis was originally applied to call control features, known as 'filtering'. However, this concept has not been applied to policy conflict detection. Within the policy domain, non-conflicting policies could be filtered by evaluating if the policies would apply at the same time. Within Homer, this would mean evaluating if a policy's when clause could occur at the same time as another policy's when clause. The term chosen to describe this notion is "policy overlap detection".

Overlap detection must determine if it is possible for two or more policies to take place at the same time. To illustrate overlapping terms, take the following three policies:

Policy A: *when* front door is open *do* ...
Policy B: *when* front door is closed *do* ...
Policy C: *when* email is received *do* ...

It is not possible for the front door to be both open and closed at the same time, therefore Policy A and Policy B do not overlap with one another. Receiving an email is independent of the door being open or closed, therefore both Policies A and B can overlap with Policy C.

4.7.1.1 *Overlap Types*

Studying the overlapping nature of Homeric policies resulted in the following four overlap types:

RELATED TRIGGERS AND CONDITIONS Related triggers and conditions always overlap with one another, and never overlap with their opposites:

***when* 1**	***when* 2**	**Overlap?**
front door closes	front door is closed	Yes
front door closes	front door is open	No

Homer supports conditions within policies which have the characteristics of a trigger. Therefore the following two policies indeed overlap:

Policy A: *when* front door opens *do* ...
Policy B: *when* front door is open *do* ...

The condition in Policy B can be both a condition and a trigger. Therefore Policy B cannot overlap with either of the following policies:

Policy C: *when* front door closes *do* ...
Policy D: *when* front door is closed *do* ...

The same applies in reverse, Policy D cannot overlap with Policy A or B. Indeed, policies A and B will not overlap with either of policies C or D. When detecting overlap this blurring in the definition of triggers and conditions must be taken into consideration.

TIME-BASED TERMS Time-based terms must be respected when compared with any other term groups. Example clauses considered to overlap with time-based terms are shown in the following table:

***when* 1**	***when* 2**	**Overlap?**
a *then* b	b	Yes (*when* a *then* b)
a *then* b	b *and* a	Yes (*when* a *then* b)
a *then* b	b *or* a	Yes (*when* a *then* b)
a *then* b	b *then* a	No
a *then* b	a *then* x *then* b	Yes (*when* a *then* x *then* b)

Homeric supports time-based (ordered) terms with the *then* operator. This introduces complications, as not only are terms conjunctive or disjunctive, they can also be restricted by order. Take the following two policies:

Policy E: *when* front door opens *then* front door closes *do* . . .
Policy F: *when* front door closes *then* front door opens *do* . . .

It is hard to decide if these policies overlap or not. Considering the generic form of the policies "*when* a *then* b" and "*when* b *then* a", the argument falls against them overlapping. However, these policies could technically overlap. Looking again at Policy E and F, imagine the front door opens. Policy E would detect this and begin listening for the front door closing. If it did close, Policy E would fire and Policy F would begin listening for the front door opening. If, within the default time limit, the front door does open, then Policy F would also fire. Therefore, we can see that Policy E and Policy F may overlap. On the other hand, it can be argued that these policies are each describing an event as two individual events occurring in a specific order. So, although the policies technically overlap, it could be argued that for end users these policies describe different events taking place and therefore do not in fact overlap.

Within Homer, it has been decided that an overlap is where a complete group of sibling terms overlap with another complete set of sibling terms. Therefore "*when* a *then* b" would not overlap with "*when* b *then* a", but would overlap with "*when* a *then* x *then* b".

PARAMETERS Parameters and their values must be correctly compared with one another. As shown in the following table:

***when* 1**	***when* 2**	**Overlap?**
t rises above 30°C	t is 25°C	No
t rises above 30°C	t falls below 35°C	Yes (*when* t is between 30°C and 35°C)
SMS received: "Hello"	SMS received: "Help"	No
SMS received: "Hello"	SMS received: "hello"	Yes (*when* SMS received: "Hello")

Parameters associated with a term are another complication for overlap detection within Homer. Different parameters have very different importance, which the following contrived example demonstrates:

Policy G: *when* email is received from Sam reading "Turn on the heating." *do* . . .
Policy H: *when* email is received from Samantha reading "turn on heating" *do* . . .

These two policies both make use of the same trigger ("email received"), which requires two parameters. The first is the name of the sender, the second is the body of the email. The first of these parameters must be exact matches to be considered the same person, whereas the second parameter is far less restricted and clear cut.

There are also numeric parameters which are far easier to compare. For example:

Policy I: *when* temperature rises above 20°C *do* . . .
Policy J: *when* temperature is 24°C *do* . . .

Policy I contains a trigger ("rises above"), whilst Policy J contains a condition ("is"). Both of these terms are concerned with the temperature, and the first parameter of each term specifies the value of concern. It is crucial that any overlap detection can understand the relationship between terms and correctly interpret the term and corresponding parameter.

CONDITIONAL ACTIONS All conditional actions must be incorporated into the overlap detection process. The following table illustrates the two examples:

Policy 1	**Policy 2**	**Overlap?**
when a	*when* a *do if* b *do* . . .	Yes (*when* a *and* b)
when a	*when* a *do* x *and if* b *do* y . . .	x: Yes (*when* a), y: Yes (*when* a *and* b)

Homeric supports conditional actions, resulting in policies which have additional constraints listed within the *do* clause. These must be taken into consideration when analysing policies for overlap. For example:

Policy K: *when* door is closed *do* . . .
Policy L: *when* . . . *do if* door is open *do* . . .

Policy K will not overlap with Policy L due to the constraint within the *do* clause.

There is additional complexity when there are multiple actions within the *do* clause, for example:

Policy M: *when* door is open *do* turn off heating . . .
Policy N: *when* door is open *do* turn on light *and if* door is closed *do* turn on heating

In this case, there are two possible outcomes. The first ignores the conditional action, and will conclude that when the door is open two actions will take place (turn off heating and turn on light). The second outcome, taking into consideration the condition action, will conclude that there is no overlap between policies M and N as the door cannot be both open and closed at the same time.

4.7.1.2 *Related Work*

Two-stage policy overlap detection is a novel concept, inspired from filtering techniques found in the feature interaction domain. The following section discusses existing work and explores the problem space to evaluate a possible means of achieving the goal of filtering non overlapping policies within Homer.

HEURISTIC Wu *et al.* [144] present a call control feature interaction solution which incorporates a naive overlap detection stage. This stage involves examining if there are any triggers in common between two features. If there are none, it is assumed that the features do not overlap, and therefore do not conflict. This is considered too primitive for Homer and a more sophisticated and accurate solution is desired.

Kolberg *et al.* [67] presented a heuristic based solution to reduce the number of features to be analysed for conflicts, by performing a pre-conflict analysis. Features are firstly split into connection equations, then interaction prone scenarios are found by applying a custom heuristic algorithm to a pair of connection equations.

Nakamura *et al.* [95] aim to solve the same feature interaction filtering problem as Kolberg, though uses an alternative approach. This approach involves use case maps, a requirement notation method, to allow the functionality of features to be fully categorised and represented. The relevant data from the use case maps which refer to a particular call scenario are then pulled together into a custom feature matrix to allow a heuristic algorithm to determine the interaction status.

Both solutions presented by Kolberg and Nakamura require detailed domain knowledge about the triggers, conditions and actions and how they relate to one another. This works for a finite domain, however it is not feasible within an unbounded system where third-party features can be added. These solutions were also designed purely around call control, therefore typically only one trigger is involved. Within Homeric there can be countless triggers (and conditions) combined using various operators.

A heuristic based solution, that can detect the sophisticated range of possible overlaps within Homer, could prove extremely challenging within such an expansive area of home automation, and particularly within Homer where third-party developers can extend the range of supported features. For this reason further options are explored.

HISTORIC DATA An alternative approach could be to make use of existing knowledge of the home and the devices within. Logs of events and data for the home could be fed into a probability algorithm which would be able to calculate the likelihood of policies overlapping. This approach had the advantage that it was completely decoupled from the components and required no extra knowledge about the terms. However, this approach would only work once a vast amount of data had been collected about the home. Even then, there would always exist rare activities within the home (such as fire alarms or burst pipes) which would never produce enough data that the policy system could accurately understand when these could overlap with other terms.

LOGIC Finally, an approach was considered to exploit the logical nature of the policy language.

Heisel [50] *et al.* present an approach that makes use of both formal logic and heuristic techniques to gauge feature interaction. The approach is designed to work on any domain which can have its states, inputs, outputs and actions modelled. When a new feature is added to the knowledge set it is first checked if it conflicts with the existing model. The first stage involves analysing the pre-conditions of the new feature with the existing model, and the second stage analysing the post-conditions. The first stage is where the filtering takes place, formal logic algorithms determine constraints with pre-conditions that are neither exclusive nor independent of each other.

However, Heisel's solution requires detailed knowledge about pre- and post-conditions up front for all the different features which is not possible within the broad and dynamic nature of home automation.

CONSTRAINT SATISFACTION Despite the drawbacks to Heisel's solution, a logical approach was explored which did not require additional knowledge about the triggers and conditions of Homer.

A SAT solver was initially ruled out due to its different focus. A constraint satisfaction solver is much better suited for the task, as it supports constraints on the range of possible values any given variable could have. Such a tool could reason about a given set of conditions to deduce if there are any contradictions amongst two policies. If no contradictions are discovered (the constraints can be met), it could be concluded that the policies are able to overlap with one another.

A standalone prototype solution was developed by Turner to explore the feasibility of a constraint satisfaction solver for detecting conflicts between primitive terms. The results were positive, so this approach was extended and fully integrated with Homer to detect overlap between Homer policies. This is described in more detail in the next section.

Currently work exists to 'filter' features that are considered to not be interacting before conflict algorithms are applied. However, all of these approaches require extensive knowledge about the features upfront which cannot readily be provided within home automation. Secondly, these approaches are typically intertwined with the conflict analysis stage and incorporate the features' actions rather than purely the triggers and conditions as is desired for the Homer approach.

The proposed approach using constraint satisfaction to determine overlap between a group of policies extends the existing state of the art. This is achieved by designing a novel two-stage approach which cleanly separates the overlap detection algorithms (involving triggers and conditions) and the conflict detection algorithms (involving actions), supporting a complex policy language, and not requiring extensive knowledge about the domain.

4.7.1.3 *Overlap Detection with Homeric Policies*

Two or more policies can be tested for overlap by imposing their list of constraints on the Java Constraint Solver (JaCoP) "store". Each term within a policy is translated into a constraint, and each constraint within a policy is combined (using convenience methods) with the relevant operators (*and*, *or*, *then*). Finally, both sets of policy constraints are combined using the *and* operator and imposed on the store. The constraints can then be searched using depth-first searching technique to determine if there are any possible solutions which can satisfy all constraints. If the policies do overlap, the first set of possible values to satisfy the constraints is saved, to later be displayed to the user (described further in the conflict resolution Section 4.7.4).

CONDITIONAL ACTIONS Homeric supports conditional actions, resulting in policies which have additional constraints listed within the *do* clause. In order to handle this, the **Conditional Actions Overlap Type**, any policy with one or more conditional actions is translated into numerous sub-policies. Each sub-policy lists all the constraints within the *when* clause and all actions within the *do* clause, eliminating any conditional actions. For example, take the following policy:

Policy A: *when* a *do* b *and if* c *do* e

This would result in the two following sub-policies:

Policy A.1: *when* a *do* b
Policy A.2: *when* a *and* c *do* b *and* e

Secondly, a more complicated example:

Policy B: *when* a *or* b
do c *and*
if (d *or* e) *do* (*if* f *do* g)
else do (h *and* i)

This would result in the two following sub-policies:

Policy B.1: *when* a *or* b *do* c *and* h *and* i
Policy B.2: *when* (a *or* b) *and* (d *or* e) *and* f *do* c *and* g

This does not effect the policy overlap reporting as the list of required terms will be reported to the user, as will any conflicting actions, without the requirement to specify a particular sub-policy. Although the above examples only result in two sub-policies, this approach is recursive and so can handle any number of required sub-policies.

ELIMINATION Each policy pair is firstly analysed for any common terms between them. If there are no terms in common, it can be concluded that, although technically possible that the policies could overlap, the chances are too slim to bother the user with. If, however, there are any shared terms (such as “Tuesday” and “weekday”, “all lamps” and “bedside lamp”) then the policies are put forward for analysis as there is a higher chance that they will overlap.

TRANSLATION Each term within a policy is translated into a constraint. The process for translating a term into a constraint is described below, for terms with and without parameters.

Terms Without Parameters For a condition term with no parameter, attempts are made to convert it into a trigger term instead. As an example, “door is open” would be translated into its trigger form “door opens”. Performing this translation where possible handles the **Related Triggers and Conditions Overlap Type**.

Once the term has been converted (if possible) to its trigger form, it is translated into a constraint. As an example, take the following two terms:

Policy A: *when* front door opens *do* ...
Policy B: *when* front door is closed *do* ...

This would result in two constraints:

Constraint A: *Name* “front door”, *Value* “opens”, *Comparator* equals
Constraint B: *Name* “front door”, *Value* “closes”, *Comparator* equals

Terms With Parameters If a term has a parameter, no attempts are made to convert conditions to their trigger alternative. Instead, each parameter is converted into an individual constraint in order to handle the **Parameters Overlap Type**.

Two examples are as follows:

Policy A: *when* humidity falls below ***20***°C *do* ...
Constraint A: *Name* “humidity”, *Value* 20, *Comparator* less than

Policy B: *when* SMS received from ***Alice*** saying ***Turn On Heating*** *do* ...
Constraint B-1: *Name* “SMS”, *Value* “Alice”, *Comparator* equals
Constraint B-2: *Name* “SMS”, *Value* “Turn On Heating”, *Comparator* case-insensitive equals

HANDLING TIME-ORDERED TERMS In order to support the notion of ordered terms (through Homer's use of *then*), additional information is required. This addresses the **Time-Based Terms Overlap Type**.

Each term is allocated a relative time value to represent the ordering. A sophisticated process is carried out to support timings of events across different policies and sets of *then* terms. As an example:

> Policy A: *when* front door opens *do* . . .
> Policy B: *when* front door closes *then* movement is detected *then* the front door opens *do* . . .

Each term in Policy B would be given a time relative to each other (e.g. front door closes at time = 0, movement detected at time = 1, front door opens at time = 2). The difficulty arises when assigning a time to the only term in Policy A. Giving the term time 0 would result in no overlap being detected, however, giving the term time 2 would result in overlap.

The following rules are used to handle *then* terms within policy overlap checking:

1. ***then* vs' *then***: A complete *then* clause must be a subset of the other policy's *then* clause, allowing additional terms in the fuller *then* clause and respecting the order of both clauses. Example: "*when* a *then* b *then* c" versus "*when* a *then* c" (overlaps when "a *then* b *then* c occurs").

2. ***then* vs' *and***: A complete subset of one clause inside the other, allowing additional terms in the fuller clause and ignoring order. Order is only recognised when describing the conditions of the overlap. Examples: "*when* a *then* b" versus "*when* a *and* b *and* c" (overlap when "a *then* b occurs, as well as c"), and "*when* a *then* b *then* c" versus "*when* c *and* b" (overlap when "a *then* b *then* c occurs").

3. ***then* vs' *or***: Any term within the *or* clause is a complete subset of a *then* clause, again allowing additional terms in the fuller clause and respecting the order of the *then* clause. Examples: "*when* a *or* b" versus "*when* b *then* c *then* d" (overlap when "b *then* c *then* d occurs"), and "*when* a *then* (b *or* c)" versus "*when* a *then* b" (overlap when "a *then* b occurs").

EXECUTION Once the policies have been translated into a set of constraints and enforced upon the constraint store, the store can be searched for possible solutions. JaCoP currently offers only a depth-first search, which is perfectly suited to the requirements of this task. Once the search has been performed the result states if the policies overlap, or if no overlaps were found.

TRANSLATION OF RESULTS If policy overlap is detected, there exists some combination of values which can satisfy the terms (constraints) within the policy. The result is analysed to produce a list of possible values for the given set of policy terms, along with the order these events must occur (relevant for the *then* operator).

Having detected if policies overlap with one another, conflict detection techniques are required to analyse if pairs of policies conflict.

4.7.2 *Conflict Detection*

Conflict detection must determine if a given policy will result in undesired effects when executed at the same time as another policy. As a simple example:

> Policy A: . . . *do* turn on lamp
> Policy B: . . . *do* turn off lamp

In this example trying to execute both policies would result in an undesirable effect: it would be extremely useful to be able to detect any such problems before runtime to be able to report them to the user, therefore deciding in advance how best to handle or avoid such problems.

4.7.2.1 *Challenges*

There are multiple challenges involved with detecting conflicts amongst policies.

CONFLICT PERCEPTION Firstly, it is often down to the individual to decide what they consider a conflict. Take the following two policies:

> Policy A: . . . *do* open the window
> Policy B: . . . *do* turn on the heating

There are some people who may feel these two policies conflict with one another, as it is undesirable to both open a window (reduce temperature) and turn on the heating (increase temperature). However, there are other people who may not consider this conflicting, as they may enjoy the fresh air whilst wanting to remain warm. Due to this human deciding factor, policies have different levels of conflict, including none, potential and probable.

INCONSISTENT EFFECTS Another problem is the notion of analysing the actions for their effect on the environment. Some effects can be straightforward, for example turning on the heating will increase the household temperature. However, opening the curtains will increase the light level only if it is not dark outside.

Positive, negative and opposing pairs of effects are not necessarily conflicting. For example, turning on the heating and opening the window have opposing effects on the temperature within the room. This could be considered a conflict. A second example, where in the evening a lamp is turned on and the curtains are closed, results in opposing effects on the light level within the room. This, however, would typically not be considered a conflict.

VARYING EFFECT DIRECTION Some actions may affect more than one aspect of the environment. As a simple example, turning on a lamp with an incandescent bulb will increase both the light level and the temperature of the room. Opening a window will increase the air flow but decrease the temperature.

ENVIRONMENTAL DATA MANAGEMENT There is the notion that actions can effect the environment (variable) or make use of an external factor (resource). This would be lot of information to enter and manage by either the developer or end user, but would be useful as Homer would have a greater understanding of what effect actions have on their surroundings. However, this also introduces challenges of how variable and resource effects are combined. For example, if we say that a telephone ringing increases the audio-level by 10 decibels and turning on the television increases the audio-level by 30 decibels, naively one may assume that if both events take place the audio-level would be 40 decibels. This, however, is incorrect as decibels are not additive.

4.7.2.2 *Related Work*

The work of Wilson [142], which was inspired by the work of Kolberg, Magill, Marples and Reiff-Marganiec [19], greatly influenced the research, design and solution of the Homer conflict detection. This section discusses the work of Nakamura and Wilson, and explores how this can be enhanced to work offline within Homer.

NAKAMURA Nakamura developed a tool that modelled the specifications (primarily the appliances and "environment properties") of a home network. Each device (appliance) within the home is regarded as an object which has properties and methods that directly relate to the actual device's state and events. Each method has a pre- and post-condition which must be met before the method can be carried out. If the conditions are not met, then the desired action is considered a conflict. As well as devices, the environment was also modelled. Each device method describes how it makes use of or affects a pre-set list of environment properties (namely power, temperature and brightness), including if it is writing or reading the environment property. If any combination of method calls results in illegal effects on the properties then they are deemed conflicting. The drawbacks to Nakamura's tool are the restricted nature of the home appliances, the assumptions made for a fixed API across appliances for the home, the limited nature of predefined, hard-coded environment properties, and finally the tightly coupled nature of the meta-information with the underlying code.

WILSON Wilson worked at a higher level, however he shared many similar approaches with Nakamura. As well as the two levels that Nakamura used (devices and environment) to describe the home network, Wilson had a third: "Service Layer". The service layer was used to combine the functionality of the devices to produce higher-level services for the home. Within Homer, this would be the policy system. Instead of hard-coding the information regarding the environment ("environment variables", or variables for short) and the devices at the code level, Wilson provided a means of obtaining the information in XML form from a remote database. This allows for a much simpler and extensible solution to managing device and environment information.

LOCKS All communication with devices in the Wilson system must go via a device layer. To manage conflicts between devices, this layer utilises a locking system to gauge when various events may be performed on devices, or environment variables may be affected. The device layer can lock a device, and a device can lock a variable. There are four types of locks available:

- **NS : Not Shared** Exclusive lock, not compatible with any other lock.
- **S+ : Shared, Increase Only** Allow only increasing effects on the device or variable, compatible with only S+.
- **S- : Shared, Decrease Only** Allow only decreasing effects on the device or variable, compatible with only S-.
- **S±: Shared** Unknown behaviour, so only compatible with S±.

This locking system allows, for example, a fan and air-conditioning to be turned on at the same time as they both decrease the temperature within the room. However, turning on the heating and the air-conditioning would not be allowed, since one would increase the temperature and the other would decrease it.

This solution works well within the home, as it is a generic access control system for the devices and environment within the home. It is extensible as any device or environment variable can be supported by simply ensuring that the information regarding it is described in the remote database. However, the approach is slightly naive as it assumes that two positives or two negatives should always be allowed, and opposing pairs are conflicts. As discussed in the challenges section (4.7.2.1), conflicts can mean different things to different people. To some people, turning on the air-conditioning and the heating at the same time is a perfectly normal occurrence.

ENVIRONMENTAL EFFECTS Explorations were carried out of the various environment factors, the range of home appliances and services, and the effects on them. Interesting issues and challenges were observed. Namely:

- Different events for the same device can each effect the environment in different ways. For example, turning on a lamp will increase light level and heat within the room, and also consume power. Turning off the same lamp will decrease light level and heat, but simply not consume power.
- The effects of an event can be in opposite senses. For example, turning on the air-conditioning will decrease the temperature but increase the noise level.
- There is the notion of resources which are consumed (gas, power, water, etc.) and environment variables that are affected (temperature, audio, humidity, etc.).
- It could be desirable to allow users to have their own resources and variables. For example, a user could want to maintain a base level of "comfort" within their home. Such a concept could be of no interest to some people but highly useful to others. Defining what affects the "comfort" variable is unique to the individual. For example,

someone may consider being comfortable to involve the house never falling below 23°C, whereas someone else may consider it to mean there is always air flow.

- It was considered that it may be desirable to write policies which interrogate the variable or resource. For example, *when* power usage is high *do* send SMS to Alice saying "Warning! The house is currently using a lot of power.", or "*when* temperature falls below 15°C *and* house is occupied *do* turn on the heating". However, it was later observed that more sophisticated information would be required about how the devices affect the resources and variables (for example, how much power turning on a device would produce or heat a lamp would emit). This leads to complicated solutions and further challenges (as discussed in Section 4.7.2.1). However in most, if not all, cases the information about the variable or resource could be obtained more accurately and simply from dedicated devices and sensors. For example, power monitors could read power usage and thermometers could read temperature.
- In the case of resources, it was observed that on the whole it is desirable to minimise their usage. For variables, it is case-dependent. See Table 4.1 for a list of sample resources and variables and their desired usage (the table is described in more depth in the following section).
- Attempting to find a solution that will work in all cases is near enough impossible. There is a vast range of possible devices and services within the home, coupled with the varying nature of possible end users. General rules should be put into place which aim to detect overlap and err on the side of caution, with the ability to handle exceptions to these rules for special cases.
- Finally, it was observed that not all resources and variables are of interest when detecting overlap. For example, observing that two lamps both increase power usage is not of relevance in comparison to the increase in light level. Similarly, water usage is of concern for some countries, but not for others.

A solution was designed to encapsulate and extend a hybrid of Wilson's and Nakamura's work alongside the new requirements and observations made in regard to Homer. Appreciating the requirement for resources and variables, the term "environ" was defined to encompass them both. Environs are introduced and discussed in further detail in the following section (4.7.2.3).

By utilising environs, and the appreciation that there exist environs that should be minimised, maximised or ignored, it is possible to determine if a set of actions conflict. For example, two actions which minimise an environ that should be maximised will result in a conflict. This is discussed in further detail in Section 4.7.2.3.

The information regarding the effects that the actions for a device may have on environ(s) would be entered by the user when adding a new device type. This is explained in more detail in Section 3.6.2.2.

4.7.2.3 *Environ Effects*

An environ is a term used to describe an environment variable or a resource of a home, similar to the notion of "variables" by Wilson *et al.* [143] and "environment properties" by Nakamura *et al.* [94]. These are used to better understand the effects that actions of devices and services have on the home environment. Some examples include temperature, water, power, light level, etc. These environs have one of three properties:

- **Minimising:** The usage or value of the environ should be minimised. Such as energy, which should be kept as low as possible. Inspired from Wilson's "S-" concept.
- **Maximising:** The usage or value of the environ should be maximised. Such as security, which should be kept as high as possible. Inspired from Wilson's "S+" concept.
- **Neutral:** The usage or value of the environ should be either minimising or maximising, however all effects should be in the same sense. Such as temperature, which is an environment variable within the home that typically is not desired to be actively

Environ	Desired Effect
Gas	Minimise
Humidity	Minimise
Noise	Minimise
Power	Minimise
Water	Minimise
Audio	Neutral
Light	Neutral
Temperature	Neutral
Security	Maximise

Table 4.1: Sample Environs and their Desired Effect

increased or decreased, but generally should be altered in the same direction. Turning on the air conditioning (reducing temperature) and the heating (increasing temperature) is typically undesirable. The neutral environ property is inspired from Wilson's "S±" concept.

Common environs and their respective properties are shown in Table 4.1. Each environ can be set to "ignored" which means the environ should not be taken into consideration in conflict detection.

For any given action performed on a particular type of device it is possible to describe how environs are affected (increased, decreased, no effect). This is done through a user interface, rather than at the code level. The information could be gathered from the system installers, the end user, or a database of default information. Some examples illustrate what might be defined:

- "turn on bedside lamp": increase light level, increase power.
- "turn on air-conditioning": decrease temperature, increase power, increase noise.

It is important to describe only the most relevant of environ effects, rather than listing all the insignificant side-effects of the action. For example, increase temperature was not included in the list of environ effects for "turning on the bedside lamp" above. This is because it is of such little importance, and at no point would the decision of turning on the lamp be affected by the slight temperature increase it would cause. On the other hand, the increase in noise when the air-conditioning is turned on was included, as many air-conditioning units are irritatingly noisy. It may be desirable to keep noise to a minimum when trying to sleep, entertaining guests or listening to music, for example. However, some house owners may have much quieter air-conditioning units and therefore this environ effect could be removed from the list of affected environs.

It is possible for a user to add their own custom environs, and describe how these are affected just as they would describe any other environ. This allows less concrete, and often more personalised, environs to be added to Homer. For example, "comfort", "security", or even "happiness".

As described previously, any action may increase ('+') or decrease ('-') zero or more environs. This information can be used to aid conflict analysis (design details can be found in Section 4.7.2.4). Any two environ effects can be compared, to calculate if they conflict using Table 4.2. This table shows that if there are two environ effects which increase a minimising environ or decrease a maximising environ then there are undesirable effects. For example, turning on the washing machine and the air-conditioning unit both increase noise, however noise is an environ that most likely is specified to be minimising, therefore it would be considered undesirable to turn on two noisy devices at the same time. The table also shows that when increasing and decreasing a neutral variable there is the potential for undesirable effects. For example, turning on a heater and a fan at the same time will both increase and decrease the perceived temperature. Temperature would typically be a neutral variable which implies that it should be affected in the same way, hence this pair of actions would be conflicting.

	++	–	+-
Minimise	✓		
Maximise		✓	
Neutral			✓

Table 4.2: Conflicting Environs

The notion of categorising the effects of actions upon the environment for the home originates from Wilson *et al.* [143] and Nakamura *et al.* [94]. Environs extends their work by being designed to work in offline conflict detection rather than online. Previous research worked by knowing what affect an action could have upon a variable, and effectively locking that variable at runtime when an action took place. If that action was to increase the variable, then whilst the lock was in place only actions that also increased the variable would be permitted. This is primitive and limited, as in some cases it may actually be a conflict to increase a variable twice, for example it could be considered undesirable to increase the noise level in the home by two different devices. This is where the novel concept of environ properties advance existing work, as variables now have additional meta-data to describe how they should be treated to help better understand conflicting behaviour.

Environs have extended the work of Wilson and Nakamura to provide offline conflict detection. This is achieved by removing the concept of locking variables and devices, and by extending the notion of environment variables through the addition of meta-data.

4.7.2.4 *Design*

The Homer Conflict Detection logic is cleanly decoupled from the Policy Server (represented in Figure 4.6). The process for detecting conflict amongst a set of actions is discussed below.

Conflict Types Firstly, there are four different conflict states to describe a pair of environ effects:

- **None**: There exist no known conflicts between the two actions. Such as "send email" and "turn on lamp".
- **Same**: The two actions are exactly the same. This can be a conflict where the action is costly or involves the user, such as "send SMS to Alice saying 'Heating turned on'." (both costly, and confusing and irritating for the user to receive twice).
- **Opposing**: The two actions will result in opposing effects on the environment. Such as "turn on lamp" and "turn off lamp".
- **Possible**: The two actions may conflict. Such as "set temperature to 20°c" and "set temperature to 30°c", or "open window" and "turn on heating".

These states are used by Homer when describing the list of conflicts detected between two actions. Table 4.3 provides examples for each conflict type.

The Process The detection of conflicts relies on comparing each action in one policy with each action of the second. If there are multiple policies then the process is carried out three times, once for each pair of policies.

For each action pair, the potential for overlaps is analysed and a list of any that are detected is returned. This list is added to the master list of overlaps already detected, and finally that master list is returned as the complete list of overlaps. Each individual conflict is given an ID for reference.

