Technical Report

UCAM-CL-TR-651
ISSN 1476-2986

Number 651

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

End-user programming in multiple languages

Rob Hague

October 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2005 Rob Hague

This technical report is based on a dissertation submitted July 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Fitzwilliam College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

End-User Programming in Multiple Languages

Rob Hague

Abstract

Advances in user interface technology have removed the need for the majority of users to program, but they do not allow the automation of repetitive or indirect tasks. End-user programming facilities solve this problem without requiring users to learn and use a conventional programming language, but must be tailored to specific types of end user. In situations where the user population is particularly diverse, this presents a problem.

In addition, studies have shown that the performance of tasks based on the manipulation and interpretation of data depends on the way in which the data is represented. Different representations may facilitate different tasks, and there is not necessarily a single, optimal representation that is best for all tasks. In many cases, the choice of representation is also constrained by other factors, such as display size. It would be advantageous for an end-user programming system to provide multiple, interchangeable representations of programs.

This dissertation describes an architecture for providing end-user programming facilities in the networked home, a context with a diverse user population, and a wide variety of input and output devices. The Media Cubes language, a novel end-user programming language, is introduced as the context that lead to the development of the architecture. A framework for translation between languages via a common intermediate form is then described, with particular attention paid to the requirements of mappings between languages and the intermediate form. The implementation of Lingua Franca, a system realizing this framework in the given context, is described.

Finally, the system is evaluated by considering several end-user programming languages implemented within this system. It is concluded that translation between programming languages, via a common intermediate form, is viable for systems within a limited domain, and the wider applicability of the technique is discussed.

Contents

Abstract **3**

Table of Contents **5**

1 Introduction **7**
1.1 From Programmer to User 7
1.2 From User to Programmer 8
1.3 Why Ubiquitous Computing? 9
1.4 Why Multiple Representations? 10
1.5 End-User Programming for Ubiquitous Computing 11
1.6 Aims 12
1.7 Dissertation Overview 12

2 Related Work **13**
2.1 Ubiquitous Computing 13
2.2 Internet Technologies 16
2.3 Home Area Networks 17
2.4 Approaches to End-User Programming 19
2.4.1 End-User Programming in Traditional Environments . . . 20
2.4.2 Programming by Example 21
2.4.3 Visual Programming 23
2.5 Programming in the Home 25
2.6 Using Multiple Representations 27
2.7 Conventional Systems for Integrating Multiple Languages . . . 31
2.8 Generation of Multiple Representations from a Single Source . 32
2.8.1 Multiple Representation in Software Visualisation 34
2.8.2 Multi-View Development Environments 37

3 The Media Cubes **41**
3.1 Introduction 41
3.2 Tangible User Interfaces 41
3.3 The Media Cube Device 44
3.4 The Media Cubes Language 46
3.5 An Example Media Cubes Program 50
3.6 The Nature of Events 52
3.7 Implementation 53
3.8 Evaluation 55

4 Translation based on common intermediate form **57**
4.1 Language Integration via a Shared Intermediate Form 57
4.2 Requirements of Mappings 57
4.3 Execution as Mapping 58
4.4 Secondary Notation in Multiple Languages 59
4.5 Structure as Secondary Notation 61
4.6 An Environment for Multi-Language Programming 63

5 The Lingua Franca Architecture **65**
5.1 An Overview of the Lingua Franca Architecture 65

5.2 The Lingua Franca Execution Model 65
5.3 Examples of Lingua Franca execution 69
5.4 The external representation of Lingua Franca 71
5.5 Operations on the Lingua Franca Corpus 73

6 The Implementation Of Lingua Franca 75
6.1 The Prototype Execution Engine 75
6.2 The LFCore Toolkit 76
6.3 Prototype Language Environments 79
6.3.1 The Media Cubes Language in Lingua Franca 79
6.3.2 VSeq — Visual Sequences 82
6.3.3 LFScript — A Textual Language 87

7 Evaluation 93
7.1 Implementation of Reversible Translation 93
7.2 Usability Evaluation 96
7.3 Cognitive Dimensions Analysis 97
7.3.1 Notation, Medium and Environment 98
7.3.2 Visibility and Juxtaposability 99
7.3.3 Diffuseness 99
7.3.4 Viscosity 99
7.3.5 Secondary Notation and Provisionality 100
7.3.6 Hidden Dependencies 101
7.3.7 Closeness of Mapping, Consistency and Role Expressiveness 101
7.3.8 Premature Commitment 102
7.3.9 Progressive Evaluation 102
7.3.10 Implications 103

8 Conclusions & Future Work 105
8.1 Summary 105
8.2 Contribution 105
8.3 Future Work 106
8.3.1 Testing 106
8.3.2 Additional Language Environments 106
8.3.3 Improvements to Lingua Franca 106
8.3.4 Further Developments of Existing Language Environments 108
8.4 Ubiquitous End-User Programming 110

A Schema for XML Lingua Franca 111

B User Questionnaire 115

Bibliography 117

Chapter 1
Introduction

In the six decades since their invention, programmable electronic computers have moved from being a tool for mathematicians, scientists and engineers, to being an indispensable part of everyday life. While this is most obvious in environments such as schools and offices, where monitors, keyboards and other paraphernalia abound, it has also had a subtler and more profound impact; computers may be found almost everywhere. One area that is particularly rich in hidden computing technology is the home. In addition to the obvious PC or Mac and games console, computing devices are used to keep the rooms warm and the food cold, to play music and record TV. However, these devices exist in virtual isolation; they cannot communicate with each other.

There has been substantial amount of research, both commercial and academic, into pervasive networks in the home. These allow devices to communicate with each other, to work together, and to be controlled remotely. Much work has been done to make these networks self-configuring and self-maintaining, so that they do not require the attention of a trained system administrator. One area that has not had the same degree of attention is the usability aspects of pervasive home networking; how can a householder utilise the potential that the network provides?

My work addresses one particular approach to pervasive home network usability, specifically that of end-user programming. Programming may seem an unusual approach to improving usability, but it will be argued that it provides significant gains over simpler techniques. The specific approach to end-user programming is based on a framework that allows users to choose a different language for each stage of program development, employing whichever language is best suited to the task at hand. This approach was initially conceived as a result of work on a novel language, the Media Cubes, which is also presented.

1.1 From Programmer to User

The first computing machines, including early digital computers such as Colossus and ENIAC, had to be physically reconfigured in order to change their functionality. A significant advance came with the advent of stored-program computers, where an operator could alter the functionality of a computer by modifying information (in other words, software) as opposed to its physical construction (hardware). At this point, it became possible to make use of a computer without having an intimate knowledge of the mechanisms by which it worked. In other words, it was possible to be a programmer without also being an engineer.

It is difficult to overestimate the importance of this transition. It enabled specialists in other areas (initially mathematics and the physical sciences) to apply digital computers to problems in their own fields, without having to become experts in the minutiae of electrical engineering involved in building and

maintaining the machines themselves. Of course, at this stage, programming was itself an arcane and difficult to master skill, but it nevertheless dealt at a level of abstraction closer to the problem at hand.

The development of FORTRAN, the first “high-level” programming language, in 1954 represented another significant step. It was designed to allow scientists to express algorithms in a form closer to the formal language of mathematics with which they were already familiar. It could be argued that this goal was not completely achieved, but in any case, FORTRAN and its successors enabled specialists in a wide range of areas to harness the ever-increasing power of computers.

Until now, we have been equating “user” and “programmer”, and indeed throughout a large part of the history of computing the two terms were synonymous; the only way to use a computer was to program it. As programming, even with the benefit of a high-level language, is a non-trivial skill to learn, researchers sought ways to allow people to take advantage of computing without the need for them to program. This lead to the concept of applications - programs that could be used as is to perform some specific task. Technologies such as direct manipulation (Sutherland 1963) meant that such applications could be made even easier to use.

Such improvements in ease of use, married with ever-increasing computational power and falling device cost, made a far wider range of applications feasible. This has led to the present situation, where computers are used daily by a wide variety of people, and the vast majority of these users do not program. Such users are termed *end users*, and typically purchase packaged operating systems and applications, and use them unmodified.

1.2 From User to Programmer

The development of computer use from engineer and programmer to end user has been an overwhelmingly positive move. However, something has been lost in the transition. An advantage that the user-programmer has over the end user is the ability to customize the software to meet specific needs. While the programmer has the ability to modify or extend the behaviour of software more or less arbitrarily, the end user does not have that choice. They must either petition the individual or body that created the application, and ask that they provide the desired feature, or, far more typically, put up with the software’s deficiency and await the release of the next version, hoping that it will be corrected. This not only applies to missing features, but errors in existing functionality. In addition, end-user applications typically provide little opportunity for automation:

> *This leaves personal computer users in an ironic situation. It is a truism that computers are good at performing repetitive activities. So why are we the ones performing repetition, instead of the computer?*
> (Cypher 1993)

The *Free Software* and *Open Source* movements (DiBona et al 1999) appear at first sight to offer a solution to this problem, and indeed many of the problems of proprietary software are relieved by allowing end users to have far greater access to the programmers of the software, or to become programmers themselves. However, this does not tackle the issue of non-programmers customising the software that they use. The point is expanded upon by Nardi:

> *End users are not “casual,” “novice,” or “naive” users; they are people such as chemists, librarians, teachers, architects, and accoun-*

> *tants, who have computational needs and want to make serious use of computers, but who are not interested in becoming professional programmers.* (Nardi 1993)

Hence, while open source development gives the user the opportunity to customise their software, they cannot take advantage of this opportunity unless they invest the time and effort to learn the programming languages and libraries used, and the peculiarities of a particular software package, and yet more time actually developing the software. This is an investment that few users (even those who are experienced programmers) are willing to make. An alternative is to make programming facilities available in a way designed to be usable by end users, and not require a high degree of training in programming techniques. This field is known as *End-User Programming*.

Designing a programming environment for end users differs in several ways from designing a programming environment for trained programmers. Such systems are usually task-specific, and based around a model that is familiar to the end user population targeted. The interface may also be significantly different from traditional, text-based programming. In some, but not all, cases, such systems forego some degree of expressibility in favour of simplicity.

Perhaps the most pervasive type end-user programming system in use today is the automated spreadsheet, introduced with VisiCalc[1], and typified by Lotus 1-2-3[2] and Microsoft Excel[3]. These allow users to create automatically updated cells, the content of which is dependent in some non-trivial way on the content of other cells (which may also be automatically calculated). Simple spreadsheets allow the relationships to be specified using familiar algebraic formulae; more advanced versions, including Excel and 1-2-3, also provide flexible programming languages, allowing sophisticated applications to be created.

Another significant example of end-user programming is HyperCard, shown in Figure 1.1. This system allows users to construct hypertext documents consisting of a "stack" of "cards" containing text and images, buttons to enable navigation, and fields to allow data entry. Scripts can be created in a programming language with a simplified syntax, allowing scripts to be associated with user actions such as clicking on a button. These scripts can examine the current state of the stack, such as the values of fields, and cause arbitrary changes to the content, including navigating to another card. The resulting system is extraordinarily flexible; it is accessible to users with little or no programming experience, yet flexible enough to produce a wide range of high-quality applications (indeed, several of the systems described in the following chapter were prototyped using HyperCard).

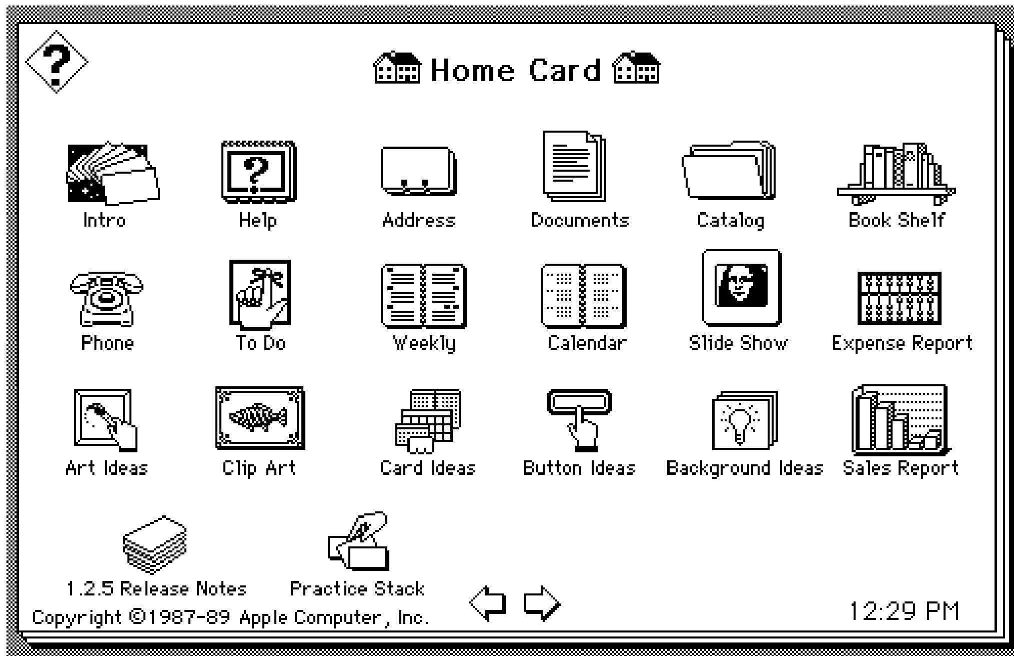
1.3 Why Ubiquitous Computing?

Weiser (1991) and Norman (1999) propose a vision of *ubiquitous computing* as computing that "disappears into the background". Whereas current information technology forces, or at least strongly encourages, the user to focus on the *technology*, ubiquitous computing allows the user to focus on the *information* — in other words, on the task at hand.

Part of this vision is the idea of task-specific *Information Appliances*, as proposed by Norman (1999), Raskin (2000) and others. However, while this is

[1] `http://www.bricklin.com/visicalc.htm`
[2] `http://www.lotus.com/smartsuite`
[3] `http://www.microsoft.com/office/excel/`

Figure 1.1: HyperCard



a step towards Weiser's ideal, it is not the whole story. To fully realize that ideal, it would be necessary for computers to be integrated seamlessly into the environment; whereas an information appliance is still an object to be focused on, a user would not consciously notice that they were using a ubiquitous computing system.

The implementation of ubiquitous computing demands a range of technologies. In addition to low-cost, power-efficient processing devices and networking technologies, it may require specialised display technologies (both very large, as in the Liveboard (Elrod et al 1992), and very small, as in a PDA or mobile phone).

Sensors are a particular class of hardware that is far more important in ubiquitous computing than conventional computing. An accurate and timely picture of the environment in which activity is occurring is a major component of most ubiquitous computing systems. In addition to novel hardware, ubiquitous computing systems often have very different needs in terms of operating systems, networking technologies and other architectural software.

Ubiquitous computing also requires a sea change in the user interfaces techniques used. For example, text entry, something that may be taken for granted when designing in a desktop environment, becomes difficult when the user does not necessarily have a room for a keyboard, a surface to rest on, or even a free hand. Hence, designers must use alternative text entry methods such as handwriting or gesture recognition, speech, or other techniques, or design applications in such a way as to avoid text entry entirely. One promising technique is *tangible user interfaces* (Ishii and Ullmer 1997), in which physical props are used as the medium for interaction. While tangible interfaces may be used in many contexts, the extension of the interface into the real-world environment is particularly suited to ubiquitous computing.

1.4 Why Multiple Representations?

Most, if not all, computing tasks may be regarded as manipulation by the user of sets of data. This manipulation is mediated by a *notation*, via which the data is represented to the user. Notations may be textual languages (such as XML (Bray et al 2000) or notes in a diary), visual languages (such as icons in a file manager, or road signs), actions (such as gestures), or combinations thereof. There is a large body of work in cognitive psychology relating to the use of notations, and frameworks for applying this work to the design of notations (for example, the Cognitive Dimensions framework described by Green (1989).)

Several studies (Cox 1996, Green and Petre 1992) have shown that task performance may be influenced by the way in which the data on which the task is performed is represented. Moreover, there is not necessarily a single, optimal representation for a given data set; different representations may be better for different tasks. Hence, systems that support the use of multiple representations of the same data may have a considerable impact on overall task performance.

Current programming tools make limited, ad-hoc use of multiple representations; for example, systems such as JavaDoc[4] and DOxygen[5] produce hypertext documentation from source code. Special-purpose notations allow more convenient expression of particular types of software. For example, Lex (Lesk 1975) allows lexical analysers to be produced using regular expressions, and the GUI design tools of Microsoft Visual Studio[6] allow the creation of user interface code via manipulations of graphical elements on a virtual canvas. However, in general, modification of programs must be performed using a single, fixed representation. Particularly, generated code is normally held to be sacrosanct, to be modified only by experts (and even then, the modifications are rarely propagated back to the original notation). A system that permitted a variety of representations to be interchanged freely would allow programmers to select the most appropriate for the task at hand. However, the design of such a system must be carefully considered if it is to be both practical and useful.

1.5 End-User Programming for Ubiquitous Computing

Fulfilling the potential of ubiquitous computing requires that the capabilities provided by the technology may be harnessed by users. As mentioned, this poses design problems far removed from those of conventional computing. In addition, ubiquitous computing moves the use of computers from a relative narrow range of contexts to a far greater one — almost anywhere, and at any time. This makes it far harder for system designers to predict, in advance, how people will wish to use a technology.

End-user programming offers an attractive solution to this problem; designers would no longer need to second-guess users' desires, as the users themselves could customise the behaviour of systems as they wish. However, producing an appropriate end-user programming system in a ubiquitous computing context is not straightforward. In addition to the user interface problems already mentioned, most ubiquitous computing systems are targeted at a wide user population, making it difficult to design a language that meets the needs of all users.

A system that supported multiple languages would allow the provision of end-user programming facilities to meet the needs of disparate user groups. In

[4] http://java.sun.com/j2se/javadoc/
[5] http://www.stack.nl/~dimitri/doxygen/
[6] http://msdn.microsoft.com/vstudio/

addition, the system would allow different languages to be used for different tasks, as appropriate. My work is directed at demonstrating the feasibility and usefulness of such a system.

1.6 Aims

My works aims to achieve the following:

- Produce a theoretical framework to support end-user programming in multiple languages
- Demonstrate the feasibility of this approach by implementing a system based on that framework to support end-user programming in the context of domestic ubiquitous computing, and several example programming languages
- Evaluate the system and languages, both from a technical and a usability standpoint

1.7 Dissertation Overview

The remainder of this dissertation describes my work towards the above aims. Chapter 2 discusses previous work in related fields. Chapter 3 describes the Media Cubes language, an end-user programming system designed for use within the networked home. Chapter 4 proposes a theoretical framework for the integration of multiple programming languages. Chapter 5 describes an architecture based on this framework, suitable for use within the networked home. Chapter 6 describes the implementation of this architecture, including both an execution environment and several example programming languages. Chapter 7 evaluates the system, and Chapter 8 presents the conclusions and discusses future work.

Chapter 2
Related Work

My work brings together several fields, including ubiquitous computing, end-user programming, and programming environments with support for multiple languages. This chapter examines the previous work in these fields, describing the context in which my work was undertaken.

2.1 Ubiquitous Computing

Ubiquitous computing, as described by Weiser (1991) and elsewhere, has the potential to become the most significant revolution in computer use since the graphical user interface. Just as the GUI made computing power far more easily available to non-programmers, ubiquitous computing may be the key to eliminating the need to engage explicitly with a computing device at all, instead allowing the user to focus on the task (as opposed to the method by which it is performed).

It is worth noting that ubiquitous computing is not simply achieved by making computing devices portable (although this is a useful aim in itself, and a prerequisite for many ubiquitous computing systems). Nor is providing such portable devices with connection to a network sufficient (although, again, this is a useful thing to do). Even a portable, connected laptop computer is still an object that demands attention. Weiser contrasts this to the written word:

> *Consider writing, perhaps the first information technology: The ability to capture a symbolic representation of spoken language for long-term storage freed information from the limits of individual memory. Today this technology is ubiquitous in industrialized countries. Not only do books, magazines and newspapers convey written information, but so do street signs, billboards, shop signs and even graffiti. Candy wrappers are covered in writing. The constant background presence of these products of "literacy technology" does not require active attention, but the information to be conveyed is ready for use at a glance. It is difficult to imagine modern life otherwise.*
>
> (Weiser 1991)

While it is indeed true that the written word pervades all aspects of life in the industrialised world, a significant number of people cannot read with the facility implied. Those who are partially sighted, dyslexic, lack access to education, or have difficulty reading for some other reason do not have access to information in such an effortless fashion, and consequently face substantial obstacles in many aspects of modern life. Nevertheless, for the majority of people, "literacy technology" has disappeared into the background; the goal of ubiquitous computing is to make computing technology do the same.

Ubiquitous computing has been approached in a variety of ways. One of these is to construct systems that are *context aware*, in that they can sense the state of their environment and modify their behaviour accordingly. This allows

Figure 2.1: Positioning technologies. Clockwise from top left: a GPS receiver, an Active Badge, an Active Bat, and an Ultrawideband tag (not to scale)

applications to update their state and alert users appropriately, without explicit intervention.

Within the field of context awareness, a common theme is *location awareness*, where the context in question is the location of users and devices. “Location” in this context can mean a variety of things, including absolute or relative physical position (expressed as a point in some coordinate system) and symbolic position (identifying which of a set of areas the unit is within). Hightower and Borriello (2001) survey a wide variety of location technologies (some of which are shown in Figure 2.1), constructing a taxonomy based on several variables (physical versus symbolic positioning, absolute versus relative, localised versus centralised awareness, identification) and considers factors such as accuracy, scalability, cost and other limitations.

The most widely used system assessed is the Global Positioning System (GPS), a system originally developed by the US Department of Defence, but made available (with artificially lowered accuracy) for civilian use. This system is based on a constellation of 24 satellites (plus three redundant backups) serving an unlimited number of receivers. The receivers calculate the distance to a visible satellite based on the timing of radio signals received from that satellite. If the

distance to four satellites (with known positions) is established, the position of the receiver may be determined using trilateration. In practice, this position is accurate to between one and five metres for longitude and latitude (and less accurate for height). This requirement for line-of-sight to several satellites is the chief weakness of the system; the system does not function indoors, and is impaired in an urban context, where tall buildings shadow large areas from GPS satellites. Nevertheless, the system is widely used by individuals for navigation, and by organisations to track mobile units such as rental cars or emergency vehicles.

Another notable technology surveyed is the Active Badge, developed at the Olivetti Research Laboratory (later AT&T Laboratories Cambridge). Each badge has a unique identifier, and periodically transmits this using an infrared beacon. Fixed base stations equipped with infrared receivers inform the network of the badge identifiers they receive. As infrared does not pass through walls, the fact that a base station receives a particular badge identifier may be taken as evidence that the badge and the base station are in the same room, allowing the system to locate badges (and, by extension, the person or object the badge is attached to) to the level of single rooms. Later systems allow smaller zones, or physical (as opposed to symbolic) positioning, using technologies such as ultrasound (used by the successor to the Active Badge, the Active Bat) or computer vision. Since the publication of the survey in question, several new technologies, notably ultrawideband radio location, have emerged.

A wide variety of applications may benefit from the addition of location awareness; for example, an application may select the printer nearest to a user's current location, without additional action on their part. Moreover, as well as augmenting traditional applications, location awareness makes possible a large number of novel ubiquitous computing applications. One example, from the Sentient Computing project at AT&T (Addlesee et al 2001), is automatic phone forwarding; when users are away from their desks, they have the option of routing calls to their present location, on a per-call basis. This service requires no user intervention to set up or activate, and only minimal intervention when a call is received (the user places his or her Active Bat over a phone and presses a button on the Bat if they wish to take the call, or does nothing if they do not).

A yet more specific example of context awareness, that constitutes perhaps the most useful variant of the technique, is that of directly selecting a device using physical proximity. In the above application, the positioning device is used to indicate the phone at which the user wants to take the call. This need not be active; a system may detect that a user and a device are collocated at a particular time, and customise the behaviour of the device in accordance with the current user's preferences. The Sentient Computing project extends this concept, constructing a time line containing the content created by a user using a variety of devices such as scanners, digital cameras and voice recorders. This content is annotated with context information, such as who else was present, and the names of people in photographs taken by a fixed camera. This greatly simplifies content management without any additional action on the part of the user. The Pepys system (Newman et al 1991) goes further, attempting to construct an "autobiography" detailing the users activities throughout the day, to serve as an aide-memoire. Particular prominence is given to "gatherings", where the user was collocated with other people. The system is intriguing, but suffers from the relatively unreliable and inaccurate nature of the data source (Active Badges), forcing the software to infer events from incomplete evidence. As a result autobiographies were often incomplete or erroneous. Nevertheless, they served as a useful reminder of events, and the system informed future work in the field.

There is a growing body of work investigating the social and psychological factors involved in the design of ubiquitous computing systems and applications. One area of investigation seeks to identify the areas of activity for which ubiquitous computing could provide support. Rodden et al (2003) describe an observational study of users revealed informal "habitats" where certain types of activity are located within the home; for example, mail that has been opened but is likely to be of interest to others is placed on the mantlepiece. Behaviours and protocols such of this may suggest ways in which ubiquitous computing could be worked into a social context.

2.2 Internet Technologies

The Internet arose from work on a decentralised network for military use, but has grown to become a global, general purpose network, widely used for myriad commercial, social and leisure activities. One of the most important factors in the success of the Internet, and particularly its applications, has been the use of standard, easy-to-implement technologies.

One of the most successful Internet applications has been the World Wide Web. This is an extremely simple hypertext system based on retrieving documents via a global address, or Uniform Resource Locator (URL). Documents described using the Hypertext Markup Language (HTML, Raggett et al 1999) may mark sections of the document (usually words or phrases) as *anchors*, and associate them with URLs. When such a document is viewed in a *browser*, activating an anchor (for example, by clicking on it with the mouse) causes the referenced document to be loaded, usually replacing the previous document.

Hypertext Transfer Protocol (HTTP, Fielding et al 1999) is used to retrieve documents via URLS. This is a straightforward and easily implemented request-response protocol. Requests consist of a method, a path to the subject resource, and a protocol identifier. The most common method is GET, which simply retrieves the specified document. Other methods include POST, used to add information to a specified document, and PUT and DELETE, used to edit documents remotely. Responses consists of a response code, a short human-readable explanation of that code, and a protocol identifier. In addition, both requests and responses may optionally include headers, name-value pairs encoded as ASCII text, and a body containing arbitrary data. The body data depends on the method; for example, the body of a GET request is empty, and the body of the corresponding response contains the document requested (or a human-readable error message if the document cannot be provided). Numerous protocols have been based on HTTP; a notable example is WebDAV (Golland et al 1999), a set of extensions providing more advanced support for editing documents on remote servers.

The simplicity of HTML and HTTP was a key factor in their widespread adoption. Similarly, Extensible Markup Language (XML, Bray et al 2000) has seen far wider adoption than its predecessor, SGML (ISO standard 8879: 1986), largely due to its simplicity. XML was originally designed for "marking up" textual data to provide additional structure. An XML *document* is represented as a sequence of Unicode characters. Typically, this sequence is a file, but this is not universal. *Comments* akin to those found in programming language source code may be inserted at most points in the document using appropriate syntax; these are conventionally ignored by processing software. A *preamble* may be included to specify the version of XML used, or to provide a *Document Type Definition* to constrain the structure of the document. Following the preamble is a single *element*, known as the *document root*. This may contain additional elements,

which may in turn contain elements, and so on, resulting in a tree structure. Elements have a name, and may optionally be annotated with attributes in the form of name-value pairs. In addition, elements may contain text in the form of unstructured character sequences.

While XML's design goals and terminology are heavily skewed towards the processing of documents, it has also been widely adopted as a standard for interchanging other types of structured data. Examples include Internet standards such as SOAP (Gudgin et al 2003) and WSDL (Christensen et al 2001), data formats such as MathML (Carlisle et al 2003) and Chemical Markup Language (Murray-Rust et al 2000), and business process description languages such as BPEL4WS (Andrews et al 2003). While other standards, most notably ASN.1 (Dubuisson 2000), provide generic mechanism for the interchange of structured data, they are notoriously difficult to realise. In particular, discrepancies between implementations make interchange between them problematic, meaning that such standards are more suitable for use within an organisation, as opposed to between organisations. This is at least partially due to the fact that such standards are based on binary data. XML is a far simpler standard, and is text based. As a result, it is far easier to develop a fully standards-compliant processor; indeed, numerous implementations are widely deployed, with few interoperability problems reported. A significant disadvantage of XML, compared to binary formats, is its relative verbosity. While this can be ameliorated, to an extent, using compression, this is an additional processing cost. Furthermore, parsing XML is typically more costly than parsing a binary format. Hence, the use of XML represents a trade-off between runtime efficiency and ease of interoperability.