Conflict	Action 1	Action 2
None	send email	turn on lamp
Same	turn on lamp	turn on lamp
Opposing	turn on lamp	turn off lamp
Possible	dim lamp to 80%	turn off lamp

Table 4.3: Sample Policy Conflicts

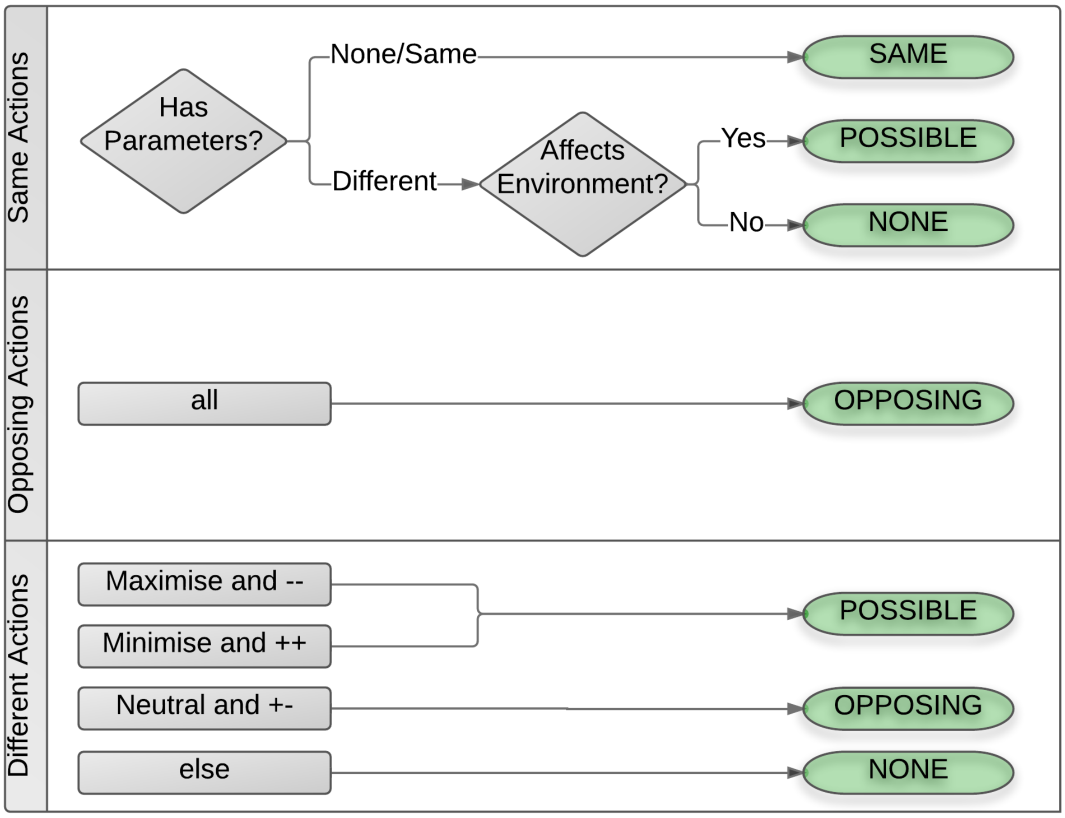


Figure 4.10: Conflict Detection Process of Two Actions for Same Device

The process for detecting overlap depends on whether the action pair is concerned with the same device or different devices. Each approach is discussed below.[2]

Same Devices The process for detecting conflict between two actions concerning the same device is shown in Figure 4.10.

Firstly it must be established if the actions are the same (e.g. both "turn on", "send", "open", etc.). This is represented in the figure with the split horizontal pane.

If the actions are the same, any potential parameters must be taken into consideration:

- The two actions are the same if there are no parameters, or the parameter values are all the same. In this case it can be reported that the two actions are identical, so they are given the conflict state "**same**". For example: "turn on desk lamp".
- However, if the parameter values differ in some way the two actions are clearly not the same. In this case, if the action is known to:

2 For all environ comparisons, it is assumed that if the environ itself is set to be ignored then it will be.

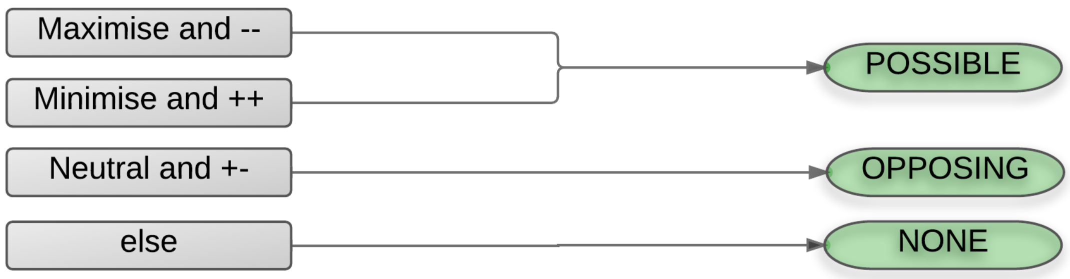


Figure 4.11: Conflict Detection Process of Two Actions for Different Devices

  - not affect the environment (an environ) in some way we can assume that the action pair does not overlap (”**none**”). For example: “send email to Alice saying ‘Heating turned on’ ” versus “send email to Bob saying ‘Dinner time!’ ”.
  - affect the environment, the best-guess is that this pair of actions may indeed cause some conflict. This is given the conflict state “**potential**”. For example: “dim lights to 80%” versus “dim lights to 20%”.

If the actions are different (shown in the bottom half of Figure 4.10) then the effects of each action upon the environment are listed (note: parameters are ignored). For the same environ, the effects are compared (shown simplified in Table 4.2) to gauge if the pair of effects could be considered conflicting. As shown in both the figure and table:

- If the environ property is *maximising* and the environ effects of the actions are decreasing, a conflict can be assumed (”**possible**”). This also applies for the opposite (*minimising* environ, with two increasing effects). For example: if light level was to be maximised, the actions are “turn off light” and “dim light to 20%”.
- If the environ property is *neutral* and the two environ effects are opposing (one increases whilst the other decreases), it is assumed that this may have an undesirable effect. Therefore the state “**opposing**” is assigned to this pair of environ effects. For example: ”turn on heating” versus “turn off heating”.

If the environ effect pair reach the end of these checks without flagging as a conflict, it can be assumed that there is no conflict between them (so is assigned “**none**”). For example: “change channel on television” versus “increase volume on television”.

Different Devices The process for detecting conflict between two actions concerning different devices is shown in Figure 4.11.

The process for different devices is much simpler than that of same devices, and in actual fact is nearly identical to how differing actions for the same device are handled. A reduced description (to save duplication of content), with examples, is given below.

For each action pair, their effects upon the environment are listed. For the same environ, the effects are compared to gauge if the pair of effects could be considered conflicting:

- If the environ property is *maximising* and the two environ effects are decreasing (or indeed the opposite – a *minimising* environ with two increasing effects), a conflict can be assumed (”**possible**”). For example: “turn on the washing machine” versus “turn on air-conditioning” (as both increase noise, which is a minimising variable).
- If the environ property is *neutral* and the two environ effects are opposing, it is assumed that this may have an undesirable effect. Therefore the state “**opposing**” is assigned to this pair of environ effects. For example: “turn on heater” versus “turn on fan” (as one increases temperature and the other decreases it, where temperature is a neutral variable).

Again, if the environ effect pair reach the end of these checks without flagging as a conflict, it can be assumed that there is no conflict between them and therefore assigned "**none**". For example: "send email" versus "turn on heating".

Handling Conditional Actions Conditional actions do not exist within the policies that are analysed as all policies are re-phrased to ensure all conditions are within the *when* clause. This process was described in Section 4.7.1.3.

Examples Table 4.3 brings together given examples throughout this section to conclude the design of the Homer Conflict Detection logic.

4.7.3 *Policy Validation*

It is possible for a user to write a policy which has an invalid *when* clause and/or a contradicting set of actions within the *do* clause. Each of these can be tested for, as discussed in this section.

4.7.3.1 *Checking The When Clause*

The Homer overlap checker is able to validate any policy's *when* clause by simply looking for overlaps amongst the terms. If the overlap detector cannot find any overlaps, then the policy can be deemed valid. The following two policies demonstrate invalid *when* clauses:

> *when* the front door is open *and* the front door is closed *do* ...
> *when* the humidity rises above 60% *and* the humidity falls below 40% *do* ...

4.7.3.2 *Checking The Do Clause*

The Homer conflict checker is able to validate any policy's *do* clause by simply looking for any conflicts amongst its set of actions. If any conflict is detected the user can be notified of such conflicts before confirming the save of the new policy. The following two policies demonstrate invalid *do* clauses:

> ...*do* turn on the desk lamp *and* turn off the desk lamp.
> ...*do* turn on the heating *and* turn on the air-conditioning.

4.7.4 *Conflict Resolution*

The final stage of conflict handling is reporting to the user any potential conflicts that may have been detected, and deciding how best to handle them. This is typically termed "conflict resolution". Within Homer, conflicts are detected at the point the user saves their policy (new or edited) or enables a policy. If Homer discovers any conflicts between overlapping policies they are reported to the user. The user must decide how they would like the conflict(s) to be handled before the policy is saved. The process for describing any conflicts and handling them is discussed in this section.

4.7.4.1 *Describing Conflicts*

When a policy is added, edited or enabled by a user, Homer analyses it against every other existing enabled policy in the database. Any conflicts detected between the new/updated/enabled policy and an existing policy are collected. Once all policies have been analysed the resulting collection of conflicts is returned to the user interface client.

Each policy pair with conflict is described to the user in natural language to make understanding the conflict as easy as possible. The following example pair of policies are used to describe and demonstrate the output process:

> New Policy (A): *when* it is a Saturday *and* the time is 11PM *do* turn on the bedroom lamp.
> Existing Policy (B): *when* the time is 11PM *do* turn off the bedroom lamp.

Homer will detect that on Saturdays when the time is 11PM Policy A and Policy B will conflict as opposing actions will take place. This information is a list of overlaps and conflicts (as described in the sections 4.7.1 and 4.7.2), which is translated into JSON for portability. A temporary ID is given in the case of it being a new policy, and is stored temporarily in the database until further actions have been requested. If no action is requested within 24 hours, it can be assumed that the policy should be deleted.

The output user interface client can decide how best to display this information to the user. The simplest choice could be conversion of the information into natural language, with a range of default options for how best to handle the given conflict.

The example policies A and B would result in the following possible natural language description of the conflicts:

> Your new policy "A" may conflict with existing policy "B" when it is a Saturday and the time is 11PM, as the bedroom lamp will turn off and turn on (which are opposing actions).

The template version of this sentence would be:

> Your <new | updated | re-enabled> policy <new/updated/enabled policy name> may conflict with existing policy <existing policy name> when <list of required triggers/conditions>, as <list of conflicts>.

The template version for one given conflict in the "list of conflicts" is:

> <device name> will <conflicting action from existing policy> and <conflicting action from new/updated/re-enabled policy> (which are <type of conflict> actions).

Along with the description of the conflicts, options must be provided for how the user would like them handled (discussed in the next subsection).

4.7.4.2 *Handling Conflicts*

Homer supports handling conflicting policies at both the conflict level and the policy level:

- **Overlap and Conflict** Options:
  - *Ignore*: There is no overlap/conflict in this case.
  - Qualified ignore (using example "turn on kitchen radio" versus "turn on desk lamp"):
    - *Ignore for all devices of these types*: There will never be overlap/conflict between either of the devices types, for example radios will never conflict with lamps.
    - *Ignore for all devices of new/edited/enabled policy*: There will never be an overlap/-conflict between all devices types of the new/edited/enabled policy against the particular device mentioned in the existing policy, for example radios will never conflict with the desk lamp.
    - *Ignore for all devices of the existing policy*: There will never be an overlap/conflict between all devices types of the existing policy against the particular device mentioned in the new/edited/enabled policy, for example the kitchen radio will never conflict with any lamp.
- **Policy** Options:
  - *Save Anyway*: The conflict is acknowledged and deemed unimportant, so save the policy.
  - *Disable <existing policy name>*: By disabling the existing policy the conflict is avoided. If the policy were re-enabled the overlap would be reported once again.
  - *Edit <existing policy name> or <new/edited/enabled policy name>*: Edit the policy will let the user amend the conflict or eliminate the overlap between them.
  - *Delete <existing policy name> or (If new policy) Delete <new policy name>*: Delete the existing policy, or in the case of a new policy effectively cancel it.
  - (If editing or re-enabling) *Cancel*: Do not save any changes made (if editing), or do not re-enable.

Device	Environ(s)
Burglar Alarm	Security
Curtains	Light
Dehumidifier	Humidity, Power, Noise
Door	Security, Temperature
Heating	Temperature, Gas
Lamp	Light, Security
Oven	Gas, Power, Temperature
Radio	Audio, Power, Security
Sprinkler	Water
SMS	
Television	Audio, Power, Security
Washing Machine	Power, Noise, Water, Humidity
Window	Temperature, Security

Table 4.4: Illustrative Devices and the Environs Affected

Again, it is entirely down to the user interface designer which and how these options should be presented to the user.

In the case of ignored pairs of devices/device types, the identifiers are stored in special tables within the database (one for overlap, another for conflict) which deals with combinations to be ignored. These are then taken into consideration when detecting overlaps and conflicts.

4.7.5 *Illustration*

A range of policies were independently defined and used to illustrate the effectiveness of the Homer overlap, conflict and resolution aspects. Firstly the environs and existing devices are given. Then the policies are listed, followed by a table showing the overlaps and conflicts between the given policies. Finally the results are collated and discussed.

4.7.5.1 *Environs*

Table 4.2 shows the eight environs involved within this illustration, along with their desired effect. For the sake of this illustration, no environs will be ignored. Table 4.4 shows the devices used within the given policies and the environs they effect.

4.7.5.2 *Policies*

The sample policies, which fully exploit all the features of Homeric, used within this illustration are:

1. *when* time is earlier than 8:30PM
   do turn on the hall lamp.
2. *when* time is 7PM
   do turn off the hall lamp.
3. *when* time is between 8PM and 10PM
   do open the 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 the washing machine.

6. *when* front door is open *or* front door is closed
do turn on the washing machine.

7. *when* receive SMS from Alice saying "On Way Home."
do if temperature is below 18°C *do* turn on heating.

8. *when* receive SMS from Alice saying "on way home."
do turn on oven *and if* temperature is warmer than 24°C *do* open the window *else if* temperature is cooler than 18°C *do* turn on 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 heating.

10. *when* (front door opens *or* back door opens) *and* time is after 5PM *and* lamp is off
do turn on hall lamp.

11. *when* day is a weekday *and* time is 7:30AM
do turn on oven *and* open curtains.

12. *when* time is 9:45PM
do turn off television.

13. *when* television turns off *then* lamp turns off
do turn on bedside lamp *and* close curtains.

14. *when* receive SMS 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 radio.

18. *when* time is 8AM *or* time is 5PM
do send SMS to Alice saying "Feed cat!".

19. *when* temperature is warmer than 25°C *and* time is 2PM
do turn on sprinkler.

20. *when* (curtain closes *then* bedside lamp turns on) *and* time is after 10PM
do turn on burglar alarm.

4.7.5.3 *Interactions*

One-by-one the above policies were added to the Homer policy server. As each one was added it was firstly validated. Then, if valid, it was checked for overlaps against all previously added policies. If any policies were considered to overlap, these were analysed for conflicts. Table 4.5 shows the results for each policy added. 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 13 shows what happens when policy 13 is added to the database which contains existing policies 1 to 12: possible conflict *e* is detected between policies 13 and 10.

Conflict handling is commutative and associative, therefore the order in which policies are added is irrelevant and does not affect the outcome.

For the sake of this exercise, all policies were added to the policy store regardless of conflict outcome, with the exception of invalid policies. The policies were added in the order they appear in the list provided in Section 4.7.5.2, hence policy 1 was the first policy added (at which point the store was empty), then policy 2 was added and compared with 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

Table 4.5: Analysis Results for Example Policies and Environs

existing policy in the store (1). When policy 3 was added it was compared to policies 1 and 2 independently. This process continued up to and including policy 20.

The only invalid policy is number 5, which would result in the user being asked to correct it because its event clause is invalid: "front door is open *and* front door is closed" cannot be satisfied.

The possible conflicts detected when adding the twenty policies are described in Table 4.6. The table states the overlap condition, the analysis outcome, and the likely user reaction to each reported case. These results are discussed in more detail in Section 4.7.5.4.

4.7.5.4 *Results*

As can be seen from Table 4.5 the Homer policy system has successfully detected the one invalid policy, and the numerous conflicts amongst the various policies. Examples of the different types of conflicts illustrated in Table 4.2 were all correctly detected and presented. The average length of time taken to add a policy and perform the analysis was 0.022 seconds.

Homer's novel approach of detecting overlaps before performing conflict detection analysis strongly reduces the number of potential conflicts reported to the user. Without such overlap detection, each new policy's effect on the environment would be compared to all existing policies regardless of the relevance. For example, without overlap analysis, if a new policy turned on a light on Mondays at 6PM and another policy turned off the lamp on Tuesdays at 6PM a conflict would be reported. In reality, these two policies can never conflict with one another since they can never fire at the same time. Homer's novel overlap detection will filter these policies and therefore such a "conflict" would never be reported.

FALSE POSITIVES There is a large degree of subjectivity as to whether a conflict is indeed genuine. This is why it is important for Homer to offer a customisable conflict detection process. The user can customise and define environs and how they are affected within their home, as well as provide feedback on presented conflicts. The more that the user performs

Case	Overlap Condition	Action Analysis	Likely Reaction
a	7PM	OPPOSING: hall lamp on *and* off (opposite actions)	change policy
b	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* 5PM *and* lounge lamp off	OPPOSING: 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	front door *and* 5PM *and* (television off *then* lounge lamp off)	OPPOSING: curtains closed *and* lounge lamp on (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

Table 4.6: Details of Possible Conflicts

these customisations, the less reports of uninteresting of irrelevant conflicts (i.e. false positives) they will receive.

Customisable environs have the added benefit that Homer is able to detect conflicts that no other existing approach can. Homer allows the user to add their own environs and to customise how policy actions affect these. Homer can then detect conflicts in a unique and personalised way. Homer is able to detect conflicts that the work of ACCENT, Nakamura and Wilson cannot.

Following user reactions to possible conflicts, the end result is a set of acceptable policies. Since conflict detection has been performed offline, at definition time, the policies should execute without conflicting (in the user's judgement).

FALSE NEGATIVES As with any conflict or interaction analysis, there is the possibility of false negatives: cases that are not detected by the analysis. This can result in policies being saved which could potentially conflict with existing policies.

There are two ways that false positives can arise within the Homer conflict analysis. The first is the situation where a possible overlap is not detected, so the policy is filtered out before conflict analysis is performed. The second is where a conflict between two overlapping policies is undetected.

Missed Overlaps There are two known situations that could result in a user believing that two policies overlap (and therefore could conflict) which Homer would not consider an overlap.

The first situation is where two policy's *then* clauses are in different orders, for example *when* x *then* y versus *when* y *then* x (this problem is discussed in more depth in Section 4.7.1.3). Homer believes that, in the case of at least one ordered clause, there must be a complete subset of one set of clauses within another. For example, *when* x *then* y *then* z overlaps with *when* y *then* z. However, the aforementioned example of x *then* y would not be considered to overlap with y *then* x, as one is not a complete subset of the other. The user, however, may perceive this to be an overlap.

The second situation that could result in a missed overlap is where the user perceives two events to overlap which are completely unrelated as far as Homer is concerned. As an example, if one policy will fire when Alice's morning alarm clock turns on and another policy will fire when Alice gets out of bed – should these be considered to overlap? Homer currently cannot detect overlap between technically unrelated devices, however a user may perceive these events to overlap as they are closely related in reality.

Missed Conflicts Missed conflicts can only arise if the user believes something to be a conflict but has not specified this in the environs. For example, if the user considers the washing machine turning on at the same time as the shower to be a conflict due to water pressure issues, then this needs expressed within the environ information (simply create a water pressure environ, state that it is desirable to minimise this environ, then tell Homer that turning on showers and turning on washing machines both increase the water pressure).

As long as the user has set up and customised environs to represent their personal model view of their home and the environs within, then Homer can correctly detect conflicts that effect the user's environs in undesirable ways.

Within Homer false negatives will not cause the system to fail. Two conflicting policies taking place at runtime may result in strange behaviour for the user (such as a lamp turning off and on very quickly), but this will not affect Homer. This is unlike telephony where a missed feature interaction could cause serious problems, such as an undetected call forwarding loop. In contrast, false negatives are not a problem for Homer. Although the user may consider them undesirable, they will not cause system errors.

PARSING OVERLAP RESULT The only results which are harder to understand are when the minimum possible value is provided for an overlap to take place. For example, this often appeared when describing a possible value for temperature as 0°C. This is simply a result of JaCoP, which provides the minimum value which satisfies the constraints (in the case of a less than operator, the lower constraint starts at zero). This can also be seen when comparing days or times, for example if two policies refer to a weekday, this is translated into a range from Monday to Friday, the overlap is presented as Monday. Similarly, for times, if one policy refers to after 2AM and another refers to time before 8PM, Homeric will report that the policies overlap at 2AM. Possible improvement when describing the overlaps to the user could be to obtain the range of possible values, or to choose the value closest to that specified in the policies.

The results demonstrate that the state of the art has been extended, by adapting and building upon the work of Wilson and Nakamura, to offer a customisable offline means of detecting conflicts between policies.

4.8 CASE STUDY

Having designed Homeric, an evaluation was performed to analyse if it meets the needs of end users. This evaluation involved 71 participants, who each wrote numerous policies in Homeric. A summary of the key details of this study is discussed within this section. The full description and analysis is published in [80].

4.8.1 *Overview*

The evaluation was a 15-30 minute exercise that was designed to evaluate both Homeric and work performed in end user programming (discussed in Chapter 5). A custom wizard tool (named the Homeric Wizard) was designed and developed to allow Homeric policies to be written by dragging individual triggers, conditions and actions of devices into their desired position in the policy. An example of the interface can be seen in Figure 5.27.

The evaluation took place over two weeks from March to April 2012. It was an Online questionnaire which was distributed to a wide audience through exponential, non-discriminative snowball sampling.[3] Participation was received from a wide range of ages and technical abilities, ensuring Homeric was evaluated by a large and varied audience.

The participants were asked to perform three types of tasks within the evaluation. The first of these was for users to describe in their own words three different example policies (presented in the Homeric Wizard). The second task involved the participants using the tool themselves to write two policies, both of which had to meet a particular goal. Thirdly, the participants were asked to write two policies of their own. All of these tasks could be performed at one of three different difficulty levels:

- Level 1: Easy
- Level 2: Intermediate
- Level 3: Advanced

More advanced language features were introduced at each level increase. Those at level 1 could write policies with only the *and* operator to join terms within the *when* clause, and only plain (non-conditional) actions permitted in the *do* clause. Level 2 allowed the additional use of *or* and *then* in the *when* clause, but still only plain actions within the *do* clause. Finally, level 3 allowed conditional actions within the *do* clause. The one language feature supported by Homer that was not exposed through the Homeric Wizard tool is durations for groups of terms within the *when* clause. This was originally offered to those at level 3, but during the evaluation pilot it was discovered that even technically competent individuals were struggling to grasp the advantages offered by such a feature. For this reason the durations language feature was removed from the evaluation.

Durations were added to Homeric as a solution to implementation problems that arose at the development stage. Ultimately, if a particular trigger or condition of a policy fires, how long should a policy stay active waiting for the remaining triggers and conditions to occur? Different times can vary the behaviour of a policy greatly. It was therefore felt that a default value should be chosen, but more technically capable users could change this for particular

3 A technique used to obtain access to a wider range of people, by asking respondents to pass the study to others and similarly asking those individuals to pass it on, and so on [24].

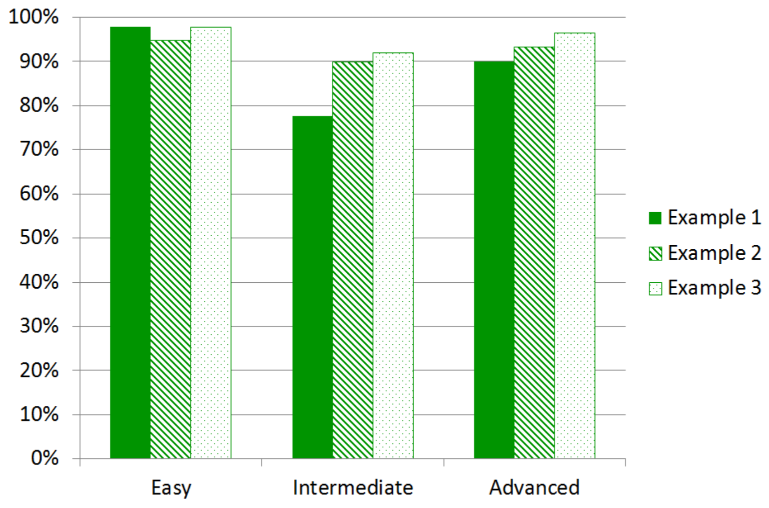


Figure 4.12: Example Correctness Scores for Each Difficulty Level

policies if they so desired. Unfortunately, it may be the case that the concept is too complicated for most people and potentially an alternative solution should be explored. However, without evaluating durations no conclusions can be drawn from the pilot study.

4.8.2 *Results*

The results of the evaluation produced a range of quantitative and qualitative data that was analysed thoroughly to evaluate hypotheses and to gain a clear understanding of Homeric's success. An overview of the findings for each of the three tasks are discussed in turn, with the full data presented in [80].

4.8.2.1 *Can Users Understand Homeric?*

Task 1 involved the user explaining, in their own words, what various example policies meant. For many, this would have been the first time they would be faced with such rule-based logic, and for all, the first time seeing the Homeric Wizard interface. Despite this, the results were very impressive. The three examples were marked on correctness, and scores were awarded for each participant attempt. Figure 4.12 shows the average scores (as percentages) for each example and at each difficulty level. As can be seen, the average scores across examples and difficulty levels were always over 85%. Secondly, it was shown that for intermediate and advanced users the average score increased for each example. This shows that, as the participants gained experience and confidence, their performance improved, even across only three examples. This shows promise that the learning curve for Homeric is short, and that Homeric language features are understandable (with no prior explanations) when presented at an appropriate user difficulty level.

4.8.2.2 *Can Users Translate into Homeric?*

The second task involved the user writing Homeric, using the Wizard tool, to produce two policies. Each policy was to meet a particular high-level goal. Similar to task 1, each policy written by the participant was marked as to how well it satisfied the goal. Despite very little introduction (a few simple screenshots in the form of a brief visual tutorial), the results were very positive, as the graph in Figure 4.13 shows. At all difficulty levels each average score was comfortably above the hypothesised 85%. This confirms that participants were able to formulate Homeric policies to meet predefined goals.

4.8.2.3 *Can Users Write Homeric?*

The final task involved the participants writing two Homeric policies to meet their own personal goals for the home. This was to help understand if Homeric offered a simple

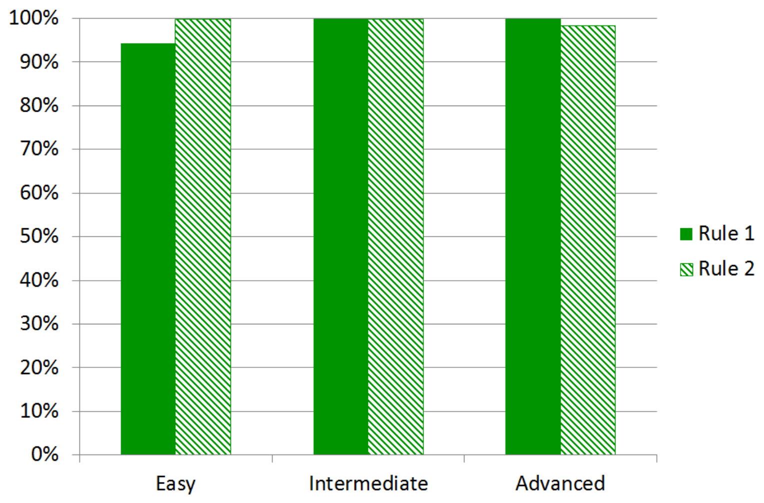


Figure 4.13: Translation Correctness Scores for Each Difficulty Level

enough language for less technical individuals, whilst also supporting the more ambitious and sophisticated policies written by more technically experienced participants. Participants wrote policies for a wide range of goals within the home, including heating management, reminders, home security and home comfort. The average number of terms used in a single policy across all users was 4.2, with a total of 93 *and*, *or* and *then* terms and 46 conditional action terms used.

Only approximately 10% of participants made any form of error. These were where a policy could never fire due to too many required conditions such as "it is Saturday *and* it is Sunday". This is potentially the fault of the user interface rather than the language, and in practice the Homer conflict detector would discover such errors and report them to the user anyway.

This task in the evaluation confirmed Homeric's success. A wide range of policies were written by all ages and technical abilities, achieving different goals and tasks for the home. The only language features requested that Homeric does not support is the *not* operator (requested by two participants) and loops (requested by one participant). Overall, participants were able to successfully use Homeric to write rules for the home and only 10% made any form of error.

4.8.2.4 *Additional Findings*

There were many comments made by participants throughout the evaluation in regard to Homeric, as well as useful observations. These are discussed below:

- It was observed that many participants made use of variable names to refer to hard-coded data in the examples, such as "bedtime", "work day" and "comfortable home temperature".
- Conditional actions proved to be a personal preference, with some participants disliking them whilst others made frequent use of them. A total of 46 conditional actions were used by the participants throughout the evaluation.
- There were many observations on the duration of a rule, in the sense of how long do a policy's actions persist and do they need to be manually undone? For example, if one turns on the burglar alarm when leaving for work, will it automatically be turned off on arriving home? This lends itself to extending the notion of *when* term durations to include action durations too, such as "turn on the radio for 10 minutes".
- There were frequent requests for sensor and actuator fusion by all technical abilities. Participants wanted to easily control groups of devices, rather than specifying multiple terms individually.

- Some participants occasionally overloaded their *when* clause to result in unnecessary and unlikely combinations of events. For example, if a policy involved locking the back door and front door at 11PM each night the participant would write "*when* time is 11PM *and* the back door is unlocked *and* the front door is unlocked *do* lock the front door *and* lock the back door". The checks of the doors being unlocked is unnecessary, and results in the policy firing only if both doors are unlocked – which is presumably not what the participant intended.

- Very few participants made reference to potential conflicts amongst policies, but those that did appeared to prefer the notion of offline conflict detection, stating how they would like to be warned about any potential conflicts when they tried to save a new policy.

- Only two participants out of the total 71 mentioned negation, and only one mentioned loops, confirming that these are not high-priority or desirable language features for the home.

4.8.3 *Summary*

The key findings from this evaluation include the success of a wide range of individuals in understanding, transcribing and writing Homeric policies. Homeric has met the needs of these 71 participants, proving itself to be a flexible and appropriate language for programming the home.

4.9 CONCLUSIONS

Three research contributions have been made within this chapter: a custom policy language for the home, novel policy overlap detection, and advancements to the state of the art of policy conflict detection. Each are now discussed in turn.

Language

This chapter has presented a new policy language, Homeric, which is custom-designed for the home.

Homeric has evolved from various user studies (discussed in Section 2.3) and research to result in a language which has been custom designed to allow a wide range of end users to program their home. The language offers flexibility and sophistication for more advanced technical users, as well as offering a simpler version for less technically minded individuals.

The following language features for the Homer policy system, as seen in the language specification in Section 4.5, offer novel features that currently do not exist in other policy languages. These include:

- **When-Do Format**: Policies are expressed in a *when – do* format, rather than in the traditional *when trigger – if condition – then action* format.

- **Ordered Terms**: Ordered triggers and conditions are supported, using a *then* operator.

- **Blurred Triggers and Conditions**: The distinction between triggers and conditions are blurred, allowing them to be interspersed within the *when* clause.

- **Conditional Actions**: Actions can be qualified with conditions with the *do* clause.

An evaluation was performed to verify the acceptability and usability of Homeric. 71 participants of varying age and technical ability were asked to read and write policies using Homeric. The results from this evaluation were extremely positive, with every participant successfully understanding and formulating policies.

Homeric has advanced the current state of the art by offering a when-do policy format with novel language features custom designed for the demands of home automation.

Overlap Detection
Overlap detection is the first part of a novel two-part process presented for detecting conflicts between policies. When a user saves a new policy the first part of the conflict detection process involves obtaining a list of all policies which could be take place at the same time at runtime. This list is considered a list of 'overlapping' policies.

Overlap detection is achieved by using a constraint satisfaction solver to gauge if policies can overlap with one another. This approach advances the state of the art as the full language features of Homeric can be supported, and no prior knowledge about the triggers and conditions are required.

An illustration is presented that demonstrates the overlap detection algorithms on a range of twenty Homeric policies which fully exploit the language features available. The results show that the approach was successful, and it enhanced the overall Homer conflict detection by eliminating conflicts which would be highly improbable.

Homer's overlap detection techniques advance the state of the art by offering a novel approach to detecting if policies can overlap with one another.

Conflict Detection
Sophisticated conflict detection techniques form the second part of the Homer conflict detection process. After non-overlapping policies have been eliminated, the remaining policies are analysed for potential conflicts.

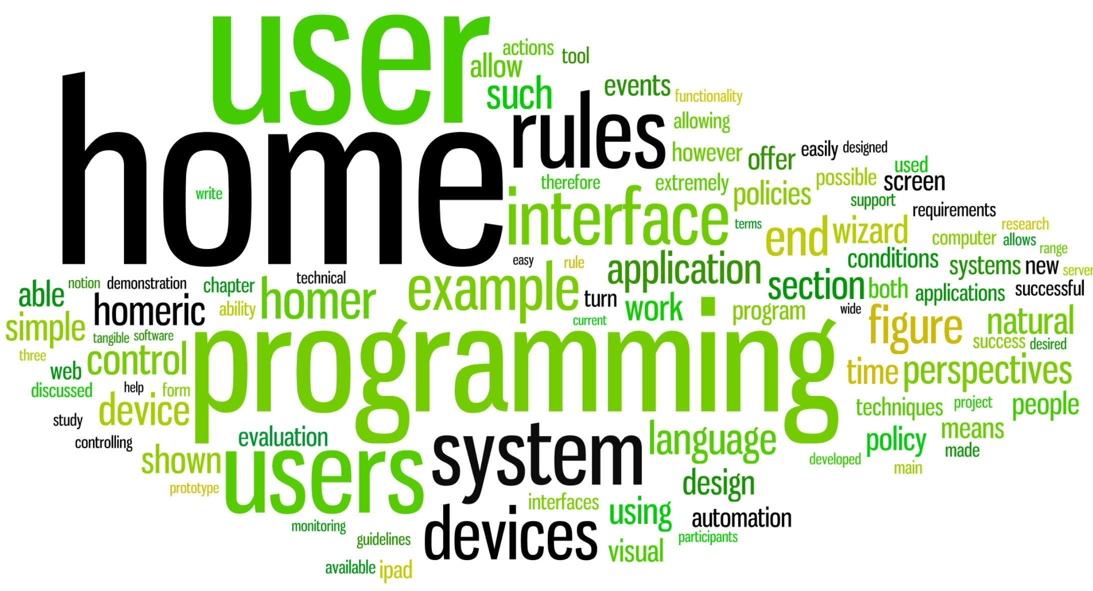
The Homer conflict detection makes use of custom user-defined environment variables (termed "environs"). Information on how any actions within Homer alter an environ is stored, allowing the conflict detection algorithms to judge if two or more actions will alter the environs (and therefore the home) in undesirable ways. The user is able to fully customise environs to offer a truly personalised conflict detection process.

An illustration presented in this chapter demonstrated that Homer correctly identified all potential conflicts between twenty sample policies. The reasons justifying why two given policies are considered conflicting is an outcome of the two-stage process and can be presented to the user. This is also highlighted in the illustration.

The conflict detection techniques for Homer extend and adapt existing conflict detection approaches to offer an advanced and customisable offline conflict detection solution.

HOMER: PROGRAMMING THE HOME

5



This chapter discusses end user applications for Homer, focusing specifically on programming the home. The work presented here provides a front-end to allow the policy language, Homeric, to be evaluated with end users, as well as means of testing and demonstrating the policy overlap, conflict and resolution work.

5.1 INTRODUCTION

It is clear that commercial home automation companies have mastered the art of providing control of the home to users. There are countless applications on varying hardware and platforms that all offer a high-quality user interface for controlling the home. Whilst the companies have been prettifying their control interfaces they have been ignoring the problem that users cannot program and customise the home themselves. Some academic projects have tried to tackle this problem with varying degrees of success. What seems to be missing from all solutions is the ability for users to both control *and* automate their homes.

Important as the underlying framework is for automated homes, the user interface also plays a very significant role. Home automation can be for anyone; no matter their age, technical ability, physical disabilities or accessibility requirements. All of these people will want to be able to interact with the home in different ways [79]. There are three main types of interaction that the user may have with the home (dependent on what the system can provide):

- Controlling – allowing the user to manipulate devices within the home, such as turning on heating or lights. This should be possible from within the home or remotely.
- Monitoring – being able to view the home so as to see the states of different devices such as the temperature or energy consumption. This is usually done remotely, from work or on holiday for example.
- Programming – a more advanced feature of home automation where the user can set rules for automating the home, for example "*when* I arrive home from work *do* play my favourite music". This would usually be defined from within the home, but could also be done remotely.

Nearly all home automation companies support control, some support monitoring, but only a handful support programming. This is not particularly surprising, as controlling a newly kitted out home is highly desirable, whereas people may not be aware of the notion of monitoring or programming their home. In terms of the user interface, companies have mastered good-looking, simple and user-friendly ways of controlling and monitoring the home (for example Control4 and Cortexa mentioned in Section 2.2.3 and 5.4 respectively). However, the major missing feature of every home automation company that has been analysed within this research project is the lack of good tools to allow users to program their home or, in many cases, the lack of support at all.

Configurability for any system is crucial, especially when it is aimed at a wide range of users. The more configuration options and the greater degree of customisation available the higher the chance of user satisfaction. Typically it is rare for any two individuals to share routines and preferences within their daily lives, even within the same family. Any home system must accommodate the wide variety of opinions, desires and preferences of the users, which will typically vary greatly between home installations. The needs of users will also change and evolve dynamically over time, therefore the ability to reconfigure the home is essential.

This chapter explores end user programming with respect to the home, and how this ties in with Homer. Firstly, the background of end user programming is outlined. The requirements for this work are stated, followed by an exploration of existing end user applications for the home. The chapter then describes design guidelines and philosophies. Next, the Homer Web Server is described, followed by the iPhone, iPad and web-based prototype interfaces. Finally, the chapter closes with a case study, evaluation and conclusions.

5.2 BACKGROUND

Researchers at MIT Media Laboratories propose the following definition for end user programming:

> *End-User Development can be defined as a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artefact.*
>
> Lieberman *et al.* [74]

Within the context of this research, end user programming is the concept of allowing any user of the home system the ability to combine device logic into mini, self-contained applications. These can be considered policies that manage the home. At the user level the concept is effectively programming the devices within the home to result in desired behaviour.

End user programming is known to be an extremely challenging problem. Many of the challenges are explained by Nardi [96], but they primarily boil down to the fact that extracting desired functionality in an unambiguous way from non-technical individuals, with no programming experience, is extremely difficult to automate.

Four different strands of end user programming research exist, each with the intention of easing the process of translating desire into unambiguous logic that computer systems can process. These strands are programming by demonstration, natural language programming, visual programming and tangible programming. Each of these are discussed in turn and examples within home automation are provided in the state of the art section (5.4).

5.2.1 *Programming by Demonstration*

Programming by demonstration, although there can be crossover, can mean two very different things:

DEFINITION 1: INTELLIGENT CONTEXT-AWARE SYSTEMS The system learns rules by observing its environment and adapting its behaviour as its knowledge base grows. An everyday example of such a system is Amazon (`www.amazon.com`), which recommends products based on previous products looked at. This approach of end-user programming is commonly used within academia in the home automation field. A home system observes the way you and your family live and manipulate your home environment; it then tries to detect patterns and attempts to automate tasks. This process is forever ongoing to continually allow more accurate and in-depth observations. Within the home the user's activities and needs can change over time, so intelligent context-aware systems offer an autonomous means of adapting to the user's dynamic lifestyle.

DEFINITION 2: MACRO-RECORDING SYSTEMS The user describes a scenario or series of events by physically manipulating the devices within the environment. The system records the series of events that take place and can then save these as rules for the home. This method aims to be a simple way for users to program their system, as instead of trying to express what they want hypothetically using some constrained user interface, they can simply carry out the tasks in real life. However, a major problem users have when trying to program is how to express what they want. For example how would a user express safely what should happen when a fire alarm or burglar alarm is activated? Another limitation of the system is context – how does the system knows which information, conditions and events are relevant. For example, is the current temperature, weather, light level, time of day, etc. important and applicable to the current macro?

5.2.2 *Natural Language Programming*

Users can define rules using natural language, for example "when it is cold outside keep my house warm". This could be provided through various means, for example speech, handwriting or keyboard input. Natural language is very attractive to the end-user due to the lack of constraints. However, programming systems which can support natural language are extremely complicated [58, 106, 139]. If the challenges of computer interpretation of logic and rules expressed in natural language can be overcome, this method of end user programming could prove successful and favourable.

5.2.3 *Visual Programming*

Visual Programming is where users can program by manipulating visual representations of programming elements. There exist many successful visual programming languages which are widely used [18]. This is a popular method for end user programming as it is a hybrid of easy interpretation by computer systems as well as a simplified means of programming for the user. Unfortunately this method is based upon logical connections and flow between elements which can prove challenging for some individuals.

5.2.4 *Tangible Programming*

This allows users to physically piece together component representations to form desired rules. In many respects, tangible programming is similar to visual programming in the sense that the end user combines individual programming elements together, minimising ambiguity and allowing the computer to interpret what the user has expressed literally. The advantage of tangible programming is the lack of screen-based user interface. By moving the user away from the typical computer and into a more physical, dynamic and non-technical environment, the user should arguably feel less intimidated and restricted and feel more free and confident to play with ideas and learn.

5.3 REQUIREMENTS

User Studies carried out and discussed in Chapter 2 produced the following list of requirements for user interaction with the home:

- **Accommodating** Cater for a very wide range of users.
- **Multi-Devices** A range of devices and platforms should be available to interact with the home system.
- **Multi-Interfaces** Touch interfaces should definitely be offered, but not necessarily excluding other modes of interface such as voice, remote control or gesture.

- **Multi-Perspectives** Allow users to be able to interact with the devices in the home from four different perspectives: location, device, personal and time.
- **Remote Control** The ability to control the home remotely, such as at work or on the move.

This list of requirements accurately describes the core requirements for any home system. The Homer framework supports all three types of functionality: monitoring, controlling and programming. All of these are exposed through Homer interfaces.

CEDIA (*Custom Electronic Design and Installation Association*, www.cedia.co.uk), the international trade organisation for the home electronic systems industry, held in London in June 2010 provided me the opportunity to obtain direct experience of many of the leading home automation user interfaces and tools. It is definitely fair to say that, for the most part, the user interfaces are of a high standard and would be difficult to improve. This visit also confirmed that controlling and monitoring homes was neither new nor novel, but companies simply looked blank when asked about providing users with the ability to program their home. So, due to the maturity of monitoring and controlling interfaces in the commercial world, the end user programming aspect was chosen as the main focus of this research. This enhances the research performed on policies, described in Chapter 4.

5.4 STATE OF THE ART

This section explores the most significant and relevant work carried out within the four main categories of end user programming, then extracts conclusions to shape the design guidelines and philosophies described throughout the rest of the chapter.

5.4.1 *Existing Work*

Research efforts within programming by demonstration, natural language, visual and tangible programming are discussed below.

5.4.1.1 *Programming by Demonstration*

INTELLIGENT CONTEXT-AWARE SYSTEMS

ACHE (*Adaptive Control of Home Environments* [93]) is a system which automates the control of heating, lighting and ventilation within the home. ACHE has two main goals: maximise user comfort and minimise costs. The system learns the user's preferences in home comfort, and tries to automate such settings for the user. If the user has to manually adjust the home (for example, turning off lighting or turning on the heating) the system can learn from this and continually try to satisfy both goals with increasing accuracy. The developer's motivation for intelligent context-aware systems is that people typically do not enjoy programming their home VCRs, so there is a question of why they would want to program their own home.

MACRO-RECORDING SYSTEMS

A CAPpella [36] is a context-aware prototyping system which can be programmed by demonstration, expressed as a situation and an associated action. A GUI is then used to select which portions of the demonstration are relevant. The next time the system detects the recorded situation, it will perform the specified action. The authors argue that their system allows users to define much more complex rules than those that have to be defined more concretely. A CAPpella's feasibility study demonstrated the users' ease and liking for the system when creating rules [36].

Alfred [40] is a natural language end user programming interface for intelligent environments which terms itself a "multi-modal macro recorder". Authors Gajos *et al.* state that human-centred computation should be "adaptive, reactive, and empower the user to configure and extend the behaviour of the system using natural modes of interaction". Alfred allows users to program the system by specifying a name for a new goal, demonstrating one or more actions that should take place, and then telling the system any conditions of the goal. The interaction is primarily by speech command, which is discussed further in section 5.4.1.2, but still supports the notion of programming the home by demonstrating physical manipulation of devices.

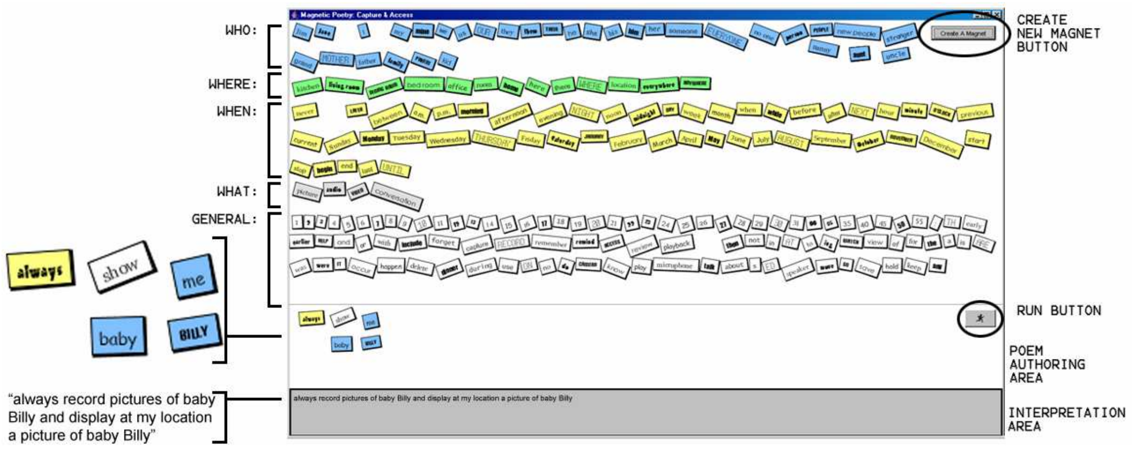
5.4.1.2 *Natural Language Programming*

Alfred [40], introduced and described above in section 5.4.1.1, is an example of a system which uses a mixture of end user programming methods, in this case: programming by demonstration and natural language. Natural language plays a strong role in this solution, allowing users to engage in a "conversation" with the system. Here is a sample dialogue taken from [40] which demonstrates a user programming a new rule within the home:

> User: I want to record a new macro.
>
> Computer: Beginning to record a macro. Say 'stop recording' when you are done.
>
> User: Turn on the main lights. Open the drapes. Turn on my desk lamp. Say 'good morning.' Stop recording.
>
> Computer: What phrase would you like to associate with this macro?
>
> User: 'Good morning, computer.'
>
> Computer: Any other phrase?
>
> User: No, I am done.
>
> Computer: Macro added!

From now on, when the user says "Good morning, computer" the lights will turn on, the drapes will open, the desk lamp will turn on and the computer will say "good morning". Alfred demonstrates effective use of natural language to allow users to create rules with instant feedback and assurance. Unfortunately, no user testing has been carried out on this project.

CAMP (*Capture and Access Magnetic Poetry* [126]) allows users to define goals and rules at a very high level by piecing together words from a library in any desired order (shown in Figure 5.1) in the style of magnetic poetry. It is used as a way of providing users with a flexible, yet computationally constrained, means of natural language programming. The system parses natural language rules into a lower level intermediate representation ready for the underlying capture and access system, Infrastructure for Capture and Access (INCA). The preliminary user evaluation reaffirmed the developers' belief that CAMP's interface was extremely simple to use and allows their users flexibility to express their desires in a way that makes sense to them.



"always record pictures of baby Billy and display at my location a picture of baby Billy"

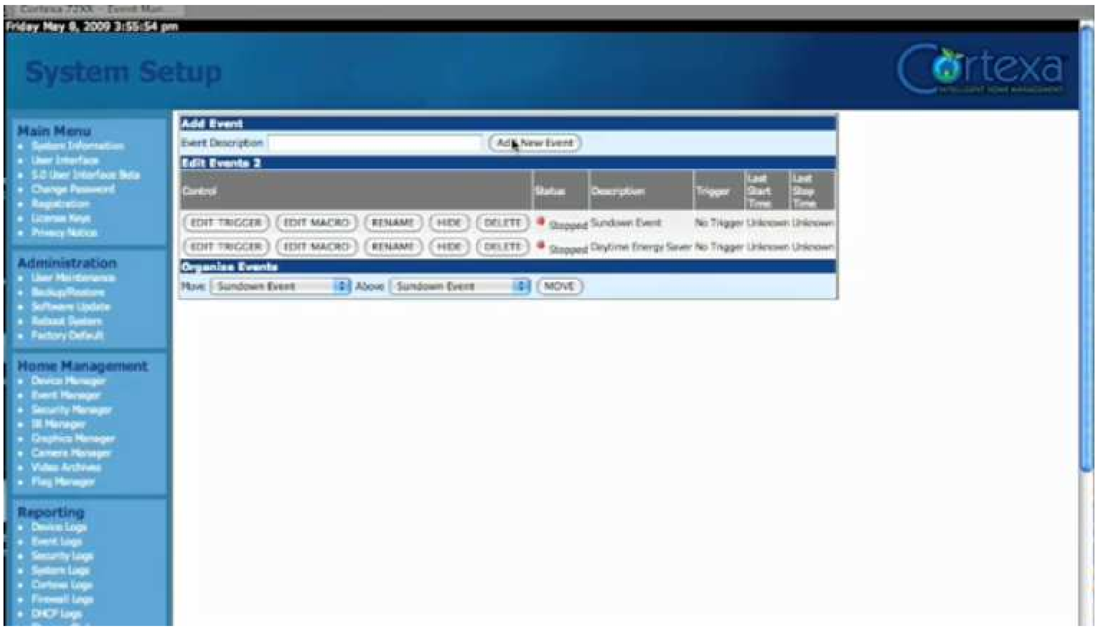
always record pictures of baby Billy and display at my location a picture of baby Billy

Figure 5.1: CAMP's Magnetic Poetry Interface [126]

Knoll *et al.* [66] demonstrates their visual scripting language for programming ubiquitous computing environments, discussing their successful experiences of user trials in creating rules using their graphical editor with either a mouse or digital pen. Their system is, however, very constrained and users can input only a limited set of commands.

5.4.1.3 *Visual Programming*

Cortexa (`www.cortexa.com`) is a top-of-the-range home automation solution that claims to offer "the most user-friendly, secure, powerful and simplistic system available". It supports a wide range of home automation hardware. The package attempts to combine and expose many services and applications through a simple user interface. Cortexa does allow end users to program the home, however this is exposed through a very administrative user interface (shown in Figure 5.2) and requires technical experience to use. This shows that even a high-profile commercial home automation company has not been able to produce a workable solution to allow end-users the flexibility and ease of automating devices in their home.



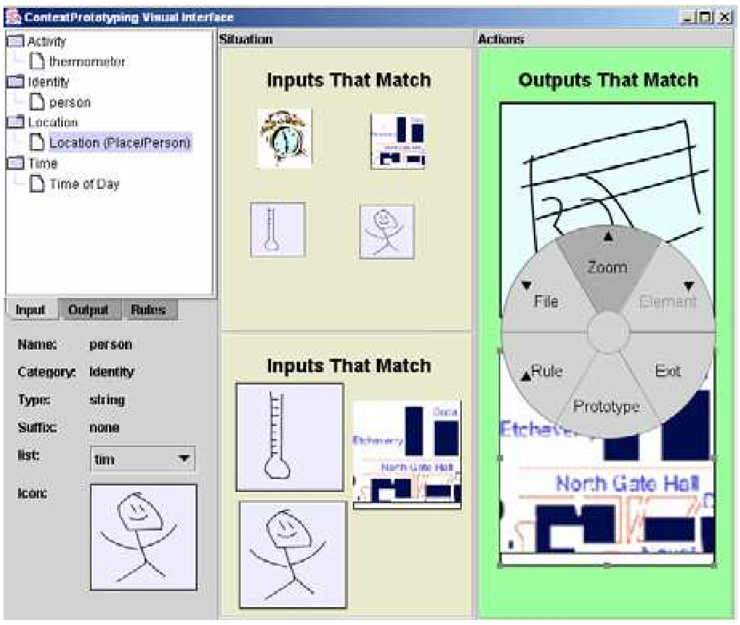
Control	Status	Description	Trigger	Last Start Time	Last Stop Time
EDIT TRIGGER, EDIT MACRO, RENAME, HIDE, DELETE	Stopped	Sundown Event	No Trigger	Unknown	Unknown
EDIT TRIGGER, EDIT MACRO, RENAME, HIDE, DELETE	Stopped	Daytime Energy Saver	No Trigger	Unknown	Unknown

Move: Sundown Event Above: Sundown Event

Figure 5.2: Sample of Cortexa's interface for creating rules within the home[31] (poor quality as copied from a video).

Girder (`www.promixis.com`) is a tool to allow the mapping of input events, such as key presses on a keyboard, to an output event, such as *play* in iTunes. It has support for most of the common home automation standards such as X10, Insteon and HAI. The software is rather immature, with a basic menu-driven interface to add and configure devices, as well as to map input and output events of these devices. This process requires technical expertise and even custom coding at times.

iCap [117] is a visual programming tool for defining rules within the home. When new devices are connected to the system, the user draws a small icon to represent the device. These can then be dragged onto two windows: situation or action (as shown in Figure 5.3). The situation window holds conditions of the rule, allowing input devices to be placed within the window. The action window holds the action events, made up of output devices, to occur if the conditions are met. The user is able to test rules in a simulation mode.

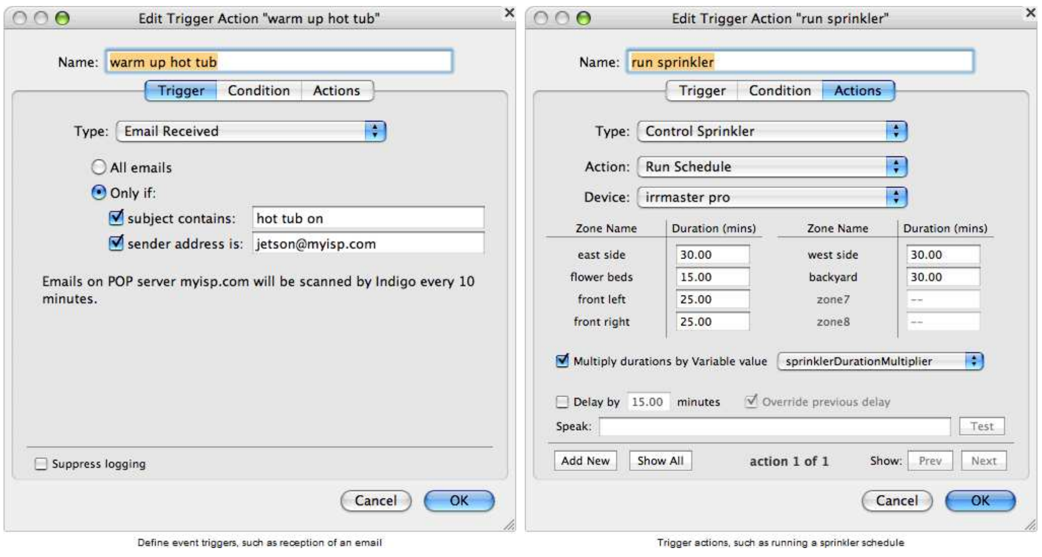


Activity
thermometer
Identity
person
Location
Location (Place/Person)
Time
Time of Day

Name:	person
Category:	identity
Type:	string
Suffix:	none
list:	tim
Icon:	

Figure 5.3: The iCap user interface [117]

Indigo (`www.perceptiveautomation.com`) is a superior home control system for X10 and Insteon devices. It provides computer, web and iPhone/iPad interfaces to allow full control over all supported devices within the home. Rules can be built for the home, supporting triggers, conditions and actions. The interfaces for defining a trigger and action are shown in Figure 5.4. This is very form-filling in nature and appears to be clear and simple to understand. Unfortunately, it is unknown how successful this interface has been for users.



Edit Trigger Action "warm up hot tub"

Name: warm up hot tub

Type: Email Received

All emails
Only if:
☑ subject contains: hot tub on
☑ sender address is: jetson@myisp.com

Emails on POP server myisp.com will be scanned by Indigo every 10 minutes.

☐ Suppress logging

Define event triggers, such as reception of an email

Edit Trigger Action "run sprinkler"

Name: run sprinkler

Type: Control Sprinkler

Action: Run Schedule

Device: irrmaster pro

Zone Name	Duration (mins)	Zone Name	Duration (mins)
east side	30.00	west side	30.00
flower beds	15.00	backyard	30.00
front left	25.00	zone7	--
front right	25.00	zone8	--

☑ Multiply durations by Variable value sprinklerDurationMultiplier

☐ Delay by 15.00 minutes ☑ Override previous delay

Speak:

action 1 of 1

Trigger actions, such as running a sprinkler schedule

Figure 5.4: Indigo's user interfaces for defining triggers and actions for the home.

Kodu is a Microsoft Xbox 360 game (Microsoft's game console, `www.xbox.com`) for allowing gamers to create their own game. Kodu is designed for any age, though is primarily aimed at children. Each object can be given a large number of rules in the *when-do* format (shown in Figures 5.6 and 5.7). The interface is extremely simple and fully icon-based, requiring very little explanation of how to use it. However, a simple and fun introduction to how the rules and interface work is still given. To confirm that their user interface can be successfully used at a young age, Microsoft performed a five month trial with 8-11 year old children which was extremely successful [113]. Another important aspect to note about this user interface for end user programming is that its sole input mechanism is an Xbox Controller (see Figure 5.5), which has a limited set of buttons.

Figure 5.5: A Microsoft Xbox Controller.

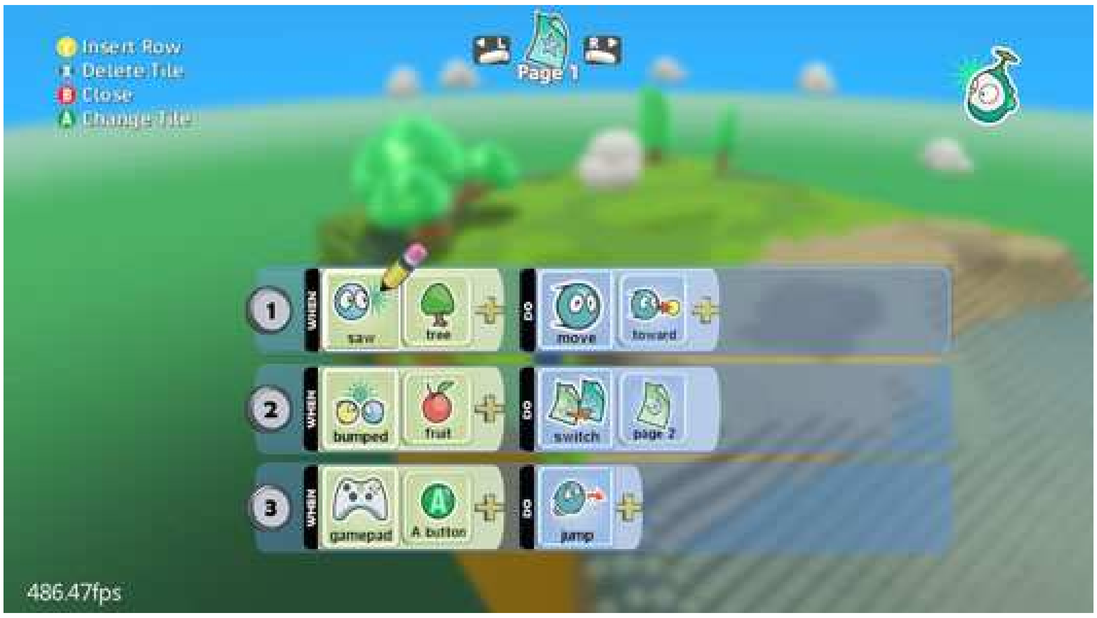


Figure 5.6: Kodu's rule-building menu [91].