In addition to the core XML standard, there are numerous standardised technologies for the manipulation and processing of XML. XSLT (Clark 1999) is a programming language for describing transformations from one XML document to another. DOM (Le Hors et al 2003) provides a standard document representation that may be produced by parsers, allowing an application based on DOM to use any parser supporting the standard. XPath (Clark and DeRose 1999) provides an expression syntax for referring to parts of an XML document. As well as specifying a path from a known node (usually the root node), predicates may be used to provide more exact criteria. For example, the following expression refers to all `title` elements that are children of `biblioentry` elements, and have at least one `biblioset` or `confgroup` sibling:

```
biblioentry/title[../biblioset or ../confgroup]
```

The simplicity of XML, combined with the rich set of supporting technologies available, makes it a good choice for many Internet applications. As well as the new applications described above, HTML, originally a dialect of SGML, has been reformulated as a dialect of XML, in order to take advantage of these factors.

2.3 Home Area Networks

There has been much recent interest, in both industry and academia, in the possible application of networking technology in a domestic setting. At the simplest level, this can simply mean making existing networking technologies such as Ethernet or WiFi (IEEE 802.11a/b/g) available in the home. When combined with a high capacity external Internet connection, this can provide a significant improvement over single host, dial-up connections, making a far wider variety of services feasible and convenient. Examples include multimedia services that rely on the transmission of large amounts of data, and services that are used

for brief intervals, for which the relatively long set-up time of a dial-up connection is a disincentive. Work in this area focuses on adapting this technology to make it more appropriate for home use: for example reducing the difficulty of installation and maintenance, as there is unlikely to be a professional systems administrator in attendance. To this end, manufacturers have used technologies such as DHCP (Droms 1993) and ZeroConf (Williams 2002) to produce "smart appliances" that configure themselves, automatically determining the specifics of any network they are connected to.

A less direct approach is to design systems that support networked control of existing appliances in the home. X10[7] provides a commercial product that allows users to control electrical devices such as lamps via a PC, and allows the integration of cameras and other sensors. Communication is over existing power lines, or via wireless links. However, device control is, in almost all cases, limited to turning on and off. This is because of a lack of a standard for the control of devices. There are some exceptions; most notably in the field of infrared remote controls. While not completely standardised, there is enough commonality, and enough information in the public domain, to allow manufacturers to produce "universal" remote controls that allow a user to user a single control to operate several devices (in practice, audio-visual components), even if those devices are from different manufacturers. Some units add the simple facilities to perform timed operation. The RedRat[8] is notable in that it allows consumer electronics devices to be controlled from a PC. This offers the possibility of arbitrarily complicated control, but is aimed at the technically sophisticated user.

There are several efforts to produce a standard by which devices can be networked and controlled, thus allowing uniform solutions to be developed, as opposed to the current, ad hoc, ones. One of the most promising of these standards is Universal Plug and Play[9] (UPnP). It is produced by a consortium with several hundred members, mostly device manufacturers and software companies, but including some academic institutions. The intention is that devices may be added to a network, and automatically discover and connect to other devices without requiring the user to install device drivers or manually configure connections.

UPnP is based on standard Internet technologies such as HTTP and XML, described in the previous section, and covers six areas: addressing, discovery, description, control, eventing and presentation. Addressing allows devices to obtain an IP address automatically, either via the centralised DHCP protocol, or, if there is no DHCP server available, via a decentralised protocol named Auto-IP. Once a presence on the network has been established in this way, the discovery layer allows devices to search for other devices on the network that provides services it is interested in, and advertise services that it itself provides. Hence, appropriate connections between devices may be made between devices without user intervention. Once these connections have been made, the description layer allows more detailed information about devices to be exchanged. This includes information such as the device name, manufacturer and model, and, importantly, a specification of the interfaces of the services that the device provides.

Control and eventing are concerned with communication between devices; these provide a set of generally applicable technologies for communication between devices. Control is based upon the synchronous, RPC-like SOAP protocol; arguably, asynchronous communication is more appropriate in a ubiquitous com-

[7] `http://www.x10.com/`
[8] `http://www.redrat.co.uk/`
[9] `http://www.upnp.org/`

puting context (Saif and Greaves 2001). Eventing is based on the asynchronous GENA protocol, but is only intended to be used for tracking changes to the state of devices. Presentation is concerned with presenting an interface to the user. At time of writing, it is limited to providing an HTML-based web interface. While this is sufficient for a variety of situations, it is not well suited to novel presentation methods that may be found in a ubiquitous computing context, for such as verbal input and aural rendering.

The intention is that UPnP is embedded in every device. This is perhaps optimistic, as the standard's minimum requirements include an IP stack. This is feasible in many devices (for example, televisual equipment) that already contain microprocessors, but is likely to be an obstacle in simpler devices such as coffee machines and alarm clocks, where both overall cost and profit margin are far smaller. In addition, UPnP relies heavily on XML, a relatively verbose data representation, and HTTP, a protocol with significant overhead for small messages. While there are numerous advantages to these choices, they also serve to increase the memory and processor requirements for UPnP devices, and as such may be considered inappropriate for very small footprint devices.

The AutoHAN project (Blackwell and Hague 2001) is a continuing project examining all areas of Home Area Networking, focusing on technologies that make a home area network practical for and usable by the typical home user. Possible uses for such a network include routing music and video to any room in the house, automating devices such as lamps and speakers to react to users' presence and actions, and allowing tasks to be performed at a certain time of day, or when certain conditions arise.

The project has investigated the reuse of legacy power and telephone wiring for high-speed data transfer, allowing home networking technology to be used where installing new wiring is inconvenient or impossible (for example, listed buildings). Many of these problems may be solved with wireless networking; however, this has the disadvantage of interference, leading to highly variable quality of service. In addition, wireless technologies are typically slower than their wired counterparts, and carry additional security implications in that the signal may be overheard. As wireless networking technology, in particular spread spectrum and ultrawideband technologies, improves, the speed and quality of service concerns will be lessened, but reuse of legacy wiring is often a valuable medium-term solution.

The project has also developed a novel variant of ATM (Asynchronous Transfer Mode) to allow multimedia data to be streamed around the home. This provides the quality of service guarantees of normal ATM without the need for complex routing hardware. This technology has been integrated into radio tuners, CD players and amplifiers, allowing high-quality audio data to be transmitted around a home network under computer control. However, while the system is technologically capable, the problems associated with interfacing (circuit-switching) ATM with the packet-switching networks that have become a de facto standard in the marketplace mean that such a technology is likely to be limited to niche markets, much like standard ATM.

Early in the project, it was noted that the user interface of the system was critical to its success. Various techniques were investigated. My work has taken place within the context of this investigation; specifically, in the provision end-user programming facilities for home networks. This resulted in the Media Cubes language, described in Chapter 3, and subsequently the Lingua Franca multi-language programming system, described in Chapter 5 and Chapter 6.

2.4 Approaches to End-User Programming

If an end-user programming system is to be successful, the traits of the target user population must be taken into consideration from the outset. In most cases, the target users will be significantly different from those of a conventional programming language (usually professional programmers). As a result, various approaches have been taken that differ to a greater or lesser extent from those taken in traditional programming contexts.

2.4.1 End-User Programming in Traditional Environments

The observations by Nardi quoted in Section 1.2 embody the prevailing view of end-user programming in offices and similar environments, where it is traditionally deployed. In these cases, the purpose of end-user programming is to enable users to create programmatic abstractions without requiring them to expend undue effort. Such abstractions allow the user to customise applications to suit their individual needs, without the need to either hire professional developers, or alter their work practices to accommodate the vagaries of the software. End-user programming is not without its problems, but offers significant benefits.

A common approach is to use a simplified language, with a feature set tailored to the application at hand, and omitting or concealing any complexity that is not immediately relevant. The language used may be ad hoc, or it may be an existing language, such as Visual Basic or Scheme, adapted to suit the specific application in question. In essence, such an environment provides a programmatic way to access functionality that is already available to the user via another mechanism (for example, a graphical user interface). This has the advantage that the user is already familiar with the functionality in question. It also opens up the possibility of "Programming by Example", described the next section.

Such environments allow users to automate repetitive tasks, potentially increasing both efficiency and accuracy. However, these long-term benefits must be offset against the short-term cost of actually creating the script, and the risk that the script will not function correctly, or will not be of enough general use that effort saved will be greater than the cost of creating it. On the basis of these criteria, the *Attention Investment Model* (Blackwell 2002) models the decision process regarding making (or not making) abstractions as an economic decision, where the scarce resource is the user's time or attention. Hence, when the user is assessing the viability of creating a particular abstraction, they estimate the cost of creating the abstraction, the benefits that it will bring, and the probability that it will be successful in bringing those benefits (or, conversely, that the process will fail, and result in costs with no benefits). An important point to note is that it is the *perceived* cost and risk that are important in this calculation. Hence, a system that is perceived to be difficult (costly or risky) to program will greatly discourage abstraction-making, regardless of the actual cost or risks involved.

In most cases, end-user programming systems of this kind are used by a minority of users, so called "power users" or "gurus". Such users can and do create large and complex applications, and in some cases come to rely heavily on them. However, when the task moves from straightforward automation of actions previously performed by a user to the creation of large, complex pieces of software, a system designed for the former proves unsuitable for the latter. Nevertheless, it is possible to construct systems to support the creation of complex software solutions by individuals who have little or no formal training in software engineering. For example, Rothermel et al (1998) describes a method by which

an environment may provide end users with the facility to test their programs, an important task as systems and the programs created with them become more sophisticated and complex.

2.4.2 Programming by Example

A promising approach for end-user programming in many situations is *Programming by Example* (also known as *Programming by Demonstration*). This involves the user performing the task to be automated one or more times, usually using a direct manipulation interface. This sequence of actions is observed, and used as the basis for a program. The following quotation from Cypher expands on the rationale behind this:

> *The motivation behind Programming by Demonstration is simple and compelling: if a user knows how to perform a task on a computer, that should be sufficient to create a program to perform the task. It should not be necessary to learn a programming language like C or BASIC. Instead, the user should be able to instruct the computer to "Watch what I do", and the computer should create the program that corresponds to the user's actions.* (Cypher 1993)

The most straightforward form of programming by example is a macro recorder that simply records a sequence of actions, and repeats them verbatim at some later time, when the user issues a certain command. Little or no attempt is made to adapt the actions to the new context. Despite their simplicity, macro recorder systems are adequate for a surprising number of automation tasks in many applications. In addition, the straightforward model of programming is relatively easy for users to grasp.

An obvious extension to this technique is to allow users to edit the recorded macro. In most cases, the purpose of this is to make the macro more generally applicable by allowing it to adapt to the context in which it is invoked. If a user is to edit the macro, it must be presented in some way; usually, this is in the form of a conventional, text-based programming language of the type mentioned above. If this language is sufficiently general, as is the case for Visual Basic for Applications[10], then the macros may provide a basis for larger, more complex scripts. This also provides a path from macro recording to the devlopment of complex programs using the scripting language.

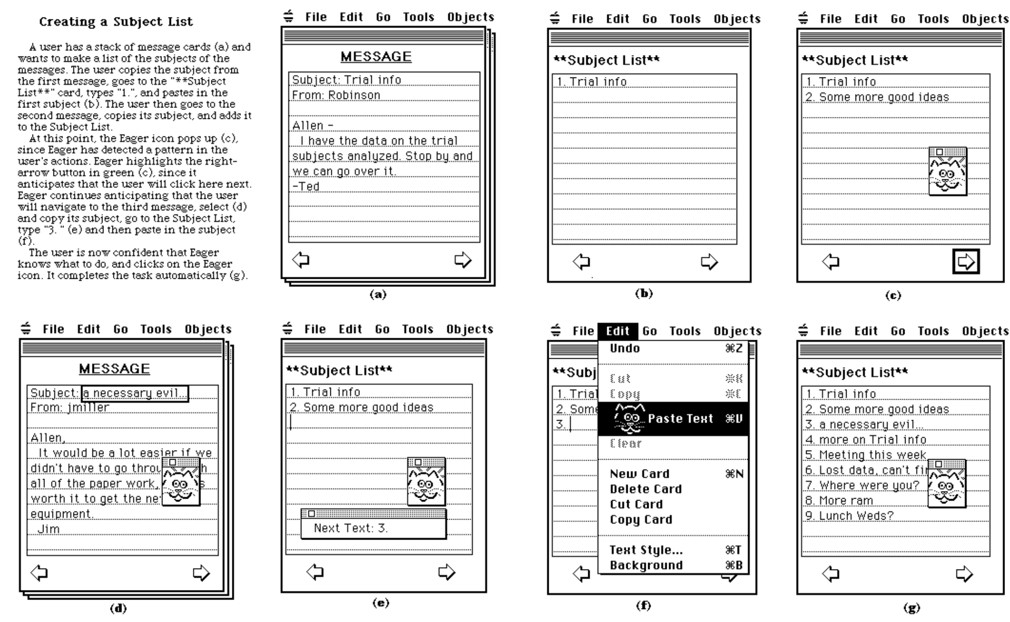
An interesting variation on this theme is found in the system ToonTalk (Kahn 1996), an end-user programming environment aimed at school-aged children. Due to the target audience, the system presents elements such as data items and programs as animated characters in an environment comparable to a video game (Figure 2.2). Programs are represented as robots that "learn" by "watching" user actions; to train a robot to perform a specific task, the user performs the intended actions on some input via the existing direct manipulation interface. When the robot subsequently sees the same input, it will perform the same actions that the user did.

Used in this way, the system acts as a simple macro recorder, with the additional constraint that the macro will only work in a single context (it will only work on a specific input value). However, once a robot has been trained in this manner, the user may generalize it by selectively removing parts of the input accepted. This relaxes the constraints on acceptable input, allowing any value to

[10] `http://msdn.microsoft.com/vba/`

Figure 2.2: **The ToonTalk programming environment**

Figure 2.3: **The EAGER programming by example system.**



be used in the specified place. This technique is analogous to the unification of logical variables in programming languages such as Prolog (Sterling and Shapiro 1994), and, despite its simplicity, is sufficiently general to allow the user to specify arbitrarily complex programs. It is also arguably less prone to errors than modification of a traditional language by a user not familiar with that language.

A different approach to Programming By Example is to attempt to infer programs automatically from observations of user actions. This may be done with or without user intervention. An early example of such a system is Eager (Cypher 1993, Chapter 9), shown in Figure 2.3. This system consists of an agent that monitors user actions, and, when it detects a repetitive sequence, offers to continue it for the user. If this offer is accepted, the agent steps through the next iteration of the sequence as it has inferred it, giving the user the chance to correct any incorrect actions. This process may be repeated for as many iterations as the user feels is necessary. Once they are happy that the agent has inferred the correct actions, it can be commanded to finish the sequence without further intervention. The user can also, at any time, cancel the agent's actions and return to the previous state.

One important aspect of Eager is that the agent is constantly monitoring user activity; no user action is required to begin the inference. Another is that the agent does not take any action without explicit indication from the user, allowing them to ignore the agent if and when they wish to. Finally, when the user does decide to allow the agent to proceed with the inferred actions, they may correct or reverse those actions at any time. An alternative approach, taken by the predictive calculator described in Chapter 3 of Cypher (1993), constantly offers the user an inferred action (with no possibility of modification), and only requires minimal interaction to accept this. In both cases, however, it is paramount that an incorrect inference is not harmful (in the sense that it costs the user undue effort or, worse, destroys any of the user's data.) Inference-based systems such as these have been successful within limited domains, but less so when applied to more general problems.

A similar training process in much wider use is that associated with Bayesian filtering of e-mail to remove Unsolicited Commercial E-Mail (UCE, or "Spam") (Graham 2004), as used in products such as Mozilla Thunderbird[11] and SpamAssassin[12]. This technique, based on naive Bayesian inference (Lee 1997), assigns to each word in an incoming message a probability that the message is UCE, based on the occurrence of that word in previous UCE messages. If the aggregate probability is greater than some threshold, the message is tagged as UCE. This is a very effective technique, but requires a large corpus of messages identified as UCE (or not) to generate the probabilities. While manufacturers can provide a seed corpus, this is unlikely to identify all messages that a user considers UCE. Hence, the software provides the facility for the user to explicitly correct assertions by indicating that a particular message labelled as UCE is in fact legitimate, or vice versa. The probability associated with the words in the message may then be adjusted accordingly. Over time, the probability table is adjusted to match the user's particular pattern of mail, and the accuracy of the software increases. Furthermore, each user's training set will be different, making it infeasible to select "known good" words that provide strong evidence that a message is legitimate, the inclusion of which would allow a sender to effectively bypass the filter. When the user is satisfied that the match is sufficiently accurate, he may instruct the software to remove UCE messages without intervention. In practice, these messages are not deleted, but placed in a "Junk" folder. This allows the user to check that the software is functioning as desired, and correct it via further training if it is not.

[11] `http://www.mozilla.org/products/thunderbird/`
[12] `http://www.spamassassin.org/`

2.4.3 Visual Programming

A *visual language* is described by Marriot et al (1998) as "a set of diagrams which are valid *sentences* in that language, where a diagram is a collection of *symbols* in a two or three dimensional space". This is reminiscent of Chomsky's definition of (spoken and textual) language as a set of valid *strings* of symbols. Hence, the key distinction drawn between visual and textual languages is that textual languages are linear (sequential, one-dimensional), whereas their visual counterparts are expressed in a two- or three-dimensional medium. Other properties, such as colour, may also be significant in some visual languages. *Visual Programming* (VP) systems use visual as opposed to textual languages to express programs. The distinction is not always clear-cut; for example, a small number of "textual" languages, notably Python (Lutz 2001) and Haskell (Peyton-Jones 2003), make some use of the two-dimensional arrangement of tokens, specifically the indentation of lines of text. In this respect, they may be considered visual languages, albeit ones in which the visual component is of limited flexibility.

Blackwell (1996) has surveyed and categorised the perceived benefits (termed *metacognitive theories*) of VP, and notes a "remarkable consistency of metacognitive concerns amongst VP researchers". The theories identified are based in many areas, including introspection by researchers, cognitive theory and folk psychology. Blackwell concentrates on the theories that Visual Programming practitioners advance to support their work, but the field also has its share of detractors. Perhaps most prominent amongst these is Brooks:

> *A favourite subject for PhD dissertations in software engineering is graphical, or visual programming, the application of computer graphics to software design. Sometimes the promise of such an approach is postulated from the analogy with VLSI chip design, where computer graphics plays so fruitful a role. Sometimes the approach is justified by considering flowcharts as the ideal program design medium, and providing powerful facilities for constructing them.*
>
> *Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will.*
>
> *... software is very difficult to visualize. ... The VLSI analogy is fundamentally misleading — a chip design is a layered two-dimensional object whose geometry reflects its essence. A software system is not.* (Brooks 1995)

Brooks softens his view in latter writings, but the overall pessimism remains, and is shared by many in the software engineering community. While VP is certainly not, as is sometimes claimed, a panacea that is better in all respects than conventional programming methods (Brooks' "Silver Bullet"), there is nevertheless convincing evidence that VP is beneficial in certain specific ways. Whitley (1997) surveys numerous studies into various aspects of visual languages, and draws the overall conclusion that visual languages are better than their textual counterparts for some tasks, and worse in others.

One of the major problems associated with visual programming languages is scalability. While they offer many advantages when dealing with smaller programs, it is difficult to extend the applicability to larger programs without sacrificing these advantages. Indeed, this is a chief criticism levelled by Brooks in the passage cited above. Burnett (1995) breaks this problem into nine more specific sub-problems, including procedural abstraction, efficiency, type safety and data persistence. The paper cites numerous examples of ways in which visual

programming environments attempt to overcome these problems, with varying degrees of success. Many of the solutions offered, such as transparent storage of state, are viable. Others, such as the separation of context and detailed views, or static and dynamic views, represent compromises that, while they mitigate certain problems, do not eliminate them. These problems may well be fundamental limitations of Visual Languages. However, in many cases, they are not an issue; in particular, in the field of end-user programming, an obvious application area for visual languages, programs may remain small.

VP systems are often based around direct manipulation, a technique more commonly associated with activities not traditionally thought of as programming. The main characteristics of direct manipulation are, to paraphrase Schneiderman (1983): continuous representation of the objects of interest, "physical" actions, rapid incremental reversible operations whose impact is immediately visible, and a layered or spiral approach to learning that permits usage with minimal knowledge. Consequently, many of the lessons learnt in the design of direct-manipulation based Graphical User Interfaces (GUIs) may be profitably applied to VP.

A common technique employed in GUI design is metaphor. The system is made to mimic another, real-world, system. The advantage of a metaphor is to allow users to "learn by analogy". However, an inappropriate metaphor, in addition to hampering learning by analogy, may cause users to make incorrect assumptions about the system. Furthermore, in some cases, a visual analogy may not provide significant benefits to comprehension (Blackwell and Green 1999).

In general, the application areas in which VP has been used most successfully are those where the problem in question has a large visual/spatial component, a notable example being GUI design. Visual layout and design tools, often found as components in Integrated Development Environments (IDEs), are now used almost universally for this task. While early systems limited the user to specifying simple, absolute layouts, modern systems are considerably more flexible, allowing users to specify complex relationships between layout parameters in an visual manner.

In addition, tools such as Apple's Interface Builder[13] go further, providing graphical representations of programmatic objects outside the user interface, and for connections between objects. This allows a significant degree of "boilerplate" code to be specified graphically, alleviating the need for laborious and error-prone manual coding. Instead of generating the code that would otherwise be generated by hand, Interface Builder stores the specification as serialised objects, allowing the connections to be edited in the same graphical way at a latter time.

2.5 Programming in the Home

When considering programming in the home, it is necessary to have a definition of the term "programming". We chose to define programming as *abstraction* over either *time* or a *class of objects*. This definition includes, in addition to the type of programming traditionally discussed by computer scientists, activities such as setting a central heating timer, and defining a playlist of CD tracks.

Programming, by this definition, is a widespread activity, and is undertaken by an extremely diverse user population. Rode et al (2004) describe an ethnographic study of programming activity in various types of households. The researchers visited volunteers at home in the evening, and brought with them a

[13] `http://developer.apple.com/tools/interfacebuilder/`

meal to share. This provides an opportunity to build up a rapport with the participants, and allows informal conversations that frequently reveal additional information pertinent to the area studied. Following the meal, the participants performed several classification tasks based on the programmable appliances in the home, using low-fidelity props to as a tool for discussion. This was followed by a questionnaire and debriefing.

The study found that “ahead of time” programming (for example, timed recording with a VCR) was more common than “repeats easy” abstraction (for example, programming a telephone number into speed dial). Programming activity was equally common amongst men and women. Some gender differences were observed in the types of appliance programmed, but the exact nature of these was ambiguous, and the authors state the need for further study to investigate these.

An important point revealed by the informal conversations with the participants is the wide variety of consequences of failure to program an appliance correctly. These range from minor irritation (such as the failure of a VCR to record a particular program) to the disastrous and potentially lethal; participants voiced the fear that an incorrectly programmed oven would burn down the house (the likelihood of this actually happening was not addressed, but, regardless of this, the fear on the part of the end user is significant in that it affects their decision as to whether to program a particular device or not). This parallels the range of failure consequences in traditional programming activities (ranging from trivial software such as screen savers to safety-critical systems such as air traffic control), but in this case, the participants are not professional programmers, and are not being paid to program, and programmable systems in the home should be designed with this in mind. The authors conclude:

> *These findings suggest that even where programmable features are difficult and risky to use, users will persevere in the face of adversity, if they have a real need for the feature. However, where there is no real need for programming, users will not bother. Thus, while programmable features may be included in items like ovens and breadmakers because they are considered selling points, these features may not in practice enhance the usability of the appliances. If such features are considered desirable, or are essential (as is the case with VCRs), perhaps designers should focus on reducing the chances of failure, and/or the associated risks.* (Rode et al 2004)

Programmability would seem to be particularly useful in the networked home, where numerous complex devices can interact to provide enhanced functionality. However, there has been relatively little work in this area. One example of such work is the Accord project (Rodden et al 2004). This project has three linked objectives; developing new tools and methods for the embedding of computation in everyday objects, investigating the ways in which new functionality and new use can emerge from interacting collections of such objects, and ensuring that people’s experience of these environments is both coherent and engaging in space and time. The project cites the work of architectural historian Stewart Brand, in which the evolution of a building is characterised by the interplay of the ”six S’s” (Site, Structure, Skin, Services, Space-plan and Stuff). In particular, it focuses on the interplay between space-plan (the layout of the home) and stuff (movable artefacts within the home).

The project has developed a component model to represent devices in the networked home. While many projects focus solely on such architectural issues, the Accord project is also concerned with allowing users to control and reconfig-

ure their home networks. Accordingly, the This model is exposed to users using a simple language based on "jigsaw-like pieces", allowing the users to interact with the model, and dynamically compose and arrange devices and services. The pieces in question may be physical devices, or elements on a screen. The project does not aim for a high degree of complexity; all of the jigsaw pieces presented have one or two connections to other pieces, and limited functionality. Some pieces act as simple triggers (for example, those corresponding to devices such as doorbells or motion sensors); others act as producers or consumers of data (for example, a camera and a display, respectively). In spite of this simplicity, useful applications may be constructed by combining preexisting components in novel ways.

The CAMP system (Truong et al 2004) aims to provide a more sophisticated programming facility for a more specific type of ubiquitous computing application in the home; that of media capture and access (for example, taking pictures when a certain set of conditions are true). This is achieved via a "magnetic poetry" metaphor (although actual magnetic poetry is not used; instead, pieces are simulated within a conventional GUI). Users specify conditions and actions by constructing "sentences" from a restricted, domain-specific vocabulary, represented as a set of discreet graphical elements ("magnets"). This representation serves to communicate the restrictions of the vocabulary to the user in a natural manner. The user selects a set of magnets to form a sentence, which is interpreted by the system as a condition. The restricted input greatly simplifies the interpretation, and feedback is provided in the form of a natural language description based on the interpreted rule.

Unlike many ubiquitous computing systems, CAMP rules are specified in terms of tasks and goals, as opposed to devices. This is arguably preferable form the user's point of view, but may make it more difficult to relate rules to concrete actions performed by the system. User studies suggest that the CAMP system is easy to learn and effective for the type of applications addressed. The authors mention that they intend to apply the techniques used in the CAMP system to other types of ubiquitous computing applications. While this is a promising approach in relatively simple cases, it is not clear that the same techniques would be applicable to more complex cases without adding significant additional structure (and hence complexity) to the representation used.

2.6 Using Multiple Representations

There is a great deal of empirical evidence that the way in which information is presented has a dramatic impact on the ease with which it can be used. Whitley (1997) surveys a large number of psychological studies that suggest that, for a wide variety of tasks, subjects perform more quickly and accurately if data is presented to them in one notation rather than another. For example, Day (1988) asked users to answer questions about when to take various kinds of medication throughout the day, based on instructions provided in one of several forms. Subjects were found to be significantly more accurate (76% correct as opposed to 56% correct) when the information was presented in matrix form, as opposed to a linear list. He found similar differences in tasks relating to cursor motion in text editors and to bus schedules; in all three cases, a visual/spatial representation of the information lead to more accurate task performance. Speed, where studied, was also improved. In an expanded study of text editing, the benefits of using a visual notation were found to be even more pronounced.