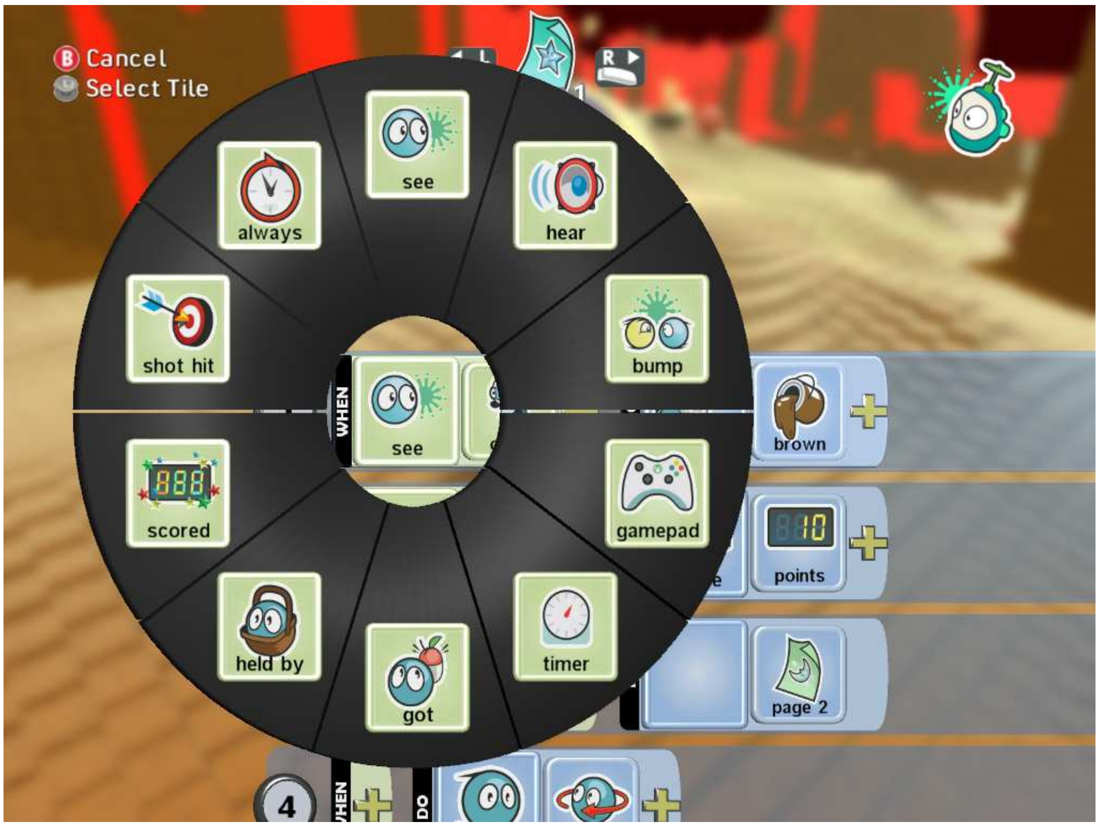
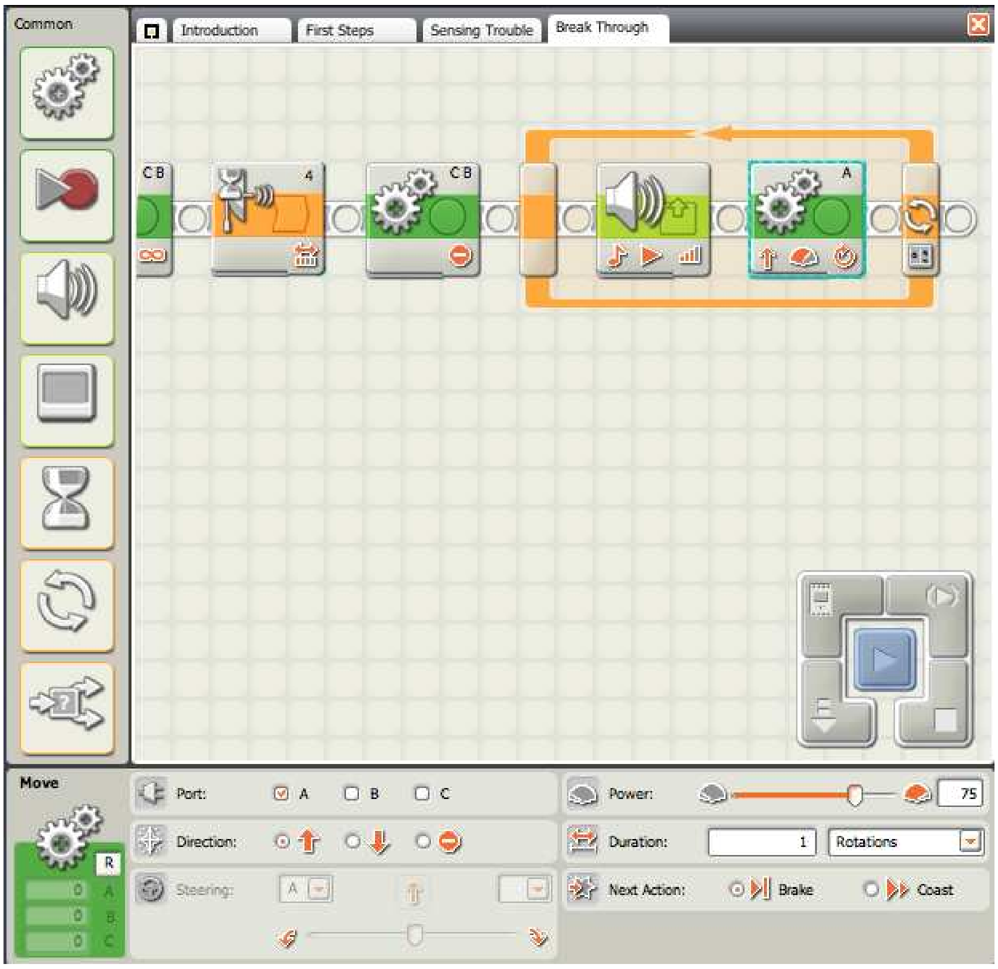


Figure 5.7: Kodu's doughnut-style menu [91].

Kolberg *et al.* present work which utilises digital paper and pen technology to offer non-technical users a simple means of controlling the appliances in their home [68], primarily focused on programming Personal Video Recorders (PVRs). Forms can be created which use the Anoto patterned paper technology to allow users to fill in a series of check boxes to define rules for their home. The home system, which they developed using OSGi, then analyses any completed form and forwards the request to the desired device (video recorder in their case). Pen and paper technology provides a simple, user-friendly and non-technical means of interfacing with a home system and Kolberg *et al.* have received a positive response from computing scientists, teachers and pupils.

Lego Mindstorms robots are programmed using a simple graphical programming language. This software treats the robot as a set of sensors and actuators. The user can piece together graphical representations of these sensors and actuators to form rules. Programming flow concepts, such as loops and branches, can be easily incorporated. A review of the software can be found in Knoll's literature study [66], where he claims that the software "can even be mastered by children". A demonstration of the software can be watched at [70] and a screenshot of this is shown in Figure 5.8.

Oscar [97] was a project working on easing the interoperability of media devices within the home. The application is interfaced through a small touch screen tablet PC (shown in Figure 5.9), where user trials were carried out and showed that users were comfortable with this medium as it was similar in form and function to remote controls. Users also found the application easy to use with no manual or prior instructions. Oscar allows users to create simple "setups" which define how components are found, selected and connected to each other to carry out the desired activity. Setups are mostly for routing media streams to hardware, which can then be run on demand. An example setup is shown in Figure 5.10 where music from the central server is requested to play in the kitchen. Two screenshots of



Port: [✓ A] B C

Direction:

Steering:

Power: 75

Duration: 1 Rotations

Next Action: Brake Coast

Figure 5.8: Lego Mindstorms [70]

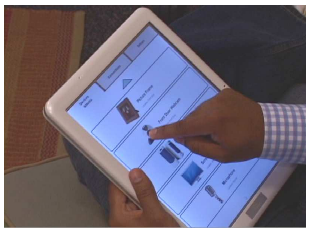
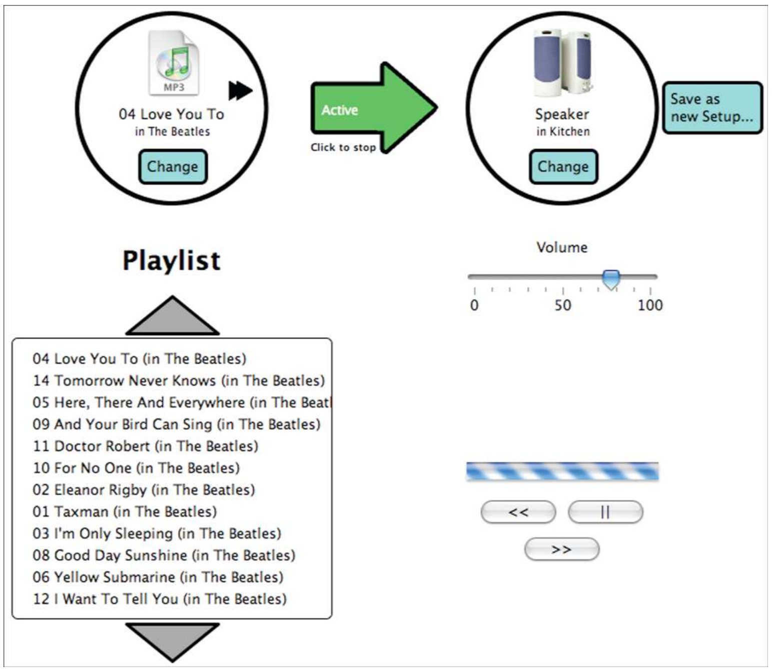


Figure 5.9: Oscar [97].



04 Love You To (in The Beatles)
14 Tomorrow Never Knows (in The Beatles)
05 Here, There And Everywhere (in The Beat
09 And Your Bird Can Sing (in The Beatles)
11 Doctor Robert (in The Beatles)
10 For No One (in The Beatles)
02 Eleanor Rigby (in The Beatles)
01 Taxman (in The Beatles)
03 I'm Only Sleeping (in The Beatles)
08 Good Day Sunshine (in The Beatles)
06 Yellow Submarine (in The Beatles)
12 I Want To Tell You (in The Beatles)

Figure 5.10: Sample Oscar "setup" [97].

the application are shown in Figure 5.11, one showing the list of devices available to the user and the second displaying any setups that have been created by the user.

Tasker is an Android application which allows users to fully program their phone. It extracts all the individual features and functions of the phone and allows the user to put these together in the form of rules. Tasker has the notion of *contexts* (application, time, date, location, event, gesture) which when activated can perform *tasks* (set of actions). A screenshot showing a series of rules can be seen in Figure 5.12. The application has proven extremely popular, winning numerous awards and gaining many rave reviews. One such quote from a journalist is:

> *"When it comes to device automation, there's just one 900lb gorilla in the Android space, and that's Tasker."* [147]

Tasker's popularity and success may be primarily with more technically-minded users, but this can still demonstrate a highly successful end user programming interface. Secondly, the passionate reviews for programming phones help confirm that there is desire for programming the devices in people's lives (and, by extension, in their homes).

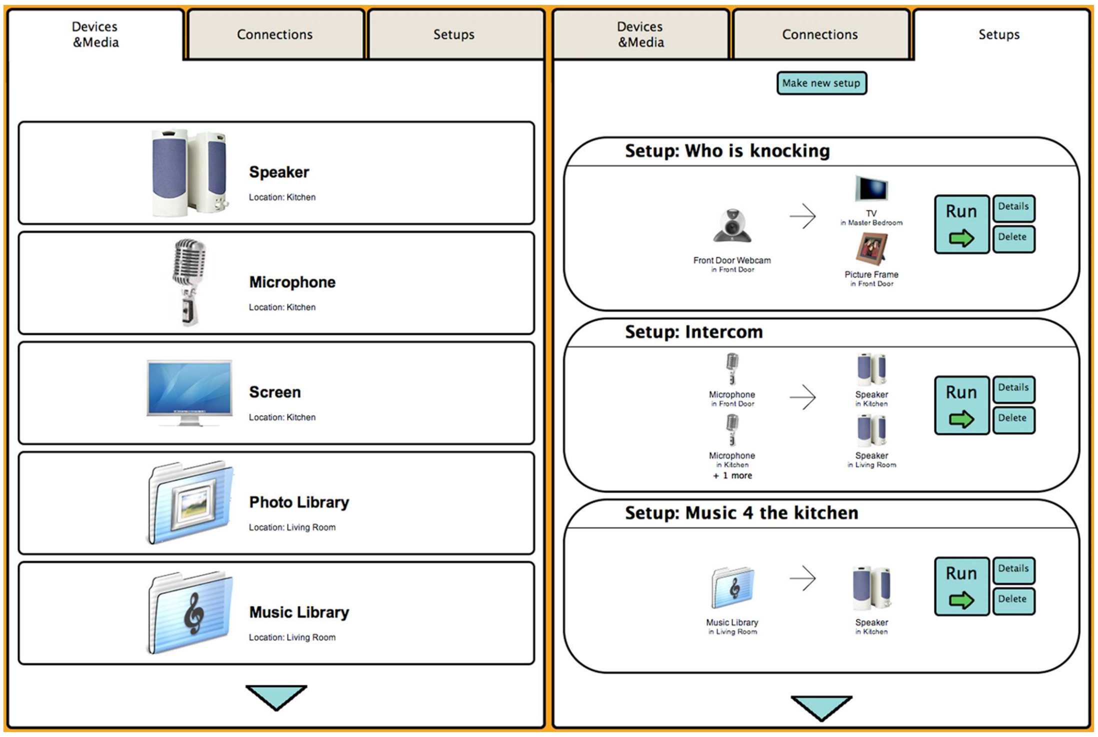


Figure 5.11: Oscar [97].

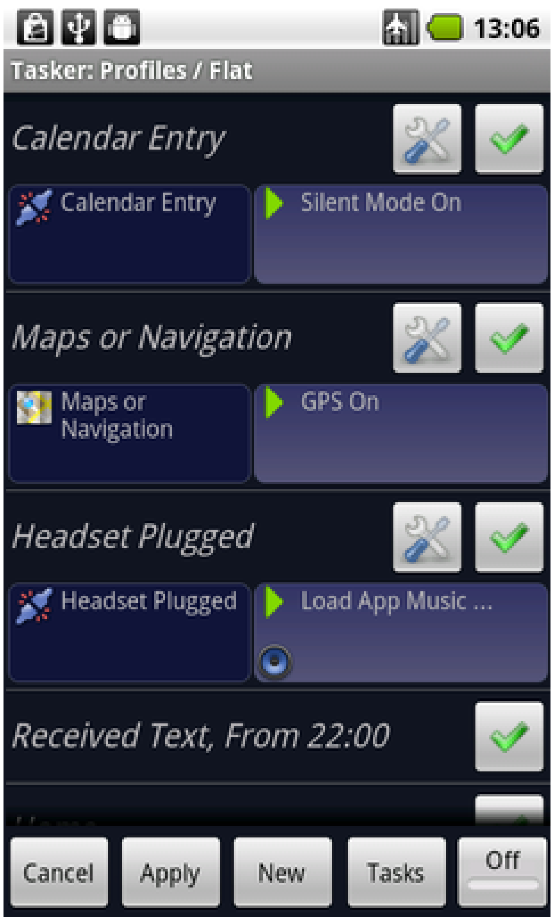


Figure 5.12: Tasker (`tasker.dinglisch.net`).

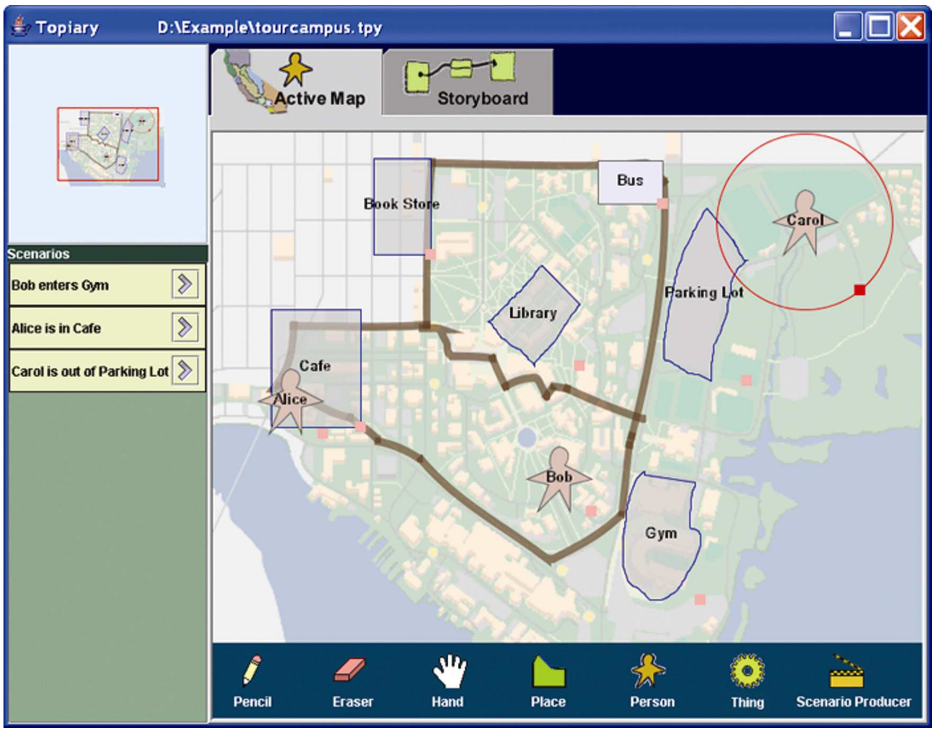


Figure 5.13: Topiary Map Interface [73].

Topiary [73] is a tool to provide a quick and easy way to prototype location-aware applications. The relevant part of this project is its user interfaces (shown below in Figures 5.13, 5.14 and 5.15). Figure 5.13 shows a map in which users can add routes, places, people, areas, and 'things' (for example cars and printers). To add such items the user is provided with a simple interface to draw areas and shapes over a map and drag-and-drop items of interest. Figure 5.14 demonstrates the process for defining 'scenarios' using Topiary. Scenarios are the concept of events and conditions. The user can select an area of the map and choose the Scenario Producer tool. Then, by the act of manipulating the objects on screen, various scenarios can be represented. Finally, with properties defined from Figure 5.13 and scenarios from Figure 5.14, the user can piece these together to form interaction sequences using a storyboard (Figure 5.15). The storyboard concept had not been used for such an application before but, despite its novel and friendly interface, user testing did not show positive results. This is mostly due to users not fully understanding the Topiary storyboard interface nor its usefulness.

Twine `supermechanical.com/twine` is sold as a cheap and easy way to:

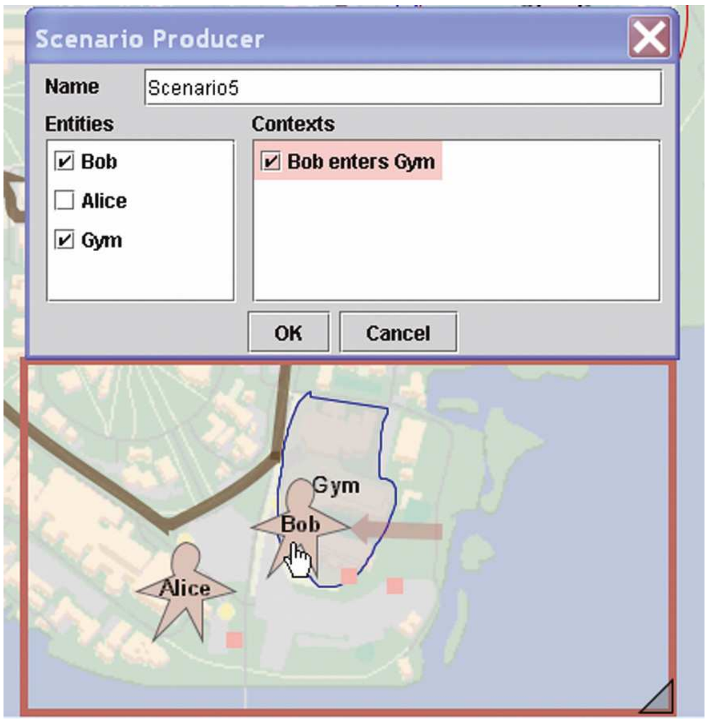
"get the objects in your life texting, tweeting or emailing"

The physical Twine device (shown in Figure 5.16) is a 2.5 inch square with various in-built sensors. These sensors can be programmed to text, tweet[1] or email. An example of the company's web-based user interface is given in Figure 5.17, showing the use of the common *when-then* policy format and simple pull-down menus that give the user a predefined list of options. This interface was designed to not require a "nerd degree", allowing non-technical users to enhance the devices within their home.

5.4.1.4 *Tangible Programming*

Physical component programming is becoming easier. This is thanks to various projects which focus on abstracting the hardware programming and providing an API for software

1 Post a Twitter message.

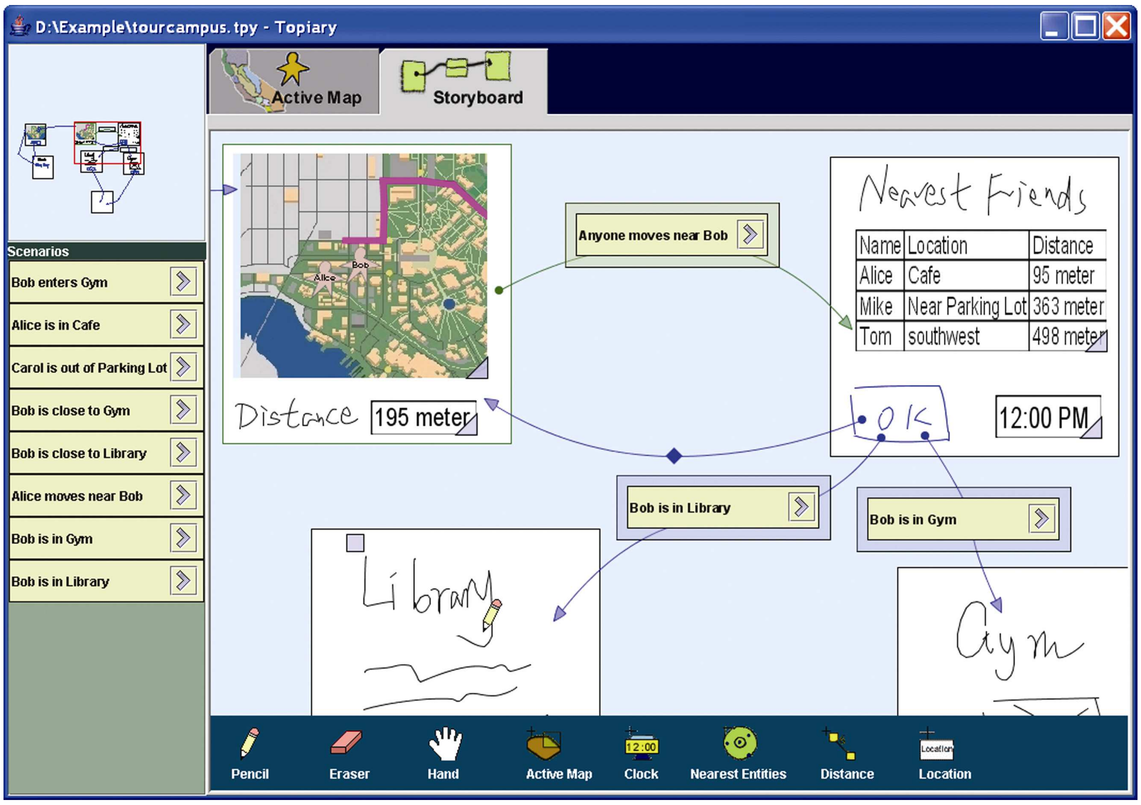


Name: Scenario5

Entities: [✓] Bob, [ ] Alice, [✓] Gym

Contexts: [✓] Bob enters Gym

Figure 5.14: Topiary Scenario Producer Interface [73].



Scenarios
- Bob enters Gym
- Alice is in Cafe
- Carol is out of Parking Lot
- Bob is close to Gym
- Bob is close to Library
- Alice moves near Bob
- Bob is in Gym
- Bob is in Library

Nearest Friends

Name	Location	Distance
Alice	Cafe	95 meter
Mike	Near Parking Lot	363 meter
Tom	southwest	498 meter

Figure 5.15: Topiary Storyboard Interface [73].

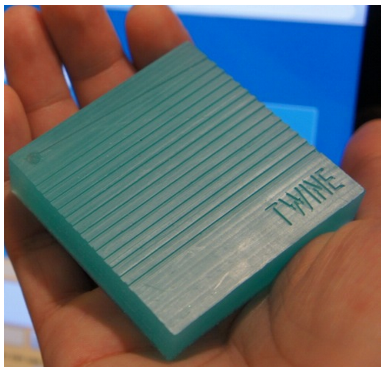
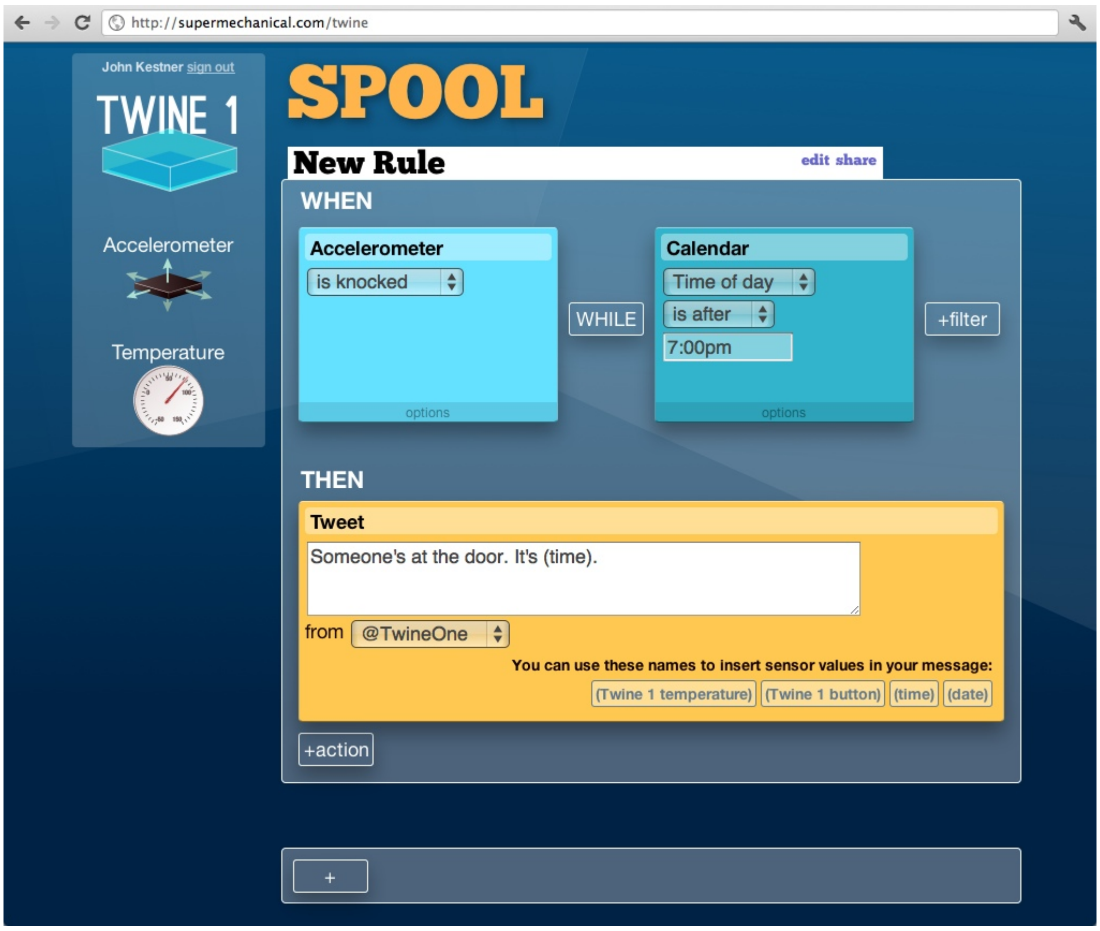


Figure 5.16: Twine Device (supermechanical.com/twine).



Someone's at the door. It's (time).

from @TwineOne

You can use these names to insert sensor values in your message:

Figure 5.17: Twine (supermechanical.com/twine).