Several other studies have shown that in specific cases a visual representation of the data involved significantly inproves task performance. However,

Figure 2.4: Forward and backward representations for conditional logic. Adapted from Green and Petre (1992)

```
if high:
  if wide:
    if deep: weep
    not deep:
      if tall: weep
      not tall: cluck
      end tall
    end deep
  not wide:
    if long:
      if thick: gasp
      not thick: roar
      end thick
    not long:
      if thick: sigh
      not thick: gasp
      end thick
    end long
  end wide
not high:
  if tall: burp
  not tall: hiccup
  end tall
end high
```

```
howl:    if honest & tidy & (lazy | sluggish)
laugh:   if honest & tidy & ¬ lazy &
         ¬ sluggish
whisper: if honest & ¬ tidy & ( nasty &
         greedy | ¬ nasty & ¬ greedy)
bellow:  if honest & ¬ tidy & nasty &
         ¬ greedy
groan:   if honest & ¬ tidy & ¬ nasty
         & greedy
mutter:  if ¬ honest & sluggish
mutter:  if ¬ honest & ¬ sluggish
```

it is not true that a visual representation is superior in all problems, nor is it the case that for a given set of data there necessarily exists a "best" representation that is unambiguously superior to all others over all tasks. Green and Petre (1992) studied user performance on representations of conditional logic represented using either forward notation ("if-then-else") or backward notation ("do-if") (Figure 2.4). The former was found to facilitate questions where the subject is given a set of inputs and have to find the resulting output, whilst the latter facilitated questions where the user worked from outputs to inputs. Other notations, such as decisions diagrams, may outperform either notation studied; nevertheless, the study demonstrates a non-trivial relationship between representation and task performance. In another study, McGuinness (1986) found that,

for familial data represented as either a matrix or a tree, the matrix representation resulted in significantly better performance on questions regarding cousin relationships, but neither representation was significantly better when the task concerned inheritance.

This matching between task and notation may be exploited by providing a way for users to switch between representations of a set of data whilst performing a task. Cox (1996) details a number of studies of subjects performing analytical reasoning problems of the type found in the GRE exam (used by US Graduate Schools to assess applicants). An initial study examined the "work scratchings" (informal notes) of subjects answering questions taken from the tests. This revealed a variety of representations used for the task, and a degree of correspondence between the nature of the question and the representation used.

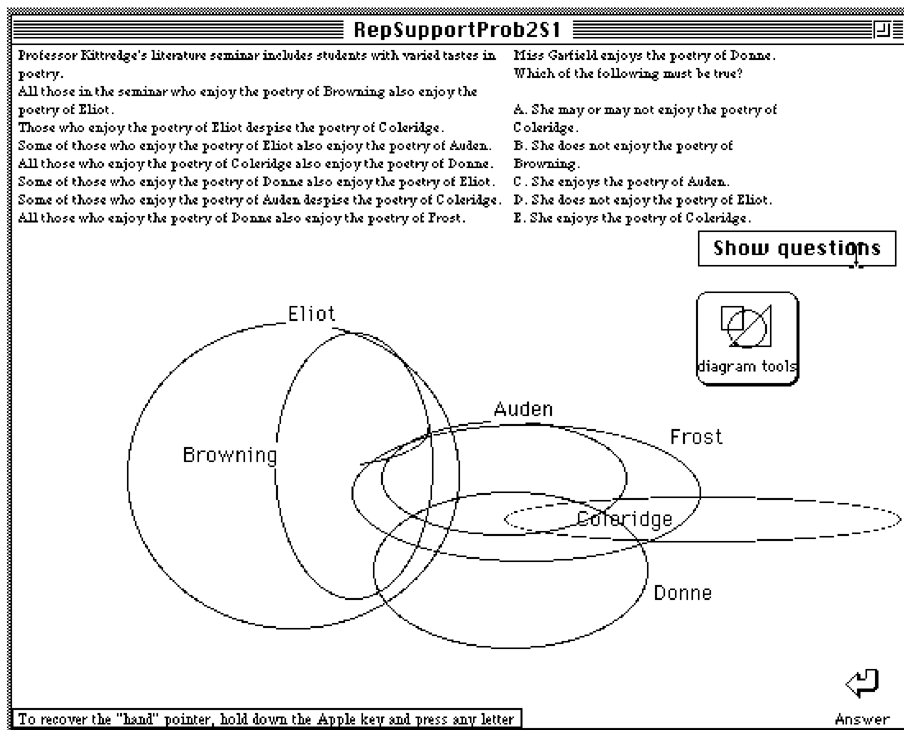
Based on this study, Cox constructed a software system, *switchERI* (shown in Figure 2.5), to provide support for use of multiple representations when solving analytical reasoning problems. This system made a set of environments available to the user, each supporting a different kind of representation, and allowed the user to switch between these environments at will. A study of users using this software showed that, while often beneficial, switching between representation was associated with a high cost. One feasible technique to reduce or eliminate this cost would be the *co-construction* of representation — having the system reflect changes made to one representation in the others. A system implementing this idea, *switchERII* (also shown in Figure 2.5), was constructed and tested.

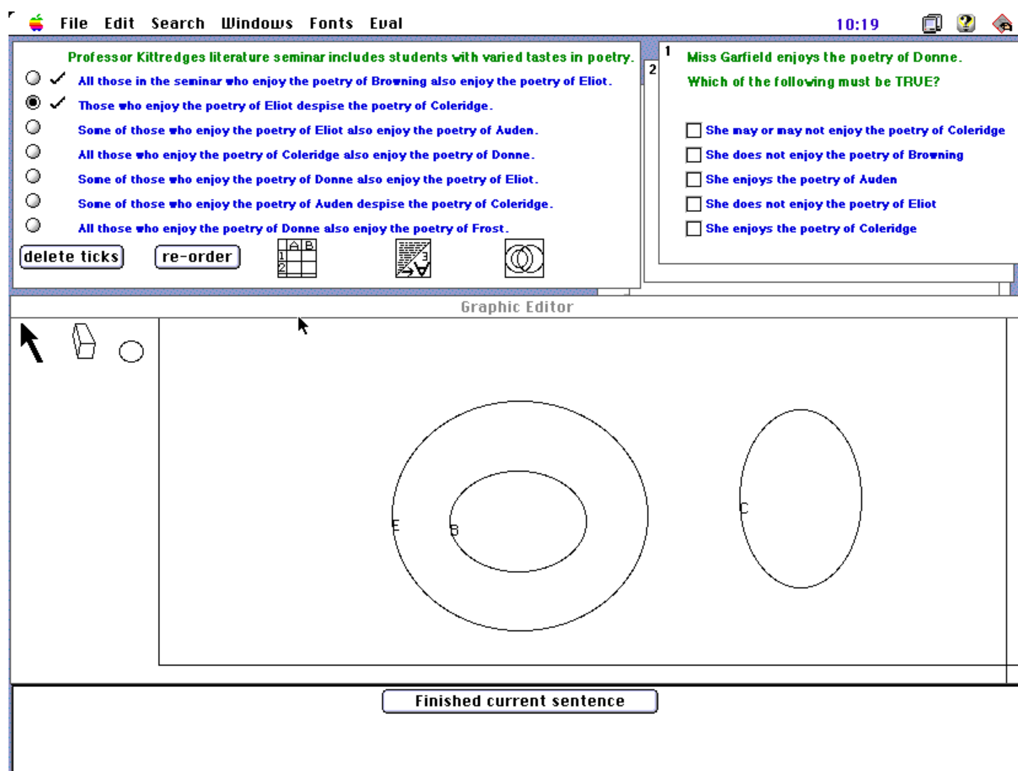
swithERII provides several external representations of the problem data (specifically, GRE analytical reasoning problems). The initial set of data is provided by the problem, but additional facts can be added as the user infers them. The key innovation is that, in some cases, changes in one representation of the data are reflected other representations automatically. This allows the user to switch between those representations at any time, with low cost. In addition, the software provides "coaching" hints that suggest the user may wish to switch to another representation if the current one is likely to be inappropriate.

The results of the study suggest several important facts regarding switching between representations. Firstly, even with the significant software support provided by *switchERII*, switching costs the user in terms of time (although less time than it would without software support). The subjects were divisible into two groups. One group, who understood the semantics of the external representations used (in particular, Euler set diagrams as shown in Figure 2.5), were able to switch "judiciously" between representations, taking advantage of the particular expressive properties of each; this is reflected in good performance. Conversely, subjects less familiar with the representations used were observed to "thrash" between representations, swapping chronically in a way Cox likens to the behaviour of virtual memory systems operating near the limit of actual memory available. While the level of switching is similar to that of the former group, it is not reflected in improved performance.

Cox notes several ways in which the system could be improved. One is that the representations are only available serially. Making multiple representations available concurrently may be advantageous, and is not significantly more difficult from a technical standpoint. However, it was not provided in *switchERII* as the user study required the experimenters to be able to observe the act of switching between representations; only allowing the viewing of one notation at once means that switching between notations is an observable act. He also mentions that providing *switchERII* with the ability to parse and interpret the user's diagrams would enable it to provide a greater level of support to switching between

Figure 2.5: **The *switchERI* (top) and *switchERII* (bottom) Intelligent Learning Environments**





representations. Cox terms this *intelligent* co-construction of diagrams.

It should be noted that the problems to which the *switchER* systems applied were both simple and highly constrained. While analytical reasoning problems are by no means trivial, and test some of the skills required in programming (such as problem comprehension and planning), they are on a far smaller scale. Nevertheless, it seems reasonable to investigate the possibility that co-construction of representations may be a profitable strategy for larger, more complex problems.

2.7 Conventional Systems for Integrating Multiple Languages

Since the invention of FORTRAN in 1954, programming languages have proliferated. Today, there are literally hundreds available to the programmer. These range from highly specialized languages such as Inform (Nelson 2001), used for the creation of interactive fiction, to C (Kernighan and Ritchie 1988), which has been used to tackle almost every programming problem imaginable. Each is more suited to some problems than others. However, in many cases, different parts of a problem suggest different programming languages. This has lead to the development of techniques to combine multiple languages.

One approach to integrating multiple languages is to allow code written in those languages to communicate via some common, well-defined interface (see, for example, COM (Williams and Kindel 1994)). This approach leads naturally to the provision of this interface over a network. Communication over a network introduces many issues that are not present in the local case, but has proved a useful mechanism for both distributed and multi-language projects, and has lead to several widespread implementations, including RPC (Birrel and Nelson 1984), CORBA (Yang and Duddy 1996) and DCOM (Horstmann and Kirtland 1997). However, such systems traditionally involve a precise specification of the interface, leading to problems both for communication across languages, and, particularly, for communication between organisations. Recent work in this area has lead to mechanisms supporting more flexible definitions of interfaces, notably Web Services (Booth et al 2004).

One problem common to all of the above systems is that, to a greater or lesser extent, the common interface provided is at a relatively course level of granularity, in that the overhead involved in inter-component communication makes small components infeasible. More seriously, such systems provide only a subset of the features available in the individual languages. In many cases, this trade-off is acceptable, and allows for greater flexibility in the choice of language for various components. However, if the goal is tight integration of components written in multiple languages, within a single organisation, these systems fall short.

In such situations, the common approach is to compile source code in multiple languages into the same intermediate form, allowing integration at the linking stage. Languages are not required to provide access to every feature made available by the intermediate form, and hence it may support disparate feature sets from differing languages (in effect, supporting the union of feature sets of the source languages, as opposed to the intersection of those features sets). Also, there is generally a smaller overhead involved than with systems based on explicit communications. Hence, such systems allow developers to implement different parts of a project in different languages at a finer granularity than would otherwise be possible. An early example of this technique, still in common use, is to

link program logic written in C or C++ (languages that support structured programming) to numerical code written in FORTRAN (which, due to its simplicity, can be compiled into very efficient code, especially on parallel architectures).

A related mechanism is commonly used by scripting languages such as Perl, Python and TCL (Ousterhout 1998). These languages provide an interface to a systems programming language, almost always C. This facilitates the use of the systems language to write extensions that would be cumbersome, inefficient, or impossible to write in the scripting language, while allowing the use of the scripting language to write the bulk of the program logic or "glue". Similarly, where an interpreter is embedded in a program written in the system language, it allows that program to be extended using the scripting language, which is often more suited to the task.

Despite the tight integration that such systems make possible, there exist difficulties when integrating languages that have significant differences in terms of data representations passed between components (this is one of the problems also faced by communication-based systems). This can be alleviated by harmonizing the data representations, function call protocols, and type systems used in such a language. A straightforward way to achieve this is to structure one language to ape the structure of another at some level; for examples, see TCL, which is designed to interface directly to C, and Jython[14], an implementation of the Python language that uses the object system of Java.

The recently introduced .NET framework[15] represents a more structured version of this approach. In addition to a common intermediate representation, it defines a common type system, and a common standard library. This has allowed compilers for a wide variety of languages to be targeted at the common environment. These languages may be integrated to a very fine granularity; for example, it is possible to define a class in Managed C++, subclass it in Visual Basic, and instantiate and use it in COBOL. This gives programmers great latitude in the choice of implementation language for any given component, allowing them to choose the best language for the job.

Another notable system that allows fine-grained interoperability between components implemented in different languages is the Rainbow system developed at the University of Manchester (Barringer et al 1997). This system allows users to design hardware systems using a combination of several languages (named Red, Yellow, Green and Blue). As with .NET, interoperability is achieved via the use of an intermediate form. The Green language is interesting in that it provides two "views"; visual and textual. While this falls short of allowing the user to view a program in an arbitrary language (for example, a program written in Green cannot be viewed as Yellow), it nevertheless provides a concrete example of the use of multiple representations of the same source code in a practical system.

Multi-view development environments are intended to integrate multiple representations of the same source code via the use of software visualisation techniques (discussed below). Such environments are described in Section 2.8.2.

2.8 Generation of Multiple Representations from a Single Source

As discussed in Section 2.6, multiple representations of the same data, with differing properties, can complement each other when used for problem solving.

[14] `http://www.jython.org/`

[15] `http://msdn.microsoft.com/netframework/`

There are numerous other circumstances where multiple representations of the same data are necessary or beneficial; for example, on-line and printed presentations of a document. However, as noted by Cox, there is often a high overhead involved with producing these multiple representations. It is therefore desirable to produce multiple representations from a single data source automatically.

The problem of producing target documents in multiple formats from a single source document is a common one, and has been approached in various ways. One approach is to add a converter to a new format to a tool used to produce an existing format. Examples include the augmentation of tools for producing printed output (for example, LaTeX and Microsoft Word) with capabilities to "export" documents to HTML format (Raggett et al 1999) for publication via the World Wide Web (specifically, external tools such as HyperLatex, and the built-in HTML Export facility of Word).

This approach is attractive, as it involves relatively small adjustments to an existing and familiar tool, but is a less than perfect solution. Given the varying capabilities of the existing and new representations, there may be aspects of the former that cannot adequately be represented in the latter. Additionally, it is difficult to take advantage of features present in the new representation but not the existing one, as the tool is designed with the latter in mind. This problem can be mitigated to an extent by adding features to the tool to cope with the new representation (e.g., the ability to add hyperlinks in Word documents), but the tool remains primarily grounded in the original representation.

An alternative approach is to produce a tool that is not related to any specific representation, but allows the user to edit the data directly. Of course, the tool must provide some representation of the data, but this need not be related to any of the final, target representations. In practice, achieving a truly neutral representation is difficult, if not impossible, as structures in one representation rarely map cleanly onto those of another. A common example is the use of a "book" metaphor in online documents, where the division of the text into pages that occur in a fixed order is, in most circumstances, inappropriate for the digital medium (Gentner and Nielsen 1996). Nevertheless, there are numerous examples of tools that attempt to separate data from presentation, and thus facilitate transformation into multiple target representations.

One example is DocBook (Walsh and Muellner 1999), an open standard for a *semantic markup language* used to prepare structured documents, ranging in size and complexity from articles to multi-volume works. It was originally developed in order to facilitate the production and interchange of technical works, notably those published by O'Reilly & Associates[16]. However, it has proved flexible enough to be useful for a wide variety of documents. One point to note is that it is designed to support a hierarchical division into volumes, books, chapters, sections, subsections and so on, a structure more suited to printed documents than hypertexts, as mentioned above. However, for many kind of documents, the structure is appropriate and sufficiently general.

DocBook itself was originally defined as a dialect of the general markup language SGML. More recently, a parallel definition as a dialect of XML has emerged; this is likely to become the focus of continued development. In either case, documents are based on standard format text files, in which certain sequences are used to specify the structure and properties of sections of text. These files may be created and edited using standard text editing tools. In addition, several dedicated tools are available, both for DocBook in particular, and SGML/XML in general.

[16] `http://www.oreilly.com/`

The main difference between DocBook and other SGML- and XML-based formats, notably HTML, is the choice of tags. Whilst the latter has tags to denote the *appearance* of text, the former has tags to denote its *meaning*. For example, whereas HTML allows the author to specify the colour and typeface of a section of text, DocBook allows text to be marked as "author", "title“, and so on (the distinction is not entirely clear-cut; HTML has some degree of semantic markup such as headings, but is however chiefly presentational in its approach, and lacks the ability to describe structures over documents). Additional structures, such as sections and chapters, are achieved by nesting such designations.

The intention behind DocBook and similar systems is that, by allowing the user to specify the data itself, as opposed to the presentation of the data, the system will have access to all the information it needs to create any representation. Transformation from DocBook to a specific presentation is handled by an external processor. Experience has proved that producing a sufficiently general processor (that is, one that will produce the desired presentation for all input documents) is difficult. In practice, many documents contain certain elements for which the correct presentation cannot be determined automatically. For these elements, the user must specify some or all aspects of the presentation. However, such specification is unlikely to be portable to all possible presentations, necessitating several different specifications to deal with various presentations. This is not ideal — the point of using DocBook was to avoid specifying multiple representations — but is acceptable if it only occurs infrequently. If such cases occur frequently, it may be necessary to reconsider the use of the given markup language.

2.8.1 Multiple Representation in Software Visualisation

Large software systems are, by many measures, amongst the most complex objects constructed by human beings. Even a modest software project may contain dozens of structures (both control structures such as loops and data structures such as classes), described by thousands of lines of source code. *Software Visualisation* systems attempt to manage this complexity by representing the structure of a piece of software diagrammatically.

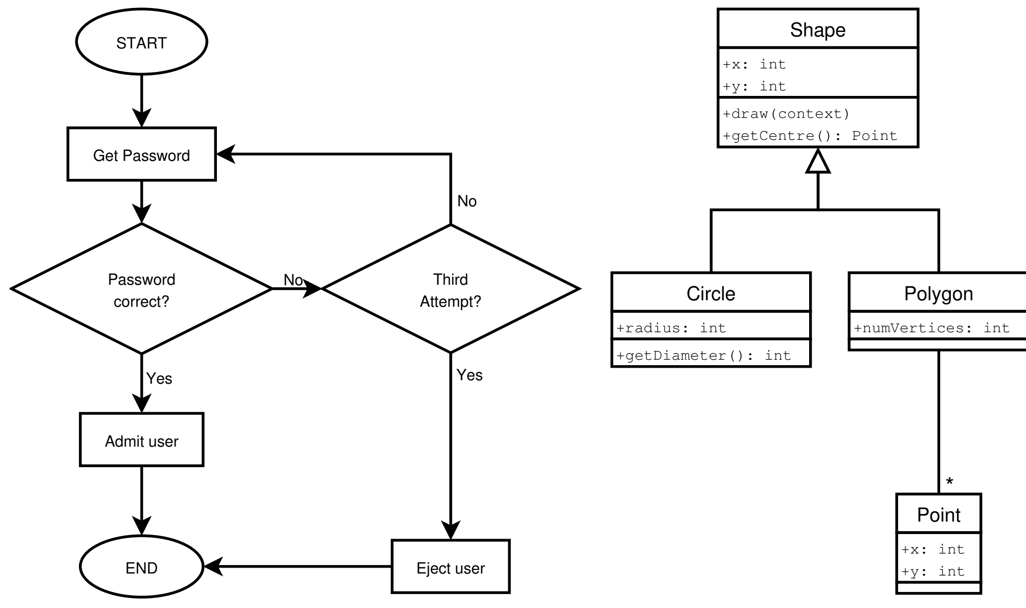
One of the earliest formalisms for the representation of software structure as a diagram is the flow chart, as shown in Figure 2.6. Labelled boxes of different shapes represent elements of the program; for example, a diamond represents a conditional branch. The boxes are connected via arrows representing control flow to form a directed graph. While this is an adequate representation of control flow, little effort is made to represent data structure.

More recently, much effort has focussed on Unified Modelling Language, or UML (Rumbaugh et al 1999). This attempts to standardise the type of diagram used to describe the structure of object oriented systems. It includes representations for data structures (classes and interfaces, shown in Figure 2.6), control flow (in terms of which methods invoke which others, and how subsystems interact), and system construction (both physical and logical).

There are numerous products available for producing UML diagrams from source code, and for producing source code from UML diagrams. Moreover, several environments allow the co-construction of UML diagrams and source code, so that changes in one representation are immediately reflected in the other. However, in all cases, the systems are constructed on an ad hoc basis, tied to both the specific representation (UML) and the underlying language.

In contrast, Grant (1999) presents a system, SVT, that attempts to generalise the task of software visualisation. This is achieved via a "semantic model of

Figure 2.6: A flow chart (left) and a UML class diagram (right)



visualisation". This system decomposes software visualisation into a number of well defined stages. These stages are defined in terms of mappings between different models of the data; raw data is mapped to a structured *data source*, which is in turn mapped to *view content* (the data that will actually be represented in the visualisation), then to *visual content* (the graphical objects representing the view content), and finally to a set of *graphical constraints*. A constraints solver is used to produce an output image that conforms to those constraints.

The output images are in fact generated as projections of 3D models, using the OpenGL rendering toolkit (Woo et al 1999). In most cases, the 3D model is in fact a 2D diagram, and the projection is a trivial transformation. It may seem that using a 3D rendering architecture for all cases where the majority may be adequately served with a 2D architecture is unnecessarily wasteful of computing resources. However, as Grant points out, 3D rendering is generally performed by specialised hardware. As a result, 3D rendering is no more expensive than 2D rendering when the latter is not supported by specialist hardware (indeed, it may be significantly *cheaper* in terms of load on the CPU). Furthermore, it brings several advantages. One is that a single rendering engine can handle both the 2D and 3D cases. Another is that zooming, an important technique for elision in the system, is easily achieved by moving the camera perpendicular to the diagram.

Building on the multistage model of visualisation, the system's main innovation is that both the mapping from data source to view content, and from view content to visual content, can be specified by the particular application, or by the user, using the logic programming language Prolog (Sterling and Shapiro 1994). This approach confers a high degree of flexibility, allowing a wide range of visualisations to be supported. The introduction of the intermediate stage, view content, between data source and visual content, allows the same subset of the data to be viewed using a variety of visualisations, and the same visualisation to be used for different subsets of the data.

The use of Prolog provides a sound theoretical basis for these mappings, assuming that non-logical constructs such as cut are not used; this is implied,

but no indication of how this is enforced is given. The language is well suited to the representation of relational data (the data source is structured as a set of assertions between atoms), and queries upon this data. In addition, it is Turing powerful, and hence can express any relationship between input and output that one may reasonably expect to be implemented on a conventional computer. That said, the expression of some relationships is unduly cumbersome, especially without the use of non-logical constructs, and execution of Prolog is slow in comparison with low-level, imperative languages such as C++.

Similarly, the use of graphical constraints, as opposed to absolute positioning as used in other, similar systems affords a high degree of flexibility and ease to the creation of new visualisations. However, the system as it is described would be somewhat cumbersome to use for spatial data (for which the system's constraints are an unnecessarily verbose representation), and hence would not be useful for the very high volumes of such data that are the traditional purview of visualisation systems. However, this is not a problem in the system's intended area of application (software visualisation), and is specific to that system as opposed to the theoretical model it embodies.

Grant describes a simple "visual type system". This prevents nonsensical mappings from being created; for example, attempting to represent an arbitrary string as a colour. The possibility of checking other constraints using the same system is alluded to, but the mechanism by which this could be achieved is not made clear. In particular, it is suggested that the type system could be used to ensure that all view content corresponds to some visual content; specifically, that the mapping between visual and view content is a bijection. This highlights another aim of the separation of view and visual content. The data that is of interest is selected as the view, and *all* of this data should be presented to the user in the graphical output. In actual fact, the rendering process may remove some information in order to present an overview of the graphic to the user, as the graphic is scaled to fit the available display space, with consequent loss of detail. Nevertheless, the information is available by navigating the view. The subject of selective elision is of particular interest in systems supporting multiple, incompatible representations of the same data, and will be revisited in later chapters.

The use of Prolog, a dynamic language capable of introspection, for the specification of mappings means that these mappings may be modified by the user, while the program is running. SVT provides several ways of doing this. In addition to loading standard Prolog code from a file, the systems provides a mutable *legend*. This primarily serves the purpose of a legend in a normal, non-interactive diagram, namely communicating the meaning of the visual forms used to the viewer. However, SVT's legend is interactive, in that the user can select a different representation for a given type of datum. While the legend cannot be used to define new representations, it provides access to all appropriate representations currently available, as determined by the type system, in a pop-up menu. The legend is itself a view, generated from the Prolog database (which includes the predicates currently used for visualisation) in the same way as the user view of the data itself.

SVT includes limited support for interaction. The rendering component, written in C++, allows views to be manipulated by, for example, panning, zooming and rotating. In addition, Prolog rules may be used to react to actions such as mouse clicks and keystrokes. Such actions may trigger arbitrary Prolog code. Navigation is achieved by switching to a new visualisation. This is sufficient for tasks such as file browsing, where the new context differs significantly from the old, but is arguably cumbersome for interactions consisting of a sequence of

small changes to the context. A set of bookmarks provides scaled versions of other visualisations of the data to facilitate such navigation.

SVT is a very general system, but is of little use on its own as it does not provide any visualisations. It is intended to be used as a basis for software visualisation systems. Grant provides one such system, Vmax, an editor for Java programs with extensive visualisation functionality.

Vmax provides multiple visualisations for Java files, packages, classes, methods, statements and expressions. These may be used alongside standard textual representations. Text buffers, implemented in C++, provide a mechanism for handling large quantities of text overlaid with a nested block structure; this is used to provide a data source based on program source code or directory listings. Text windows are provided to allow users to edit source code. This functionality could, in theory, have been implemented in Prolog and SVT, but is not for performance reasons.

In addition to displaying the graph structure of file systems and Java packages, and rendering program source code in a variety of forms, Vmax allows the user to visualise the state of the program at various points as it runs. This is achieved by inserting "trace points" into the program source code to record the value of a specified variable at that point; the user may do this using a menu item, or visually by dragging a representation of the variable to the desired point. Vmax inserts statements that output the value of that variable to a trace file, which may then be read in and visualised in a variety of ways. Grant notes that it would be desirable to have available dynamically updated run-time visualisations, but Vmax does not have this capability.

Vmax also provides a flexible view for Prolog programs. This view uses SVT's interaction capabilities to allow users to edit the program in visual form. This effectively turns Prolog into a visual language, and, as Prolog is the specification language used in SVT, allows users to specify new visualisations using a visual language. While this "visual Prolog" has numerous advantages (chiefly, that the user is constrained to only produce syntactically correct rules), it suffers from many of the usual problems of visual languages (see Section 2.4.3), and does not alleviate the need for the user to understand Prolog. Notably, the fact that it is a visual language does not necessarily make it more suitable for specifying visualisations, as the visual form of the rules has no intrinsic relationship to the visualisations they embody.

A limitation of SVT encountered by Vmax is that there is no structured way for changes made to a visualisation to be reflected in the underlying data. Where this is achieved, such as in the legend or the Prolog visualisation described above, it is implemented on an ad hoc basis using SVT's interaction capability. A more systematic approach would allow the creation of mutable visualisations purely within the framework, allowing visual notations to be easily produced for any data. This would be an important step towards the use of multiple notations for manipulation of the same data.

SVT also lacks explicit support for secondary notation, in that a representation of a data set is entirely determined by that data. This is not problematic for one-way translation from data to representation, and is largely acceptable in the case of a single mutable representation with translation in both directions, as in both cases secondary notation may be encoded alongside the data. However, it is a serious obstacle to the creation of a system to support multiple, interchangeable representations, each with its own secondary notation. Chapter 4 discusses this issue further, and describes one approach to solving it.

2.8.2 Multi-View Development Environments

Multi-View Development Environments (MVDEs) generalise the concept of software visualisation such that multiple views of a software system may be used to both examine and manipulate that system. The advantages of doing so have already been covered in previous sections. One early such system was PECAN (Reiss 1985). This system provides a variety of views of the program under construction. Some views, including a syntax-directed editor and a structured flow graph editor, may be used to edit the program; others are read-only, and serve to provide alternative views of certain aspects of the program structure. The system provides a high degree of integration between the views. However, it was too resource-intensive to be practical on the systems of the time, and, more seriously, was limited to a fixed set of views, with no explicit provision for extension.

Meyers (1993) aims to produce a generic multi-view development environment with an “open-ended set of views”. In particular, the user should be able to add new views to the system easily; this is seen as a necessary requirement for evolving software development systems. This generality places severe requirements on the mechanism used to integrate the views into a coherent system. Meyers examines the integration mechanisms in a variety of previous systems, and finds them inadequate for the specified task. Accordingly, he settles on a mechanism based on a “canonical representation”. This representation may be used to generate the type of read-only views typical of all software visualisation. Furthermore, if the mapping between a view and the canonical representation is invertible, then that view may be used to edit the underlying system.

Previous software visualisation systems based on the use of a canonical representation of the underlying system have represented the system in various ways; however, none of these are deemed sufficient for supporting a system including bidirectional translation. Abstract syntax trees and graphs represent the features of a particular view, and hence lack the generality required for multi-view development environments. General hypertext systems have the inverse problem, in that they typically do not define semantics, leaving no common base for disparate views. Similarly, the Plan Calculus is considered to be at too high a level of abstraction for the particular purpose.

Consequently, Meyers develops a new representation for his system. Semantic Program Graphs (SPGs) are directed hypergraphs (directed graphs in which edges may have multiple sources and multiple sinks) that encode both executable and non-executable information in the model. The latter includes both structure and additional information such as comments and whitespace. Meyers examines the representation of structure in various views, but does not address in detail the representation of other non-executable information.

Meyers identifies three programming paradigms that SPGs are required to model; sequential control flow, data-flow based computation, and parallel control flow. He goes on to show how this modelling may be achieved in all three cases, demonstrating that SPGs meet this requirement. The modelling and visualisation of data manipulated by the programs is considered to be largely orthogonal to the task of representing the programs, and has been addressed in depth by other work in the field (as discussed in the preceding sections).

The particular semantics of SPGs were chosen in order to match those of a particular class of views; specifically, a certain level of abstraction. Hence, the semantics are simplified by omitting features solely required to “support unusually high level languages like Prolog… nor for unusually low level programming languages like assembler.” This simplifies the representation, and the implemen-

tation of views, at the cost of placing limits on the views that may be easily implemented.

In addition to the representation itself, Meyers describes an architecture to support the implementation and integration of views based on it. A piece of software named the "SPG Manager" provides a consistent platform upon which views may be implemented. This is the only piece of software in the system that may manipulate the program (as represented by an SPG) directly. Views communicate with the SPG Manager using a well-defined protocol. This allows views to both examine and manipulate the program. Such an approach not only eases the task of the view implementor by abstracting away the details of SPG management, but also allows views to be kept consistent with the underlying program.

Meyers identifies several problems specifically associated with producing invertible mappings between views and a canonical representation. The first arises when features in the canonical representation have no corresponding feature in the view. Possible solutions to this problem include omitting the feature, approximating it, or representing it as an opaque "black box". The converse situation, where the view has features that the canonical representation lacks, does not have similar solutions. This implies that the features in views are a subset of those of the in the canonical representation. This is a necessary condition for invertible mappings (as features not in the canonical representation would be lost when mapping from the view to the representation and back again), and is an acceptable limitation if the chosen canonical representation is sufficiently expressive. Prolog is an example of a view with features that mean that it is incompatible with SPG; Meyers states that, in such cases, there is no satisfactory way to implement the view in the context of the canonical representation, without making major changes to one or both.

Several examples of translation between SPGs and various views are presented. These include translation both to and from finite state automata and Pascal. Meyers notes that producing these translations is a complex task, the difficulty of which is compounded by the flexibility of SPGs. This flexibility makes it necessary for each translation to handle a wide variety of features, some of which may (as noted above) be absent from the view in question. Accordingly, the example translations, while extensive, omit some details (such as whitespace in Pascal) in order to simplify both implementation and explanation. It is acknowledged that such details would have to be fleshed out in order to produce a practical multi-view development environment. Nevertheless, the examples strongly suggest that the approach in question is sound.

The SPG architecture provides a strong basis for the development of multi-view development environments. While the approach taken is very general, the specific implementation is specialised to programming in a conventional setting, using certain types of programming languages. Hence, to adapt the technique to novel contexts or languages, it would be necessary to use a different canonical representation. In addition, the framework as presented does not go far in addressing the issues associated with non-executable information (secondary notation); these issues require further examination, particularly where views support a wide variety of secondary notation features.

Chapter 3

The Media Cubes
A tangible programming language

This chapter discusses the Media Cubes, a novel input device designed to allow users to interact with a home network in a convenient manner, and a programming language that has been designed around such devices. We refer to this language as a "tangible programming language", as the tokens that are manipulated are physical objects as opposed to diagrams or text represented to the user via some interface.

3.1 Introduction

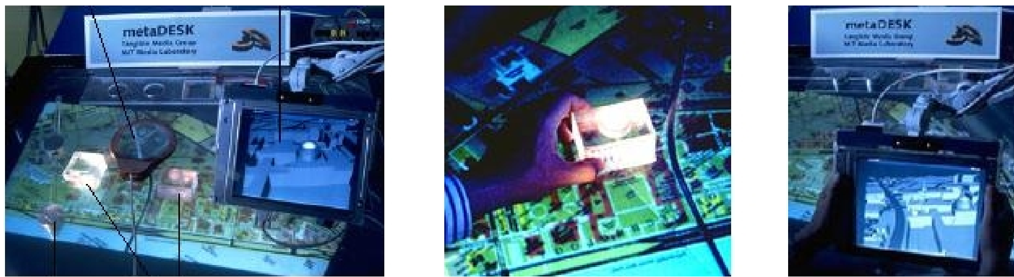
While much of the AutoHAN project, and domestic ubiquitous computing in general, is focused on simplifying or automating many complex tasks associated with the networked home, such as setting up devices and enforcing security or usage policies (Saif et al 2001), it is necessary to provide some means for users to control the system. Accordingly, the project investigated a range of interaction techniques. One notable example is the use of voice recognition, combined with language understanding to disambiguate incomplete statements based on the user's context (for example, the command "Turn on the TV" would be interpreted as referring to the television in the same room as the user). However, it was also believed that, in addition to direct control of devices, the user must be able to program new behaviour into the system in order to take full advantage of it.

A systems programming language, Iota.HAN (Bierman and Sewell 2003), was developed for the project. While this has many advantages (a strong formal basis, flexible communication primitives, and a static type system that includes support for XML data), it is aimed squarely at experienced programmers, and is not suitable for general users. The Cambridge Event Language (CEL), another language developed as part of the project, is simpler, but still not suitable for most end users.

As well as the mismatch between traditional programming languages and the user population to be addressed, traditional programming environments do not sit well in the context of the networked home. They are based on the manipulation of complex visual objects (almost exclusively text), and as such require a large, high resolution display to be effective. In the networked home, convenient displays are likely to be low resolution (a television), small (a PDA), low resolution *and* small (a mobile phone), or absent entirely. Moreover, appropriate input devices for text or spatial data may not be available, and when they are may be of low fidelity, or cumbersome to use.

The *Media Cubes* were conceived by Blackwell (2000) as an experimental programming system to overcome the difficulties of both traditional programming languages, and traditional user interfaces for simple programming tasks, in the domestic environment. The central idea is that of associating physical objects with abstract entities (such as "the time when a user enters a room", "the user who has entered the room" or "the action of playing a user's preferred music when they enter the room"), creating a *tangible programming language.*

Figure 3.1: **The metaDESK TUI (left), and users interacting with a phicon (centre) and an activeLENS (right)**



3.2 Tangible User Interfaces

The Media Cubes language aims to provide an interface to computing functionality (in particular, programming) via the manipulation of physical props. This approach, while unusual, is not without precedent. Many other projects have used physical tokens as a component of, or the basis of, their user interface; for example, Mackay and Pagani (1994) present a system for editing video based via the manipulation of physical pieces of paper forming a storyboard. Ishii and Ullmer (1997) describe a view of human computer interaction based on a distinction between two "realms"; the physical environment, comprised of atoms, and "cyberspace", comprised of bits (Negroponte 1995). They observe that interaction between these two realms is, at present, largely confined to conventional graphical (and textual) user interfaces, to the neglect of haptic interactions and peripheral awareness. To counter this, they propose the development of *Tangible User Interfaces* (TUIs), in which physical objects (*phicons*, or physical icons) correspond to virtual objects. This leads to a "graspable" interface between the physical and virtual realms that is not limited to the VDU and keyboard, but can extend throughout the physical environment.

Numerous examples are provided, including a "tangible geospace" — a TUI to an interactive map of the MIT campus (Figure 3.1). The map is displayed on a horizontal desk surface, and the position of the map is determined using a phicon of a particular landmark, placed on the desk; the map is translated such that the landmark's position matches that of its phicon. Orientation and scale may be manipulated by placing two landmarks on the map, and varying their relative positions. In addition, an "activeLENS" — an LCD display mounted on an articulated arm with sensors to monitor its position and orientation — allows different views of the map, such as satellite imagery, to be accessed within the same physical space.

The mediaBlocks (Ullmer et al 1998) system is another example with particular relevance to the Media Cubes. MediaBlocks are phicons corresponding to sequences of digital multimedia data. Such data is not stored in the block itself, but is associated with the block's unique identifier. The blocks themselves are exceedingly simple, using only a passive technology allowing the identifier to be read by a more complex device (various technologies are used in different versions of the system). A number of additional devices were instrumented with appropriate readers, allowing various manipulations of the data associated with blocks. Blocks may be browsed using a dedicated device, and printed using an instrumented printer. Another dedicated device, the *media sequencer* allows the contents of blocks to be composed into new sequences. Items may be added and removed via readers attached to PCs, allowing the blocks to serve their primary

Figure 3.2: The mediaBlocks, shown in the context of the *media sequencer*.

function of acting as containers to transport data from one device to another. While an interesting system, the mediaBlocks are limited to a very narrowly defined role, essentially moving multimedia data from one location to another. The Media Cubes are intended to provide a more general and flexible interface, but nevertheless subsume most, potentially all, of the functionality of mediaBlocks.

The key difference between the majority of these systems and the Media Cubes is that, whereas previous systems provide a direct interface to some functionality, the Media Cubes aim to provide a programming interface to the functionality of the AutoHAN system, as discussed earlier. AlgoBlock (Suzuki and Kato 1995) is another example of tangible user interface techniques used in the context of programming. It may be loosely described as a physical syntax for the Logo language. It is intended to be used by young children in an educational setting. The chief reason for using a physical language in this context is to facilitate collaboration between members of a group working on a program. While the work on Algoblock suggests that physical languages do indeed confer such benefits, collaboration is not an area we are presently studying, and hence these particular results are only peripherally relevant to the Media Cubes language. However, the work stands as an example of the feasibility of programming languages based around tangible interface techniques.

The Jigsaw system mentioned in Chapter 2, represents another end-user programming language with tangible components (in this case optional). However, while more sophisticated in terms of behaviour than mediaBlocks, the range of behaviours expressible in the system is intentionally limited to simple sequential actions. Conversely, the Media Cubes are intended to provide a level of expressibility more akin to that of AlgoBlock and non-tangible end-user programming languages.

Traditionally, TUIs are difficult to develop, as they require skills in a broad range of areas, such as hardware development, computer vision and user interface design. However, as with GUIs, programming toolkits such as CTK (Salber et al 1999) and, more recently, Papier-Mâché (Klemmer et al 2004), have emerged to simplify the process by abstracting away the details of specific implementation technologies. The latter has, unusually for a programming toolkit, been studied in terms of usability, and has been used to implement a variety of applications. One of the applications created as part of the Paier-Mâché project, SiteView, is a tangible interface to home automation . However, as the Media Cubes language

Figure 3.3: **Media Cubes (working prototype), designed by Daniel Gordon and constructed by Dick Kimpton**

Figure 3.4: **Media Cubes (non-working mockups)**

is based on custom hardware for which drivers would have to be implemented, the benefit of using such a toolkit would be marginal, and as such no toolkit was used.

3.3 The "Media Cube" Device

As mentioned, the Media Cubes language is a programming language in which programs are not constructed using words or graphics, but by manipulating physical objects. The objects in questions are small blocks, instrumented with sensors and labelled, first described in Blackwell (2000). Figure 3.3 shows working prototypes of these devices, built by Daniel Gordon and Dick Kimpton. Figure 3.4 shows non-working mockups that more closely resemble the intended final product.

Like mediaBlocks, the Media Cubes themselves are relatively simple devices. Each is a small cube with a single button mounted on the uppermost face (the working prototypes also include a three-colour LED; this feature was not used in the language design). The bottom face is left blank. On each of the four

other faces, a sensor is mounted that may detect when the Cube is next to another Cube. The Cubes are also equipped with wireless networking hardware, an internal speaker, and a simple microprocessor. Each is identical aside from a unique ID. A single function may be invoked by pressing the button, and programs are constructed by placing Cubes adjacent to one another.

An obvious extension to the language would be the introduction of "Cubes" that are not in fact cubic, but are instrumented in such a way that they may interact with the Cubes and be used as components in programs. This would allow users to directly refer to the device in a program, by placing a Cube next to it. It may also be useful to have Cubes that incorporate analogue controls for quantities such as temperature or time.

Programs are not reflected in the state of the Cubes; they exist only in a software process running on a server. This allows the Cubes to be simple devices, and to be interchangeable to an extent; if a given Cube is lost or broken, it is simply a question of updating the software representation of its functionality to correspond to a physical Cube with a different unique ID. It also allows us to make changes in the language by modifying the server-side software, avoiding the more difficult process of embedded software development.

The current Media Cubes prototypes, shown in Figure 3.3, are around 75mm along each edge, contain a PIC microprocessor, and communicate with the base station via an infrared (IR) transmitter/receiver pair. The latter fact has caused problems in actual use. The IR link is adversely affected by environmental infrared, such as fluorescent lights and direct sunlight. More seriously, it is directional, forcing the user to orient the Cubes in a particular way if communication is to be successful. Future prototypes will use a radio link, alleviating these problems.

Adjacent Cubes are detected using a coil on each face. This coil is pulsed around sixteen times per second. The current on the coil, when it is not being pulsed, is monitored; hence, a current induced by the coil on an adjacent Cube is detected. This system only detects the faces of a Cube that have other Cubes adjacent to them; it does not determine which Cubes these are. The information from several Cubes is used to infer the arrangment. In the current language, Cubes are only combined in pairs, so this inference is trivial.

The Media Cubes are intended to provide a smooth progression from direct manipulation to programming. In isolation, they may be used as direct manipulation devices, and particularly simple devices at that; the button mounted on the top of the Cube may be used to control a single function, or to alternate between functions, depending on the Cube in question. In this context, Cubes may be viewed as simplified remote controls. As such, they provide little barrier to adoption, even for users that lack technical confidence. A user may start by using Cubes that correspond to familiar functions of home appliances, and then move on to Cube that correspond to more abstract concepts such as media streams (see below). Once a user is comfortable with using Cubes in isolation, the concept of using them in combination may be introduced. Hence, the user makes a relatively seamless transition from direct manipulation to programming.

Another way that the Media Cubes are intended to reduce the cost of creating an abstraction is by providing a concrete representation of abstract entities in the system, such as media streams. This is intended to allow the user to refer to these entities more easily. Having a representation for abstract concepts is also important with respect to our earlier definition of programming; the Media Cubes give concrete representation to concepts such as "7:30pm" and "whenever the doorbell rings", allowing the user to create abstractions over time, and "every

device that makes a sound", allowing them to create abstractions over classes.

3.4 The Media Cubes Language

The Media Cubes language, as it stands, is based around a small number of cubes, along with instrumented devices. These cubes are combined *dynamically*, in that, once two faces have been touched together, the cubes may be separated without removing the association. This contrasts with AlgoBlock, in which the arrangement of cubes at any instant determines the program, and separating two cubes removes the corresponding association.

The advantage of not building a fixed model of the program is that the language is not subject to the constraints inherent in the model. The first of these is the number of components. If each component must remain in place, the size of the model built is limited by the stock of components available. This problem is not found in conventional programming languages — no-one has ever been unable to complete a Lisp program because they have run out of parentheses — but is important in the context of tangible interfaces, where there is a finite, and in most cases, severely limited, supply of interface elements. The problem is exacerbated by the fact that physical components may be lost or damaged. The dynamic strategy adopted for the Media Cubes language allows the reuse of cubes, allowing the language to be based on far fewer cubes.

In addition, a language in which components must be arranged into a model representing the program is limited by the geometry of three-dimensional space. For example, it is not possible to place faces of two cubes against the same face of another cube simultaneously, assuming that faces must overlap completely. Even if this constraint is relaxed, fitting more cubes against a single face becomes increasingly awkward, and introduces additional geometric constraints, as more cubes are added. If the model is built in three dimensions (as opposed to two), the user is also likely to encounter problems relating to the model's balance and structural integrity. Moreover, if components are rigid, specifying one relationship may determine others in an unintended way. These problems may, to an extent, be circumvented (as in AlgoBlock) by introducing extra components, some of which may be non-rigid (for example, connecting wires). They do not occur in systems based on dynamic arrangement.

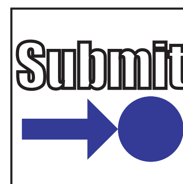
The dynamic arrangement approach is a good fit for a ubiquitous computing context, as such systems are long-lived, and should be highly reliable, making large numbers of difficult-to-replace components problematic. In addition, it may not be convenient to construct a model, as this requires a relatively large area, and a work surface of some kind. In contrast, the Media Cubes may be used while standing or sitting anywhere, and do not require a surface to work on. However, while this approach has significant benefits, it also has numerous problems. These are discussed in Section 3.8.

Most cubes in the Media Cubes language follow the standard design of a cubic block with four "active" faces (i.e., faces instrumented with sensors that can detect the presence of other active faces, allowing the system to infer that two active faces are combined), and a single button. The button is used to reset the cube to its starting state, removing any associations it currently has.

The key cube upon which the Media Cube language rests is the “Do-When” cube, used to specify causal relationships. This cube has four active faces. “Do” accepts an event to emit, or a script to activate, when the specified conditions occur. “When” and “Whenever” accept conditions in the form of atomic or compound events. The former implies that this is a “one-off” script, that disappears after being triggered once, while the latter implies a persistent script. The condition and behaviour associated with the most recently touched face are used. Finally, the “Script” face provides the script described by the other faces to other cubes; this face can be associated with the Do face of another Do-When cube to specify sequential causality (“If A happens, then B happens, do C”).

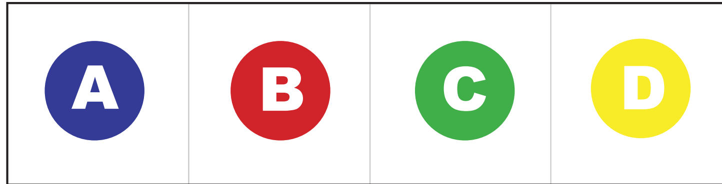
Atomic events are provided by instrumented devices. For example, a light switch could be instrumented such that it may act as a cube face, providing the event switching the state of the given light (but see Section 3.6). Similarly, a bar code reader in the kitchen could provide an event corresponding to scanning an item, parameterised on the bar code of the item scanned. It may also be useful to provide a way for the user to browse the recent history of events that the system has seen, either via a dedicated device or an existing device such as a TV or PC, and incorporate these events into Media Cubes programs. This would provide a convenient way for users to explore and access the events produced by the system (they simply need to perform an action, and then browse for the resultant events), but care would have to be taken lest the user is overwhelmed by the sheer number of events presented.

For development purposes, it is useful to produce a simulator for the Media Cubes language. In order to provide access to events that would be obtained from devices in the home network, an “Event cube” is introduced. This “cube” effectively has an infinite number of active faces; a request for a face `X` is successful if `X` is a valid event specification (a notification type and subtype, separated by a slash (/), in the current system), and the resultant face provides the corresponding event. This allows programs to be created within the simulator without specifying every instrumented device individually.



The “System” cube provides an interface to the AutoHAN infrastructure. At present, the only face provided is “Submit”, which accepts a script and “activates” it. Prior to activation, a script is inert, and does not react to events in the system. When activated, the script will receive events in which it has registered an interest (in practice, events that match it’s top-level condition), and react accordingly. Further developments may result in additional faces being added to the System cube, for example to assign functionality to newly purchased cubes, or to modify overall system behaviour. If this results in more faces than the four accommodated by the standard cube design, the functionality could be split

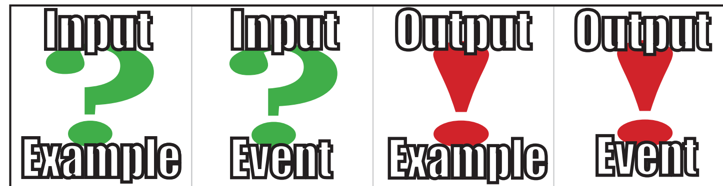
over multiple cubes, or an alternative physical design (such as a palette) could be used.



It has already been mentioned that the dynamic construction approach allows the Media Cubes language to employ fewer components, and leaves it relatively unconstrained by geometry. However, it is still the case that two faces of the same cube cannot be placed next to each other, and that a cube cannot be in two places at once. Hence, if one wishes to, for example, use one conditional script as the action of another, two Do-When cubes are required. An alternative is to provide a "Clone" cube, such that faces of other cubes may stored temporarily on faces of the Clone cube. Each face is comparable to a clipboard, as found in desktop user interfaces.

In the case of the previous example, the inner condition would be constructed in the normal way, and the Script face of the Do-When cube is cloned (by combining it with a face of the Clone cube). The Do-When cube is then reset (by pressing its button), and the same face of the Clone cube may be used as the action of the new (outer) conditional.

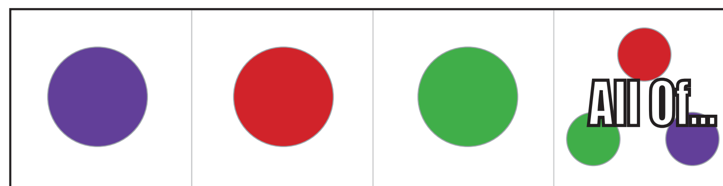
There are two obvious approaches to implementing the Clone cube. Either the face of the Clone simply provides the same value as the face cloned provided at the time the two were associated (passive cloning), or the face of the clone cube acts as an alias to the cloned face (active cloning). The two are analogous to call-by-name and call-by-value in Algol, and are equivalent in simple cases, where a given face of a cube always provides the same value. However, in more complex cases, the two differ if the state of the cube has changed between the act of cloning a face, and the use of the clone. Active cloning of individual faces quickly becomes problematic; for example, what happens if the cube owning the cloned face is reset? Active cloning of entire cubes has clearer semantics, but is still more complex than passive cloning of faces, both in terms of implementation, and of comprehension by the user. Hence, the current Clone cube has passive semantics.



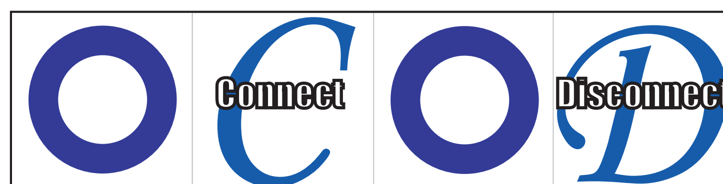
The "Generalisation" cube allows users to produce programs that react to a *class* of events, as opposed to a single, specific event. The particular semantics of the cube assume that events have a type, corresponding to the class of event, and a subtype, identifying a specific event with that class. For example, an event may have the type `OutOf`, signifying that it is an event corresponding to something running out, and a subtype `Milk`, identifying the particular product that has run out. Another event may have a type `Order` and subtype `Milk`, signifying that the given product should be reordered. To generalise over an event type, an example event that has that type is provided to the "InExample" face. Subsequently, the "InEvent" face corresponds to the *any* event of that type when used as a condition.

In most cases, the script reacting to the generalised event is parameterised such that its behaviour is specialised to according to the subtype of the actual event received. This is achieved by providing an event with the desired type to the "OutExample" face of the Generalisation cube; the "OutEvent" face then provides an event with that type, and the same subtype as the concrete event received.

Generalisation is undoubtedly the most complex aspect of the Media Cubes language, and may be too complex for the typical user. A way of generalising scripts that act on specific events, similar to the scheme employed by ToonTalk, may be preferable, though it is difficult to see how this could effectively achieved without visual feedback.



The "AllOf" cube allows the creation of compound events. Three of the faces accept events, and the fourth face provides a compound event corresponding to the events provided on the other faces. In addition, scripts may be provided in the place of some or all of events; in this case, the fourth face does not provide a compound event. In either case, the fourth face may also provides a script that, when activated, activates the scripts, or emits the events, associated with the other face.

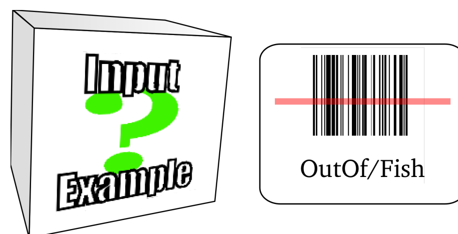


The “Connector” cube is an example of a “higher level” cube. When appropriate device connection events are provided to the two “Endpoint” faces, the “Connect” face provides a script that, when activated, connects the two devices, and the “Disconnect” face a script that disconnects them. This could be achieved using the other cubes, but is a common operation, and as such it is convenient to have a single cube provide access to the functionality.

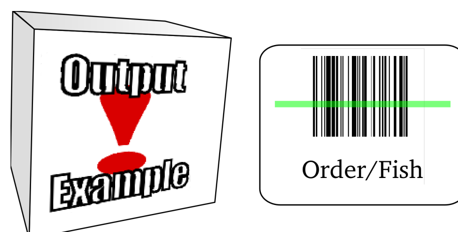
3.5 An Example Media Cubes Program

The Media Cubes language is best presented using a small example. The program described is intended to be a simple automated shopping list; when an `OutOf` event is received for a particular product, then, the next time the `GoShopping/-` event occurs, an `Order` event is generated for that product. This is achieved by producing a script that, whenever an `OutOf` event is received, produces another script. This “inner” script reacts to the next `GoShopping/-` event by generating an `Order` for the appropriate product.

Each interaction with the cubes is shown via graphic representations of the cube faces involved, or other devices as appropriate, and followed by an explanation of that stage. Note that there is some flexibility in the ordering of these interactions; for example, the first and second can be exchanged without altering the meaning of the program.

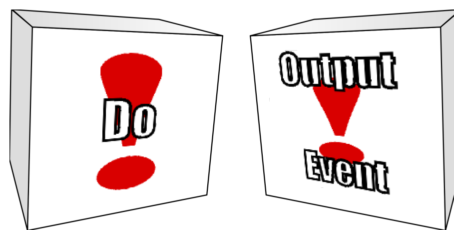


The `OutOf/Fish` event is provided as an example input event to the Generalisation cube. This event could be generated by, for example, a barcode reader attached to the waste bin; empty packages are scanned before they are thrown away.

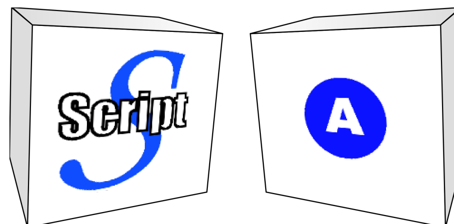


Similarly, the `Order/Fish` event, is provided as an example output event to the Generalisation cube. This event could be generated by another barcode reader, perhaps provided by a store. Note that, while the subtypes of the two example events are the same in this case, this is not necessary; only the types are taken into account.

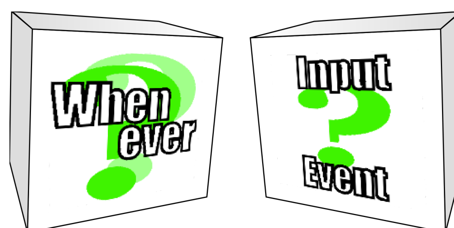
Next, the inner conditional is constructed using the Do-When cube. A `GoShopping/-` event is provided as a trigger. This event corresponds to some user action, with no prior meaning. The system provides “blank” event generators, such as unlabelled buttons or cubes, for this purpose. The When face is used, as we wish this to be a one-off conditional; only the next `GoShopping/-` event should cause the order event to be emitted.