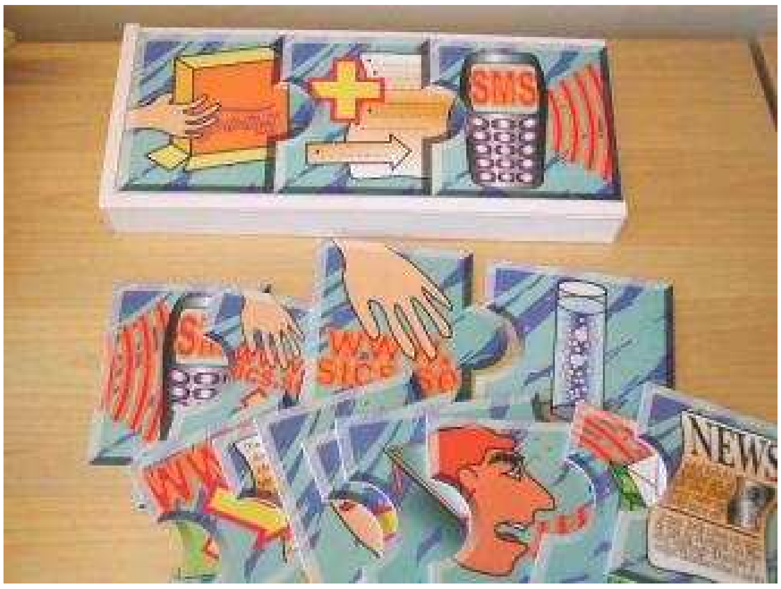


Figure 5.18: Accord Project, showing the jigsaw pieces and a sample policy (get a shopping list for groceries as an SMS to your mobile phone) on the custom jigsaw reader [1].

programmers. This allows for faster and easier means of making use of physical components in higher-level software applications and systems. Examples of such projects include iStuff [8], Phidgets [46] and Papier-Mâché [65]. The most relevant examples of tangible programming are discussed below.

ACCORD (Administering Connected Co-Operative Residential Domains) [53, 107] is a framework for allowing dynamic configuration of a library of components. These components can be combined to form policy-style rules within the home. The policies are created using physical wooden jigsaw pieces connected together as shown in Figure 5.18. User trials carried out in [53, 107] were very successful, highlighting that users were able to easily grasp the notion of jigsaw pieces representing various devices or functions and combining them to form connections. The users were able to create sets of connected components to solve example problems, they also suggested further components with sample applications.

Akesson *et al.* [2] developed a barcode scanner technique for programming devices within the home. Their work is similar to the Accord project and has substantial crossover with the second definition of programming by demonstration discussed above. A system has been developed which supports the creation of rules within the home by using a barcode reader and attached Personal Digital Assistant (PDA) offering further options. Devices are connected by linking their functions. The barcode reader can also be used to allow the user to see what rules a particular device is involved in.

Media Cubes is a project which uses three dimensional wooden cubes (shown in Figure 5.19) as a means of defining rules within the home [13]. The cubes communicate using infrared and can be paired or placed beside devices within the home to signify different requests. The latest work to happen on this project was 2001 [45], so unfortunately the technologies used are rather dated. However, their work remains unique.

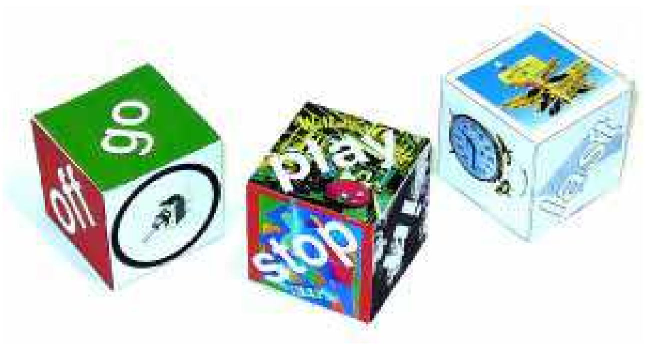


Figure 5.19: 3D Media cubes [13].

SiteView [10] is a project for programming an office, though intended to be extendible to a home environment. The model is based on conditions and actions, whereby up to three conditions are specified using RFID tagged cards which represent conditions: times of day, weather and temperature. These are placed in designated places (as shown in Figure 5.20 labeled "condition composer"). An action is then specified by placing another RFID tagged card on a 2D floor plan of the office. There is a screen (also shown in Figure 5.20, labelled "environment display") which displays a photograph of the office in the state of the conditions specified. When an action is placed, the photograph changes to show what the office would look like if that action were carried out.

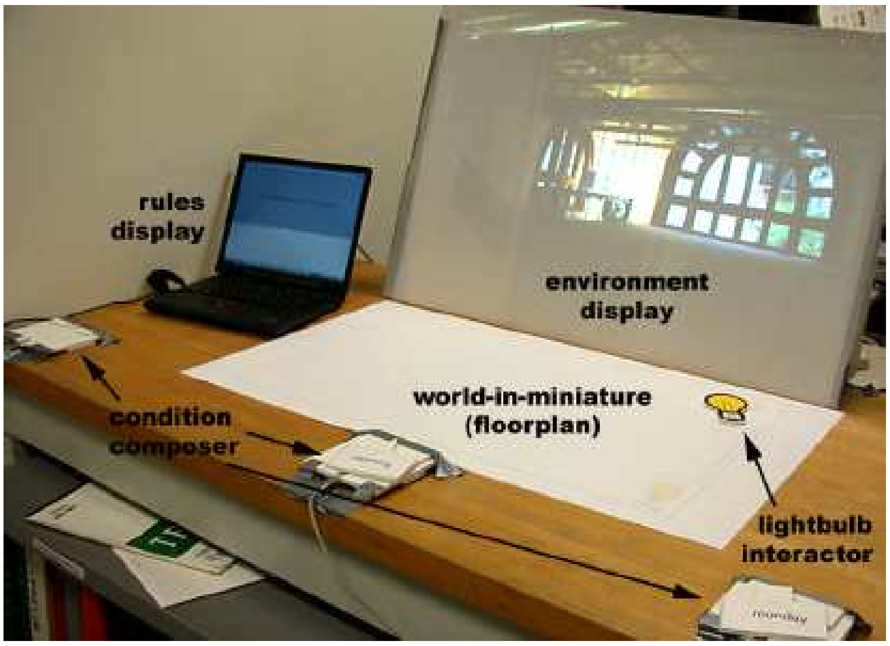


Figure 5.20: SiteView [10].

5.4.2 *Analysis*

There is no one-solution-fits-all in end-user programming. Most of the solutions above focus narrowly on one technique instead of trying to explore a hybrid. This can result in rigid, inflexible and uncustomisable solutions. Take, for example, the Accord project; it allows users to piece together three wooden jigsaw pieces to form a home policy. This is extremely restrictive as policies are then tied to the form input-process-output, where each device state must be represented as a separate jigsaw piece. Since there is no way to specify details about a device on a particular jigsaw piece, there is no way to query or verify what the newly formed policy will do or receive feedback of any kind. Suffering from similar restrictions, yet using a different method, is a CAPpella. This system allows users to program rules within their home by demonstrating desired scenarios and events through physically manipulating the devices. This solution does not allow users to deal with situations that are difficult to demonstrate. For example; consider “when the fire alarm is activated”, “when the temperature reaches 18°c in the living room” or “when it is sunny outside”. Both Accord and a CAPpella suffer in functionality due to restricting the control and setup of rules within the home to one type of programming method.

Many solutions are still very device-centric. Other researchers in the field also agree:

> *"Despite their use of simplified input languages and mechanisms, these systems tend to be device-centric rather than user-centric, task-centric, or goal-centric. They require that users approach the configuration of ubicomp applications from the perspective of a developer, by treating application development as the configuration and integration of devices and sensors rather than a domestic goal or task that a user is trying to achieve."*
>
> Truong *et al.* [127]

Programmers typically create very device-oriented home frameworks and then attempt to make them usable for end-users. It is crucial that the user's goals are focused on, rather than the developer's view. Truong *et al.* carried out a study to analyse how users think about context-aware capture applications:

> *"As we had hypothesized, the results of our study showed that people who had no experience developing ubiquitous computing applications tended to frame the descriptions of their desired applications in terms of their domestic goals and needs rather than in terms of device behaviours."*
>
> Truong *et al.* [127]

These findings reiterate how important it is to focus on user goals, and avoid restricting the user by offering only one view of the system.

For many of the solutions described above, it would be extremely difficult to extend or to support the addition of new devices and features. SiteView, for example, displays photographs of the environment matching the given conditions set by the user and then what the environment will look like if the chosen action is carried out. This will work well for a small set of possible conditions and actions. However, in Homer, possible conditions and actions can be added at any point. Photographs will therefore be difficult to keep up-to-date, especially photographs of mixed conditions which would result in an exponentially growing number of photographs required. On a similar note, other examples of the requirements necessary to extend supported functionality on projects discussed above include: new jigsaw pieces would need to be developed and programmed for each new device added to the Accord project, new vocabulary and rules would need to be added to Alfred to allow the system to understand and handle new devices and scenarios, and new rules and training for ACHE would be needed to handle other services of the home. A home and its devices change and develop over time. Any home system needs to be able to handle this dynamically and easily, and ideally with a large degree of automation.

Each of the four main categories of end user programming has both advantages and disadvantages which are summarised below.

- **Programming by demonstration**
  - **Intelligent Context Aware Systems** the lack of user input is the major plus point. Users need not feel responsibility for customising and maintaining the system, as they can just leave the system alone and relax in the knowledge that it has some level of intelligence and is making the design decisions for them. However, such a system takes away the feeling of control from the user. This can be a good thing, but some users may like the ability to override or specify particular rules within the home.

  - **Macro Recording Systems** can offer a very simple means of specifying conditions and events for the system. Mixed with other means of programming, this can result in a relatively flexible and simple to use system. However, the technique is limited in terms of what can be demonstrated.

- **Natural Language Programming**, when done well, can offer a truly natural interaction method for programming. Users can think about rules very differently from each other, so the flexibility and freedom natural language can offer when specifying rules can be extremely useful. However, poor implementation can result in problematic systems where users are required to remember a very particular and limited vocabulary and set of phrases. These would need to be extended each time a new device is added to the system.
- **Visual Programming** offers a flexible environment that can be extended and adapted as the underlying system changes and new devices are added. The resulting interface has the possibility of having very few dependencies, and therefore can be accessed from anywhere through some type of computer interface such as a web browser. Once the user is familiar with the interface, they can program rules relatively quickly and easily. Visual programming, however, can prove challenging for end users if not made simple enough. The flip-side of an overly simplistic user interface means programming functionality has to be seriously reduced. Making the approach easy to understand by any user and yet still offering the full desired functionality is no easy task. Attention should be drawn to the success of Kodu, a purely visual programming solution designed for children to control with only a games console controller. It's success is both extremely impressive and a prime example of a well designed visual programming solution.
- **Tangible Programming** takes users away from the computer in any conventional sense and therefore reduces any preconceptions, nerves or discomfort from the user, and is potentially more appealing to a wider audience. However, although it can prove simple and novel for end users, in reality it is restrictive and primitive. Tangible programming systems are also typically hard to extend, customise and obtain feedback from.

These techniques individually offer both advantages and disadvantages, however the most successful projects above were ones that had major crossover with other techniques. Examples include Alfred (demonstration and natural language) and CAMP (natural language and tangible). These projects offer a far more flexible solution, allowing the user to find ways that work best for them, rather than being forced into one particular programming method.

5.5 DESIGN GUIDELINES

Having critically reviewed the current techniques and solutions for end-user programming and home systems, the following design guidelines are recommended when producing a successful solution for programming the home:

- **The Interface Must**:
  - Allow users to feel in control of their home; even if they are happy with the home taking care of them, users need to know that they can both:
    - take control at any point, and
    - query the system as to why certain events occurred.
  - Keep the interface and user programming methods simple.
  - Be easily extended and adapted as new technologies come along.
  - Avoid forcing users into one particular way of thinking, instead make use of the notion of perspectives to allow users to interact with a home system in a way that is comfortable and familiar to them.
  - Provide instant feedback to users for reassurance that they have done the right thing. For example; show the user the rule which they have just created and confirm that the system has saved this rule.

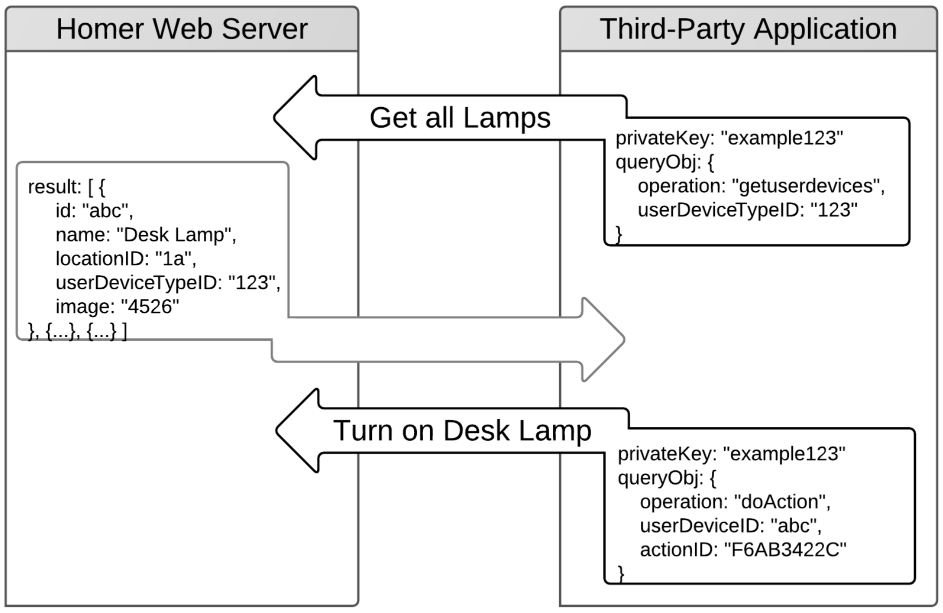


Figure 5.21: Example Communication with the Homer Web Server.

   - Support the expression of rules and events in logically equivalent forms. For example: the user should be able to express "when I get home turn on the kettle" and "turn on the kettle when I get home".
- **The Interface Should**:
   - Provide a means of doing more complicated tasks if the user so desires, but do not scare more timid users with too many options.
   - Allow more than one underlying platform so users can choose which suits them best at any given time. The home works for the user, not the other way around. For example, provide both a mobile phone application and PC version.
   - Accessible from anywhere. For example; whilst at work or on holiday.
- **The Interface May**:
   - Provide an option that the system can function with no configuration by the user. For example, a context aware system might be used.

To conclude, the quote written in Chapter 2 is highly appropriate:

"Simple things should be simple and complex things should be possible."

Alan Kay [71]

These guidelines are revisited in the following sections with regard to Homer.

5.6 HOMER WEB SERVER

The Homer Web Server was developed to allow external access to the functionality offered by Homer. By exposing the functionality through a web API, endless end user applications can be supported which expose *any functionality* through any *web-enabled device*. As examples, it is possible to create a website for monitoring the home, a mobile phone application for controlling the home and a tablet application for monitoring, controlling and programming the home. These applications can be developed by any third-party developer granted permission to access the Homer web API.

The Homer web API operates over HTTP/HTTPS, using JSON for a lightweight and highly popular means of data interchange. The API is exposed through the embedded OSGi web server. When Homer starts, the dedicated OSGi Homer Web Server bundle is also started. This bundle initialises the OSGi web server and installs the Homer API as a servlet.

Any (approved[2]) developer can contact the servlet, exchanging information to support the end user application. The list of supported functionality is:

- **Obtain Information** – All Homer information regarding devices, locations and events can be requested, optionally with given filters. Some examples include: all locations with devices that have actions associated within them, all devices within the kitchen, all X10 devices.
- **Event Handling** – A callback URL can be provided by applications as a means of registering a listener for triggers occurring (again, with any desired filters). Conditions can be verified and actions requested in a simple request.
- **Policies** – All features of the policy system are exposed including writing, editing and deleting a policy, checking for conflicts between policies, and enabling and disabling policies.
- **Home Setup** – Add, edit and delete locations and devices within the home, allowing full customisation of a home installation on any device.

To illustrate the Homer Web Server a simple example is outlined in Figure 5.21 to demonstrate the communication required to obtain a list of all lights within the home, and then to turn one particular light on.

5.7 PROTOTYPE USER INTERFACES

Sample applications were written to provide prototype interfaces for Homer. Firstly, an iPhone application was developed to offer control over the home. Secondly, an iPad application was developed to allow policies to be written. Thirdly, a web-based application was developed to also allow policies to be written. Each application is discussed in turn below.

5.7.1 *Homer for iPhone*

The iPhone application makes it possible to browse devices by location or type, as shown in Figure 5.22. The current state of a device can then be viewed, its history of past events is available, and the device can be asked to perform selected actions, as shown in Figure 5.23. A live Twitter feed is also available for all events within the home.

5.7.2 *Homer for iPad*

The iPad application is intended to be the main user interface for the home, meeting the design guidelines stated earlier in this chapter, as well as design philosophies discussed in Section 5.8. This prototype primarily focused on programming the home. The full design should offer monitoring and control over the home as well as the ability to view, write and edit policies. The choice of users, platform and design of the prototype are discussed in this section.

5.7.2.1 *Users*

There are four main sets of user groups that could be satisfied: the younger generation who are technically savvy, the middle generation who are technically competent, the healthy and active ageing generation who are technically capable, and the older generation who are, on the whole, much less technically capable. The healthy and active ageing population was chosen as the target user group. It is appreciated that this solution will therefore be more than usable by those in the younger and middle generations, but understood that the older and less able generation may be missing out. This was a conscious decision as I believe that it is

2 A developer is "approved" after requesting access to Homer from those in charge, and being granted permission. At this stage the developer will be given a private access key which must then be provided with all HTTP requests to Homer.

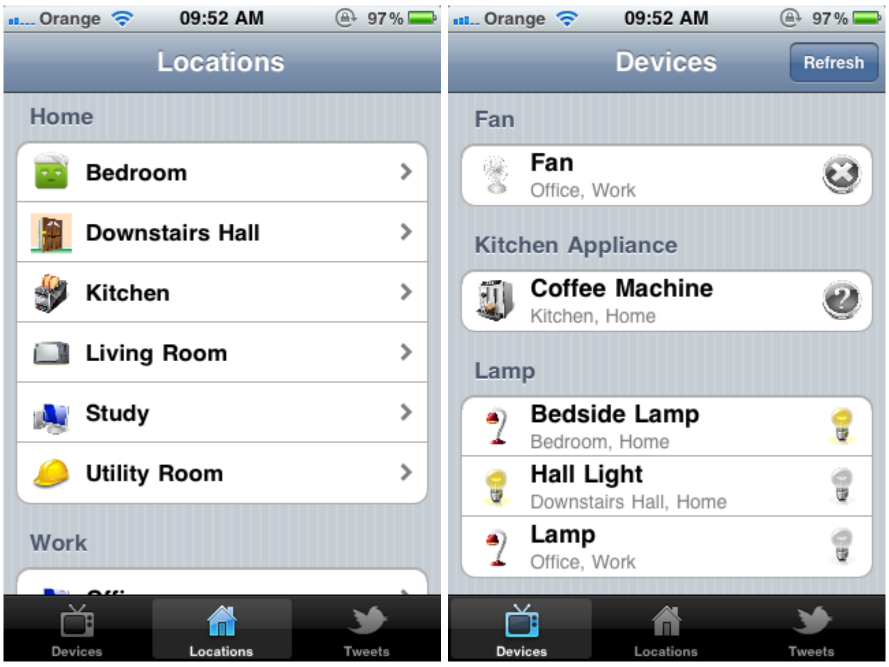
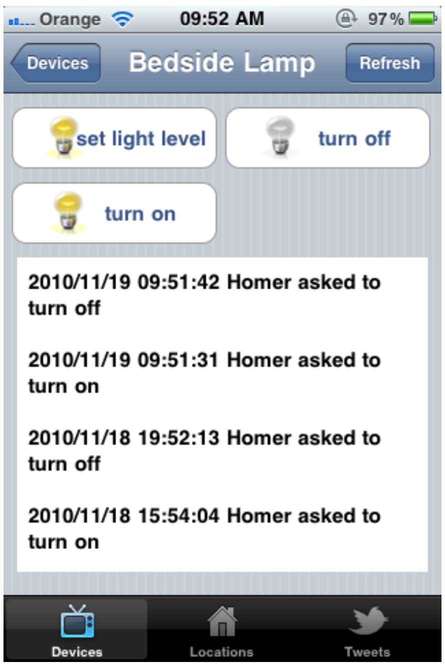


Figure 5.22: Two screenshots of the iPhone Prototype: Browse by Locations or Devices.



2010/11/19 09:51:42 Homer asked to turn off

2010/11/19 09:51:31 Homer asked to turn on

2010/11/18 19:52:13 Homer asked to turn off

2010/11/18 15:54:04 Homer asked to turn on

Figure 5.23: Screenshot of the iPhone Prototype: View a Device.

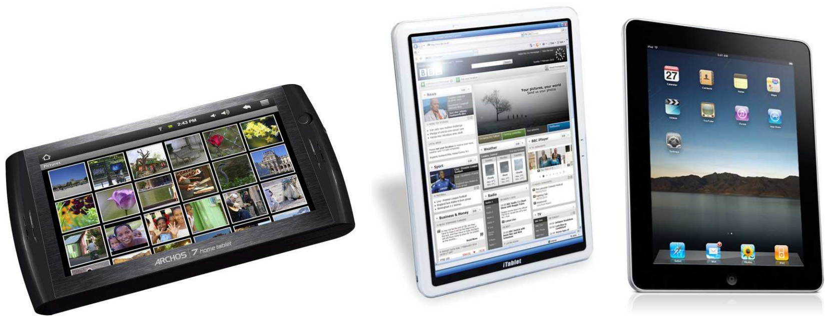

(a) The Archos 7 Home Tablet. (b) The X2 iTablet. (c) The Apple iPad.

Figure 5.24: Possible Devices.

too late to aim for the older generation as by the time research meets the commercial world it will be time to cater for the next generation. This will massively reduce the current focus on accessibility and assistive technologies and will change the focus to developing better designs and solutions.

5.7.2.2 *Platform*

From the user study discussed in Section 2.3, people showed a strong desire for touch control of their home [79]. When asked how likely they would be to use touch control in their home an impressive 61% answered "very likely" and 38% answered "likely". This is mirrored in both commercial and academic systems. The most successful home control systems use a touch panel interface, to name a few: Cortexa (`www.cortexa.com`), O2 Joggler (`yourfamily.o2.co.uk`), OmniQare (`www.omniqare.com`) and Oscar [97]. The latter, as stated in Section 5.4.1, showed that users liked the touch-screen tablet PC interface as it is similar in form and function to a remote control. Due to a mixture of these reasons it was decided to design and develop a system for a touch-screen interface. The options available on the market at this time (June 2010) were:

- a touch-screen monitor attached to a PC
- a small hand-held touch-screen device
- a tablet PC.

The first on the list, a touch-screen monitor attached to a PC, was ruled out due to the desire for a portable device which could be carried anywhere within the home. The second, a hand-held touch-screen device, was ruled out due to lack of screen real-estate that could be offered by the average hand-held touch-screen device. This left tablet PCs, which unfortunately in 2010 (the time of development) there was an extremely limited range of availing hardware on the market. The main manufacturers were Archos, X2 and Apple. Archos offered a 5- and 7-inch portable touch-screen device which ran Windows 7 (Figure 5.24a), X2 offered 10.2- and 12.1-inch portable touch-screen devices which also ran Windows 7 (Figure 5.24b) and finally Apple offered a 9.7-inch touch-screen device (iPad) (Figure 5.24c).

The three options were reviewed to gauge which would be the most sensible within the context of the home and for the selected user group. All three had wireless capabilities, were able to run a full web browser, and could offer multimedia features. The hardware of the Archos and iTablet let them down, with the Archos devices using a resistive touch screen which was not very responsive and the iTablet being both thick and heavy. The iPad on the other hand is extremely slick, lightweight and beautiful to hold. However, the iPad has its own limitations: the lack of Flash support and the restrictive nature of iOS development. The Archos and iTablet, both running on Windows 7, allowed any windows program to run and make full use of the hardware capabilities of the device. Appreciating both the advantages and disadvantages of each of the devices, the iPad was chosen as the most sensible device for the home and target user group. The iPad, although still new at the time, was judged to be extremely simple and easy to use as it runs on the same operating system as the extremely popular iPhone and iPod Touch. This made it perfect for the target user group who are perhaps not confident with technology.

The major home automation companies such as Crestron and Control4 had developed applications for the iPad only weeks after the iPad was released. Many of these companies are predicting that the iPad could help give home automation the next major breakthrough by making the technology attractive, readily available and usable by a very wide audience.

5.7.2.3 *Design*

To demonstrate the possibility of writing policies for Homer on a tablet PC, an iPad application was developed which featured a prototype interface for writing, editing and uploading a policy.

The screenshot in Figure 5.25 shows a policy for turning off the hall light when leaving the house. The bottom part of the screen allows users to select clauses, with multiple ways of selecting the same clause through the use of perspectives. The top part of the screen is dedicated to the policy, which aims to read naturally in English. Each clause within the policy can be dragged around to change order, selected and altered using the clause selection at the bottom of the screen, or removed from the policy altogether.

The purpose of this application was to show one example of a policy editor, and how the designer of such an application is free to make any decisions they desire. The only requirement is that the resulting policy sent to Homer conforms to the JSON format.

5.7.3 *Homeric Wizard*

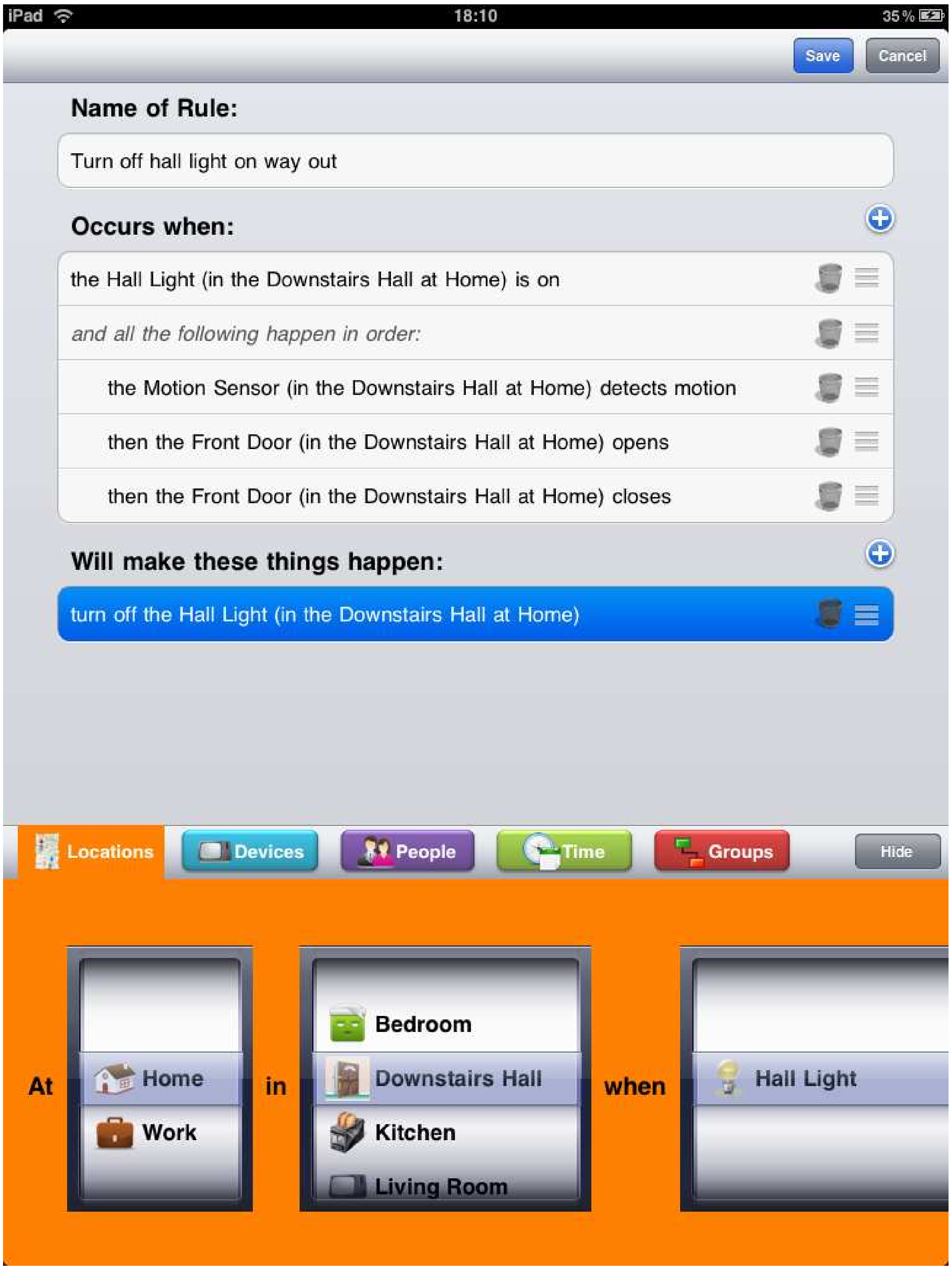
A prototype web-based application was developed to allow Homeric rules to be written for the home by both technical and non-technical people. Unlike Homer for the iPhone and iPad, the Homeric Wizard tool is not connected to Homer. This is because the wizard was developed to allow evaluations of Homeric, perspectives and end user programming design techniques. It was therefore desirable to have an unrestricted collection of devices and possible triggers, conditions and actions to ensure that the end user has a large library of choices, so as to not interfere with the main goal of the tool.

The Homeric Wizard consists of one main screen which has three sections. The first is a library of components, their devices and respective triggers, conditions and actions. There is then a *when* panel section and a *do* panel section which allow the triggers, conditions and actions from the library to be dragged-and-dropped within. Terms within the panels can be rearranged or deleted by simply dragging the terms individually to their desired location. An example of the three sections, with overlay descriptive text, can be seen in Figure 5.26.

The Homeric Wizard was designed to use a hybrid of visual and (restricted) natural language programming techniques to offer the user a familiar and easily understood tool. The full description of the tool can be found in [80]. An example of a simple policy written using the tool can be seen in Figure 5.27. The policy reads like a sentence: "*when* any of the following occur: the back door opens or the front door opens, *do* turn on the lights in the kitchen and turn on the table lamp in the hall". In order to formulate this sentence the correct "building blocks" needed to be dragged into place, hence the inclusion of visual programming. This is required to ease the computation involved in parsing the policy greatly, as well as providing inspiration and reminders to the end user about the functionality available to them.