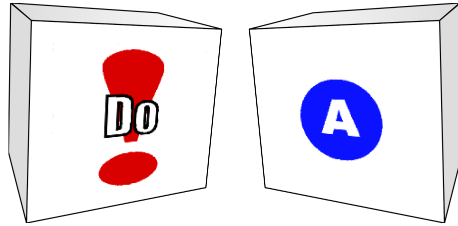
The Do face of the same cube is used to set the action performed; in this case, to the output event from the Generalisation cube. This event will have the same subtype as the event received when the input event from the same cube is used as a trigger.



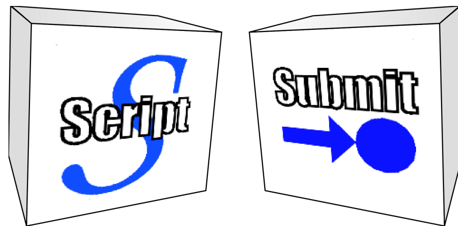
The Script face of the Do-When cube, corresponding to the script that created in the previous steps, is cloned by placing it on a face of the Clone cube. This allows the script to be used later, independently of further actions on the Do-When cube. Hence, the latter cube may now be reset.



The outer conditional is now constructed. First, the condition is specified using the Whenever face of the Do-When cube; this conditional will react to all events that match its trigger. The use of the input event from the Generalisation cube means that wherever corresponding output event occurs in the action of the conditional, it will have the subtype of the received event (and the type of the output example).



The face of the Clone cube holding the inner script is used as the action for the Do-When cube. Note that, even though the Do-When cube has been reset and reused since the script was created, the cloned version is still accessible.



Finally, the script is submitted to the system by placing the Script face of the Do-When cube onto the Submit face of the system cube; it is then activated, and will receive future events with the same type as the input example (`OutOf`).

3.6 The Nature of Events

The design of the Media Cubes language is, in general, independent of the underlying event system. All constructs consider events to be atomic units, and do not operate on the internal structure of an event, with the exception of the Generalisation cube, which only relies on each event having a type and subtype. As a consequence, the system designed has great latitude in terms of the events the system generates and responds to.

Nevertheless, the events made available to the user may have considerable impact on programming. For example, consider the events corresponding to switching a device on and off. In physical devices, there is almost always a single control (a switch) used to toggle between the two states. It is therefore tempting to have a single event correspond to the operation of this control. This approach is acceptable in the case of direct manipulation, where the current state is readily observable, but is unsuited to programming in which the state of the device when the program is invoked cannot be know ahead of time. Separate events for on and off are preferable, as they are independent of the current state of the device. However, if such events are used, it is unclear which event the physical switch corresponds to, making integration of the physical device with programming systems such as the Media Cubes problematic. Moreover, if the physical switch itself emits the event when used, it must now be more complex, as it must query the state of the device to determine the event to emit.

It may be preferable to construct the system in such a way to support both types of events. The most obvious way to do this is to synthesise events of one type in reaction to the other. Then, the system may produce "low-level" events, which in turn cause "high-level" events to be emitted. These high-level events are used by end-user programmers. Different sets of events may be more suitable for different types of language. However, the proliferation of events may cause confusion when moving from one language, or one level of the system, to another, and hence should be kept to a minimum where possible.

An alternative way to introduce such concepts into the end-user programming components of a system is the use of "higher-level constructs" such as the

Connection cube mentioned in Section 3.4. Such constructions offer a high degree of flexibility without requiring changes to the underlying system, but may lead to inefficient generated code. Accordingly, a combination of a well chosen event scheme, with judicious use of higher-level constructs, is likely to be the most productive approach.

3.7 Implementation

The language, at first sight, would appear to translate directly to an object-oriented system, with each cube being an object, and faces representing methods (or messages) for that object. However, it differs from such systems in a crucial respect: the interactions used to construct programs are *symmetric*, as opposed to asymmetric method calls (message passing) in object-oriented programming, where the subject–object relationship is made explicit. A statement `Object.method(Subject)` or `[Object methodOn: Subject]` makes it clear that `Object` is "in charge", and determines the semantics of the operation; conversely, when faces of two cubes are touched, it is not evident which cube "owns" the interaction. Clearly, a different approach is needed.

Multimethods, as found in the Common LISP Object System (Steele 1984) (where they are know as "generic functions") and Dylan (Shalit 1996) provide a possible direction. In a conventional call to a virtual method of an object, the run-time type of that object is used to determine the concrete method implementation to call. In contrast, a multimethod system uses the runtime types of *all* participants (object and subject) to determine which concrete method to call. Note that this is distinct from overloading in Java and C++, in which the static types of parameters are used to distinguish between similarly named methods of an interface at compile time; runtime method selection is still entirely determined by the type of the object to which the called method belongs.

The multimethod systems in these languages are comprehensive, and, as a result, complex. Two observations allow the system adopted for the Media Cubes language implementation to be far simpler. Firstly, there are always exactly two participants in a Media Cubes interaction, and the concrete method to call is determined by both. This eliminates the need for complex priority algorithms. Secondly, in any valid interaction, one cube is supplying data, and the other is acting upon it (in other words, a subject–object relation exists, but the direction is determined by the order of the operands); however, it is not the case that a face will always be either a source or a sink (for example, consider the Clone cube).

Based on these observations, the Media Cubes interpreter resolves interactions as follows. Firstly, it obtains cube face objects for both faces involved in the interaction. Secondly, it queries each face object involved in the interaction to obtain a list of types that it can provide ("source" types). For each source type, the other face object is queried to determine if there is an appropriate "sink" method to accept values of that type (or a supertype of that type). If a single matching pair of source and sink methods is found, the sink method call is called with the result of calling the source method with no arguments. If no methods, or several pairs of methods, are found, an exception is thrown.

This algorithm is implemented as a static method of the `Face` class, shown in Figure 3.5. This class has an instance method `getSourceForSinkTypes(Face)` to return the `Class` returned by the (unique) source method of one face that corresponds to a sink method of another, which in turn employs the instance methods `getSourceTypes()` and `getSinkForType(Class)`, returning `null` if no method is found, and throwing an `InteractionException` if more than one

Figure 3.5: The Face Class (with method implementations omitted)

```
public abstract class Face {
    public Method getSourceForType(Class k) { /*...*/ }
    public Method getSinkForType(Class k)
        throws InteractionException { /*...*/ }
    private Class getSourceForSinkTypes(Face otherface)
        throws InteractionException { /*...*/ }
    public Enumeration getSourceTypes() { /*...*/ }

    public static Object interact(Face a, Face b)
        throws InteractionException { /*...*/ }
}
```

Figure 3.6: The Cube Class (with method implementations omitted)

```
public abstract class Cube {
    protected Dictionary faces;
    public Face getFace(String name) {/*...*/}
    public void addFace(String name, Face face) {/*...*/}
}
```

method is found. The method is called on each face, passing the other face as a parameter. If exactly one of these calls returns a class, the corresponding source and sink methods are used. Otherwise, an `InteractionException` is thrown; this may be reported to the user using an auditory warning from the cube's internal speaker.

The default implementation of the methods `getSourceTypes()` and `getSinkForType(Class)` uses the Java Reflection API (Arnold and Gosling 1996) to examine the methods available for a given face object. This allows the common case, where a face has a fixed set of source and sink methods (often, a single source or sink method) throughout its lifetime, to be implemented simply by defining sink method with appropriate parameter types, and source methods with appropriate return types. Faces with more complex semantics, such as those of the Clone cube, override the default implementations, allowing them to return source and sink types appropriate to their current state.

Cubes themselves are instances of subclasses of `Cube`, a simple abstract class providing only an interface to manage a named set of faces (Figure 3.6). Concrete subclasses of this class typically add new faces in their constructor. Each subclass

defines one or more custom `Face` subclasses implementing the behaviour of each face. In most cases, these classes must access private members of the associated `Cube` subclass. Java's inner classes provide a convenient mechanism to provide this access without violating encapsulation, and also allows each `Face` subclass instance to be implicitly associated with a "containing" instance of the associated `Cube` subclass.

Interactions are managed by instances of the `Interpreter` class, which takes a stream of interactions as input, invokes the interaction resolution algorithm described above, calls the resultant methods, and passes side effects (such as errors, or scripts being submitted to the system) to an `Environment` subclass. By providing appropriate input streams and environments, the same interpreter code may be used for simulating the behaviour of the Media Cubes language, and for the implementation of the language itself.

3.8 Evaluation

The Media Cubes language design described above has numerous positive characteristics. However, it also has significant problems, the foremost of these being the lack of an external representation of the program under construction. This forces the user to keep the program in mind throughout.

The first consequence of this is that the program is limited in size to that which the user may hold in their short-term memory. The size of this "working memory" is commonly taken to be seven items, plus or minus two. This estimate is acceptably accurate when the items in questions are discrete units such as digits or words, but it is less readily applied to complex structures as found in programming languages. Furthermore, the estimate is predicated on the assumption that the items are arbitrary and independent. This is clearly not the case here, as the items are part of a structure, and are being selected deliberately to contribute towards some goal. However, while these factor may raise the number of elements that may be held in memory above the traditional estimate, there is still a relatively low limit on program size.

The second consequence is that, once created, the program cannot be examined. This means that the original programmer cannot check that the program has the intended behaviour, and others may not examine the program to determine what it does or how it works. More seriously, the lack of an external representation precludes editing. The Media Cubes language is genuinely write-only. As the programs cannot be modified, programming errors may only be corrected by rewriting the program from scratch.

These limitations restrict the usefulness of the Media Cubes to small, simple programs. It would be possible to redesign the language such that an external representation was constructed, as with AlgoBlock, and largely overcome these problems, but this would entail the problems already discussed. I chose to address the problems in another way. By providing an alternative representation of programs created with the Media Cubes, the problems caused by the lack of an external representation may be eliminated, without changing the language and sacrificing its positive aspects. I chose to develop a system to support arbitrary translations between notations, to allow users to select the most appropriate notation for a given task. The specific problems faced in implementing such a system are detailed in the following chapter, with subsequent chapters describing the system implemented.

Chapter 4
Translation based on common intermediate form

The Media Cubes language described in the previous chapter has the unusual property that it is write-only. As a result, it is impossible to modify programs created using the language, or even examine them to determine their function. These limitations greatly reduce the utility of the language. It would be possible to redesign the language to reduce these problems, but in doing so some of its desirable qualities would be lost. I have taken an alternative approach, namely to design a framework in which the write-only nature of the language is accommodated.

The central idea of this framework is to employ multiple languages, each of which have specific strengths and weaknesses. These programming languages are integrated at a far finer granularity than in previous systems (see Section 2.7). Specifically, a program that was created in one language may be viewed and edited in another. Languages may be freely combined, allowing the user to select the most appropriate language for the task at hand.

4.1 Language Integration via a Shared Intermediate Form

The close integration of languages is achieved via a common intermediate form for programs. Unlike the intermediate forms used in compilers, however, this form retains sufficient information to reconstruct the source code. Crucially, it not only allows the reconstruction of a functionally equivalent program, but also the reconstruction of the exact source code, including "secondary notation" such as comments. This removes the need to retain source code; the intermediate form is sufficient.

Furthermore, if such reversible mappings exist between the intermediate form and several different programming languages, the intermediate form can act as a bridge, allowing translation from one language to another. This technique has been profitably employed in previous multi-view development environments, as described in Section 2.8.2.

While most languages would provide mappings to and from the intermediate form, this is not required. For some languages, including the Media Cubes, there is a mapping from the language to the intermediate form, but not from the intermediate form to the language. In this case, the language can be used to create programs, but not view them. Conversely, if there is a mapping from the intermediate form to a given language, but no mapping from that language to the intermediate form, then that language may be used to examine programs, but not create them. An aural representation of a program, produced via speech synthesis, would fall into this category.

Modification of programs is achieved by translating the program into a given language, editing that program, then translating the modified program back into the intermediate form. For obvious reasons, languages that do not provide mappings both from and to the intermediate form may not be used to edit programs.

4.2 Requirements of Mappings

The relationship between the intermediate form and the various programming languages may be viewed as a set of mappings between the set of all possible programs that may be represented in the intermediate form (termed simply *programs*), and various sets of all valid representations of programs in a given language. Borrowing from semiotics, these representations are termed *texts*, but are not limited to sequences of characters; a text in this sense could equally comprise of a sound, a graph or a series of gestures.

In the case of bidirectional translation, these mappings are not arbitrary, but must satisfy certain constraints. Let f be the mapping from a particular language to the intermediate form, and f^{-1} be the inverse mapping from the intermediate form to the language. f must be a *bijection* between texts in the given language and programs, in that for every object in the domain (texts in the language), there is exactly one object in the codomain (programs) that is related to that object, and vice versa.

Another way of stating this requirement is to say that f composed with f^{-1} must be the identity on texts, and f^{-1} composed with f must be the identity mapping on programs. The first property requires that each text maps to a unique program. This may be achieved by ensuring that any redundancy in the source language is reflected in the intermediate form. Similarly, the second property requires that each program corresponds to a unique text. This would be equally straightforward if there were only a single language. However, the purpose of the system is to allow translation between multiple languages. As the representational capabilities of these languages are not necessarily equivalent, texts must be structured in such a way as to ensure that information is not lost when translating from the intermediate form to any given language. One method of achieving this is described below.

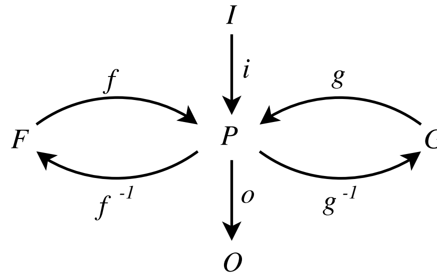
In contrast to bidirectional translation, the mapping associated with a language that provides translation in only one direction (either to or from the intermediate form) is largely unconstrained, in that there is no need for the mapping to have an inverse. The only requirement is that the domain (for translations *to* the intermediate form) or codomain (for translations *from* the intermediate form) be the universe of programs, and, for translations from the intermediate form, that every program is mapped to a text. There is no requirement that different programs must correspond to distinct texts. In theory, it is even permissible to relax the constraint that the correspondence must be a map, and allow a program to map to (or be mapped to) different texts nondeterministically. However, it is difficult to envisage a situation where such a language would be useful. Figure 4.1 presents the possible mappings for input, output and bidirectional languages.

4.3 Execution as Mapping

As well as mapping between the intermediate form and languages used by programmers, the intermediate form is also the basis for execution. One approach would be to map the intermediate form to some executable form, for example machine code or bytecode, and then execute it in the normal way. This mapping could be treated in exactly the same way as other unidirectional mappings from the intermediate form.

An alternative would be to model execution on *term rewriting*. This is a technique used in many theoretical frameworks for computation, such as the λ-calculus. In this type of framework, execution is described as a series of steps, each of which consists of a term in the language being rewritten as another

Figure 4.1: Mappings between programs P, an input language I, an output language O, and two bidirectional languages F, G



term in the language according to a set of well-defined rules. An example is β-reduction in the λ-calculus, arguably the most important rule in that system. This is defined $(\lambda x.M)y \rightarrow M[y/x]$ for expressions y and M, where the latter has a free variable x; the parameter value y is substituted for x wherever it occurs in M.

In the case of a deterministic system, such as the λ-calculus with a fixed reduction order, the set of term rewriting rules describe a map from programs to programs. In the case of non-deterministic systems such as the π-calculus, the reduction rules do not describe a map, as each term does not necessarily have a unique successor that it evolves to. For example, consider the following:

$$x(y).P|x(y).Q|x < z > .R$$

$$\swarrow \qquad\qquad \searrow$$

$$x(y).P|Q[z/y]|R \qquad\qquad P[z/y]|x(y).Q|R$$

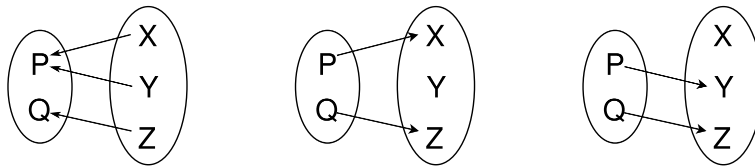
In this case, the message z may be received by either the $x(y).P$ or $x(y).Q$ (but not both). The "choice" of process is non-deterministic, and cannot be predicted in advance.

One advantage of using a term rewriting approach in a multi-language setting is that it provides a convenient method of debugging; the running program may be examined after any step. This program may be translated, using the existing mechanism, into a text in any supported language. This allows the user to view the program at any stage of execution, in any supported language, and gives them the same latitude of choice of language for debugging as for other tasks.

4.4 Secondary Notation in Multiple Languages

As mentioned above, for bidirectional translation to function correctly it is necessary that each text in a given language corresponds to exactly one program, and conversely that each program corresponds to exactly one text in that language. However, the same program is also mapped to exactly one text in each of the numerous other languages in the system. These texts are not necessarily equivalent. A similar phenomenon occurs in natural languages (see Lyons (1981), § 5.3). For example, the English words "sheep" and "mutton" (the meat of a sheep) are both translated to a single word, *mouton*, in French, while the French words *libre* (unrestricted) and *gratis* (without charge) are both translated (in modern, everyday English) as "free". Similar disparities also occur at the phrase level, making it impossible to find an exact analogue for a piece of text in one language in another; information is "lost in translation".

Figure 4.2: A non-bijective mapping (left) and its sections (centre and right)



Similarly, the texts of one language may differ in nuances not expressible in texts of a second language. If a reversible mapping is to be achieved, additional, hidden features must be added to the texts of the second language in order to reflect these nuances. As with natural languages, this works both ways; there may be nuances expressible in the second language, but not the first.

More precisely, execution is best viewed as an operation on programs rather than texts. As such, programs differing in features that affect execution must be distinct. However, for most languages, texts have features that are not reflected in execution. In conventional compilation, such features are discarded, in effect mapping two texts to the same program. This map can be viewed as grouping texts into equivalence classes. It does not have an inverse, but, assuming that for every program there is at least one text that maps to it, it does have *sections* such that the map composed with one of its sections is the identity on programs. In effect, a section maps each program to one of the (possibly many) texts that maps to that program. For example, consider the case with two programs, P and Q, and a language with three texts, X, Y and Z. Suppose the translation from texts to programs maps X and Y to P, and Q to Z. This map has two sections (shown in Figure 4.2); one maps P to X, and the other P to Y. Both sections map Q to Z.

For any given mapping, many sections may exist, as a program may be mapped to any text that translates to it. This suggests that the set of texts in a language with such a map may be partitioned into subsets of texts that map to the same program (an equivalence class). Given that these texts are equivalent in terms of execution, what distinguishes them from one another?

The term *Secondary Notation* (Petre 1995) is used to describe the features of a text that do not affect execution, but that the user may nevertheless modify (such as comments and spacing in textual languages). These are the features in which “equivalent” texts differ. While it does not affect execution, secondary notation is of immense importance to humans who read, write and modify the program. Oberlander (1996) examines the effects of secondary notation in visual languages, introducing concepts from linguistic pragmatics. In particular, he suggests that a viewer will assume that secondary notation features have been included for a reason, and associate meaning with them. If the features are accidental, there is a significant risk that the user will use them as the basis for incorrect inferences about the system.

The types of secondary notation available vary significantly between languages. Examples in textual programming languages include naming of features such as variables and functions, and indentation and layout of source code. In visual languages, however, factors such as positioning and scaling, colour and shape may be used. It is not possible, in general, to provide a satisfactory translation between these disparate forms of secondary notation in all cases.

If the map between a language and the universe of programs is to have an

Figure 4.3: Structural Variations in C Programs

```
int f(y) {
  int x=1, i=0;
looptest:
  if (!(i < 10))
    goto loopend;
  x *= y;
  i += 1;
  goto looptest;
loopend:
  return x;
}
```

```
int g(y) {
  int x=1, i=0;
  while (i < 10) {
    x *= y;
    i += 1;
  };
  return x;
}
```

```
int h(y) {
  int x=1, i;
  for (i=0; i<10; i+=1) {
    x *= y;
  }
  return x;
}
```

inverse (as opposed to several sections), texts that are distinct only in terms of secondary notation must map to distinct programs. It follows that, to ensure that the map for a second, different language also has an inverse, these distinct programs must correspond to distinct texts in that language. This presents problems if texts in the second language do not have appropriate features to represent the distinction.

Consider the case where the first language is a visual language, and the two texts differ only in the shape used to represent a certain element of the program (a circle in one text, a triangle in the other). These two texts are mapped to distinct programs, which are in turn mapped to distinct texts in the second language, a textual language. While the same program element is represented in all four texts, the distinction between a circular representation and a triangular one is meaningless in the second language. As this is the only difference between the two programs, it would seem that the two texts in the second language must be indistinguishable. However, they must be distinct texts in order to fulfil the requirements for bidirectional translation.

For certain sets of languages, it might be possible to devise a uniform scheme of secondary notation, in which all types of secondary notation have representations in all languages. However, this is difficult in all but the simplest cases, and is likely to place undesirable constraints on the types and flexibility of secondary notation. A better solution is to extend languages such that texts that differ only in some secondary notation feature not representable in the language are distinct. The form of this extension takes the form of adding "hidden" features to texts in the language, allowing superficially similar texts to be distinguished.

4.5 Structure as Secondary Notation

The presentation of secondary notation has been limited to arbitrary information associated with a particular program element. Perhaps surprisingly, the way in which code is structured also fits the definition of secondary notation given above.

Consider the three functions in Figure 4.3. All three functions are identical in terms of execution (in that they produce identical object code, aside from labels, when compiled). Moreover, annotation of program elements (for example, naming of variables), is comparable in all three versions. The significant difference between the three functions is the way the code is structured; this does not affect execution, and hence falls into the definition of secondary notation.

When attempting to decompile code into a language such as C, such structures present a problem. The generated object code (in most environments) will correspond almost exactly to the first version of the code (aside from things such as labels and variable names). However, the second or third form is likely to be more comprehensible to programmers. Hence, decompilers attempt to recognise patterns in the generated code corresponding to higher-level structures. This approach is hampered by the fact that optimising compilers may reorder instructions in potentially complex ways to improve performance. Because of this, the approach is fragile, in that it does not work well if applied to files generated by a different compiler, much less from a different source language. Additionally, there is no way of distinguishing between functionally equivalent forms that may occur in the source code. As such, while decompilers are an invaluable tool to aid in the reverse engineering of software, they are a far cry from the exact reconstruction of the original source code.

The vast majority of structures in most programming languages may be described in terms of properly nesting groups of program elements. For example, a C “while” loop is a group of two groups of program elements; a condition and a body. Similarly, a “for” loop is a group containing another group, containing three subgroups (the initialiser, condition and advancement), and a group constituting the body of the loop. This suggests that grouping may be a useful basis for the representation of structure.

Syntactic structures such as loops are represented by a group containing program elements and groups. All groups are annotated with a “role”, describing the structure or substructure represented by the group. This may be used to determine the representation of the structure in a particular languages. There are four possibilities:

- If the role corresponds to a structure representable in the language, it may be fully represented.
- If not, it may be represented by grouping.
- The group may be ignored, and the contents are represented in the same way as they would be if not annotated.
- The group may be represented “opaque”; the group is represented as an atomic object of some kind, and the contents are not represented at all.

These choices are analogous to the choices that Meyers presents for representing features of semantic program graphs not available in a particular view. For most structures, the contents of the group should be restricted. For example, the C while loop should contain exactly two groups. If a group does not conform to such restrictions, it is not possible to simply represent that group as the desired structure. If this is the case, it may be necessary to fall back to one of the alternative representations suggested above. Restrictions on the content of groups may be enforced using a simple type system modelled after XML Document Type Definitions (Bray et al 2000) or Schemas (Fallside 2001), (Clark 2001).

In addition to their role, groups may have other secondary notation annotations associated with them. One particularly common annotation is a name.

Others might include shape, colour or position. In this context, a group need not necessarily reflect any structure beyond grouping; it may simply signify that a set of program elements are associated, and certain annotations are common to all of them.

4.6 An Environment for Multi-Language Programming

It would be possible to implement the proposed architecture as a set of command line tools, both "compilers" and "decompilers", to translate between files in the intermediate form and files in various programming languages. Equally, it could be implemented in the form of an Integrated Development Environment similar to Microsoft Visual Studio[17] or Eclipse[18]; indeed, both of those systems have sufficiently powerful extension mechanisms to implement the system as a component. However, neither implementation meshes well with novel, non-textual languages such as the Media Cubes. Furthermore, they demand the user's attention, and are tied to traditional computing environments (keyboards, monitors and mice), and as such are far from ideal components of a ubiquitous computing system.

A more appropriate solution is a centralised program database, with which other components communicate via a network. This approach, taken by Meyers' SPG architecture, decouples implementations of the specific language mappings from generic functionality such as storage and version control. More importantly, as the "real" code (i.e., the code held in the database) may only be modified via a well-specified interface, integrity checks may be enforced, ensuring that the code in the database is valid. A network interface allows a wide variety of programming front ends, ranging from command line compilers to tangible programming languages, to communicate with the system with equal ease.

---

[17] `http://msdn.microsoft.com/vstudio/`
[18] `http://www.eclipse.org/`

Chapter 5
The Lingua Franca Architecture

This chapter discusses Lingua Franca, a framework designed to facilitate the use of multiple scripting languages. Unlike most other multi-language architectures, the Lingua Franca framework allows different languages to be used for the same program, at different stages in its development. This is achieved by translating between various source notations and an intermediate form, dubbed Lingua Franca.

5.1 An Overview of the Lingua Franca Architecture

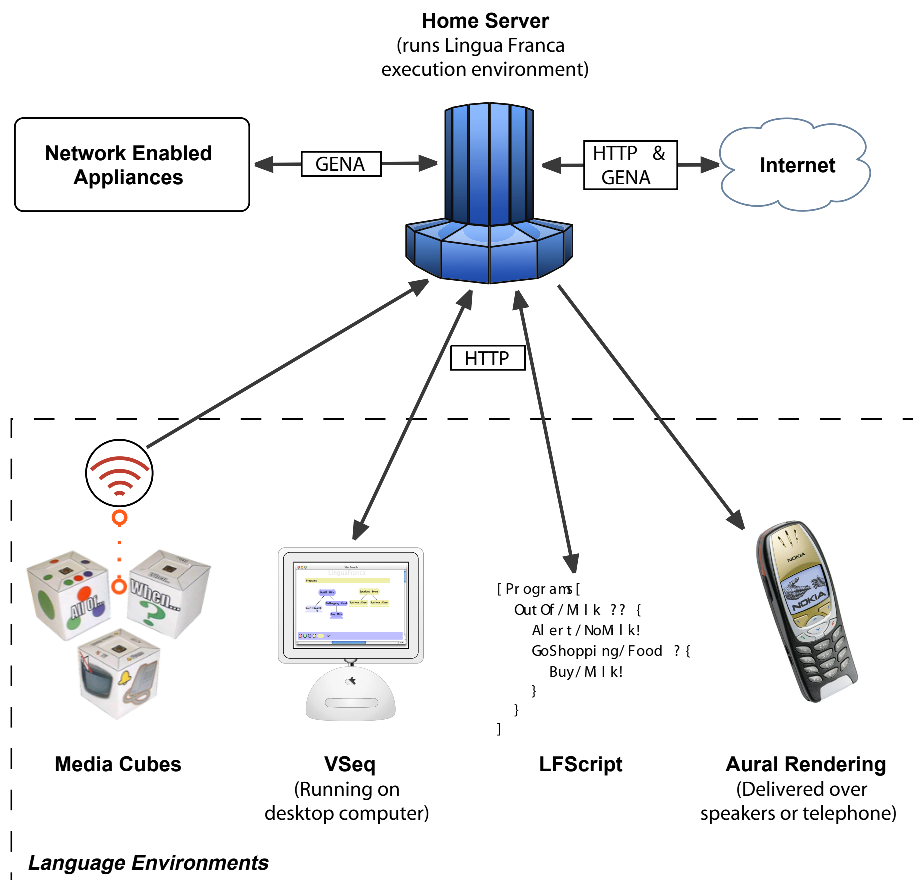
I use the term "Lingua Franca" to describe two things; an intermediate form supporting reversible translation, in which programs may be represented, and the software architecture supporting the use of this intermediate form. In this section, I give an overview of the latter, in order to provide a context for the description of the former.