Perspectives were integrated into the Homeric Wizard to ease browsing and locating specific devices in the library. The tool is also personalisable by allowing names to be entered, which are seamlessly integrated into the various locations (e.g. **Alice's** Bedroom), devices (e.g. **Alice's** Mobile Phone), events (e.g. **Alice's** Birthday) and parameters (e.g. Send text to **Alice**). The wizard also supports three difficulty levels to allow the user to write policies at a level most comfortable to them; for more details see Section 4.8.1 or [80].

The evaluation of the Homeric Wizard is discussed in Section 5.9.



Name of Rule:

Turn off hall light on way out

Occurs when:

the Hall Light (in the Downstairs Hall at Home) is on

and all the following happen in order:

the Motion Sensor (in the Downstairs Hall at Home) detects motion

then the Front Door (in the Downstairs Hall at Home) opens

then the Front Door (in the Downstairs Hall at Home) closes

Will make these things happen:

turn off the Hall Light (in the Downstairs Hall at Home)

Figure 5.25: Screenshot of iPad Prototype: Writing a Policy.

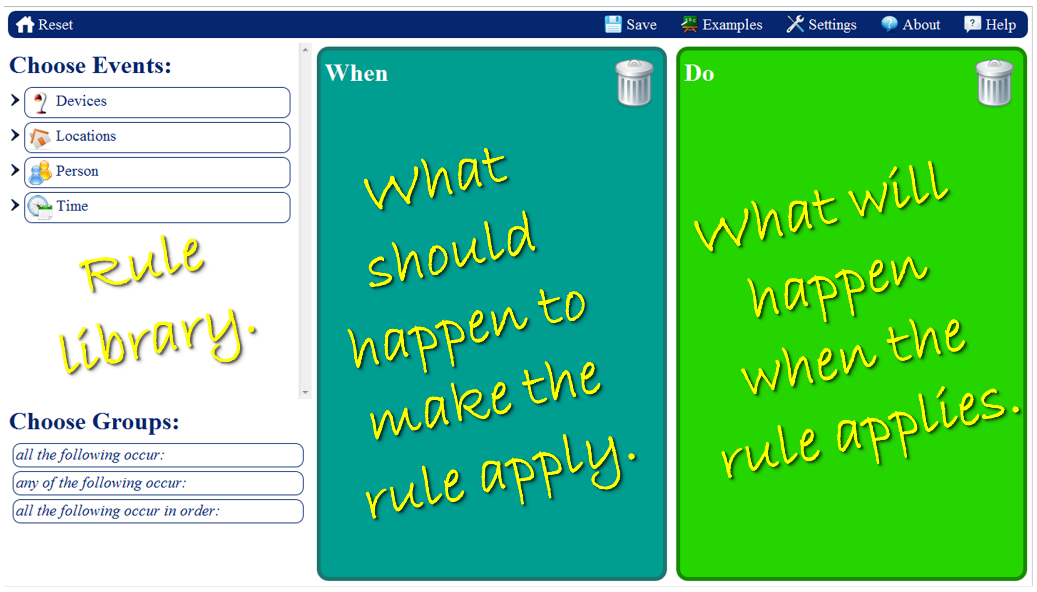
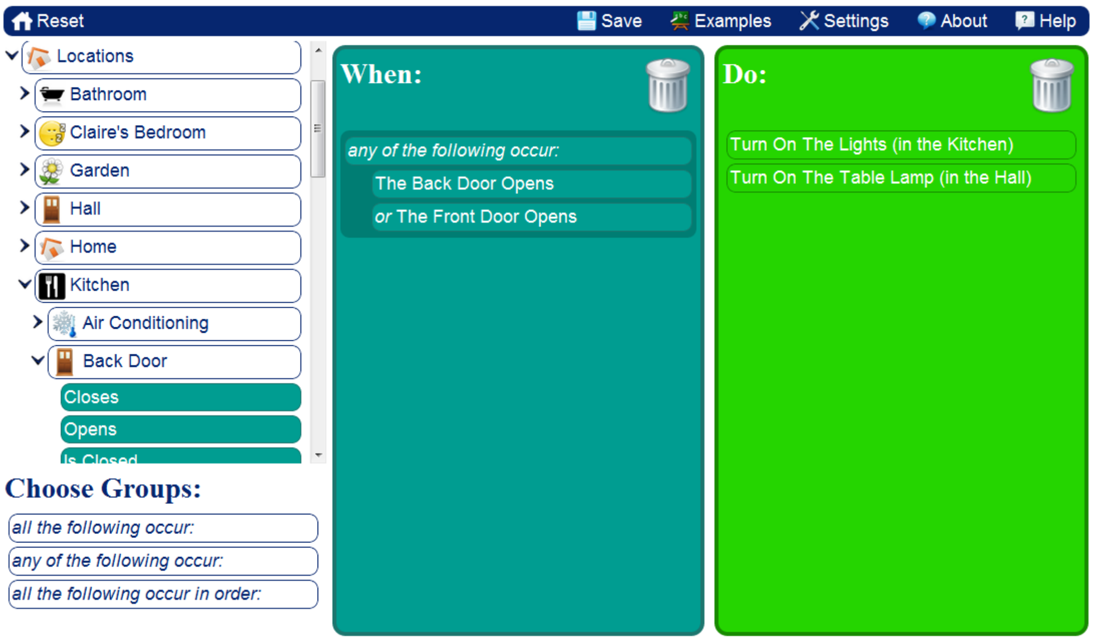

Reset
Save
Examples
Settings
About
Help
Choose Events:
Devices
Locations
Person
Time
Rule library.
Choose Groups:
all the following occur:
any of the following occur:
all the following occur in order:
When
What should happen to make the rule apply.
Do
What will happen when the rule applies.

Figure 5.26: The Homeric Wizard Application (with Overlay Text)



any of the following occur:
The Back Door Opens
or The Front Door Opens

Turn On The Lights (in the Kitchen)
Turn On The Table Lamp (in the Hall)

Figure 5.27: The Homeric Wizard Application (with Example Policy)

5.8 Conceptual Design of a Home Interface

Having thoroughly researched the fields of end user programming in conjunction with home automation, the following section describes design ideas and philosophies for a conceptual user interface which can monitor, control and automate the home. This is termed a "home management" interface throughout the following section. Sample mock-up screenshots are provided for illustration of a house called "Craigengall", with two residents "Mum" and "Dad". This proposed interface makes use of the research performed to produce a hybrid of successful techniques and design decisions, whilst also remaining true to the requirements outlined in Section 5.3. So far, the interface has been designed and mocked-up, but is not yet implemented.

5.8.1 *Touch Control*

Through both the literature reviews and user studies performed through this research, touch control is clearly the most desired interaction means for a home system. Therefore, any home management interface should be designed for this means, most likely to be used on a lightweight hand-held tablet device.

5.8.2 *Vocabulary*

Instead of using computer science terminology, user friendly terms should be used whenever possible to help simplify the system for the user. For this reason, "rule" should replace "policy" and "programming" at all times, as it is a less technical and more familiar word.

5.8.3 *Combine Control and Monitoring with Rules*

As noted in the user study in Section 2.3.2 generally end-users feel comfortable controlling their home, though they tend to fear the prospect of programming it. They desire the ability to be able to have such rules, and that these rules be easily modified and adapted. However, they do not like the idea of having to program them. Given the current choice of methods available to users, this is unsurprising. I hypothesised that if programming aspects were blended into the general control and monitoring aspects of a home user interface, the user would be less likely to notice or fear the programming parts. The user would also not necessarily have to make a conscious decision to program the home by opening a new piece of software or accessing the "advanced control" parts of their current home interface. Instead, the user should be able to easily apply rules to any object which they may currently be controlling/monitoring.

Kodu is one of the few success stories of end-user programming. The approach also tightly integrates the programming aspects with all elements of the game. Although its philosophy is still relatively logical and simplistic (simply *when-do*), it has been proven to be easy to use and understand. Their idea is that any object can be programmed, so in the context of the home everything within the home could be programmed. Instead of thinking about programming the home, the user would be programming things within the home directly. People themselves also play a major role and could have rules attached to them. I believe that by allowing people to apply rules to objects directly, people are forced to think more concretely, deciding what they are programming and what they want to happen. This is what they need to be able to do, no matter what interface they are using to communicate this. The Kodu idea allows people to tell the system exactly what they want directly.

By tightly integrating the programming aspects with the rest of the home system, the process and time involved in defining new rules for the home can be simplified.

5.8.4 *Customisation*

There exist users who prefer default settings and "vanilla" installations, so it is crucial that the home management system supports minimal setup and customisation, as well as sensible default settings.

However, there are many users who like to customise their products to suit their daily needs and requirements. Attempting to satisfy this user group will increase product satisfaction and suitability, and decrease user frustration. An example solution is a fully customisable widget-style panel, where the user can select which widgets to display. The right-hand side panel in the mock-up screenshots of Figures 5.28-5.33 show five sample widgets: current

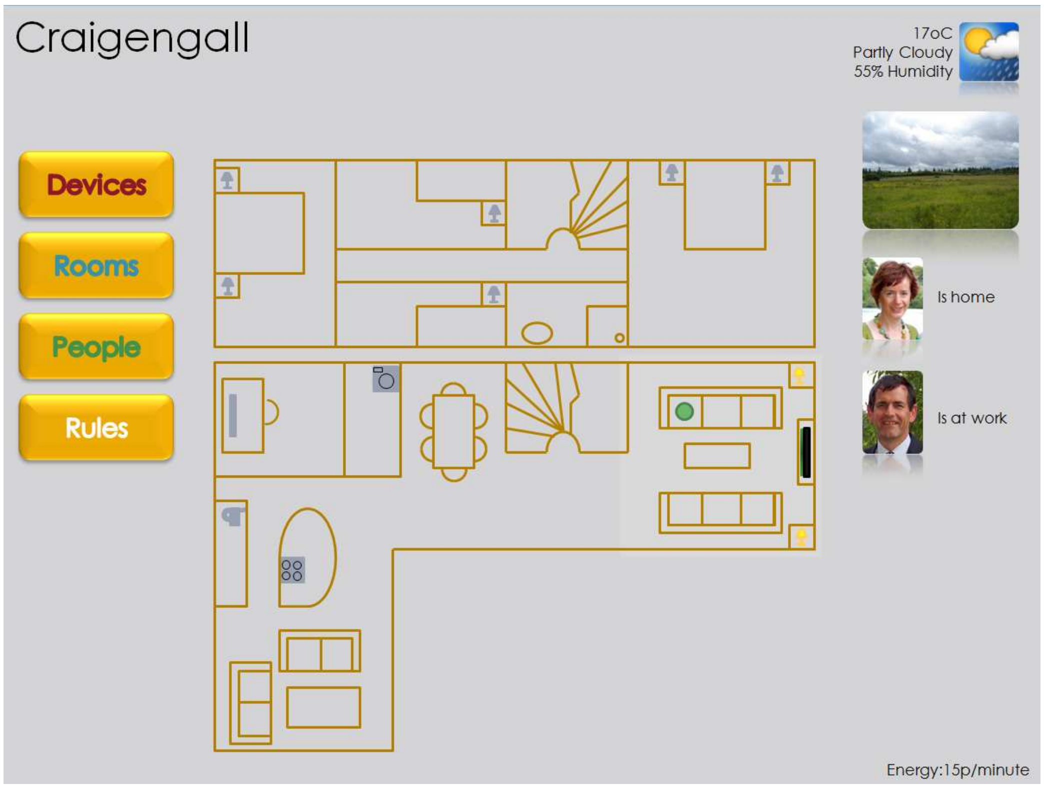


Figure 5.28: Mock-up: Home Screen.

weather, current view of security web-cam, status of residents and current energy usage. This panel should be shown on all pages of the interface, providing consistency and personalised relevant information throughout. Additionally, the widgets themselves should act as short-cuts, allowing users to quickly and easily navigate to elements of the home system which are of most interest to them.

5.8.5 *Home Page*

The home page by nature should be an instant gateway to the home. During a prototype evaluation a user said that the most important aspect of any home system would be the status of their home:

> *"Home is the most valuable thing you'll ever buy in your life. You want to know it's OK."*

The main central page of a home system should offer an instant overview of the home yet still allow quick navigation to key features. The most popular home screen amongst existing home automation companies is to display a plan view of the house. However, this static plan does not indicate the live status of the home, and is therefore limited and, at times, misleading. Instead, a live and interactive plan of the home should be shown, illustrated in the mock-up in Figure 5.28. The plan should show the user exactly what is on and off and where people are if such information is available. Every element of the map should be interactive: clicking on a device should lead to the device's page where it can be controlled, lights can be toggled on and off, and rooms will lead to a dedicated page just for that room (location).

5.8.6 *Navigation*

It is crucial for any end user application to be easily navigated, and a home system designed for a whole range of possible users is no exception. The general Human Computer Interaction (HCI) principles for navigation are relevant, however attention should be made to ensure that the home screen is always easily accessible through one button from anywhere, the user always knows where they are, and finally no menu structure should be too deep (for example, a maximum of three levels deep). These rules help to ease interaction even if this application was left in an intermediate state. For example, one member of the household may be in the process of doing something with the application and have to stop for unrelated reasons. The next person to interact with the application needs to easily know where they are, how to get to the home screen, and also to have the option of pressing a back button to exit the current task.

5.8.7 *Scenarios*

Sensor fusion is the notion of taking a collection of triggers and translating them to some higher-level event. An example of this could be "front door opens, then hall movement detected, then front door closed", which could be fused to say "someone walked into house".

The dictionary defines "scenario" as:

> *A sketch, outline, or description of an imagined situation or sequence of events.*
>
> Oxford English Dictionary.

This definition confirms that the word "scenarios" could be used as a simple and less technical word for the *when* clause of a policy. Further examples of these scenarios within Homer are "when Mum gets home from work", "when it is bedtime" or "when no one is in the kitchen". The intention is to dissolve the gap between control and programming for the user, to soften the concept of rules and to allow the user to write rules using statements that make sense to them. For example, it is far easier to define how Mum gets home from work once, then in rules simply say "*when* Mum gets home from work", instead of specifying the lower level events in each rule.

Each scenario would need to be defined in some way. This could be done through the same interface as is used for defining a *when* clause. The user could save any *when* clause as a scenario, allowing it to be reused within other rules.

The user should be able to view all events that will occur when specific scenarios occur, having the option to activate and deactivate each of these events as well as edit, delete and add new ones.

5.8.8 *Perspectives*

Users typically think about problems in different ways, so there should be differing means of programming logically equivalent rules for the home. For example: "turn on the coffee machine when bed becomes empty" and "when I get up turn on the coffee machine". This also applies to viewing rules. For this reason rules should be tightly integrated with the home management user interface.

User surveys described in 2.3 showed that the four most commonly used perspectives by users when referring to elements within the home are: people, locations, devices and time. Each such perspective should offer elements that can be monitored, controlled and programmed easily and consistently. By doing this the user would be able to add rules to whatever element of their house they desire, accessed in multiple ways.

Each of the four perspectives is now discussed in turn.

5.8.8.1 *People*

The occupiers of the house should feature predominantly within the home management interface. A person-centered menu, such as the mocked up one in Figure 5.29, should provide instant access to all means of communications currently available and known for that person. Each person within the household should also have a dedicated screen (such as that shown in Figure 5.30) to offer further functionality. The person page should display any rules involving that person, a means of contacting them, any personal preferences, and the ability to create a new scenario or rule involving them. This allows individuals to view any rules which they are involved with from a personal point-of-view, being able to see what happens when they, for example, "get out of bed", "go to work", or "get home from work". This would allow for easy and quick creation and management of self-centred rules within the home.

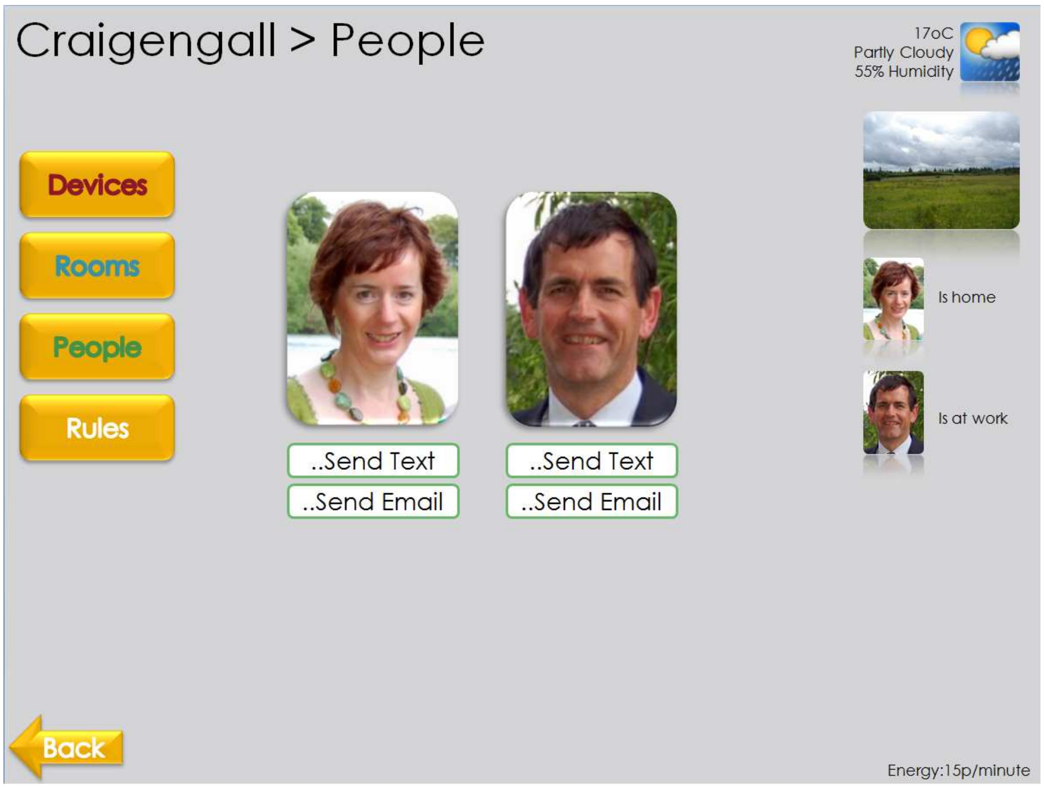


Figure 5.29: Mock-up: People Screen.

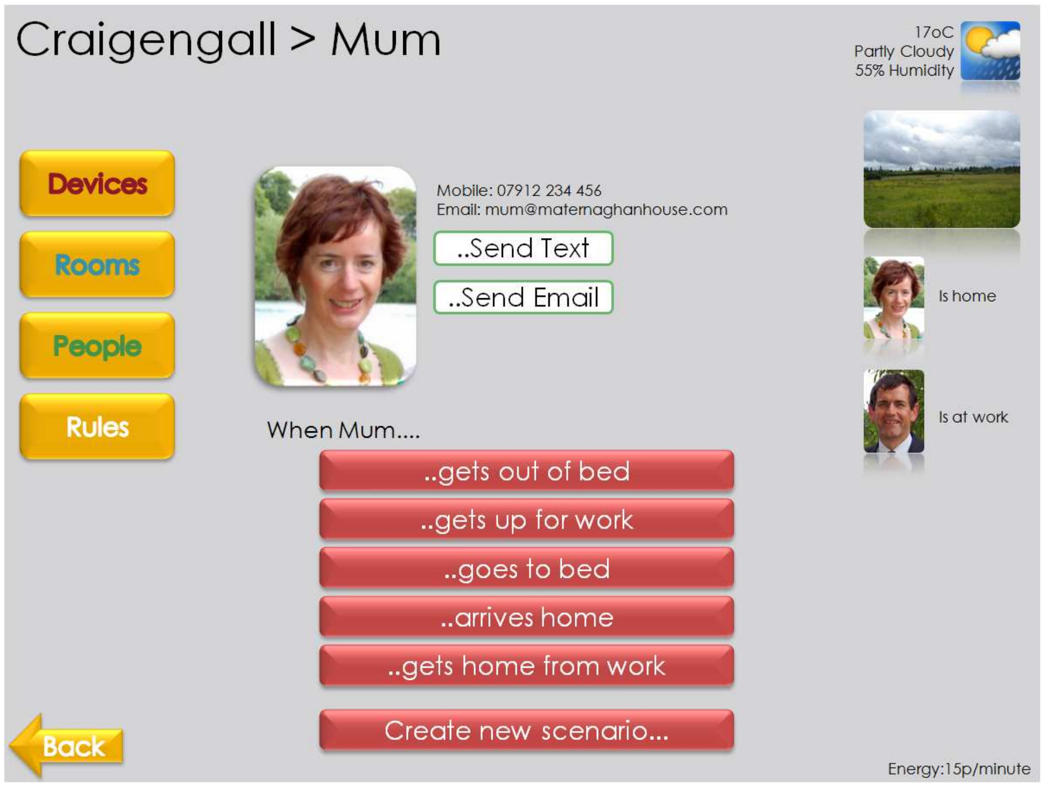


Figure 5.30: Mock-up: Person Screen.

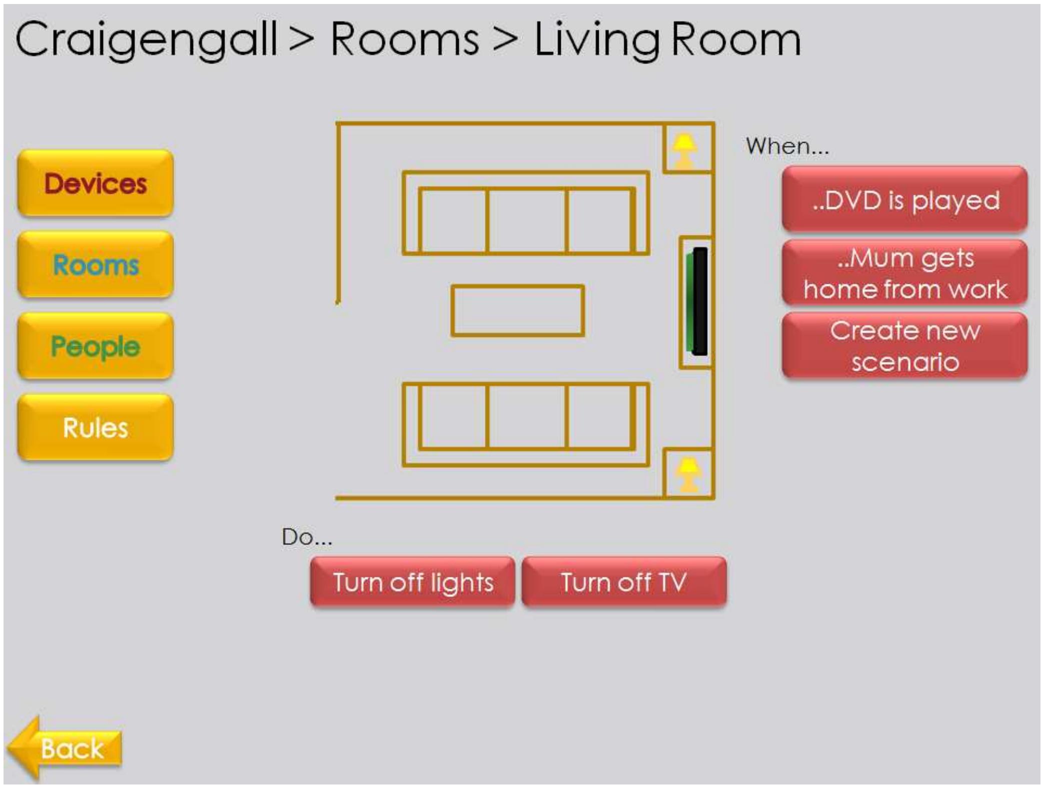


Figure 5.31: Mock-up: Location Screen.

5.8.8.2 *Locations*

Locations, like people, should be able to be browsed, monitored, controlled and programmed. An example layout is shown in Figure 5.31. A live plan of the room should be shown in some form, where the status of the devices within the location would be shown visually and users could interact with the devices to control them. All possible actions which can be performed within the location at that point in time should be shown.

Similarly, all rules or scenarios which involve the location, or devices within it, should be easily accessible. By doing so the separation between rules and the home would be reduced, and the process to manage, edit and write rules would be made more accessible. As an example, if Mum (in the mock system) were to write a rule from her personal screen (such as that shown in Figure 5.30) "*when* I get home from work *do* turn on the television", it should be associated with "Mum", "living room" and "television" screens. Despite how the rule is written and the various elements involved, the rule should be made known and accessible from all relevant elements.

5.8.8.3 *Devices*

Again, similar to people and locations, devices should also be capable of being browsed, monitored, controlled and programmed. The page should offer similar functionality as the locations page, offering easy and immediate access to control the device and manage any rules which involve it.

Figure 5.32 shows an example television page. In this case, the television is currently on and so the user is able to change the volume and channel, turn off the television, or setup a program to be recorded. Again, the rule "when mum gets home from work turn on the television" is shown as it involves the television.

5.8.8.4 *Time*

Any home system should support time as a condition for rules. The user should be able to choose at what times they would like something to occur, and hence create a scenario and/or

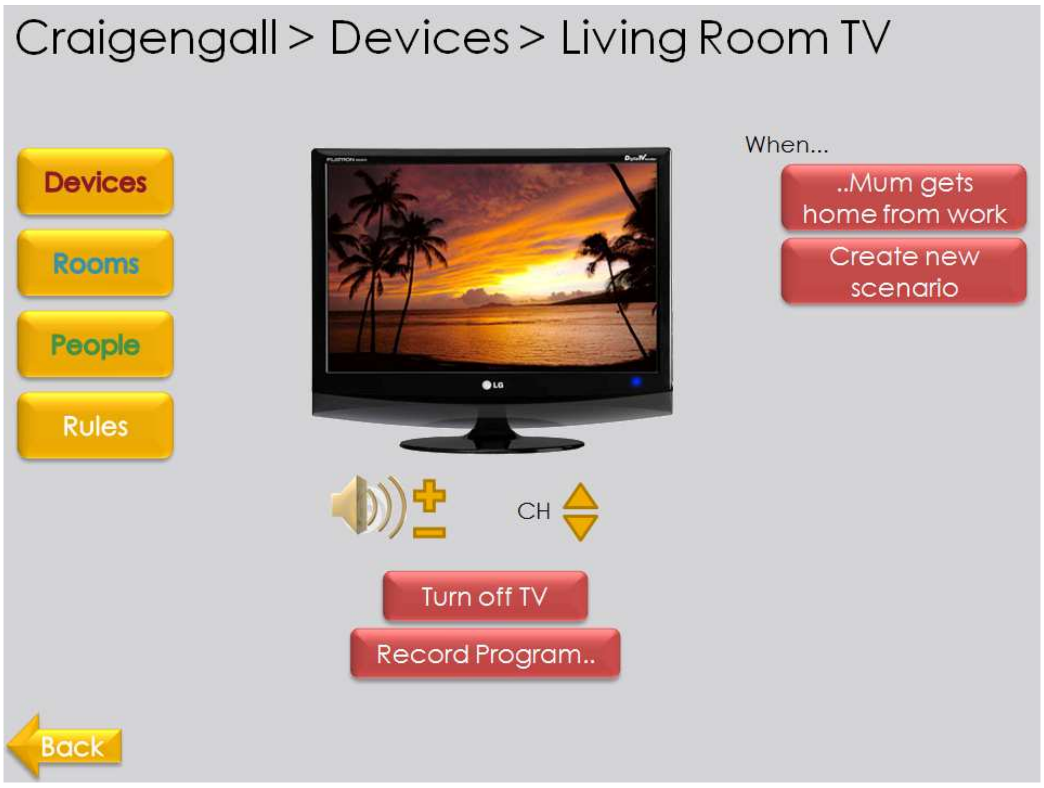


Figure 5.32: Mock-up: Device Screen.

rule. Time could mean any notion of hour, day, week, month, year, special dates or events (such as spring, sunrise or sunset). Against any category users should be able to specify any events that they would like to occur (resulting in rules). Figure 5.33 shows an example time screen with common and specific time conditions which users can easily select to view events that take place when these occur, as well as add and edit events.

5.8.9 *Rules*

The reality of writing rules for the home can be challenging for users, dependent on their technical ability. The following sections describe means that could be implemented to help ease the process.

5.8.9.1 *Feature Control*

It is crucial that advanced language features be hidden from less technical or experienced users. Advanced features will only intimidate these users. Suggested levels for Homeric, Homer's policy language 4.5, for differing capabilities are:

- **Simple** ("I'm a little scared."): Offers basic capabilities such that triggers and conditions can only be combined with *and*, and actions are a simple list. An example would be "*when* trigger1 *and* condition1 *do* action1 *and* action2".
- **Medium** ("I'd like to give it a go."): Adds the capability to use the *or* and *then* operators within the *when* clause. An example would be "*when* trigger1 *then* (condition1 *or* trigger2) *do* action1".
- **Advanced** ("Let me do everything."): Support for time intervals on events within the *when* clause is added, as well as conditional actions. An example would be "*when* trigger1 *and* trigger2 occur within 5 minutes *do* action1 *and if* condition1 *do* action2".

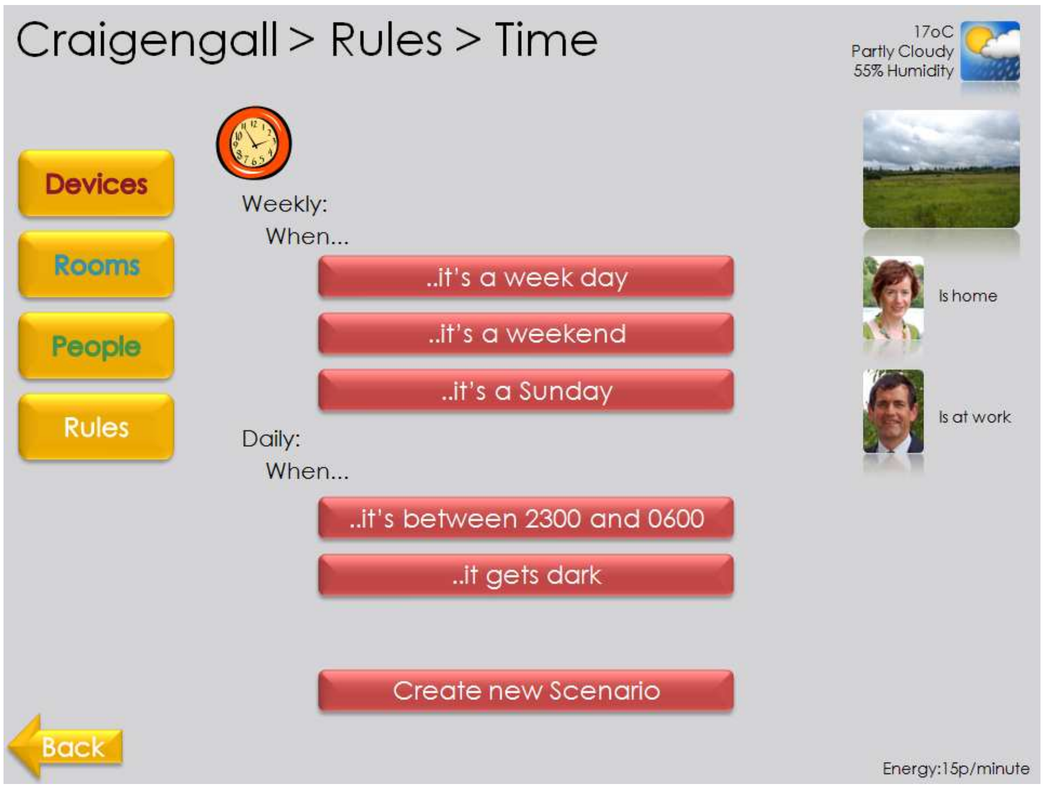


Figure 5.33: Mock-up: Time Screen.

5.8.9.2 *Intelligent Automated Tutorials*

Users could be taught the language features of the home system, if they so desire. This help could be provided in the form of either walk-through tutorials or intelligent automated tutorials. For example, when the system detects that the user has written a rule (or rules) which could be more simply represented using a different language feature, it could introduce and teach the user about such a language feature. This would help to introduce features only as they become relevant, rather than overloading the user at one point in time.

5.8.9.3 *Templates*

Template rules could be used for common automated tasks within the home. The user could then easily fill in these templates to suit their needs. This would help users to write rules and expose them to the language features available, as well as offer a source of inspiration and encouragement for using rules within the home.

5.8.9.4 *Library*

A library of rules could be made available publicly to provide inspiration and to help users when automating their home. Users could make use of and contribute their own rules to this global store, which would help users to share and discover useful household rules.

5.9 CASE STUDY

Having designed the notion of perspectives and explored various end user programming techniques, an evaluation was designed and carried out to analyse their success. 71 people participated in the evaluation, of varying ages and technical abilities. Full details of the evaluation can be found in [80].

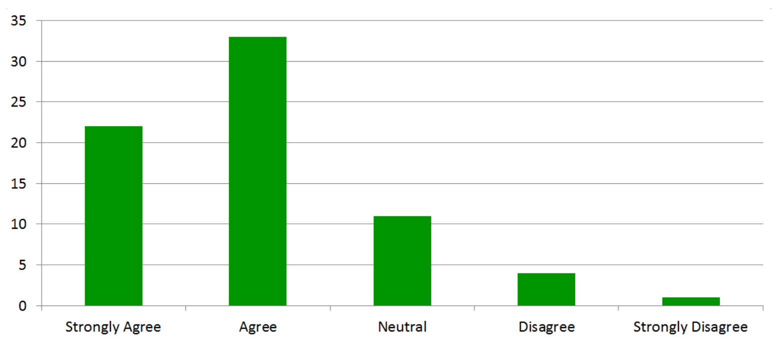


Figure 5.34: Counts for "Using Perspectives Made it Easier to Write Rules"

5.9.1 *Overview*

In order to evaluate perspectives and end user programming techniques a custom tool was required, and so the Homeric Wizard was used. The user needed to write Homeric policies for this evaluation. For this reason, this case study is paired with that of the policy system. The policy system study was interested in the success of Homeric, which required a tool to formulate policies, whereas this evaluation was interested in the success of the tools, which required the user to write Homeric. A full overview of the evaluation can be found in Section 4.8.1.

5.9.2 *Results*

Within this evaluation two main aspects of programming the home were analysed: firstly, the success of perspectives, and secondly the success of visual and (restricted) natural language end user programming techniques. Both are discussed in turn.

5.9.2.1 *Perspectives*

Perspectives proved very successful within the evaluation. Users were introduced to the notion of perspectives through a simple graphical tutorial. They were then asked to write four "rules", two of which transcribed a given goal and the other two could be anything. For each task, the Homeric Wizard would use perspectives for one part and not for the other.

It was proven that, on average, participants made use of more than one perspective per policy, showing that indeed participants made use of perspectives.

Many assumptions had been made originally, assuming that there would be correlation between the demographics of the user and the perspectives that they used. However, no such evidence could be observed in the data. This shows that one cannot make assumptions about the type of user of the application, and that perspectives therefore are even more useful as they can be offered to everyone, and the user can make use of those that they prefer. It was found that devices and time were the most common of perspectives.

At the end of the evaluation, participants were asked if they found perspectives made it easier to write rules in the Homeric Wizard. Figure 5.34 visually shows the very positive results, confirming that people found perspectives helpful. Feedback was left by many participants, some of whom took the time to express how much they liked perspectives. Some quotes include "perspectives make it easier" and "Perspectives = Good".

5.9.2.2 *Visual and Natural Language Programming Techniques*

The Homeric Wizard tool proved to be a success, with a wide range of users being able to use the same tool to write policies for their home at a level that they preferred.

The participants were asked to describe how challenging they found writing the "rules" within this evaluation. The graph in Figure 5.35 shows this data visually, with only 18.3% of participants claiming that writing the rules was challenging. This is extremely positive, as such a wide array of people, with very little training, were able to formulate rules for perhaps the first time and not find the task challenging.

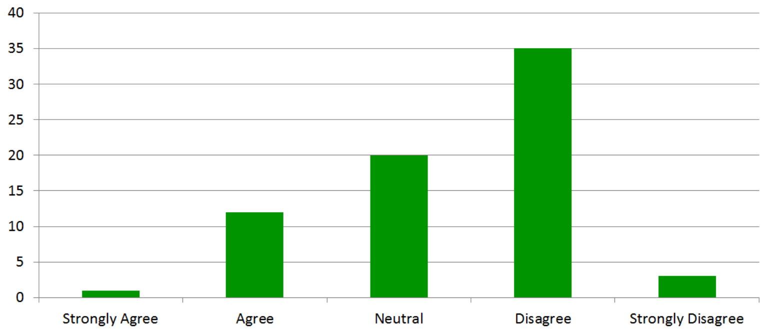
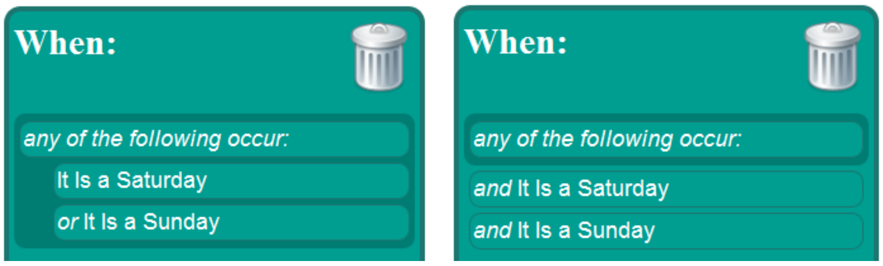


Figure 5.35: Counts for "The Evaluation was Challenging"



(a) Correct (b) Incorrect

Figure 5.36: Nesting Terms in the Homeric Wizard

Participants were also asked if they would be likely to ever program their home with a tool like the Homeric Wizard. 60 out of the total 71 participants responded that they would be likely or very likely. Unfortunately, it is unknown if those who chose unlikely or very unlikely were voting against ever wanting to program their home, or if they were voting against the Homeric Wizard itself. Regardless, such a high number of people voting in favour is extremely positive and further shows the success of the Homeric Wizard.

The only problem detected with the user interface of the Homeric Wizard was when nesting terms within groupings (*and*, *or* and *then* nodes). A fair number of participants did not appreciate that terms within a group should be visually indented and coloured to appear as children within a group. Figure 5.36 shows an example of correct and incorrect nesting. This problem would need addressed in future iterations of the Homeric Wizard.

Many positive comments were received regarding the Homeric Wizard, including praise for its design, simplicity and ease of use. One participant said: "Use of plain language and icons made it simple to set rules".

5.9.3 *Summary*

The key findings from this evaluation include the success of perspectives and also using a combination of natural language and visual programming techniques. Together, these help offer an interface for programming the home to a wide array of ages and technical abilities.

5.10 CONCLUSIONS

5.10.1 *Summary*

This chapter has presented an extensive background review of existing state of the art for end user programming in regard to the home. From this, design guidelines and ideas have been produced, presented and evaluated.

Through the requirements and consideration of the state of the art it was possible to draw a list of observations and conclusions to help produce a short and concise collection of design guidelines. These design guidelines express the key requirements for a successful home automation interface.

A key observation made throughout the research performed within this chapter was the simple notion that people typically address the range of devices in their home through varying means. This has been termed perspectives, and has proven key in allowing a wide range of possible end users to interact with the same home system in a way that is comfortable and natural to them.

To aid evaluation of various design principles and ideas stated within the chapter, a tool was designed and developed to support writing policies for the home. This effectively allows end users to program their home, where these end users can vary widely in age, experience and technical ability. Natural language and visual programming techniques were coupled to form a new style of end user programming for the home. This was embedded within the Homeric Wizard.

A case study involving 71 individuals was undertaken to evaluate the notion of perspectives and the Homeric Wizard. The results from the evaluation were extremely positive, with every participant being able to make use of the tool to express policies for the home, despite their potential lack of experience or technical ability. Perspectives proved to be liked by most participants, and they certainly eased the task for many. Due to the success of the policies formulated and the positive feedback, it can be concluded that natural language and visual programming techniques offer a possible solution for programming the home.

5.10.2 *Review*

Section 5.3 stated a number of requirements for the user interaction aspect of Homer. These requirements will now be revisited to assess to what degree they have been met.

- *"**Accommodating**: Cater for a very wide range of users."*

  All through the research and design phases of this work it can be seen that the interactivity between user and system has always focused on simplicity and suitability for users ranging in both age and technical ability. The design guidelines stated in Section 5.5 reflect this. Having designed the Homeric Wizard for this wide range of users, an evaluation was conducted to review the interface. The user evaluation, presented in the case study in Section 5.9, involved 71 participants ranging in age from under 20 to over 60 and from technically poor to technically expert. Every participant was able to successful understand, translate and write policies using the Homeric Wizard. This requirement has therefore been met.

- *"**Multi-Devices**: A range of devices and platforms should be available to interact with the home system."*

  Through the design and implementation of the Homer Web Server, presented in Section 5.6, any device or platform with a web connection is able to support a Homer interface. Three example interfaces have been made to demonstrate this: an iPhone application shown in Section 5.7.1, an iPad application shown in Section 5.7.2, and finally a web-based application called the Homeric Wizard shown in Section 5.7.3. This requirement has therefore been met.

- *"**Multi-Interfaces**: Touch interfaces should definitely be offered, but not necessarily excluding other modes of interface such as voice, remote control or gesture."* As explained for the previous requirement, the Homer Web Server supports any device or platform with a web connection. Currently the iPhone and iPad offer a touch interface to Homer, therefore meeting this requirement. At present there are no other modes of interactivity supported, however the successful work of projects such as Alfred, Accord and SiteView which make use of voice and tangible objects to program the home are acknowledged and feature within the future work section below.

- *"**Multi-Perspectives**: Allow users to be able to interact with the devices in the home from four different perspectives: location, device, personal and time."*

  As can be seen within the design guidelines stated in Section 5.5, and the notion of perspectives described in Section 5.8.8, multi-perspectives are fully integrated and supported by Homer. The case study in Section 5.9 evaluates the success of perspectives from a 71-participant study. This requirement has therefore been met.

- *"**Remote Control**: The ability to control the home remotely, such as at work or on the move."*

  Due to the design of the Homer Web Server, it is possible for any Homer interface to work from anywhere that has Internet connectivity. This requirement has therefore been met.

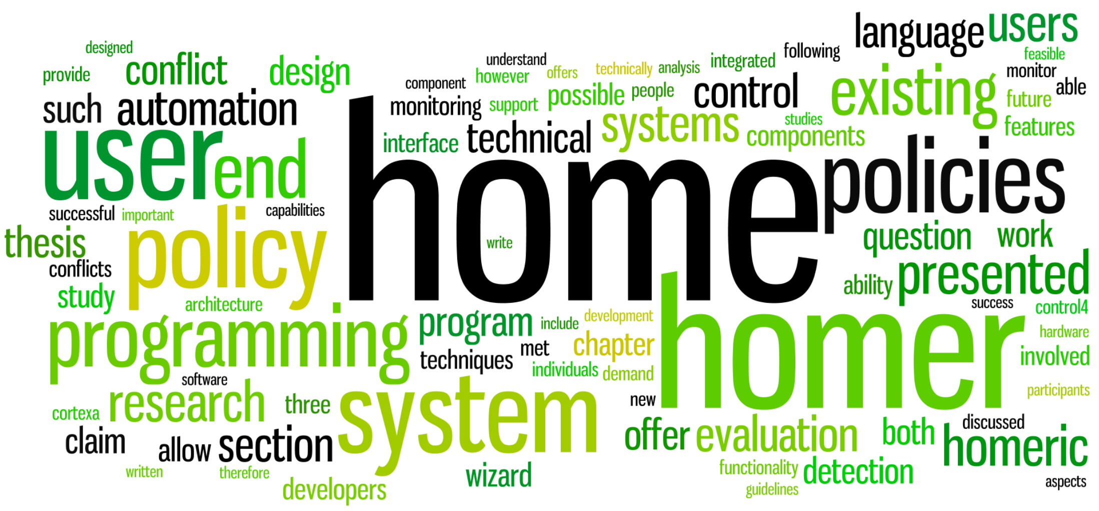
This chapter has presented guidelines and prototype applications to address the requirements stated in Section 5.3. These requirements have been individually assessed and it was shown that they have all been met within this chapter.

Although there exist many sophisticated home user interfaces from leading home automation and telecare systems such as Control4, Cortexa, Homeseer and OmniQare, no home system has integrated a means for non-technical individuals to program their home. Standalone research projects such as Alfred, CAMP, Girder, iCap and Oscar have tackled various aspects of end user programming challenges to varying degrees of applicability to the home, integration with a home system, and successful user evaluations. The Homeric Wizard presented in Section 5.7.3 is a unique example of an end user programming application tailored purely for the home and designed for both technical and non-technical individuals, with a successful evaluation involving 71 participants.

Part III

HOME RUN

6 CONCLUSION



This chapter will summarise the thesis, discussing its achievements, research contributions, applicability, limitations and potential future work.

6.1 THESIS SUMMARY

This thesis has presented Homer, a home automation system designed to allow end users to fully customise their home. This section will discuss the key aspects of the thesis.

A policy system was presented in Chapter 4 which allowed three research contributions to be made: custom home policy language, novel policy overlap detection, and advanced customisable conflict detection.

After an in-depth review of existing policy systems and their representations and implementations, a custom home language (entitled Homeric) was designed for the policy representation. The policy system was fully embedded within Homer to allow any new component to be automatically supported with policies.

A new technique for policy overlap detection was presented, with the use of the constraint satisfaction solver JaCoP, allowing any policy to be both validated (checked for logical correctness) and analysed for existing policies that could be applied at the same time. Additionally, policies can be checked statically for conflicts through the use of customisable environs, then any conflicts detected can be expressed to the user. To conclude this chapter, a case study was presented which evaluated the success, acceptability and appropriateness of Homeric. The Homer policy system is described in [78], [82], [83] and [84].

The policy system aimed to offer a possible means of allowing end users to customise and program their home at a higher level. To better understand the requirements three user studies were carried out. These studies helped gather the needs and requirements for a home system from the end user's point of view. The first user study gathered ideas and goals that people had for what their home could do for them. This showed the demand for automation features within the home, and often many desires were perfectly feasible with today's technology. A second, larger study was carried out to understand how these people would like to interact and control their home. Many positive results were unveiled from this study and documented in [79]. The overall trend was the desire to interact with the home through touch devices from anywhere (whether somewhere in the home, at work, or even whilst on holiday). Finally, a third study was conducted to learn about the challenges of programming the home. This study helped to design and shape the home language, observing the key findings that a *not* operator is effectively redundant and that often the difference between triggers and conditions is misunderstood.

A test-bed, introduced in Chapter 3, was designed and developed to allow the research contributions of the thesis to be grounded. After a thorough exploration of existing home systems and architectures a service-oriented design was chosen. OSGi was used to develop Homer; making use of its modular, loosely-coupled nature to result in a plug-and-play component architecture. The thesis discussed how third-party developers are able to provide a Homer Java wrapper for existing components, be it hardware or software based, to seamlessly integrate their functionality within Homer. Much of the work within this Chapter is presented in [78], [81], [82].

Additionally, end user programming research was undertaken to allow the policy work to be evaluated. Chapter 5 provided design guidelines for end user programming support, tailored for the home. This allowed the policy work of Chapter 4 to be demonstrated and evaluated. An extensive literature review was presented, analysing existing work for the four main techniques of end user programming: programming by demonstration, and visual, tangible and natural language programming. This review, partnered with the user studies performed earlier in the thesis, allowed design guidelines to be drawn up for the notable features of a home user interface. The notion of perspectives was also presented, integrated thoroughly with Homer and offering multiple ways of locating items within the home. Three sample Homer user interfaces were given: iPhone, iPad and web applications. The guidelines and ideas from this chapter were confirmed to be successful after an evaluation was conducted, given at the end of the chapter within a case study. Some of the principles and ideas from this chapter are presented in [78] and [83], and results from the evaluation in [80].

6.2 ACHIEVEMENTS

6.2.1 *Review*

Four objectives were stated in Section 1.3, which will now be reviewed to help gauge the success of the thesis.

- *Design a language for both technical and non-technical users which can allow the combining of the home functionality in an expressive, flexible and unambiguous form.*

  This objective involved designing a custom policy language for the home. This was key in allowing both technical and non-technical individuals to represent policies for their home using the same language. Homeric, the custom policy language for Homer, evolved from thorough research into existing policy systems (Section 4.4) and languages used for programming the home (Section 5.4), as well as from user feedback obtained from a dedicated user study (Section 2.3.3).

  The language was evaluated by 71 participants, ranging in technical ability and age, through the Homeric Wizard tool (described in Section 5.7.3). The evaluation, presented fully in [80] and summarised within Section 4.8, revealed that users were able to understand, transcribe and write Homeric policies with very little background introduction or tutorial. The average correctness of translated policies was 92.5%, for transcribed policies was 98.7%, and 90% of participants made no errors when writing policies.

- *Design advanced offline detection mechanisms. As there are high chances of conflicts between user written rules (policies) for the home these must be detected and reported to the user at the time of writing and saving a policy.*

  Existing policy conflict detection techniques analyse solely the resulting actions of any policy, whether at runtime or offline. Since offline conflict detection was chosen (for reasons, see Section 4.7), it was decided that all reported conflicts should be minimised, to help eliminate any unnecessary burden on the user when saving policies. To achieve this a novel technique was used to detect overlap between policies. If it was decided that two policies could potentially overlap, and therefore could take place at the same time, potential conflicts would be checked between these two policies. This eliminates unnecessary conflict detection analysis, and reduces the burden of filtering reported conflicts for the user. As a side-effect of this technique, policies are also validated to ensure that their *when* clause is feasible.

Offline conflict detection is handled by many existing policy systems such as ACCENT, KAoS and Ponder. However, these rely on ontologies and policy priorities, and are typically too complex for Homer. Instead, inspiration was drawn from the techniques of Nakamura [94] and Wilson [142] with enhancements to allow a richer set of conflict types and customisation.

An illustration was provided in Section 4.7.5 to show a range of policies being validated, analysed for overlaps, and then analysed for conflicts.

- *Offer end user programming techniques to expose the custom home language to both technical and non-technical users, since the policies must be expressible from a very high-level by an extremely wide range of users.*

  Chapter 5 presents the Homeric Policy Wizard which emerged from thorough background research and analysis of existing end user programming techniques, discussed in Section 5.4. The most successful and inspirational tool reported was CAMP [126], which made use of natural language words and phrases (expressed using a fridge magnet style) which could be pieced together freely to form rules. CAMP, however, allowed users to write ambiguous rules which were difficult for the system to parse. The Homeric Wizard dealt with this issue by adding restrictions to the policy definition interface.

  The Homeric Wizard was woven into an evaluation to assess the Homeric language itself, and also whether both technical and non-technical people can use the wizard to successfully write policies for the home. This evaluation revealed that all participants, regardless of age or technical experience, were able to successfully understand, transcribe and write policies for the home using the Homeric Wizard. The successful results from the evaluation include: over 50% of the participants found none of the tasks within the evaluation challenging, and over 85% of the participants agreed that they would like to use a tool like the Homeric Wizard to program their own home.

- *Design a flexible system to support the vast range of existing hardware and software for the home, as well as future devices through third-party support. The system must allow a means of flexibly combining the functionality of these devices at a higher level.*

  Homer is a component-based platform that allows developers to quickly and easily write components for the home, which become seamlessly integrated into the system. This is presented in Chapter 3, forming a test-bed for the following objectives.

6.2.2 *Contributions*

The following three research contributions have been presented:

- **Policy Language**: A policy language, named Homeric, is presented which is designed and tailored specifically for the home. It allows policies to be expressed which can combine the functionality of devices and services in the home in flexible and unambiguous ways. A user evaluation was performed to confirm the language's suitability and acceptability for the home, revealing the success of the language with both technical and non-technical users.

- **Policy Overlap Detection**: The Homer policy system provides a novel technique to validate a policy and examine if it overlaps with existing policies. This was achieved using constraint satisfaction tools and demonstrates a novel concept of overlap detection within the policy domain.

- **Policy Conflict Detection**: The state of the art of policy conflict detection has been advanced to allow Homer to detect conflicts using customisable environment variable information. An illustration was presented to demonstrate and validate both the overlap and conflict detection work.

6.3 APPLICABILITY ON A WIDER SCALE

Homer, although designed for the home setting, could readily be applied to other policy-ready domains (those which lend themselves to functionality of triggers, conditions and actions).

Homeric, the new policy language presented in this thesis, could be readily applied to many other domains. Sample policies which utilise Homeric for the most popular existing policy domains include:

- Access Control: "*when* (it is a weekday *and* (time is before 8AM *or* time is after 5PM)) *or* it is a weekend *do* lock all access doors *and* turn on employee entrance card activation machines"
- Call Control: "*when* call received from Alice to Bob *do* redirect Carol"
- Home Care: "*when* a fall is detected *then* no movement is detected for 3 minutes *do* alert a neighbour"
- Quality of Service: "*when* bandwidth usage is greater than 95% *do if* video enabled *do* lower video quality *else* lower audio quality"
- Sensor Networks: "*when* water level reaches 60CM *do* open the flood gates"

As can be seen, Homeric works with ease in these domains. Additionally, by offering unique language features such as blurred triggers and conditions, ordered terms and conditional actions Homeric has offered advanced policy representation and extended existing policy language functionality in these domains.

However, there is one primary function that Homeric does not currently support: prohibition policies. This form of policy is used to disallow certain behaviour, and is highly beneficial, sometimes crucial, within policy domains. Although deemed less important within the home, this poses a limitation within some domains. Examples of policies which could not be expressed include:

- Call Control: "Alice is not allowed to call Bob"
- Medical: "Only doctors are allowed to prescribe medicine"
- Sensor Network: "No wind turbines may be turned on when wind speeds are greater than 45MPH "

Despite this lack of prohibition policy support, Homer offers advanced and unique language features which are applicable in a wide range of policy domains.

The second aspect of this work explores the detection of overlap between policies. This was considered important within the home to reduce the number of irrelevant conflicts reported to the user.

The Homer overlap detection techniques work best in domains with a wide range of triggers and conditions which generally represent states or values. Some examples include:

- Access Control: files (opened/closed, locked/unlocked), doors (opens/is open, closes/is closed, locks/is locked)
- Home Care: curtains (opens/is open, closes/is closed), medicine (is taken/is not taken)
- Quality of Service: call (enables video/disables video, video enabled/video disabled)
- Sensor Networks: water level (rises/falls, is/is higher than/is lower than)

With such triggers and conditions the full benefits of Homer's blurred triggers and conditions can be realised. For example, "call enables video" (trigger) can be used interchangeably with "call has video enabled" (condition). This in turn can be detected by the Homer overlap algorithms and allows Homer to know if two policies simply cannot overlap. For example, "call enables video" cannot overlap with "call has video disabled".

The Homer overlap detection is designed to filter irrelevant policies from the list of potential conflicting policies. This was deemed important within home automation in order to reduce the burden on the user, however, this may not be as important in other domains. For example, if only trained professionals wrote policies for a given wind farm, it may be more beneficial to report all potential conflicts and not perform pre-filtering.

The Homer conflict detection relies on knowing how the different actions affect the environment. This can involve altering variables such as temperature or humidity, or consuming resources such as energy or water. Examples of such environmental data in popular policy domains include:

- Home Care: temperature, safety
- Quality of Service: bandwidth
- Sensor Networks: wind speed, water level

When such information is available then the Homer conflict detection will be able to successfully detect conflicting policies which affect the environment in undesirable ways. This will work with no additional heuristics and will work with no alterations to the existing algorithms.

However, policy domains which do not lend themselves to environmental modelling will be unable to make use of the Homer conflict detection. Such domains include access and call control, where actions typically do not affect the environment.

To conclude, the Homer system has been designed around the home and lends itself to similar domains such as home care and sensor networks. As can be seen, the individual contributions of the policy language, overlap detection and conflict detection each can be applied in other policy domains to varying degrees of success.

6.4 LIMITATIONS

There exists some limitation of Homer: the central hub nature of the underlying architecture, lack of sensor and actuator fusion, and no support for runtime conflict detection. Improvements and additional features that could enhance Homer and the work presented here is discussed in the following Future Work Section.

The core Homer architecture was implemented to provide a test-bed for grounded research to take place within the policy domain, so as such Homer was designed to run on one single central computer within the home to eliminate distribution complexities. However, this has disadvantages:

- there is a single point of failure, so if something went wrong with the central hub then the whole home system would stop working
- multi-user limitations, for example if the owner of a block of flats installed Homer to automate and manage the building, it would not lend itself to then allowing individual flat residents to individually automate and manage their flat due to the fundamental Homer design of one installation per home
- finally scalability can be of concern, where sizeable homes/buildings with many devices, hardware connections, users and policies are all interacting with one central system.

A better design for a home system could be to make use of a more distributed architecture, allowing hardware communication, policy management and general control to be propagated and distributed to various systems around the home. The cloud could be used to store the user data and policies, ensuring that there is one accessible source for the latest information.

For example, in such a hypothetical system one could formulate policies in the cloud. The cloud could then analyse the policies and propagate them to appropriate users, homes, rooms, or even devices. Such nodes can be managed by the cloud, so in the case of failure alternative mechanisms could be used.

A second limitation involves the nature of programmability within Homer. Homeric combines raw triggers, conditions and actions to produce policies. However, typically within the home these events are low-level and rather device oriented. Often multiple events must be combined to represent a higher-level activity, and this can be time-consuming and frustrating for the user to express in multiple policies.

As an example, what if the user would like to express a policy which turns on the hall light when they walk in the front door, but they do not own any devices to know if the front door has been opened or closed? The goal could be achieved by piecing together multiple low-level sensors, such as "*when* garage door opens *then* movement detected in garage *then* garage door closes *then* movement detected in hall". This achieves the desired policy, however if the user wishes to write further policies involving walking in the front door this logic would need to be duplicated, which is cumbersome. Additionally, if the user bought a front door sensor at some point in the future then all the policies which involved the original logic would need to be individually and manually updated to make use of the new single sensor.

Whilst a home system is running, conflicts are possible. This is especially the case where there are multiple residents within the home, competing for the same resources with their own goals and objectives.

Currently, Homer has no way of detecting nor handling such runtime conflicts. Although the Homer system does not suffer as a result of such conflicts, the users will most likely end up confused and frustrated. If one user requests the heating to turn on as she leaves work in order to warm the house for her return, there could be another user within the home trying to turn off the heating as he's been cooking and is too warm. Each would be unaware of the other's actions, and the net result is that neither resident would be satisfied.

Additionally, the users have no way to query the home. It could be highly useful and desirable to provide a mechanism for users to be able to question why certain events took place, in order to better understand why the home is behaving as it is. In the previous example, the resident in the kitchen at home could query the system to answer why the heating is on despite him requesting that it be turned off. This would expose the desire of his partner turning it on for her mobile phone only a few minutes earlier.

6.5 FUTURE WORK

There are a number of areas presented in this thesis that could be advanced. These include:

Policy Language

- **Policy variables** currently exist in ACCENT but not within Homer. They would be useful within the Homer policy language to allow users to refer to different values in different aspects of writing a policy. For example: "*when* SMS received from <person> saying "What is the house temperature?" *do* send SMS to <person> saying "The house temperature is: <house temperature>°C.". Research into how best to support this in an anonymous dynamic way and expose the notion in a simplistic yet flexible way for the user could be undertaken.

- **Sensor and actuator fusion** are terms for the notion of relating lower-level events to higher-level events and could be used to handle the second limitation discussed in the previous section. An example of both sensor and actuator fusion include:
  - Sensor Fusion: "*when* front door opens *and* hall movement is detected *and* front door closes" could be used to describe someone walking into the house.
  - Actuator Fusion: "*do* turn on the heating *and* turn off the air-conditioning" could be used to describe heating up the house.