Central to the Lingua Franca architecture is the *execution engine*, a piece of software continually running on a host connected to the home network (the "home server"), as described in Section 4.6. This software is responsible for the storage of the *corpus*, the body of Lingua Franca code held in the system, and handles execution of that code. It provides an interface for adding, removing and modifying code in the corpus over the network, and also interfaces Lingua Franca code with other devices and software in the home network.

The architecture makes use of several of the technologies described in Section 2.2. The external representation of Lingua Franca, used when transferring code between components of the system, is a dialect of XML. Modification of the corpus is performed via HTTP, using standard request types. The interface is comparable to, but simpler than, WebDAV. GET requests retrieve a portion of the corpus, POST requests insert new code at a specified point, and PUT requests replace a specified portion of the corpus with new (in practice, modified) code. The corpus is represented as a single XML document, and the path of the request is interpreted using the XPath language (Clark and DeRose 1999), allowing the requester to specify any point in the document tree. XPath expressions are used to represent a path from the document root to a node (or multiple paths to multiple nodes), based on node types, attribute values and other criteria. In particular, it allows queries based on a node identifier, optionally constrained to a subsection of the document. This type of query is by far the most common in the case of Lingua Franca.

The end-user programming languages implemented on top of Lingua Franca are embodied within *language environments*. These pieces of software run on a client device somewhere on the network, and communicate with the execution engine over the network, via HTTP as described above. The relationship is shown in Figure 5.1.

Figure 5.1: The Lingua Franca architecture



5.2 The Lingua Franca Execution Model

The Lingua Franca execution model is loosely based on the π-calculus (Milner 1999), with certain important differences. These stem from the decision to model events in a similar way to GENA (Cohen et al 2000), a publish-and-subscribe protocol that is used for communicating asynchronous events in both UPnP and the AutoHAN project.

The model is based on the notion of communication between *processes*, via discrete messages (*events*). Following GENA, events consist of a *notification type* (NT), and a *notification subtype* (NTS), and optionally additional data (the *payload*). The entire event may be considered to be a set of name-value pairs. There must be at least a value corresponding to the name `nt`, which is taken to correspond to the notification type, and, if there is a value corresponding to the name `nts`, this is taken to correspond to the notification subtype. The payload may provide one or more additional name-value pairs. Such pairs have no inherent meaning to the Lingua Franca, but their values may be used as the notification type or subtype of subsequently emitted events. This is analogous to passing a name over a channel in the π-calculus. Note that GENA manages subscriptions and delivery on the basis of notification type only; subtype and payload are ignored.

The state of the execution engine is modelled as a set of *processes*. Each process is listening for events meeting some specific criteria. When a process receives such an event, it evolves into zero or more processes, as dictated by the nature of the process and the event received. In addition, one or more new events may be emitted, again based on the specific details of the process and the received event.

Lingua Franca is based *multiple dispatch*, as opposed to *single dispatch* as used in the π-calculus. In the former scheme, an event is received by *all* processes listening for such events. Conversely, in the latter scheme, exactly one process (selected nondeterministically) receives any given event. The use of multiple dispatch allows Lingua Franca to model the underlying event mechanism directly, at the expense of being unable to apply theoretical results about the π-calculus to the system.

Several other theoretical frameworks also share similarities to the Lingua Franca model. CBS (Prasad 1995) is a "CCS-like calculus where processes speak one at a time and are heard instantaneously by all others"; this model of communication is comparable to that of Lingua Franca, but CBS is a first-order calculus, and requires an underlying language to be Turing-complete. HOBS (Ostrovský et al 2002) is a Turing-complete, higher-order calculus based on broadcast communications, with extensive support for encapsulation. It is, however, complex, and a simpler framework was considered adequate for the application in question. The asynchronous π-calculus (Boudol 1992) explores the use of asynchronous, but still point-to-point, communication in a π-calculus like system. Finally, the bπ-calculus (Ene and Muntean 2001) takes a similar approach to Lingua Franca, namely to produce a variant of the π-calculus based on broadcast communication. This system is the most similar to Lingua Franca, and would provide the best starting point for a formal analysis of the latter.

The simplest type of process dispatches an event and then terminates. This is represented as follows:

$$E/e!$$

Where E is the notification type, and e is the notification subtype. Note that, following Turner (1995) and similar work, dispatching processes in Lingua Franca always terminate immediately; this differs from the pure π-calculus as described by Milner (1999), but simplifies both theory and implementation without significant loss of expressiveness. In some cases, events do not have a notification subtype; in such cases, a hyphen is used in its place. A dispatched event may optionally be augmented with a "payload". The following dispatch processes show these features:

$$E/-! \qquad E/e(name_1 = value_1, name_2 = value_2)!$$

Another fundamental type of process receives an event, and then evolves into another process. This is represented:

$$E/e?P$$

This process would evolve to the process P when an event with notification type E and notification subtype e was received. It is possible to leave the notification subtype unspecified. This is represented:

$$E/*?P$$

In this case, an event with notification type E would cause the process to evolve into P regardless of its notification subtype. Furthermore, it is possible to bind a name to the specific event received:

$$E/*(x)?P$$

In this instance, the name x is bound to the specific event received when the process evolves. This binding is limited in scope to the evolved process P. Within this process, the name may be used to access *components* of the events, using the following notation:

$$x.c$$

Where x is a name bound to an event, and c is the name of a component. All events have components named nt and nts, corresponding to the notification type and notification subtype respectively, and other components may be derived from the event payload. Components may be used in the place of notification types and subtypes, as in the following example:

$$E/*(x)?E'/x.nts!$$

Here, an event with notification type E is received and bound to the name x. The process then evolves to $E'/x.nts!$ with that binding in scope. This process dispatches an event with notification type E' and the same notification subtype as the received event (which was bound to x), and then terminates.

New names may also be introduced using the ν operator, borrowed from the π-calculus:

$$\nu x(E : x!)$$

Here, x is bound to a "fresh" event; the scope of the binding is the specified process, in this case $E : x!$. The generated value has only a single component (`nt`), and hence the component specifier may be omitted without ambiguity. Each instance of the operator generates a unique notification type, that is, one that does not occur anywhere else. As the scope of the binding is limited, the notification type may be used to ensure that a messages may only be received by a limited set of processes; this is particularly useful in combination with replicated input, described below. It is important to note that, as notification types may be passed to other processes via the subtype or payload of an event, the new notification type may become known to processes outside the ν expression. This phenomenon is known as *scope extrusion*, and its analogue in the π-calculus is discussed in Milner (1999).

Multiple processes may be combined into a single process using *parallel composition*:

$$P|Q$$

Where P and Q are processes. The parallel composition allows each process to react to events as it would otherwise. Assuming that $P \xrightarrow{E} P'$ (meaning that event E causes process P to evolve into P') and $Q \xrightarrow{F} Q'$, then:

$$P|Q \xrightarrow{E} P'|Q$$

$$P|Q \xrightarrow{F} P|Q'$$

Furthermore, if $P \xrightarrow{G} P''$ and $Q \xrightarrow{G} Q''$, then:

$$P|Q \xrightarrow{G} P''|Q''$$

Both composed processes receive the event. This behaviour is described as *multiple dispatch*. In contrast, the π-calculus has *single dispatch* behaviour, in which only each event is received by only one process (chosen nondeterministically). The two behaviours are comparably expressive; Lingua Franca is based on multiple dispatch in order to model the behaviour of GENA.

The processes described so far are finite, in that they interact with the environment a finite number of times, and then terminate. Following Milner (1999),

Chapter 9, it would be possible to model infinite processes via named definitions, but the same effect may be achieved more generally by introducing replication. As with many practical π-calculus based systems (Turner 1995), we restrict this to replicated input, represented:

$$E/e(x)??P$$

The difference is that, instead of evolving to

$$P'$$

(P with the received event bound to x) on receipt of a matching event, this process evolves to P composed with the original process, i.e.:

$$E/e(x)??P|P'$$

Note that, unlike the analogous construct in the single-dispatch π-calculus, Lingua Franca replicated input cannot be defined in terms of simple input and a generic replication operator, as a single event should not react with more than one copy of the replicated input.

Lingua Franca represents alternate choices using the summation construct, consisting of a set of receive processes joined with the + operator:

$$(A/a?X/-!) + (B/b?Y/-!) + (C/c?Z/-!)$$

At most one of the receivers in a sum reacts to an event. When one receiver reacts, the others are removed. For example, the above process reacts to an `A:a` event and evolves to:

$$X/-!$$

The sum construct, in combination with the ν operator and replicated input, allow a wide variety of structures to be implemented within the framework.

5.3 Examples of Lingua Franca execution

Many automation tasks in the home are comparatively simple; for example, muting the telephone whenever the doorbell rings. Assuming that the system generates and responds to appropriate events, such tasks translate directly into Lingua Franca:

$$\texttt{Doorbell/Rings}??\texttt{TV/Mute}!$$

Whenever the `Doorbell/Rings` event occurs, the above process evolves into the parallel composition of a copy of itself and an dispatch process. The latter immediately emits the `TV/Mute` event, and terminates. The end result is identical to the original process, and hence will respond to further `Doorbell/Rings` events in the same way.

Simple processes such as this are useful but limited, in that they only react to a specific event. The binding constructs of Lingua Franca may be used to generalise over all events with a given NT. For example:

$$\texttt{OutOf}/*(x)??\texttt{Order}/x.nts!$$

This process behaves in a similar way to the previous example, with one important difference; the emitted `Order` event has the same NTS as the `OutOf` event

that generated it. More complex actions are also possible; for example, the process may be modified to delay emitting `Order` events until some other event has occurred:

$$\texttt{OutOf}/*(x)??(\texttt{Reorder}/-?\texttt{Order}/x.nts!)$$

Whenever the modified process receives an `OutOf` event, it evolves to a copy of itself, composed with a receiver for the `Reorder/-` event, with x bound to the particular event received. For example, an `OutOf/Milk` event would cause the process to evolve to:

$$\texttt{OutOf}/*(x)??(\texttt{Reorder}/-?\texttt{Order}/x.nts!)\Big|\texttt{Reorder}/-?\texttt{Order}/\texttt{Milk}!$$

The copy of the original process may receive further `OutOf` events, and generate further receivers. Thus, an `OutOf/Bread` event would cause the above process to evolve to:

$$\begin{aligned}\texttt{OutOf}/*(x)??(\texttt{Reorder}/-?\texttt{Order}/x.nts!)&\Big|\texttt{Reorder}/-?\texttt{Order}/\texttt{Milk}!\\&\Big|\texttt{Reorder}/-?\texttt{Order}/\texttt{Bread}!\end{aligned}$$

When a `Reorder/-` event is received, the waiting receivers evolve into their result processes. In this case, the result is:

$$\texttt{OutOf}/*(x)??(\texttt{Reorder}/-?\texttt{Order}/x.nts!)\Big|\texttt{Order}/\texttt{Milk}!\Big|\texttt{Order}/\texttt{Bread}!$$

The two dispatch processes then immediately emit the appropriate events (`Order/Milk` and `Order/Bread` respectively), and terminate, leaving only the copy of the original process.

Replicated input in Lingua Franca introduces a process that listens *forever*; there is no way to terminate such a process. In practice, most programs are intended to be finite in their extent, and this therefore presents a problem. Termination of such program may be provided by the environment (external to Lingua Franca), but it is also possible to construct finite but repeating processes within the architecture.

While all replicated receivers are perpetual, it is in some cases possible to guarantee that a particular replicated receiver will not receive any further events (at which point it may be safely ignored or removed). Such a guarantee may be made when the receiver is predicated on a new name, and there are no potentially active processes remaining in the scope of the ν operator. For example:

$$\nu x\Big(x/*(y)??y.nts/y.nts!\Big|x/A!\Big|x/B!\Big|x/C!\Big)$$

Three events with the fresh notification type are dispatched. These react with the replicated input, resulting in the dispatch of events that do not contain this notification type. For example, the first dispatch emits x and terminates. The event reacts with the replicated input, resulting in:

$$\nu x\Big(x/*(y)??y.nts/y.nts!\Big|x/B!\Big|x/C!\Big|A/A!\Big)$$

Consequently, the event A/A is emitted. The processes $B/x!$ and $C/x!$ have similar effects. Overall, the process emits the events A/A, B/B, and C/C (in some order, chosen nondeterministically), and evolves into the following process:

$$\nu x(x/*(y)??y.nts/y.nts!)$$

Figure 5.2: A First-In, First-Out Queue in Lingua Franca

$$
\begin{array}{l}
\texttt{Queue}/*(q)??(\texttt{Append}/q.nts(x)? \\
\quad \nu\ tail(\\
\quad\quad \texttt{Queue}/tail! \\
\quad\quad |\texttt{Append}/q.nts(z)??\texttt{Append}/tail(vnt = z.vnt, vnts = z.vnts)! \\
\quad\quad |\texttt{Get}/q.nts?(x.vnt/x.vnts!|\texttt{Get}/q.nts??\texttt{Get}/tail!) \\
\quad) \\
)
\end{array}
$$

It is clear in this case that the replicated receiver will never be activated again, as the name `x` is unknown outside the scope of the ν operator. In more complicated cases, it is not as clear-cut, as it is possible to pass names within events, leading to scope extrusion as mentioned previously.

Using this technique, it is possible to define processes with a “guard” event, such that they repeatedly receive events of one kind, until an event of another kind occurs, at which point they become inactive. This is the basis of *until* clauses in LFScript; see Section 6.3.3.

As a final example, Figure 5.2 shows a process that implements a first-in, first-out (FIFO) queue. A queue is constructed by emitting an event with the NT `Queue`; the NTS of this event is used as an identifier for the queue in further operations. Events of the form `Get`: q cause the next event on the queue, if any, to be emitted. The queue is initially empty; there are no receivers for `Get` events. Events may be added to the queue using an event of the form `Append`: q, with a payload mapping `vnt` and `vnts` to the NT and NTS of the event to add to the queue. When the first such event is received, an empty queue evolves into three processes. The first dispatches a `Queue` event with a fresh identifier, creating a “tail” queue to represent all other items on the queue. The other two respond to the queue events; the first forwards all further `Append` events for this queue to the tail queue, and the second responds to a `Get` event by emitting the event appended to this queue, and evolves to a process that forwards all further `Get` events to the tail queue. Each “cell” of the queue (corresponding to a single event) is perpetual. As such, this implementation of a queue is inherently inefficient, and grows more so over time, as the cost of a get or append operation is proportional to the number of events that have *ever* been in the queue. Nevertheless, it serves to demonstrate the flexibility and expressiveness of the system.

5.4 The external representation of Lingua Franca

In order for Lingua Franca code to be transmitted over a network, or stored in a file, it is necessary to define an external representation, or serialised form, for it. As well as representing the executable part of Lingua Franca, it must also be able to include arbitrary data associated with secondary notation. In addition, given the heterogeneous nature of home networking systems, it must be cross-platform.

I have chosen XML (see Section 2.2) for the external representation of Lingua Franca. This is a marked contrast to most programming languages, and most intermediate forms, which use ad hoc text syntaxes and binary formats respectively. However, in the case of Lingua Franca, interoperability with other components of the home network is essential, while the rate of transactions, and

Figure 5.3: **XML representation of the execution features of Lingua Franca**

```
<repreceive xmlns="urn:linguafranca" bindevent="x">
  <nt>A</nt>
   <receive>
     <nt>B</nt>
     <nts>b</nts>
     <dispatch>
       <nt>C</nt>
       <nts><param event="x" name="nts"/></nts>
     </dispatch>
   </receive>
 </repreceive>
```

hence the need for runtime efficiency, is likely to be low. Consequently, XML is an appropriate choice of interchange format.

Several existing programming languages use XML as a concrete syntax. The most notable of these is XSLT (Clark 1999), a language used to transform XML documents. The use of XML, as opposed to an ad hoc syntax, allows processors to use established and commonly available parsing software where they would otherwise have to implement a custom parser (a relatively complex and error-prone activity for all but the most trivial syntax). Again, the chief aim is interoperability. A major application of XSLT is for a program (or *stylesheet*) written in the language to be downloaded to a web browser, in order to transform another document prior to display. The success of this application rests on compatibility between different XSLT implementations.

The representation of Lingua Franca in XML is based upon a straightforward translation of the expressions used to describe the execution model. Figure 5.3 shows the XML representation of the Lingua Franca corresponding to the expression:

$$A/*(x)??(B/b?C/x.nts!)$$

This is a process that, each time it receives an event with a notification type A, creates a process that, upon receiving an event with notification type B and notification subtype b, dispatches an event with notification type C and the same notification subtype as the event originally received, and then terminates.

The XML representation of Lingua Franca makes use of XML Namespaces (Bray et al 1999). The executable elements of Lingua Franca reside in the namespace `urn:linguafranca`. The example uses the `xmlns` attribute to specify this namespace as the default, obviating the need to specify a prefix for each element name.

There is a one-to-one correspondence between the XML elements and the components of the expressions previously used. The `receive`, `repreceive` and `dispatch` elements corresponds to ?, ?? and ! respectively. Within these ele-

ments, the `bindevent` attribute is used to specify binding (x), and `nt` and `nts` elements are used to specify notification type and subtype (E/e). If the latter is omitted, the containing expression matches events with any NTS ($E/*$). `dispatch` elements a required to specify both NT and NTS, and may have a payload, specified by `payload` elements with `name` attributes. Other children of `receive` and `repreceive` elements specify the resultant process. The `param` tag is used to recall the values previous bound to names. A feature not shown in the example is parallel composition of processes; this is achieved simply by placing the processes as siblings. The ν operator is represented by the `new` element, with a `bindevent` attribute to specify the name introduced, and children representing the processes affected. Summation is represented by the `sum` element, which contains a collection of `receive` children.

The mapping used produces results that are verbose, even by the standards of XML. This is in large part due to the use of elements as opposed to attributes in almost all cases. The reason for this is chiefly to allow the use of an element (`param`) to dereference bound names. In an obvious alternative formulation, the NT and NTS of events would be specified by attributes as opposed to child elements. As XML attribute values are unstructured, this formulation makes the task of distinguishing between name dereferences and literal event specifiers problematic. One solution would be to use some special syntax (for example, preceding the name with a `$` character) within the value. However, this syntax would be unknown to XML tools, so would require non-standard processing, and would not be checked via the standard techniques. Another alternative would be to introduce a parallel set of attributes (say, `bnt` and `bnts`) corresponding to dereferences. However, this would also necessitate integrity checks (for example, that no element had both `nt` and `bnt` attributes) beyond the scope of standard tools. A third solution would be not to distinguish between bound names and literals at all syntactically, and dereference any name that is bound in the given context, treating all other names as literals. This would be likely to make the language more confusing (as it is not immediately clear if a given name was intended to be literal or not), could lead to inadvertent variable capture, and makes passing names as event data problematic. None of these solutions are satisfactory, and hence the verbose notation is used. The verbosity is less problematic than it would be in a conventional textual programming language, however, as the XML representation of Lingua Franca is a mechanism for exchanging programs between software, and is not intended to be manipulated directly by programmers.

In parallel to the executable elements, Lingua Franca also includes elements corresponding to the general secondary notation structures described in Chapter 4. `group` nodes represent grouping. The `name` attribute provide name for use by programmers; this is typically displayed by language environments. It may also be used in XPath expressions to find the group. Optional `creator` and `role` attributes are used to identify, respectively, the language environment that created the group and the specific type of group. These attributes are used by language environments to determine how (or if) the group should be displayed. `group` nodes may have arbitrary children.

`note` nodes are used to add point annotations to Lingua Franca programs. As with `group` nodes, `creator` and `role` attributes are used to determine the exact nature and presentation of the node. The content of a note node depends on its role; it may be text or XML, and encodes whatever data the node represents.

5.5 Operations on the Lingua Franca Corpus

Clients (specifically, language environments) query and modify the Lingua Franca corpus indirectly, via the execution engine. In the tradition of object-oriented programming, a small set of operations are provided, and the corpus may only be modified via these operations.

Clients access this functionality over the network using HTTP 1.0. The path component of the URL specifies a node in the corpus, using XPath language (Clark and DeRose 1999). This language allows searching based on path from the root (document) node, and provides a wide range of flexible predicates to narrow the search further. However, Lingua Franca subtree specifications typically only use a subset of this functionality; queries are generally solely based on either path from the corpus root, or an `id` attribute.

GET requests are used to retrieve the section of the corpus consisting of the node specified by the path and all of its children. This is primarily useful for presenting Lingua Franca code to the user. In certain language environments, the presented code may be modified and resubmitted to the corpus; This allows programs to be edited.

PUT requests replace the node at the path, and all its children, with new Lingua Franca code specified in the message body. These requests are assumed to replace existing content, so there is a danger that a user will inadvertently (or maliciously) destroy data in the corpus that is still of use. Fortunately, the separation between language environment and program database, combined with the HTTP protocol's support for returning an informative error message upon failure, allow an integrated versioning system to be added to the program database easily.

POST requests are used to add new content to the corpus, as a child of the node specified by the path. This is distinct from PUT requests in that no existing content is modified. In theory, it would be possible to use PUT for both cases by allowing PUT requests to specify a non-existent node. However, there are numerous problems with this approach. For example, if the node was specified by its position (e.g., the third child of a given node), it would be easy to inadvertently replace an existing child that had been added since the corpus was last examined. Support for revision control would allow clients to avoid this problem, but at the expense of significant additional complexity and possible extraneous requests to implement what is a very common operation. The inclusion of POST requests to add new content explicitly avoids these difficulties.

Finally, DELETE requests are used to remove subtrees from the corpus. Again, versioning may be built into the database to allow a user to roll back this action, and hence reduce the danger associated with it.

Chapter 6

The Implementation Of Lingua Franca

This chapter describes the implementation of the Lingua Franca architecture. In addition to the execution engine, which also serves as the program database, three language environments were produced in order to exercise the translation features of the architecture.

6.1 The Prototype Execution Engine

To enable investigation of the Lingua Franca system, a prototype execution engine was developed. This was not intended to reach the standards of reliability and runtime efficiency that would be necessary for a component of a home network, but rather to implement the execution and processing of Lingua Franca, and the external interfaces, such that clients (language environments) may be implemented and tested, and the techniques and issues involved with programming in multiple notations explored.

The prototype was implemented in Python[19], a bytecode-interpreted, dynamically-typed, object-oriented scripting language with an extensive standard library that includes XML support. It also includes support for higher order functions, and has several high-level data types built in. These attributes make it a good choice for rapid prototyping of applications. In addition, its chief deficiencies (lack of static typing, and poor runtime efficiency compared to languages compiled to native code) are not of concern for the development of the prototype.

The prototype design is based around standard XML technologies. In particular, it uses DOM (Le Hors et al 2003) as the data model for Lingua Franca programs, and XPath to specify areas of the program to act upon (for example, elements that are to be updated in response to a given event). This facilitates an implementation-independent view of the execution process embodied in the prototype, and provided flexibility that was invaluable as the syntax and semantics of Lingua Franca evolved. However, DOM trees, being a general representation of any XML document, are more complex than is required to represent Lingua Franca programs, and allow invalid Lingua Franca programs to be represented. For this reason, a dedicated Lingua Franca representation was developed when the architecture had matured.

Performance is not a priority in the prototype execution engine, and hence a very simple interpreter is sufficient. An instance of the `Handler` class manages the state and behaviour of the interpreter. The state consists of an XML document representing a Lingua Franca corpus, represented as a DOM tree. The main loop of the interpreter first selects the first active `dispatch` node (i.e., the first that is not an descendant of a `receive` or `repreceive` node), using the following XPath expression:

```
/lf/corpus/descendant::dispatch
    [not(ancestor::repreceive or ancestor::receive)][position()=1]
```

[19] `http://www.python.org/`

It then removes it from the corpus, and processes any `receive` or `repreceive` nodes that react to the event. If no visible `dispatch` exists, the interpreter pauses until one does.

The nodes that may potentially receive the event are those `receive` and `repreceive` nodes that are not descendants of another `receive` or `repreceive` node (and are hence active). Such nodes are found using the following XPath expression:

```
/lf/corpus/descendant::*
    [self::repreceive | self::receive]
    [not (ancestor::repreceive or ancestor::receive)]
```

This expression may be extended to find only those nodes that actually react with the given event simply by adding another predicate examining the `nt` and `nts` children of matching nodes. Receivers are processed by copying their children to the top level (specified by the XPath expression `/lf/corpus`), and, in the case of `receive` nodes, removing the original node. If the receiver binds the received event (using the `bindevent` attribute), any `param` nodes referring to the bound event are replaced with the appropriate value drawn from the particular event received.

Any visible `new` nodes are eliminated by generating a fresh event type and subtype, and replacing references to the bound event with values drawn from the fresh event.

`sum` nodes are handled by ensuring that, when one of its `receive` children is removed during a reaction, the `sum` node is also removed, along with its other children. The case where multiple `receive`s with the same `sum` parent react to the same event is handled by removing all but one (chosen nondeterministically) from the list of receivers before it is processed. A refinement, not presently implemented, would be to give priority to receivers that specify a notification subtype in addition to a notification type over those that specify only a notification type.

External events are handled by connection to a GENA subscription arbiter, specified when the execution engine is launched. Whenever an event is processed, the engine sends a corresponding GENA event to the arbiter. Similarly, when a GENA event is received from the arbiter, a corresponding `dispatch` node is appended to the corpus. The execution engine subscribes and unsubscribes to notification types such that the types subscribed to correspond exactly to the types received by currently active receivers. GENA subscription is based on notification type only, and events cannot be filtered on the basis of notification subtype; hence the events received are a superset of those that the corpus reacts to.

In addition to instantiating and interfacing with the `Handlers` class, the execution engine also includes an HTTP server to allow language environments to access and modify the corpus. The program is typically runs in “server” mode, in which it runs indefinitely and responds to network input. Two other modes of operation are provided for debugging purposes. GUI mode provides a graphical user interface to allow execution to be paused, or stepped through one reaction at a time, and allows the corpus to be edited directly. When run in batch mode, execution proceeds as normal, but exits when there are no pending reactions. This, in combination with the option to dump dispatched events and the final corpus state to be dumped to standard out, allows offline testing of Lingua Franca programs without the need for additional software.

6.2 The LFCore Toolkit

Early prototypes of Lingua Franca components were written in an ad hoc fashion, using existing technologies. The use of XML was a major advantage in this situation, as mature technologies with freely available implementations exist for parsing and document representation. However, as the design of the Lingua Franca architecture stabilised, the flexibility of generic technologies became less important. Hence, a dedicated toolkit was designed to facilitate the creation of Lingua Franca components, chiefly language environments.

LFCore provides a rich representation of Lingua Franca programs, and an API allowing the extension of this representation to include application-specific data. The latter is important to enable flexible handling of secondary notation.

LFCore is implemented in Java. That language was chosen for a variety of reasons. Foremost amongst these are its cross-platform nature, static typing and extensive support for both XML and standard networking protocols. Traditionally, Java has been seen as inefficient, in terms of run-time performance, compared to languages such as C. This is far less true of modern Java environments, and in any case is not a major issue for LFCore, as the intended application area does not require high throughput.

Lingua Franca programs and program fragments are represented as a tree, with each of the nodes represented by an object of some subclass of `LFNode`. Each node has links to both its parent and any children it may have. These fields are protected, and accessor methods ensure that they remain consistent across the entire tree.

A common problem associated with heterogeneous trees in object-oriented languages is extending the interface after its initial creation. Typically, nodes in the tree are represented by instances of classes conforming to some specific set of operations. This makes adding a new node type after the initial design simply a matter of creating a new class that implements the appropriate interface. However, adding another operation is difficult, as all of the existing classes must be modified. The situation in LFCore is the opposite of this; the node types in Lingua Franca are unlikely to change (while the definition of Lingua Franca remains the same), whereas each language environment is likely to add additional operations.

The *visitor pattern* (Gamma et al 1994) describes a structure to handle this situation. The `visitor` interface contains a `visit` method for each type of node. The node supertype has an `accept` method that takes a visitor as a parameter. This is overridden in each concrete node class to call the appropriate `visit` method. New operations are added simply by implementing a new subclass of the visitor class. LFCore has a single visitor interface, `LFVisitor`, with a `visit` method for each `LFNode` subclass (The method name `visit` is overloaded, based on the type of the parameter).

While the visitor pattern makes adding new operations easy, it also makes adding new node types difficult, as this requires a change to the visitor interface, and consequently to all classes implementing it. `LFVisitor` partially avoids this problem by providing a `defaultVisit` method that is called by the default implementation of each `visit` method. The default method allows visitors to provide sensible fallback behaviour for node types not known at the time of writing. However, it is not possible to provide customised behaviour for unknown subclasses, so this is only a partial solution.

In addition to implementing new operations, language environments are also likely to define nodes for novel forms of secondary notation. This is achieved

by subclassing the NoteNode class. As mentioned above, adding new classes to the visitor is problematic. Accordingly, the new classes are handled in the same way as instances of NoteNode by visitors. Java's reflection (run-time type information) facilities are used to distinguish between note classes. In most cases, a visitor will not act upon secondary notation nodes directly. Instead, the NoteNode children of a node are typically examined to obtain (and store) information regarding the presentation of the node. To this end, LFNode provides methods that return children of a specific class. These methods facilitate the use of novel NoteNode subclasses to associate arbitrary application-specific data with parts of a Lingua Franca program.

LFNode encapsulates the transformation of Lingua Franca programs to their XML external representation. An auxiliary class, LFParser, encapsulates the transformation from XML to an LFNode-based tree. It uses the event-based SAX XML parser interface, and so may employ any of the numerous XML parsers that implement that interface.

In most cases, there is a one-to-one correspondence between LFNode subclasses and XML tags. The exception is the note tag; as mentioned, language environments subclass this to store application-specific data. When a note tag is encountered, the parser determines the class to instantiate by examining the tag's role attribute. If this exists, and corresponds to a known class, then that class is instantiated. Otherwise, the generic NoteNode class is used. Subclasses may optionally override the methods used to transform an instance's data to and from its XML representation in order to parse the data provided.

Certain types of note node specify general information that may be of use to multiple language environments. There is nothing preventing one language environment from using note node subclasses defined by another, but to avoid multiple language environments implementing the same type of data in different ways, LFCore provides standard implementations. One such class is the BGColorNode class, used to associate a background colour with a node and all of its descendants; however, if a descendant has its own BGColorNode child, this overrides the ancestor's colour for the node and its own children. The content of the node is interpreted as a string encoding of a 24-bit integer value. Any encoding parsed by the decode method of the standard Java Integer class is accepted, but a hexadecimal encoding preceded by a hash (#) is preferred, and is the encoding generated by the class when the node is serialized to XML. An example is shown below:

```
<xax:note role="rgh22.linguafranca.common.BGColorNode"
          creator="VSeq">
  #aaaaff
</xax:note>
```

The creator in this case is typically unimportant; a language environment that associates meaning with the BGColorNodes should treat them identically regardless of the language environment that created them.

In addition, LFCore provides classes that may be generally useful when implementing language environments. An example is the IndexVisitor class. This is a subclass of LFVisitor that allows access to nodes in a tree via numeric index. When instantiated with a root node and optional class, the visitor traverses the

tree, recording all nodes that match the class (or all nodes if no class is provided) in the order they are visited (corresponding to depth-first order). Subsequently, the `get` method of the `IndexVisitor` instance may be used to obtain a node by index, and the `indexOf` method used to find the index of a particular node. The class is used to identify and retrieve unrendered `note` nodes in LFScript, as described in Section 6.3.3.

The LFCore library is organised in two packages (all Lingua Franca Java code — including LFCore and the various language environments — is organised in packages under the `linguafranca` package). The first, `lfcore`, contains the key components of the library — `LFNode` and its direct subclasses, `LFVisitor`, and `LFParser` — along with their dependencies. The second, `common`, contains utility classes such as `BGColorNode` and `IndexVisitor`. This division allows the core library to be imported without the (possibly unused) utility classes, and separates the essential components of the library from those that are more likely to change.

6.3 Prototype Language Environments

The following sections describe the language environments that have been implemented for the Lingua Franca architecture. The particular languages have been chosen to exercise the translation mechanism, and make use of the facilities that the environment provides. They are all designed for the purpose of programming a home network in a ubiquitous computing context, but serve different user groups, and are tailored to different programming tasks.

6.3.1 The Media Cubes Language in Lingua Franca

The Media Cubes language has been implemented as a Lingua Franca language environment. The implementation broadly follows that described in Chapter 3. As it is a write-only language, representing or tracking `note` nodes is not an issue. It is sufficient to provide a means of producing scripts in Lingua Franca form, and a method for submitting these to the server.

Producing a script in Lingua Franca form is a matter of mapping Media Cubes constructs onto those of Lingua Franca. This is largely straightforward. The conditionals produced by the Do-When cube correspond exactly to the `receive` and `repreceive` nodes. Atomic events specified as conditions are translated directly into `nt` and `nts` nodes used for event specification, while those used as scripts are translated into a `dispatch` node with the appropriate event specification.

Figure 6.1 shows the Lingua Franca script produced by the "shopping list" example presented in Section 3.5. This program will also be used in the following sections, to allow direct comparison of the languages produced.

The System cube's Submit face accepts a script, and make an HTTP POST request containing an XML representation of this script to add the script to the corpus. A fixed base URL for scripts created by the Media Cubes language environment is provided when the environment is launched, specifying both the network address of the execution engine to communicate with, and the point in the corpus to add created scripts to. Conventionally, this point will be an appropriately named group that is a direct child of the corpus root. Consequently, once scripts are submitted, they are immediately activated, and can react to events with no further action on the part of the user.

Figure 6.1: The shopping list program in Lingua Franca, in both formal notation and the equivalent XML

$$\texttt{OutOf}/{*}(x)??\left(\texttt{Reorder}/{-}?\mathrm{Order}/x.\mathrm{nts}!\right)$$

```
<repreceive xmlns="urn:linguafranca" bindevent="x">
  <nt>OutOf</nt>
    <receive>
      <nt>Reorder</nt>
      <nts/>
      <dispatch>
          <nt>Order</nt>
          <nts><param event="x" name="nts"/></nts>
      </dispatch>
    </receive>
</repreceive>
```

The Connection cube produces a higher-level structure in Lingua Franca, consisting of a `group` containing two `dispatch` nodes; these nodes dispatch connection events to each endpoint. The use of a group is not strictly necessary, as the Media Cubes language does not support reversible translation, and hence it need not be possible to reconstruct the higher-level structure from the generated Lingua Franca. However, it reflects the relationship between the nodes, and allows other language environments to render the construct appropriately, either as grouped nodes, or as a special Connection construct if one exists.

The Generalisation cube is implemented by adding a `bindevent` attribute binding the received event to a fresh name to the conditional node generated with the InEvent as a condition. Similarly, the OutEvent generates a `dispatch` node where the `nt` specification contains a `param` node referencing the same name.

The union of actions (scripts and emitted events) generated by the All-Of cube simply corresponds to parallel composition of the equivalent Lingua Franca elements. Producing a union of conditions (which we take to mean “All of these events happen, in any order, with no limit on time frame”) is not as clear cut, as Lingua Franca has no inherent notion of such a condition. Hence, another higher level structure must be introduced.

As is common in higher level structures, a `new` node is used to introduce a fresh event. This event is used in the place of the condition on the generated receiver; the action of this receiver is generated as normal. In addition, a tree of `sum` nodes is used to encode all possible orderings of the conditions; if these conditions all occur (in any order), the new event is emitted. As the `new` node guarantees that the name is fresh, the only receiver that reacts to the event is the conditional previously mentioned. An example of the generated Lingua Franca (for the union of three events `A/-`, `B/-`, and `C/-`) is shown in Figure 6.2

This formulation provides the desired functionality, but carries the risk of hugely inflating the size of programs; the need for explicitly encoding each pos-

Figure 6.2: **Lingua Franca for a union of conditions generated by the All-Of cube**

```
<group>
  <new bindevent="FRESH">
    <sum>
      <receive><nt>A</nt>
        <sum>
          <receive><nt>B</nt>
            <receive><nt>C</nt>
              <dispatch>
                <nt><param event="FRESH" name="nt"/></nt>
              </dispatch>
            </receive>
          </receive>
          <receive><nt>C</nt>
            <receive><nt>B</nt>
              <dispatch>
                <nt><param event="FRESH" name="nt"/></nt>
              </dispatch>
            </receive>
          </receive>
        </sum>
      </receive>
    </sum>
```

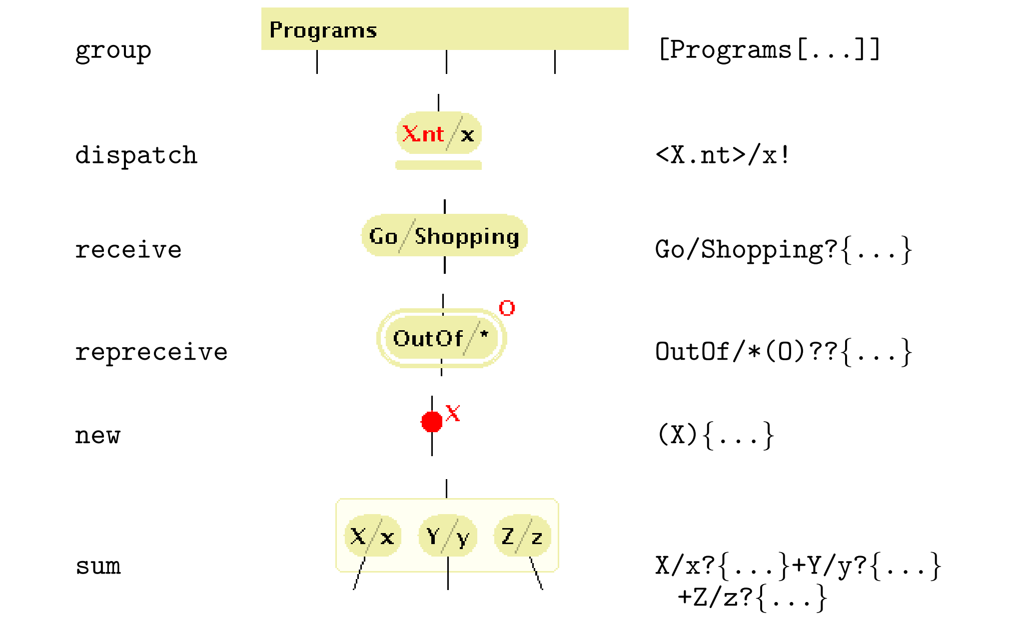
(Similar sum nodes for sequences starting with B and C)

```
    <receive>
      <nt><param event="FRESH" name="nt"/></nt>
```

(Generated action)

```
    </receive>
  </new>
</group>
```

Figure 6.3: **Lingua Franca Nodes in VSeq (centre) and LFScript (right)**



sible ordering of events results in programs growing with the *factorial* of the number of events that make up the compound. This is clearly unacceptable if the number of events is to grow beyond a handful. However, as mentioned in Chapter 3, other factors constrain the size of Media Cubes programs, so, while there is a problem in theory, it is unlikely to occur in practice. For similar constructs in other languages, an alternative, more efficient, formulation would be used.

6.3.2 VSeq — Visual Sequences

VSeq ("Visual Sequences") is a language environment that allows the user to manipulate Lingua Franca programs in the form of event diagrams, in which causal relationships are represented via a tree of nodes. It is likely to be the main environment for the manipulation of mid-sized scripts, as it offers visual access to the full range of Lingua Franca functionality, but may become unmanagable for large programs.

Language

VSeq represents a Lingua Franca program as a tree diagram broadly mirroring the structure of the underlying Lingua Franca code. Figure 6.3 gives a summary of the representation of various node types in VSeq. The basic nodes related to event handling are represented by lozenge-shaped elements labelled with a notification type and subtype. Without any additional graphical elements, such nodes correspond to `receive` nodes. Nodes with an outer border represent `repreceive` nodes. Nodes underlined with a bar represent `dispatch` nodes. As dispatch is terminal in Lingua Franca (i.e., a dispatch process always evolves into the empty process), such nodes are always leaf nodes of the diagram. At present, VSeq does not represent payloads for dispatched events; however, future revisions of the language would include this feature.

Figure 6.4: A VSeq `group` node, unfolded (left) and folded (right)

Binding is represented by adding the event name, in red, to the top right of a node (as in the `repreceive` node shown). Similarly, a red event specification component represents a reference to a bound name. `new` nodes are represented by a red circle, with a bound name represented as previously described. `sum` nodes are represented as a light-coloured box containing their `receive` children, represented in the normal way (but without a connecting line to the parent).

`group` nodes are represented by rectangular strips labelled with the group name. The automatic layout algorithm sizes the strip to be the combined width of all of the group's children, providing a visual representation of the extent of the group. To support selective elision of parts of the program, groups may be "folded" (and unfolded) by double-clicking on the title strip. When folded, only the title strip is shown; the children are hidden (as illustrated in Figure 6.4). While the children cannot be edited, the entire group may be repositioned, and moved from one parent to another, by manipulating the title strip.

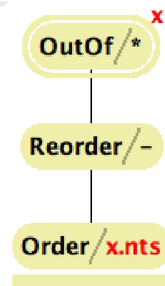
`note` nodes are not directly represented in the diagram. Instead, nodes with a creator "vseq" are used to determine properties of the graphical representation of their parent nodes, such as background colour. Other `note` nodes are not displayed. Secondary notation in VSeq consists of visual properties specific to VSeq, such as the size and position of an element's representation, and more general properties, chiefly background colour. These are encoded separately, to allow other language environments to express generic properties whilst ignoring those useful only in the context of VSeq.

The ordering implied by the tree corresponds to causality; the effect of a node is conditional on the event described by its parent. Edges in the diagram are directional, but are not labelled as such. Instead, the relative vertical position of connected nodes indicates the direction of the edge between them; the upper node is the parent, and the lower node is the child. The same convention is used in Hasse diagrams to indicate partial order over a set; indeed, causality forms a partial order over the set of nodes of the program.

Aside from the constraint that a node must be strictly below its parent on the page, graphical elements may be positioned as desired by the user. Additionally, VSeq incorporates a simple automatic layout algorithm, in order to position elements that have not yet been positioned manually by the user. When a piece of Lingua Franca code is initially loaded into the language environment, the algorithm examines each node of the tree. If the node is of a type that has a graphical representation, but it does not have an appropriate `note` child describing the specifics of that representation (such as position), an appropriate `note` is added. The position encoded by this node is selected such that the child is a standard vertical distance from its parent, and at a horizontal position such that is does not overlap any of its siblings representations in their *default* position (as they would be laid out by the algorithm).

The horizontal ordering of nodes does not affect the meaning of the program, in terms of execution. However, it corresponds to the ordering of elements in the underlying Lingua Franca document, which is itself an implicit form of secondary

Figure 6.5: The shopping list program in VSeq



notation (distinct from explicit secondary notation represented by group and note elements). This ordering will be reflected in most, if not all, language environments.

Figure 6.5 shows the shopping list program from Figure 6.1 as represented in VSeq. The language provides a succinct and readable view of simple programs such as this, while still having the capacity to show the majority of Lingua Franca structures.

Implementation

The layout algorithm is implemented via `LayoutVisitor`, a subclass of `LFVisitor` (see Section 6.2). `VisualPropertiesNode`, an application-specific `NoteNode` subclass, is defined to store the position of nodes. Instances of this class are added by the layout algorithm, and updated when the layout is modified by the user. Position is stored as an offset from the parent. The primary reason for this is to allow any part of a Lingua Franca corpus to be rendered without reference to any particular "root"; the relative positioning scheme allows any node to be designated as the root and placed at the origin. A useful side effect of the scheme is that the desired interaction behaviour, where any subtree may be repositioned by moving its root element, is achieved at no cost. The chief disadvantage of the scheme is that to determine the absolute position of a node, it is necessary to examine all nodes in the path from that node to the root. In practice, this has not been a problem. If necessary, `VisualPropertiesNode` could be extended to cache absolute position. Such cached data would not be represented in the XML form of the program, and hence would be calculated afresh each time the program was loaded into the language environment (as well as when user modifies the diagram).

A second subclass of `LFVisitor`, `RenderVisitor`, traverses the tree, rendering nodes based on the properties held in their `VisualPropertiesNode` children. It also examines any `BGColorNode` children (as described in Section 6.2) to determine the background colour. The background colour of an element's representation is a concept that makes sense in many contexts, and hence the common `NoteNode` subclass is used. In contrast, the position of an element within a VSeq diagram is only useful in a single context, and hence is represented by an application-specific class. Figure 6.6 shows a fragment of Lingua Franca XML containing both `VisualPropertiesNode`s and `BGColorNode`s.

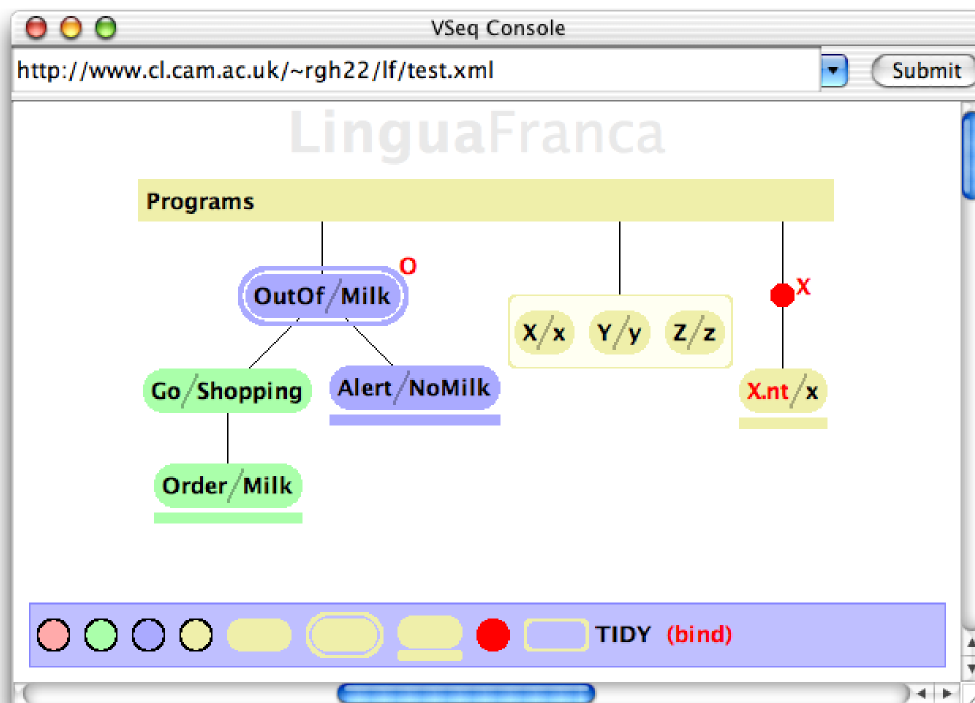
A third subclass of `LFVisitor`, `MouseVisitor`, is used to determine which node (if any) a mouse click hits. It is neccesary to use a visitor for this due to the relative positioning scheme mentioned above. Again, `VisualPropertiesNodes` are examined to determine the positions of nodes.

Figure 6.6: **Lingua Franca XML showing VSeq `note` nodes**

```
<corpus xmlns:xax='urn:xax' xmlns='urn:linguafranca'>
  <xax:group name='Programs'>
    <repreceive bindevent='O'>
      <nt>OutOf</nt>
      <nts>Milk</nts>
      <receive>
        <nt>Go</nt><nts>Shopping</nts>
        <dispatch>
          <nt>Order</nt><nts>Milk</nts>
          <xax:note creator='VSeq'
              role='rgh22.linguafranca.vsec.VisualPropertiesNode'>
            0,60,93,27
          </xax:note>
        </dispatch>
        <xax:note creator='VSeq'
            role='rgh22.linguafranca.vsec.VisualPropertiesNode'>
          -58,60,106,27
        </xax:note>
        <xax:note creator='VSeq'
           role='rgh22.linguafranca.common.BGColorNode'>
          #aaffaa
        </xax:note>
```

A "console" application, shown in Figure 6.7, allows the user to edit the diagram interactively. This is a standard desktop application, implemented in Java using the LFCore toolkit described in Section 6.2. It runs on any platform that provides Java 2 Standard Edition and the Swing user interface toolkit. The console may be used comfortably in a conventional desktop computing environment. However, the graphical elements and interaction methods are designed to be suitable for a lower resolution display than that typically found on desktop systems. In addition, text entry is kept to a minimum, and the automatic layout facilities and large graphical elements mean that accurate manipulation via the pointing device is not necessary. This is intended to allow the system to be used on low-fidelity displays such as televisions, and small devices such as personal digital assistants or web tablets. In addition, it should make the system more accessible to users with reduced motor control or visual acuity.

Figure 6.7: The VSeq console



The console presents a large drawing canvas on which the diagram may be assembled. The user may navigate around this canvas using standard scrollbars. A *watermark* (the grey text reading "Lingua Franca") is displayed at the origin. This gives the user an unobtrusive indication at to their position on the plane (and the position that will be visible next time the application is launched). If this indication was not present, successive editing operations may cause the diagram to "drift" away from the origin, and as a result be off-screen when the same program fragment is subsequently displayed.

The console allows the user to both view and edit the diagram. Nodes may be moved by dragging with the pointing device. Horizontal ordering of siblings is reflected in the sequential ordering of elements in the corresponding Lingua Franca program. Dragging one node over another adds it as a child of that node, provided that the resulting tree would correspond to a valid Lingua Franca program.

An important task performed by the VSeq console, and indeed all Lingua Franca language environments, is to constrain user input such that the program produced is a valid Lingua Franca program. In addition, language environments may enforce constraints specific to the particular notation in hand. In the case of VSeq, this includes ensuring that the vertical ordering remains consistent with the parent-child relationship; if a child is moved to a position above its parent, it is detached from that parent and added to the root (there is no visible representation of the root node; children of the root appear to be disconnected). Preserving this invariant means that vertical ordering of connected elements in a Vseq diagram reflects the causal ordering embodied by the corresponding program.

The console also includes a *toolbox* (the blue area towards the bottom of the

window), enabling a wider range of manipulations of the diagram. These tools are modeless; a tool is dragged from the toolbox and dropped on the diagram element that it is to be applied to. The four coloured circles allow the user to colour parts of the tree. This is purely secondary notation, and has no effect on execution. The next five tools are used to add new nodes to the tree: they correspond to `receive`, `repreceive`, `dispatch`, `new` and `sum` nodes respectively. Each tool is shown as a stylised representation of the nodes it creates.

The "TIDY" tool lays out a node and all of its children in an orderly fashion, without changing the ordering or containment properties. This is achieved using `LayoutVisitor`. A parameter is passed to modify the visitors behaviour such that it will overwrite any existing `VisualPropertiesNode`s. The "(bind)" tool is used to add or edit the name bound by a given node. The notification type and subtype of an event may be edited by double-clicking on the node; at this point, a dialog box is shown allowing the user to edit the event specification.

6.3.3 LFScript — A Textual Language

The LFScript language is a textual language that allows access to the full range of Lingua Franca functionality. It is designed primarily for users familiar with other programming languages. In addition, the AutoHAN project postulates the emergence of specialist "software plumbers". These are not necessarily software developers in the traditional sense, but specialise in customising the behaviour of programmable systems in the home. Their relationship to the homeowner is similar to that of a plumber or electrician, in that they perform tasks that the homeowner has the opportunity to perform, but possibly not the inclination, time or necessary skills. LFScript would be an ideal language for these specialists.

In keeping with traditional languages, texts in the language are plain (ASCII) text files. This has the advantage of enabling the programmer to use the plethora of tools that have been developed for the manipulation of plain text files. Such tools are typically designed specifically for use by programmers (frequently, by programmers themselves), and have been refined over decades. As such, they are well suited for the task at hand, and the target user group (those with programming experience) will be familiar with at least some of them.

The key problem with using ASCII as texts for the language is that they provide no way to associate "hidden" information with points in the program, and hence including non-representable secondary notation is problematic. An alternative would be to tie LFScript to a specific client, in a similar fashion to VSeq. However, this would prevent users from employing existing tools. The approach taken by LFScript is to include a reference to the Lingua Franca program from which the text was generated, in the form of the URL used to acquire the program from the server. Given this reference, it is possible to record unrepresentable `note` nodes by placing an identifier in the appropriate location in the text. This identifier consists of the "@" character, followed by an integer corresponding to the index of the `note` node in the original program. When the (possibly modified) text is translated back into a program, the `note` nodes may be reconstituted by retrieving the original program (via the URL reference) and querying by index to retrieve individual nodes.

There are two problems with this approach. The first is that the insertion of such identifiers affects the readability of the source code. This may be mitigated by using the syntax highlighting available in all modern programming editors to render the identifiers in a neutral colour. Some editors allow parts of the text to be hidden, while keeping it in the same position in the file — this eliminates the problem completely. This approach does not allow the programmer to see the

Figure 6.8: The shopping list program in LFScript

```
OutOf/*(x)?? {
    Reorder/-? {Order/<x.nts>}!
}
```

locations to which secondary notation has been attached by other environments; this is unlikely to cause problems, as there is little a programmer can productively do with such secondary notation.

A more serious problem occurs if the same program is edited by multiple users concurrently. In this case, the referenced URL included in the file may not correspond to the same program when the text is submitted as when it was originally created. If this happens, the indices encoded in the identifiers are unlikely to correspond to the same `note` nodes, and hence the wrong nodes will be inserted into the new program.

It would be possible to store a copy of the original program, as Lingua Franca XML, and use this to reconstruct the secondary notation. This would guarantee that the original program was available, but would require the programmer to maintain two files in parallel. A more comprehensive solution to the problem is to incorporate a versioning system into the server. With such a system in place, it is possible to encode a URL that will always retrieve the same result — a subtree of a specified version of the corpus — by adding a *query* term (delimited by a question mark) to the URL, after the path. This allows the correct `note` nodes to be retrieved, and the program to be constructed correctly.

The key syntactic constructs in LFScript correspond directly to those in Lingua Franca, allowing programmers to access the full range of Lingua Franca functionality is a straightforward way. They are summarised in Figure 6.3. LFScript is in many ways similar to the formal notation introduced in Chapter 5, as may be seen by comparing the LFScript representation of the shopping list program in Figure 6.8 to the formal representation in Figure 6.1. The key differences are the enforced use of braces to delimit the children of a node, and the additional support for secondary notation elements.

An event specification consists of the notification type and subtype, separated by a slash (`/`). If the specification corresponds to any event with the given notification types, the subtype is replaced by an asterisk (`*`). Similarly, if the specification corresponds to an event with the given notification type and no subtype, the subtype is replaced by a hyphen (`-`). Event components in the form of `param` nodes are represented `<E.n>`, where `E` is the name to reference, and `n` is the component of the referenced event to use (the `name` attribute of the node).

Dispatch nodes are represented by an event specification followed by an exclamation mark. The event specification may, in this case, be supplemented with additional name-value pairs to specify payload items. These consist of a slash followed by the name, an equals sign, and a value, either a literal value in quotes, or a `param` node as described above. The following example shows both cases:

```
nt/nts/foo="bar"/baz=<X.nts>!
```

Receivers are represented by an event specification followed by either a single

question mark (for `receive`) or two question marks (for `repreceive`), an optional binding specification, and a list of children, enclosed in braces ({}). Binding specifications consist of a name in parentheses (`()`). `new` nodes are represented by a binding specification and list of children with no receiver specification. `sum` nodes are represented by a list of receivers interspersed with plus signs (`+`).

LFScript introduces two types of note nodes; whitespace nodes and comment nodes. Comments begin with two slashes, and continue until the end of the line, as in C++ and BCPL. Both comments and whitespace may be inserted between any tokens in the source code. This unconstrained placement of `note` nodes means that the relative placement of secondary notation elements may not be determined simply by their position in the Lingua Franca tree. For example, consider the two texts "`Order/ Milk!`" and "`Order /Milk!`". Both correspond to `dispatch` nodes with equivalent `nt` and `nts` children, and a whitespace `note` node in between those children, but in the first case the whitespace follows the slash, and in the second it precedes it. In order to distinguish between potentially ambiguous cases such as this, LFScript inserts additional, empty `note` nodes corresponding to non-optional (as opposed to optional, secondary notation) syntactic elements - in this case, a node with the role `slash`, either before or after the whitespace node.