The act of combining terms to produce higher-level terms is advantageous for the user, as commonly expressed events can be written once and used many times. It also allows for changes to be made to the higher-level terms in one place, which will mean all policies relying on the term will naturally inherit the new logic. This is a feature of ACCENT, however with research the current rigid style as seen in ACCENT could be extended to integrate with the dynamic and anonymous nature of Homer.

- **Prohibition policies** are a popular policy format offered by existing policy engines discussed in Section 4.4. A prohibition policy allows a user to specify things that should not be permitted, overriding existing policies at runtime. Such as "*when* the home is unoccupied *do not* turn on heating". Further research could be undertaken to make this concept suitable for end users, and to integrate into Homer.

Policy Conflict Detection

- **Advancements to Environs**
  - **Environ Assignment**: A more sophisticated means of assigning environ effects would be desirable in Homer. This would involve offering an extensive range of default values and settings for typical devices within a home (to minimise manual user input), as well as more customisable environ effects. As a simple example, describing a particular effect on an environ within Homer is currently rather rigid. Consider opening a window: if it was a dry day, the humidity level of the home will tend to fall, while, on a wet day it may well rise.
  - **Environ Values**: Environs could be enhanced by associating values with the action effects upon the environ. For example, when turning on a washing machines energy usage will increase by, say, 700 Watts/hour and 90 litres of water will be consumed. Research into how these values could be specified, how this information could aid conflict detection and the value they may add to a general home automation system could greatly enhance the existing work presented in this thesis.
  - **Environ Limits**: Following on from environ values, virtual limits could be associated with environs and incorporated into the policy language. For example, a user could write a policy to not allow more than 3000 Watts to be consumed at any one time. Again, this could enhance the home automation experience for users.
- **Conflict Resolution Features** would enhance the existing conflict detection work of Homer. Although it was argued that offline conflict detection was far better suited to the Homer philosophy of the user always being in charge, the work of ACCENT, KAoS and Ponder is acknowledged. Homer lacks the sophistication of these policy solutions due to lack of runtime conflict detection. It would be desirable to offer runtime conflict handling that is customised and controlled by the user through the use of resolution policies (inspired from ACCENT). For example, the user could say that if a requested action conflicts with another, the policy server should choose the action belonging to the newer policy.
- **Conflict Loops** are currently undetected by the Homer policy engine. As a concrete example, take the following two policies: "*when* the temperature is below 20°C *and* the heating is off *do* turn on the heating" and "*when* power usage exceeds 4kW *do* turn off the heating". These two policies could cause a loop, whereby the temperature is low so the heating is turned on, which in turn pushes the power usage over 4kW turning the heating off again. The effect of turning off the heating will cause the first policy to evaluate and fire once again. Whilst the temperature remains below 20°C and the heating puts the power usage over the 4kW limit these policies will continue to trigger, causing the heating to alternate between turning on and off.

  Through the use of environs Homer could realise that turning on the heating will consume power, which in turn could invoke the second policy. When these two policies are compared for conflict the turn on and turn off of the heating would result in a conflict reported to the user. This suggested solution needs further research and integration with the existing Homer policy engine.

Policy Handling

- **Policy Explanations** can be highly desirable in a home to allow the user to query why certain events did (or did not) take place. Currently little work exists in this field, so research could be performed to extend Homer to allow support of policy explanations to help provide a means of justification and reassurance for the users.
- **Library of User/Template Policies**: for some households there could exist very similar rules as those found in other households, so much so that the notion of a rule 'store' or 'library' may prove both useful and inspiring for end users. Template rules could be written which could then be filled-in by end users, people could share their rules for others to view and try, or even suggested rules could be integrated to help encourage automation of common routines. This would also bring the advantage of helping teach users about the Homer policy language and what is possible within their home.
- **Multi-Occupancy** within the home brings many challenges that have not been addressed within this thesis. These challenges include:
  - **Access Restrictions** are desirable to ensure that the various members of the household can only control and alter the home in ways deemed appropriate by the home owner/head of household. Can policies aid the general access rules for the home for the various users? For example, perhaps a policy could be set which disables the children within the household adjusting any devices out with their bedroom, or that limits the children to only one hour of television a day.
  - **Conflicts** will arise more frequently if there exist multiple different people defining them for the one space. Research exists which explores this problem within telecare, where different stakeholders (such as doctors, nurses, family, friends) may all be writing policies for one home [128]. The solution presented makes use of policy hierarchies, whereby policies authored by the stakeholder with highest authority will be prioritised in all conflicting situations. Research needs to be performed to evaluate if this approach can work within the home, and how best to integrate this into a home policy system such as Homer.
  - **Home Anonymity** can be an issue with existing home technology, whereby the home will often be unable to distinguish the various presences in the home. Unlike in a one person home, where sensors are activated and the home can assume this was because of that one person. Multi-occupancy can cause problems with directing information, evaluating the whereabouts, and personalising and automating events for a particular person in the home. Work exists which explores the notion of wearable sensors [121, 123, 138] to help the home obtain a clearer idea of who is who, however this relies on the users remembering to wear the sensors at all times and can therefore prove unreliable and problematic.

6.6 CONCLUDING REMARKS

Home automation is inevitable. Technology has taken over our factories, our cars, our workplaces and our homes. The desire and technical feasibility exist to automate our daily lives through personalised home automation systems. As highlighted in Section 2.2.4, there are currently five main issues which hinder home automation: high costs, disjointed hardware, inflexibility, complicated user interfaces, and security concerns. This thesis has aimed to contribute to research within the inflexibility space, exploring how to improve general home customisation and programmability aspects.

This thesis has presented a fully implemented and advanced home system. The system is one of the only which can offer end users the ability to control, monitor *and* program their home. By making this requirement a reality Homer has shown what is possible and has laid a foundation for future home automation.

Three research contributions have been presented in this thesis: a custom policy language for the home, novel policy overlap detection, and advancements in existing policy conflict detection techniques.

To conclude, the research and development presented in this thesis has extended the state of the art within multiple domains, met the many requirements and objectives discussed throughout, and offered future directions for various aspects of the work.

Part IV

BACK MATTER

[1] Accord Project. Understanding and using the tangible toolbox. www.sics.se/accord/release/docs/html/D3.2.htm, Sept. 2002.

[2] K. P. Akesson, A. Bullock, T. Rodden, B. Koleva, and C. Greenhalgh. A toolkit for user re-configuration of ubiquitous domestic environments. *Companion to Proceedings of UIST*, 2002:1–2, 2002.

[3] A. Alkar and U. Buhur. An internet based wireless home automation system for multifunctional devices. *Consumer Electronics, IEEE Transactions on*, 51(4):1169–1174, 2005.

[4] Amigo Project. Amigo project description. www.hitech-projects.com/euprojects/amigo, Dec. 2008.

[5] T. Andrews, H. Dholakia, Y. Goland, B. Klein, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services. 2003.

[6] A. Arabo and F. El-Mousa. Security framework for smart devices. In *Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on*, pages 82–87. IEEE, 2012.

[7] J. C. Augusto. Past, present and future of ambient intelligence and smart environments. *Environments*, 67:1–15, 2010.

[8] R. Ballagas, M. Ringel, M. Stone, and J. Borchers. iStuff: A physical user interface toolkit for ubiquitous computing environments. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI)*, volume 16, pages 2–4, 2003.

[9] C. Becker, M. Handte, G. Schiele, and K. Rothermel. Pcom – A component system for pervasive computing. In *Proceedings of the 2nd International Conference on Pervasive Computing and Communications, Orlando, Florida*, pages 67–76. IEEE, 2004.

[10] C. Beckmann and A. Dey. Siteview: Tangibly programming active environments with predictive visualization. In *Adjunct Proceedings of UbiComp*, pages 167–168, 2003.

[11] G. Bell and J. Kaye. Designing technology for domestic spaces: A kitchen manifesto. *Gastronomica*, 2(2):46–62, 2002.

[12] P. Bergstrom, K. Driscoll, and J. Kimball. Making home automation communications secure. *Computer*, 34(10):50–56, 2001.

[13] A. F. Blackwell and R. Hague. AutoHAN: An architecture for programming the home. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 150–157. IEEE, 2001.

[14] L. Blair, J. Pang, K. J. Turner, S. Reiff-Marganiec, T. Gray, P. Perry, and J. Ireland. Policy support for call control. *Computer Standards Interfaces*, 28(6):635–649, 2005.

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

[16] R. Boutaba and I. Aib. Policy-based management: A historical perspective. *Journal of Network and Systems Management*, 15(4):447–480, 2007.

[17] A. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home automation in the wild: challenges and opportunities. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, pages 2115–2124. ACM, 2011.

[18] M. Burnett. Visual language research bibliography. web.engr.oregonstate.edu/~burnett/vpl.html, 2009.

[19] M. Calder, M. Kolberg, E. Magill, D. Marples, and S. Reiff-Marganiec. Hybrid solutions to the feature interaction problem. In *Proceedings of Feature Interactions in Telecommunications and Software Systems VII FIW Ottawa Kanada*, pages 295–312. IOS Press, 2003.

[20] M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[21] G. A. Campbell. A goal-directed and policy-based approach to system management. Technical Report CSM-180, University of Stirling, May 2009.

[22] G. A. Campbell and K. J. Turner. Goals and policies for sensor network management. In D. Urška, editor, *Second International Conference on Sensor Technologies and Applications*, pages 354–359. IEEE, 2008.

[23] G. A. Campbell and K. J. Turner. Policy conflict filtering for call control. In *Proceedings 9th Int Conf on Feature Interactions in Software and Communications Systems*, pages 93–108. IOS Press, May 2008.

[24] J. J. Castillo. Snowball sampling. www.experiment-resources.com/snowball-sampling.html, 2009.

[25] R. Cerqueira, C. Cassino, and R. Ierusalimschy. Dynamic component gluing across different componentware systems. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 362–371. IEEE, 1999.

[26] R. Charette. This car runs on code. *IEEE Spectrum*, 46(3):3, 2009.

[27] M. Chetty, J. Sung, and R. Grinter. How smart homes learn: The evolution of the networked home and household. *UbiComp 2007: Ubiquitous Computing*, pages 127–144, 2007.

[28] Control 4. About us. www.control4.com/about-us/, 2012.

[29] D. Cook and S. Das. *Smart environments: Technology, protocols and applications*, volume 43. Wiley-Interscience, 2004.

[30] P. Corcoran and J. Desbonnet. Browser-style interfaces to a home automation network. *Consumer Electronics, IEEE Transactions on*, 43(4):1063–1069, 1997.

[31] Cortexa. Cortexa system setup. www.youtube.com/user/cortexa2009#p/u/3/kg4BrHbu06Q, 2009.

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

[33] C. N. Darrah, J. Freeman, and J. A. English-Lueck. Busier than ever: Why american families can't slow down. *Anthropology of Work Review*, 30(1):288, 2007.

[34] S. Davidoff, M. K. Lee, C. Yiu, J. Zimmerman, and A. K. Dey. Principles of smart home control. In *Proceedings Ubiquitous Computing*, pages 19–34, 2006.

[35] D. de Wit. *The Shaping of Automation*, volume 13. Uitgeverij Verloren, 1994.

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

[37] C. Douligeris, J. Khawand, and C. Khawand. Communications and control for a home automation system. In *Southeastcon'91., IEEE Proceedings of*, pages 171–175. IEEE, 1991.

[38] W. K. Edwards, M. W. Newman, and J. Z. Sedivy. The case for recombinant computing. Technical Report CSL-01-1, 2001.

[39] A. Endpoints. The ActiveBPEL engine. `www.activevos.com/community-open-source.php`, June 2009.

[40] K. Gajos, H. Fox, and H. Shrobe. End user empowerment in human centered pervasive computing. In *Pervasive 2002*, number 2414 in Lecture Notes in Computer Science, pages 134–140. Springer, 2002.

[41] A. Gárate, N. Herrasti, and A. López. Genio: an ambient intelligence application in home automation and entertainment environment. In *Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies*, pages 241–245. ACM, 2005.

[42] J. Gershuny. Busyness as the badge of honor for the new superordinate working class. *Social Research*, 72(2):287–314, 2005.

[43] C. Geyer. About BPEL. `bpel.xml.org/about-bpel`, 2007.

[44] P. D. Gray, T. McBryan, N. Hine, C. J. Martin, N. Gil, M. Wolters, N. Mayo, K. J. Turner, L. S. Docherty, F. Wang, and M. Kolberg. A scalable home care system infrastructure supporting domiciliary care. Technical Report CSM-173, University of Stirling, Aug. 2007.

[45] D. Greaves. Autohan project. `www.cl.cam.ac.uk/research/srg/han/AutoHAN/oldindex.html`, 2000.

[46] S. Greenberg and C. Fitchett. Phidgets: Easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, page 209. ACM Press, New York, USA, 2001.

[47] R. Grinter, W. Edwards, M. Chetty, E. Poole, J. Sung, J. Yang, A. Crabtree, P. Tolmie, T. Rodden, C. Greenhalgh, et al. The ins and outs of home networking: The case for useful and usable domestic networking. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(2):1–28, 2009.

[48] R. Grinter, W. Edwards, M. Newman, and N. Ducheneaut. The work to make a home network work. In *ECSCW 2005*, pages 469–488. Springer, 2005.

[49] R. Harper. *Inside the smart home*. Springer, 2003.

[50] M. Heisel, J. Souquieres, et al. A heuristic algorithm to detect feature interactions in requirements. *Language Constructs for Describing Features*, pages 143–162, 2000.

[51] T. Hjorth and R. Torbensen. Trusted domains in home automation. *Computers & Security*, 2012.

[52] S. Hughes. SHAKE users group. `www.dcs.gla.ac.uk/research/shake`, Aug. 2007.

[53] J. Humble, A. Crabtree, T. Hemmings, K.-P. Akesson, B. Koleva, T. Rodden, and P. Hansson. Playing with the bits: User configuration of ubiquitous domestic environments. In *Proceedings of UbiComp 2003*, Lecture notes in Computer Science, pages 256–263. Springer, 2003.

[54] S. Intille. Designing a home of the future. *Pervasive Computing, IEEE*, 1(2):76–82, 2002.

[55] JBoss. Jboss Drools. `jboss.org/drools`, Feb. 2008.

[56] B. Johanson and A. Fox. The Stanford interactive workspaces project. *The VLSI Journal*, pages 1–30, Aug. 2004.

[57] L. Kagal. Rei: A policy language for the me-centric project. Technical Report HPL-2002-270, HP Labs, 2002.

[58] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, pages 63–74, Washington, DC, USA, 2003. IEEE Computer Society.

[59] D. Keck and P. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *Software Engineering, IEEE Transactions on*, 24(10):779–796, 1998.

[60] S. L. Keoh, K. Twidle, N. Pryce, A. E. Schaeffer-Filho, E. Lupu, M. Sloman, S. Heeps, S. Strowes, J. Sventek, and E. Katsiri. Policy-based management for body-sensor networks. In *4th International Workshop on Wearable and Implantable Body Sensor Networks*, pages 92–98. Springer, Mar. 2007.

[61] J. Kientz, S. Patel, B. Jones, E. Price, E. Mynatt, and G. Abowd. The georgia tech aware home. In *CHI'08 extended abstracts on Human factors in computing systems*, pages 3675–3680. ACM, 2008.

[62] T. Kim, L. Bauer, J. Newsome, A. Perrig, and J. Walker. Challenges in access right assignment for secure home networks. *Proc. HotSec 2010*, 2010.

[63] K. Kimbler and L. G. Bouma. *Feature Interactions in Telecommunications and Software Systems V*. Ios PressInc, 1998.

[64] J. King, R. Bose, H. I. Yang, S. Pickles, and A. Helal. Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces. In *Proceedings 31st Conference on Local Computer Networks*, pages 630–638. IEEE, 2006.

[65] S. R. Klemmer, J. Li, J. Lin, and J. A. Landay. Papier-maché. In *Proceedings of Human Factors in Computing Systems*, pages 399–406. ACM Press, New York, USA, 2004.

[66] M. Knoll, T. Weis, A. Ulbrich, and A. Brändle. Scripting your home. In *Proceedings of the Second international conference on Location- and Context-Awareness*, LoCA'06, pages 274–288, Berlin, Heidelberg, 2006. Springer-Verlag.

[67] M. Kolberg and E. H. Magill. A pragmatic approach to service interaction filtering between call control services. *Computer Networks*, 38(5):591–602, 2002.

[68] M. Kolberg and E. H. Magill. Using pen and paper to control networked appliances. *IEEE Communications*, 44(11):148–154, 2006.

[69] T. Koskela and K. Väänänen-Vainio-Mattila. Evolution towards smart home environments: empirical evaluation of three user interfaces. *Personal and Ubiquitous Computing*, 8(3):234–240, 2004.

[70] Lego Mindstorms. Software demo. `mindstorms.lego.com/en-us/Software/Default.aspx`, 2010.

[71] B. Leuf and W. Cunningham. *The Wiki Way*. Addison Wesley, 2001.

[72] N. Li, M. V. Tripunitara, and Q. Wang. Resiliency policies in access control. *ACM Transactions on Information and System Security*, 12(4):113–123, 2009.

[73] Y. Li, J. I. Hong, and J. A. Landay. Topiary: A tool for prototyping location-enhanced applications. In *User interface software and technology*, pages 217–226. ACM, 2004.

[74] H. Lieberman, F. Paternó, and M. Klann. End-user development: An emerging paradigm. *End User Development*, 9:1–8, 2006.

[75] E. Litvinova and P. Vuorimaa. Engaging end users in real smart space programming. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 1090–1095. ACM, 2012.

[76] B. Margolis and J. Sharpe. *SOA for the Business Developer: Concepts, BPEL, and SCA*. MC Press, 2007.

[77] D. J. Marples, S. Tsang, E. H. Magill, and D. G. Smith. A platform for modelling feature interaction detection and resolution. In *Proceedings 3rd Feature Interaction Workshop*, pages 185–199. IOS, 1995.

[78] C. Maternaghan. The homer home automation system. Technical Report CSM-187, University of Stirling, Dec. 2010.

[79] C. Maternaghan. How do people want to control their home? Technical Report CSM-185, University of Stirling, Dec. 2010.

[80] C. Maternaghan. Can people program their homes? Technical Report CSM-191, University of Stirling, 2012.

[81] C. Maternaghan and K. J. Turner. A component framework for telecare and home automation. In *Proceedings 7th Consumer Communications and Networking Conference*, pages N4.1–N4.5. IEEE, Jan. 2010.

[82] C. Maternaghan and K. J. Turner. A configurable telecare system. In *PETRA*, Crete, Greece, 2011. ACM Press.

[83] 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, Nov. 2011.

[84] C. Maternaghan and K. J. Turner. Programming home care. In *Proceedings Advances in Techniques and Technologies for Care at Home*, pages 5.1–5.7. IEEE, May 2011.

[85] M. Mazurek, J. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, et al. Access control for home data sharing: Attitudes, needs and practices. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 645–654. ACM, 2010.

[86] T. McBryan and P. Gray. A model-based approach to supporting configuration in ubiquitous systems. In *Proceedings Interactive Systems. Design, Specification and Verification*, pages 167–180. Springer, 2008.

[87] T. McBryan, M. R. McGee-Lennon, and P. Gray. An integrated approach to supporting interaction evolution in home care systems. In *Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments*, pages 167–180, New York, USA, 2008. Springer.

[88] T. McByran and P. Gray. A model-based approach to supporting configuration in ubiquitous systems. *Lecture Notes In Computer Science*, 5136:167–180, 2008.

[89] M. D. McIlroy. Mass produced software components. In *Software Engineering Concepts and Techniques*, pages 88–98. NATO Science Committee, 1968.

[90] M. E. T. McMurdo. A healthy old age: Realistic or futile goal? *British Medical Journal*, 321:1149–1151, 2000.

[91] Microsoft. Kodu. `research.microsoft.com/en-us/projects/kodu`, 2010.

[92] G. Mori, F. Paternò, and C. Santoro. CTTE: Support for developing and analyzing task models for interactive system design. *IEEE Transactions on Software Engineering*, 28(8):797–813, 2002.

[93] M. C. Mozer. The neural network house: An environment that adapts to its inhabitants. In *Proceedings AAAI Spring Symp. Intelligent Environments*, pages 110–114, 1998.

[94] M. Nakamura, H. Igaki, and K. ichi Matsumoto. Feature interactions in integrated services of networked home appliances. In *Proceedings Feature Interactions in Telecommunications and Software Systems*, pages 236–251. IOS, June 2005.

[95] M. Nakamura, T. Kikuno, J. Hassine, and L. Logrippo. Feature interaction filtering with use case maps at requirements stage. In *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00), IOS Press*, pages 163–178, 2000.

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

[97] M. W. Newman, A. Elliott, and T. F. Smith. Providing an integrated user experience of networked media, devices, and services through end-user composition. Pervasive '08, pages 213–227. Springer, 2008.

[98] D. F. Noble. *Forces of Production: A Social History of Industrial Automation*. Transaction Publishers, 2011.

[99] R. Nunes and J. Delgado. An internet application for home automation. In *Electrotechnical Conference, 2000. MELECON 2000. 10th Mediterranean*, volume 1, pages 298–301. IEEE, 2000.

[100] Open Health Tools. Why OSGi? `www.projects.openhealthtools.org/sf/wiki/do/viewPage/projects.stepstone/wiki/WhyOSGi`, 2010.

[101] Oracle. Oracle BPEL process manager. `www.oracle.com/technology/products/ias/bpel/index.html`, 2012.

[102] T. Owen, I. Wakeman, B. Keller, J. Weeds, and D. Weir. Managing the policies of non-technical users in a dynamic world. In *Proceedings 6th International Workshop on Policies for Distributed Systems and Networks, Stockholm, Sweden*, pages 251–254. IEEE, 2005.

[103] E. Poole, M. Chetty, R. Grinter, and W. Edwards. More than meets the eye: transforming the user experience of home network management. In *Proceedings of the 7th ACM conference on Designing interactive systems*, pages 455–464. ACM, 2008.

[104] S. Reiff-Marganiec and K. Turner. Use of logic to describe enhanced communications services. *Formal Techniques for Networked and Distributed Sytems—FORTE 2002*, pages 130–145, 2002.

[105] S. Reiff-Marganiec and K. Turner. "feature interaction in policies". *Computer Networks*, 45(5):569–584, 2004.

[106] J. Rimmer, T. Owen, I. Wakeman, B. Keller, J. Weeds, and D. Weir. User policies in pervasive computing environments. In *User Experience Design for Pervasive Computing, Pervasive 2005*, Munich, Germany, May 2005.

[107] T. Rodden, A. Crabtree, T. Hemmings, B. Koleva, J. Humble, K.-P. Akessonn, and P. Hansson. Configuring the ubiquitous home. In *Proceedings 6th Int. Conf. on The Design of Cooperative Systems*, pages 215–230. ACM Press, 2004.

[108] J. Rode, E. Toye, and A. Blackwell. The domestic economy: a broader unit of analysis for end user programming. In *CHI'05 extended abstracts on Human factors in computing systems*, pages 1757–1760. ACM, 2005.

[109] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 20(4):74–83, 2002.

[110] N. Sadeh, J. Hong, L. Cranor, I. Fette, P. Kelley, M. Prabaker, and J. Rao. Understanding and capturing people's privacy policies in a mobile social networking application. *Personal and Ubiquitous Computing*, 13(6):401–412, 2008.

[111] P. Sánchez, M. Jiménez, F. Rosique, B. Álvarez, and A. Iborra. A framework for developing home automation systems: From requirements to code. *Journal of Systems and Software*, 84(6):1008–1021, 2011.

[112] A. Schaeffer-Filho, E. Lupu, M. Sloman, S. L. Keoh, J. Lobo, and S. Calo. A role-based infrastructure for the management of dynamic communities. In *2nd International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, number 5127 in Lecture Notes in Computer Science, pages 1–14. Springer, 2008.

[113] K. P. Schools. Explorer kodu club. `koduclub.org/default.aspx`, 2010.

[114] E. Shehan and W. Edwards. Home networking and hci: what hath god wrought? In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 547–556. ACM, 2007.

[115] M. P. Singh and M. N. Huhns. *Service-Oriented Computing*. Wiley, Chichester, UK, nov 2004.

[116] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. Koala: A platform for OS-level power management. *Power*, pages 289–302, 2009.

[117] T. Sohn and A. K. Dey. iCAP: An informal tool for interactive prototyping of context-aware applications. In *Extended abstracts on Human Factors In Computing Systems*, pages 974–975, New York, USA, 2003. ACM.

[118] J. Sousa and D. Garlan. Aura: An architectural framework for user mobility in ubiquitous computing environments. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Proceedings of the 3rd Working Conference on Software Architecture*, pages 29–43. IEEE, Aug. 2002.

[119] N. Sriskanthan, F. Tan, and A. Karande. Bluetooth based home automation system. *Microprocessors and Microsystems*, 26(6):281–289, 2002.

[120] T. Starner, J. Auxier, D. Ashbrook, and M. Gandy. The gesture pendant: A self-illuminating, wearable, infrared computer vision system for home automation control and medical monitoring. In *Wearable Computers, The Fourth International Symposium on*, pages 87–94. IEEE, 2000.

[121] M. Stikic, D. Larlus, S. Ebert, and B. Schiele. Weakly supervised recognition of daily life activities with wearable sensors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(12):2521–2537, 2011.

[122] D. Strassberg. Home-automation buses: Protocols really hit home. *EDN*, 40(8):69–84, 1995.

[123] A. Subramanya, A. Raj, J. Bilmes, and D. Fox. Recognizing activities and spatial context using wearable sensors. *arXiv preprint arXiv:1206.6869*, 2012.

[124] The Knopflerfish Project. Knopflerfish android. `www.knopflerfish.org/releases/3.2.0/docs/android_dalvik_tutorial.html`, 2012.

[125] R. Torbensen. On the emergence of pervasive home automation. 2011.

[126] K. Truong, G. Abowd, and J. Brotherton. Who, what, when, where, how: Design issues of capture and access applications. In *Proceedings Ubiquitous Computing*, pages 209–224. Springer, 2001.

[127] K. Truong, E. M. Huang, and G. D. Abowd. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. *Proceedings Ubiquitous Computing*, pages 143–160, 2004.

[128] K. Turner. The accent policy system. Technical report, Technical Report CSM-188, Department of Computing Science and Mathematics, University of Stirling, UK, 2011.

[129] K. Turner, S. Reiff-Marganiec, L. Blair, G. Cambpell, and F. Wang. "appel: An adaptable and programmable policy environment and language". 2007.

[130] K. J. Turner. Device services for the home. In *Proceedings 10th Int. Conf. on New Technologies for Distributed Systems*, pages 41–48. IEEE, May 2010.

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

[132] K. J. Turner and K. L. L. Tan. Graphical composition of grid services. In *Proceedings 6th Rapid Introduction of Software Engineering Techniques*, number 4401, pages 1–17. Springer, May 2007.

[133] K. Twidle. Ponder2: PonderTalk. `www.ponder2.net/cgi-bin/moin.cgi/PonderTalk`, 2008.

[134] K. Twidle, N. Dulay, E. Lupu, and M. Sloman. Ponder2: A policy system for autonomous pervasive environments. In *Proceedings 5th International Conference on Autonomic and Autonomous Systems*, pages 330–335. IEEE, 2009.

[135] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Proceedings 4th International Workshop on Policies for Distributed Systems and Networks*, pages 93–96. IEEE, 2003.

[136] A. Uszok, J. M. Bradshaw, J. Lott, M. Breedy, L. Bunch, P. Feltovich, M. Johnson, and H. Jung. New developments in ontology-based policy management: Increasing the practicality and comprehensiveness of KAoS. In *Proceedings Workshop on Policies for Distributed Systems and Networks*, pages 145–152. IEEE, 2008.

[137] W3C. Web services choreography description language version 1.0. `www.w3.org/TR/2004/WD-ws-cdl-10-20041217`, 2004.

[138] L. Wang, T. Gu, X. Tao, H. Chen, and J. Lu. Recognizing multi-user activities using wearable sensors in a smart home. *Pervasive and Mobile Computing*, 7(3):287–298, 2011.

[139] J. Weeds, B. Keller, D. Weir, I. Wakeman, J. Rimmer, and T. Owen. Natural language expression of user policies in pervasive computing environments. In *Proceedings Workshop on Ontologies and Lexical Resources in Distributed Environments*, 2004.

[140] T. Weis and K. Geihs. Components on the desktop. In *Proceedings 33rd International Conference on Technology of Object-Oriented Languages*, pages 250–261. IEEE Computer Society, 2000.

[141] T. Weis, M. Handte, M. Knoll, and C. Becker. Customizable pervasive applications. In *Proceedings 4th International Conference on Pervasive Computing and Communications*, pages 239–244. IEEE, 2006.

[142] M. E. Wilson. *An Online Environmental Approach to Service Interaction Management in Home Automation*. PhD thesis, University of Stirling, 2006.

[143] M. E. Wilson, E. H. Magill, and M. Kolberg. An online approach for the service interaction problem in home automation. In *Consumer Communications and Networking Conference*, pages 251–256. IEEE, 2005.

[144] X. Wu and H. Schulzrinne. Handling feature interactions in the language for end system services. *Computer Networks*, 51(2):515–535, 2007.

[145] C. Yerrapragada and P. Fisher. Voice controlled smart house. In *Consumer Electronics, 1993. Digest of Technical Papers. ICCE., IEEE 1993 International Conference on*, pages 154–155. IEEE, 1993.

[146] B. Yuksekkaya, A. Kayalar, M. Tosun, M. Ozcan, and A. Alkar. A gsm, internet and speech controlled wireless interactive home automation system. *Consumer Electronics, IEEE Transactions on*, 52(3):837–843, 2006.

[147] E. Zukerman. Tasker for android: A mobile app that caters to your every whim. www.makeuseof.com/tag/tasker-android-mobile-app-caters-whim, 2011.