The insertion of non-optional `note` nodes raises the issue of handling programs where the LFScript secondary notation is of an unexpected form; for example, a required node is missing, or there are two adjacent whitespace nodes (the LFScript parser generates a single whitespace node for each contiguous sequence of whitespace characters). This may occur because some rogue language environment has inserted or removed LFScript secondary notation without respecting the constraints imposed by the language. However, a far more likely cause is that the functional (as opposed to the secondary notation) structure of the program has been edited in some other language environment; for example, a node between two whitespace elements may have been removed. As other language environments are not in general aware of the additional constraints placed on `note` nodes by LFScript, this is unavoidable. Additionally, if a given program fragment has not been processed by LFScript before, it will obviously lack the required `note` nodes.

When the processor finds a node with missing or inappropriate `note` node children, it removes *all* existing whitespace and non-optional nodes that are children of the node in question. It then inserts non-optional and whitespace nodes that conform to the constraints, in such a way that the resultant source code will be "pretty printed" (indented to reflect structure, with inter-element spacing to further improve readability). Comments are preserved, but their position relative to whitespace and non-optional elements may change.

An important aspect of the above process is that only the immediate descendants of a non-conforming node are necessarily modified. If indirect descendants of the node (i.e., those not directly connected to it) have conforming `note` nodes, they are left unaltered. This allows secondary notation to be preserved in elements that are moved in another language environment.

Unlike `note` nodes, all `group` nodes may be represented in LFScript. As is standard, `group` nodes without a role correspond to named groups with no particular semantics. This is represented using double brackets: `[`*Name*`[...]]`. If the group has a creator other than "lfscript", the creator and role are added: `[`*Name*`<`*Creator*`-`*Role*`>[...]]`. Groups with the creator "lfscript" correspond to higher level structures in LFScript. Each role corresponds to a different structure; if the role is not recognised, the group is rendered as a non-LFScript group.

Figure 6.9: **Lingua Franca equivalent of an LFScript *until* clause (omitting `note` nodes for readability)**

```
<group>
  <new bindevent="FRESH">
    <repreceive><nt><param event="FRESH" name="nt"/></nt>
      <sum>
        <receive><nt>A</nt>
          ...
          <dispatch><nt><param event="FRESH" name="nt"/></nt></dispatch>
        </receive>
        <receive><nt>B</nt>
          ...
          <dispatch><nt><param event="FRESH" name="nt"/></nt></dispatch>
        </receive>
        <receive><nt>C</nt>
          ...
        </receive>
        <receive><nt>D</nt>
          ...
        </receive>
      </sum>
      <dispatch><nt><param event="FRESH" name="nt"/></nt></dispatch>
    </repreceive>
    <dispatch><nt><param event="FRESH" name="nt"/></nt></dispatch>
  </new>
</group>
```

In addition to the constructs provided by Lingua Franca, LFScript also allows the programmer to define "until" clauses, in which a set of `receive` nodes are active until a specified event occurs, at which point the entire clause is deactivated. The LFScript syntax for such a construct is a sum, representing the active receive nodes, followed by one or more "guards", represented by `receive` nodes preceded by minus signs, for example:

```
A/*?{...} + B/*?{...} - C/*?{...} - D/*?{...}
```

When the above construct is activated, it repeatedly receives `A/*` and `B/*` events, performing the appropriate actions, until a `C/*` or `D/*` is received, at which point the corresponding action is performed, and the entire construct is

deactivated.

This construct is translated into a Lingua Franca `group` node with creator "lfscript" and role "until". A `new` node is used to introduce a fresh name, and contains the rest of the generated code. This code consists of a `repreceive` listening for the fresh event, which in turn contains a `sum` including all of the specified `receive` nodes. Those that correspond to "positive" terms are augmented with an additional `dispatch` child that emits the fresh event; those corresponding to guards do not. Finally, a `dispatch` emitting the fresh event is placed in parallel with the `repreceive`. As an example, the generated code corresponding to the above expression is shown in Figure 6.9

Chapter 7
Evaluation

The Lingua Franca system described in Chapters 5 and 6 aims to implement the concept of a multi-language programming system based on reversible translation via a shared intermediate form for a limited domain. To assess the degree to which the system succeeds in this goal, it is necessary to establish the correctness and utility of the reversible translations implemented. It is also necessary to examine the usability of the language environments produced, to ensure that the system produced is in fact useful to users. The following sections evaluate the system according to these criteria, using a variety of techniques.

7.1 Implementation of Reversible Translation

Unit tests were performed throughout development to ensure that the software was consistent with the functional specification. For components implemented in Java (LFCore and the language environments), this was achieved using the JUnit[20] framework. Test cases were written (in Java) for all major functionality before or as it was implemented, and such tests were retained throughout the development process. These proved invaluable in determining when modification to one piece of functionality impinged upon separate, previously implemented functionality.

The unit tests for the execution engine consist of a Lingua Franca corpus that reacts to certain events of the form `Test/`*X*, where X is the name of a particular test. The execution engine was run in batch mode with this initial corpus and an appropriate set of injected events, and the emitted events were compared to the expected results. As with the JUnit tests, the entire test suite was run periodically to check that newly added functionality had not introduced problems in existing code.

The key aim of the Lingua Franca system is to enable reversible translation between multiple languages. To verify that a particular translator achieves this aim, it is necessary to test that, for a given program, translating from the program to a text, then translating this text into a program, results in the original program, and that a similar assertion holds for the complimentary sequence of translations (text to program to text). Hence, the unit tests provide numerous example texts including all of the features of the texts in question, and example programs including all of the features of Lingua Franca programs, and test that the above assertions hold for these examples.

A prerequisite of implementing such a testing regime is a definition of equivalence over both programs and texts. It is tempting to define equivalence of programs as two programs resulting in equivalent texts when translated into any language, and similarly equivalence of texts as two texts translating into equivalent programs. While equivalence relations useful in this context will indeed have these properties, such a circular definition does not provide a sound basis for testing.

[20] `http://www.junit.org/`

I have chosen to define equivalence of programs in this context as follows. Two nodes are equivalent if and only if they are of the same type, they have the same attributes (in the XML sense) defined, the values of those attributes are equal (using the standard, case-sensitive definition of equality of strings), and their content is equivalent. In the case of `receive`, `repreceive`, `dispatch`, `sum`, `new`, `param` and `group`, their content is equivalent if and only if they have the same number of children, and those children, in the order they occur, are pairwise equivalent (by the current definition). In the case of `note` nodes, the content is equivalent if and only if the content of the node in its XML representation is equal, including whitespace. In the case of `nt` and `nts` nodes, if each node has a single `param` child, they are equivalent if their children are equivalent. Otherwise, they are equal if neither has a `param` child, and the textual content of the nodes, with leading and trailing whitespace removed, is equal (case-sensitive string equality). Two nodes that are not equivalent by the rules stated are not equivalent. With this definition, equivalences of texts can be defined in terms of equivalences of programs, as described previously.

A problem with using this definition of equivalence of programs to test reversible translation occurs in cases where the language environment adds `note` nodes to record default information (such as element positions in VSeq, or pretty-printing in LFScript). In this case, translating a program to a text and back again results in the original program with additional `note` nodes; the two programs are clearly not equivalent by the above definition. It would be possible to construct language environments such that they removed any `note` nodes added automatically, and not modified by the user, thus avoiding the problem. However, this would require the default values to be calculated, potentially a costly operation, each time a fragment is loaded into the environment. Hence, it is beneficial to retain this information in the form of `note` nodes if possible.

Given that the language environments may legitimately add note nodes to a program, the testing strategy must be modified. One possibility is to relax the constraints of equivalence used by the test, such that the test is passed if the two programs *with* `note` *nodes where the* `creator` *corresponds to the language environment being tested removed* are equivalent. This assumes that a language environment will preserve its own secondary notation elements, and if it does not, that it will fail the reverse (text to program to text) testing. This was considered to be sufficient to test the desired behaviour, namely that the translation will preserve all salient features of programs.

Unit tests written based on this testing regime were implemented for the two language environments supporting reversible translation (VSeq and LFScript). Examples were chosen to exercise the translation of all features of programs and texts, in both directions. Particular attention was paid to the *until* clauses in LFScript, as correctly reconstructing higher-level structures is an essential, yet difficult to implement, feature of Lingua Franca. On the basis of these tests, both language environments were found to implement reversible translation correctly.

While reversibility of translation is a requirement of the system, it is not sufficient. If the system is to be useful, it must have the property that the difference between two programs (specifically, a program and a modified version generated by translating the program into a text, editing the text, and translating the changed text back into a program) is reflected by an appropriate and analogous difference in the corresponding pair of texts in any given language.

Firstly, it should be made clear that, for certain pairs of programs, and certain languages, an "appropriate" difference in texts is no visible difference at all. This is the case where the programs differ only in secondary notation features of some other language, as mentioned in Chapter 4. Moreover, if a language omits

certain program nodes (a common case being those that are children of `groups` with an unrecognised `role`), two different programs may produce the texts that appear identical. This is to be expected.

Where the two texts differ, a desirable property is that the "degree" of difference is comparable to that between the programs. In particular, a small change in a program should not result in a large change in the corresponding text in any language; this would inhibit editing the program in multiple languages. This criteria is, at least in part, subjective, as it is based on the *perceived* degree of difference between two programs, as observed by the user. Roast et al (2000) define quantitative measures of viscosity (the degree to which a notational system is resistant to change), based on the number of "unitary actions" taken to transform a program from one state to another. While it would be possible to produce a quantitative measure of the degree of difference between two programs in a similar manner, this measure would not necessarily correspond to the degree of similarity in the programs. Green and Blackwell (1998) define a viscose system as one in which "a single goal-related operation on the information structure requires an undue number of individual actions." It seems likely that programs that differ in a single goal-oriented operation would be considered similar, regardless of the effort required to turn one into the other. Consequently, this measurement does not appear to be a good match to the desired criteria, and as such the evaluation of the criteria is restricted to qualitative assessment.

A standard test program utilising all Lingua Franca features was used as a starting point for this evaluation. The program was then translated into both VSeq and LFScript, modified in various ways, and resubmitted to the execution engine. A text may then be generated with the other language environment, and the difference between that text and the one generated from the unmodified program examined, allowing the degree of difference to be assessed.

VSeq fared well in this assessment. Most program changes produced appropriate and analogous changes in the corresponding VSeq text. The results were immediately useful. In the cases where this did not occur, the Tidy tool provides a simple way to correct any deficiencies.

LFScript is less successful in this respect. While the translation is reversible, and modified programs are by no means unusable, whitespace is typically not reproduced as it would be by a human programmer when elements are moved or deleted. The pretty-printing of "damaged" subtrees goes some way to mitigating this problem, but the results are still less than perfect. In particular, the placement of comments is frequently incorrect in the new program. Nevertheless, the system still enables users to edit programs in several languages in succession.

Another, more serious problem observed in the case of LFScript is that of higher-level structures. A small change in one language (e.g., VSeq) may disrupt a higher-level structure in another language (e.g., LFScript). If this results in a malformed structure that cannot be rendered, the higher-level structure becomes decomposed into its constituent parts — a change that is generally neither analogous or appropriate. The labelling of groups with their role and creator greatly reduces the likelihood of this occurring accidentally, but does not eliminate the possibility. The obvious alternative would be to design language environments such that they disallowed modification of the internal components of unrecognised structures; this would solve the problem, but would make such structures far less useful in a multi-language context. A more satisfactory solution would be to employ some kind of schema language to ensure that the composition of higher level structures remains valid. This possibility is expanded upon in the following chapter.

In general, the system is successful in its aim of facilitating multi-language programming. Reversible translation is correctly implemented in those environments that aim to support it, and results when modifying programs are acceptable. However, there is still room for improvement in some areas, particularly the reconstruction of LFScript texts and higher level structures in general after program modification.

7.2 Usability Evaluation

As the primary purpose of the languages produced was to illustrate the feasibility of multi-language programming environments based on reversible translation, as opposed to being immediately useful languages in their own right, full usability testing has not been performed. Nevertheless, an informal study was carried out on a limited number of users to obtain preliminary, qualitative results about the two languages aimed at general end users (Media Cubes and VSeq).

For the purposes of the test, the Media Cubes language was realised in a "Wizard of Oz" fashion using the simulator and non-working prototype cubes; all other components of the system were the working prototype implementations. Subjects were first given a brief written description of the Media Cubes language, and then shown several small examples of Media Cubes programs (based on the examples in Chapter 3). Throughout the study, subjects were encouraged to ask questions if they needed clarification on any point. Once the language had been presented, they were asked to construct a program to accomplish a specific task (mute the TV whenever the phone or the doorbell rings), using a provided vocabulary of events. When they were satisfied the constructed program would fulfil the requirements, the test moved on to the VSeq language.

Subjects were given a similar written description of VSeq, and a demonstration of how to use the console to edit programs in the language. They were then shown the previously constructed program in VSeq, and asked to use the language to add additional behaviour (when the doorbell, but not the phone, rings, switch the hall light on). Finally, they were asked to fill in a short questionnaire about both languages, and the relationship between the two. This questionnaire is included in Appendix B.

The study was undertaken with five subjects. Three of these had a strong technical (programming) background; the remaining two did not. All subjects successfully completed both tasks. Most were hesitant with the Media Cubes, often asking for the explanation of a particular cube to be reiterated, or for confirmation that their understanding of the function of a cube was correct. In some cases, subjects made false starts before producing the correct program. When errors were made, all were detected either by the system, or by the subject. In contrast, subjects asked fewer questions regarding the VSeq language, and produced the correct modifications to the program directly. This may be partly explained by the fact that VSeq was presented after the Media Cubes, and hence subjects were already familiar with the underlying Lingua Franca concepts, but also seems to be in part due to deficiencies in the Media Cubes language itself. There was no noticeable difference between the performance of technical and non-technical users with either language.

One feature of the Media Cubes language that caused particular confusion, amongst both technical and non-technical subjects, was the Generalisation cube. A common mistake was to assume that both the input and output examples and events had to be used; in fact, the more general solution of the given task required only the use of the input faces. Most subjects were also unsure of the behaviour of the cube in general. This may be a documentation issue, in that

the written explanation, and the naming of the cube faces, does not adequately express the function of the cube. Conversely, it is possible that the Generalisation cube concept is inherently flawed, and an alternative mechanism for creating generalised scripts is required.

The questionnaire was designed to obtain additional information about the subject's perceptions of the system. In most cases, subjects were happy with the correspondence between the Media Cubes program and its VSeq representation. In general, subjects were more confident that their solution to the VSeq programming task was correct than their solution to the Media Cubes task; several explicitly cited the lack of feedback as a reason for this. One subject described the Media Cubes language as "too much of a memory test". Most subjects felt that having multiple representations was beneficial; those that did not considered the Media Cubes superfluous.

When asked if they would use the languages to program a home network, one subject responded that they could not say as they did not know enough about the alternatives available, and another said that she would not have a home network. The other subjects answered in the affirmative for VSeq, but not the Media Cubes, citing some of the problems mentioned above.

Overall, the study suggested that, while VSeq is largely usable as it stands, the Media Cubes language requires significant further attention. That said, the Media Cubes aroused significantly more interest from subjects, perhaps due to the relative novelty of tangible programming languages. This is encouraging in the context of the Media Cubes as an introductory programming language, and may be of significant benefit in terms of engaging new users with the system.

7.3 Cognitive Dimensions Analysis

In addition to user testing, it is also informative to examine the languages in terms of usability on a theoretical basis, both to highlight any usability deficiencies caused by the underlying architecture, and to inform further development. I have chosen the Cognitive Dimensions framework as a tool for this evaluation (Green 1989). This framework is particularly suited to the evaluation of programming systems. Other methods are less suitable; for example, the Keystroke Level Model (Card et al 1980) is too low-level to provide a useful overall evaluation, and does not address problem solving, while Cognitive Walkthrough (Rieman et al 1995) requires a specific goal, and a specific set of steps to achieve this goal. Programming tasks are not generally amenable to this kind of specification. Furthermore, the Cognitive Dimensions framework is equally applicable to static and interactive systems:

> *In this respect, [the Cognitive Dimensions] approach is unlike most evaluation methods in HCI, which are solely aimed at interactive devices, such as editors and the like. The focus of those approaches is on the process of interaction. Therefore, they do not work well for non-interactive artefacts.* (Green and Blackwell 1998)

While two of the systems under examination are interactive (the Media Cubes language and VSeq), the third is a traditional programming language, where the notation is largely independent of the environment used to manipulate it. Cognitive Dimensions provide a framework where all three may be discussed and compared on an equal footing. Blackwell (2003) addresses the particular issues involved with the application of cognitive dimensions to tangible programming languages.

A disadvantage of the framework is that it does not provide quantitative data about systems. It is a "broad-brush" tool for discussion and evaluation of notational systems. As such, it provides a good way to access early prototype systems and direct further development.

Lingua Franca constitutes the underlying *information structure* for all language environments. A number of points may be made about this structure that are common to all language environments. *Hidden dependencies* are present, in the form of notification types; there is no inherent link from a piece of code dispatching an event to all possible receivers of that event. The possibility of passing names within events compounds this; however, this is an advanced technique, and although it used in the underlying implementation of some higher-level structures, it is unlikely to be encountered by the majority of users. Similarly, comprehending and writing programs that pass names within events requires *hard mental operations*, and constitutes making an abstraction. However, a large amount of functionality can be used without these techniques; hence, the system is not *abstraction hungry.* It is, however, *abstraction tolerant*, as users are free to make abstractions by inventing events and writing programs that react to them. This capacity also reduces *repetition viscosity.*

7.3.1 Notation, Medium and Environment

The notation used in VSeq is that of mutable diagrams. The medium is a conventional display device, either a computer monitor or a television. The console provides the environment via which users view and manipulate them.

The VSeq environment has a single distinct subdevice, namely the dialog used to edit the event specification associated with a node. This consists of a simple dialog with a single line of explanatory text, a text entry area containing the event specification to be edited, and "OK" and "Cancel" buttons, all using the host platform's standard user interface. The chief criticism that may be raised is that of a lack of closeness of mapping leading to poor role-expressiveness; a single text entry box is provided to enter two conceptually separate values (notification type and subtype). To further complicate matters, each quantity may be one of two types (either a literal event component, or a reference to a bound name). A simple syntax is introduced to derive these values, and determine their types, from the single text string. This is needlessly complex, and is likely to lead to error-proneness. The subdevice should be modified (or, given its simplicity, entirely rewritten) such that it provides a better match for the task in hand.

The LFScript language is much like a traditional language in that the notation is that of plain text files. The environment is also similar, in that it consists of a processor that transforms a static source code file into an executable form. However, the processor differs from its counterparts in traditional programming environments in that it is also used to obtain source code prior to editing. In this respect, it is more like the interface to a revision control system. In order to analyse the system, it is also necessary to specify the text editor used. At present, LFScript is edited using a standard programmer's editor with no additional support for the language. However, the main target user group of the language ("digital plumbers" and enthusiastic householders) are unlikely to use such an editor. Hence, for the purposes of this evaluation, we hypothesise a dedicated editor with the following characteristics:

- Standard GUI editing features (e.g., insertion of text, cut and paste, search and replace, mouse-driven navigation and scrolling)
- The ability to open multiple views of the same document simultaneously, in different windows.

- Automatic indentation and syntax highlighting for the LFScript language.
- Integration with the LFScript processor, such that code may be retrieved and submitted from within the editor.

The environment of the Media Cubes language consists of the cubes themselves, manipulated in the medium of physical space. While other components, such as a base station to connect the cubes to the network, are necessary to support the language, they are not involved in user interaction, and hence do not form part of this evaluation. The notation in this case is that of a series of actions performed with the cubes. This notation is transient, as described by Blackwell (2003).

7.3.2 Visibility and Juxtaposability

Visibility in VSeq is generally good, unless the diagram is very large. Conversely, juxtaposability is poor; it is not possible to view two distant sections of a diagram simultaneously. This problem could be addressed by adding "bookmarks" to allow users to move to defined points in the diagram instantly, and by allowing them to open multiple windows viewing different parts of the same diagram. Improving juxtaposability would also aid the visibility of large diagrams. More extensive solutions to these visibility concerns are discussed in Section 8.3.4.2.

The visibility and juxtaposability of LFScript are largely determined by the facilities provided by the editor. The hypothetical editor explicitly includes support for juxtaposing arbitrary sections of text. Syntax highlighting aids visibility by making it easier to distinguish syntactic features from one another. The relatively terse representation (see below) improves both visibility and juxtaposability by allowing a greater amount of code to be viewed in a given area.

The Media Cubes language differs from the other systems analysed, and from other programming languages, in that it is write-only. Hence, the *visibility* of the system is zero; programs cannot be examined at all. This is a feature common to all systems that make transient marks. However, the ability to view programs created in the Media Cubes language in other Lingua Franca languages goes some way towards mitigating this problem.

7.3.3 Diffuseness

The notation chosen for VSeq is relatively diffuse; this was deliberate, with the intention of making the system less error-prone. However, it also hampers visibility by reducing the amount of code that may be viewed in a given area. Conversely, the notation in LFScript is particularly terse; this improves visibility, as mentioned above.

The Media Cubes language may be regarded as diffuse in two respects. Firstly, unlike on-screen languages, they require an amount of physical space to work in (however, in this respect they are terse compared to model-building languages such as AlgoBlock). Secondly, a large number of distinct actions are required to achieve any particular effect. As opposed to conventional languages, where diffuseness typically reduces error-proneness by making information more explicit, the diffuseness of the Media Cubes language may in fact increase error-proneness by exacerbating the problems caused by the requirement for the user to keep the program under construction in memory.

7.3.4 Viscosity

The VSeq environment allows easy modification of all structures in the diagram, and there are few dependencies. This means that viscosity is generally low. In particular, the Tidy tool, combined with the ability to move an entire subtree by dragging its root node, serves to reduce knock-on viscosity by simplifying any operations made necessary by a change. However, there is no support for manipulating an arbitrary set of nodes at once, nor for modifying all instances of an event name. Implementing these features would further reduce viscosity.

The viscosity of LFScript is largely a function of the editor, as this determines the ease with which dependent changes may be made. The hypothetical editor does not have any particular support for *refactoring* — automatically making a collection of related changes (for example, changing all instances of a bound name) with a single operation — but textual searching makes carrying out the same process manually easier and more accurate.

Due to the Media Cube's lack of visibility, and the consequent inability to edit programs in the language, its viscosity is extremely high (arguably, infinite): the only way to make a change to a program is to delete and rewrite it.

7.3.5 Secondary Notation and Provisionality

The secondary notation provided by VSeq supports redundant recoding of information in the form of colour and layout; both of these may be used to emphasise existing structures. VSeq has little support for escape from formalism; the only feature that can be used in this way is named groups. The ability to attach textual notes to program nodes would be one obvious way to correct this. Another extension might be the ability to add arbitrary strokes to the diagram. This would be highly appropriate when using a pen-based interface such as a Tablet PC, but less so when using devices less suited to drawing. In addition, the manipulation of these strokes once they had been laid down would need careful consideration.

The VSeq console does not include any support for provisionality. This may be addressed by a facility to deactivate or "comment out" subtrees; this could be encoded relatively simply in Lingua Franca, and hence would not require any change to the underlying architecture.

In keeping with most conventional languages, LFScript provides two forms of secondary notation. Whitespace allows redundant recoding of structural information, while comments provide a flexible means to escape from formalism. In addition, they provide a rudimentary level of support for provisionality, in that sections of code may be deactivated by placing them inside comments. However, this is not perfect; identifiers referring to unrendered `note` nodes would be preserved in comments, and, if the code was reintroduced into the "live" program, these identifiers are likely to refer to the wrong elements. Editor support for automatically stripping such identifiers from comments before submitting code would help, but this is only a partial solution. Encoding provisional code in Lingua Franca as described above would avoid the problem entirely.

The question of secondary notation in the Media Cubes language is an interesting one. While the languages provides no features for annotating the program, the user has ample opportunity to annotate the environment, for example, by attaching notes to Cubes (such notes may be regarded as sub-devices). In a traditional environment, such auxiliary annotations are separate from the information being manipulated, and rely on cross-referencing to establish correspondences between elements. Conversely, the tangible nature of the individual elements of the

Media Cubes environment allow such annotations to be associated with elements directly, making the correspondence substantially more immediate.

7.3.6 Hidden Dependencies

The hidden dependencies of VSeq are those of the underlying information structure. It would be possible to address these in the environment, perhaps by highlighting all receivers of an event, or uses of a bound name, but such techniques reduce visibility and juxtaposability, and increase diffuseness, and were therefore avoided. Similarly, LFScript does not introduce any new hidden dependencies beyond those inherent in Lingua Franca. These are mitigated to an extent by the searching facilities of the hypothetical editor, which allow other uses of a name to be more readily located. These facilities could be extended to be syntax-aware, and, as with VSeq, the environment could highlight dependencies in some manner. In both languages, adding such highlighting would increase diffuseness and reduce visibility, but, unlike many design trade-offs, the choice may be made moment-by-moment by the user, by allowing dependency highlighting to be turned on or off.

As programs cannot be examined in the Media Cubes language, all dependencies in that language are hidden dependencies. However, the types of program likely to be created with the Media Cubes has few dependencies, mitigating the problem somewhat. Hidden dependencies are often a problem in tangible user interfaces; Blackwell (2003) observes that "there is no convenient tangible equivalent to the ubiquitous node and link formalism that is used to indicate relationships between independent elements."

7.3.7 Closeness of Mapping, Consistency and Role Expressiveness

The closeness of mapping of VSeq is reasonable; the diagrams represent the control flow of the program directly. However, there is no explicit indication of what each type of (graphical) node corresponds to. This may present a learning problem for new users. The representation is also consistent, in that the same elements are always represented in the same way. However, the uniformity of the representation leads to poor role-expressiveness; in particular, the representations for `receive`, `repreceive` and `dispatch` nodes are all very similar.

In the case of LFScript, the closeness of mapping is reasonable, assuming that the user has understood the execution model of Lingua Franca. I believe that this model is reasonably comprehensible, but user testing would be required to establish this.

The role-expressiveness of LFScript is relatively good, in that it follows the convention of using question marks to signify input (`receive` and `repreceive`) and exclamation marks to signify output (`dispatch`). The syntax is highly consistent, in that a single form only ever means one thing (for example, although it is used in numerous contexts, an identifier in parentheses always corresponds to binding that identifier). The only exception to this is the use of angle brackets (`<>`) to express both references to bound names and the creator and role of foreign `groups`. The context should be sufficient to distinguish between the two cases. The identifier tags inserted to allow the reconstruction of unrendered `note` nodes also cause an issue with regard to role-expressiveness, as they do not correspond to anything in the user's model of the program. However, the user never needs to insert or manipulate such tags, merely leave them alone, and as mentioned they may be deemphasised or hidden by the editor as part of syntax

highlighting. One final point to make is that similarity between the representations of `receive` and `repreceive`, while increasing role-expressiveness, may also lead to confusion, and a consequently make the system more error-prone.

One of the key benefits of the Media Cubes, and of tangible user interfaces in general, is good role-expressiveness. This chiefly arises from the use of familiar objects to model information. In the case of the Media Cubes language, devices in the home network also constitute elements in the notation of the Media Cubes language. This results in particularly good role-expressiveness and closeness of mapping.

7.3.8 Premature Commitment

In both LFScript and VSeq, programs may be constructed in any order, and hence the only issues with premature commitment are those related to the underlying information structure. These are limited to the choice of new names, and new event types and subtypes. These aspects of the program may be modified later (subject to the viscosity of the language in question), and as such there is relatively little premature commitment in these languages.

In contrast, premature commitment is perhaps the most serious problem with the Media Cubes language as it stands. The language strongly encourages bottom-up creation of programs (that is, creation of small components that are then combined). Top-down programming is possible, but is made needlessly difficult as it requires a greater number of intermediate steps to be stored using the Clone cube. This, coupled with the need to keep the program in mind during its construction (*hard mental operations*) present a serious usability problem. As mentioned, these problems are believed to be manageable for small programs, but impose a limit on program size.

7.3.9 Progressive Evaluation

None of the languages under evaluation provide any support for progressive evaluation. This is a serious deficiency; without it, there is no way to test the functionality of a program without loading it into the live system. This is particularly acute in the case of LFScript, a language designed for the development of relatively large and complex programs.

A "sandbox" environment that enabled users to simulate the actions of a program and test out possible inputs would allow progressive evaluation, greatly enhancing the environment. Similarly, it has been noted that the expression rewriting style of the Lingua Franca architecture allows a snapshot of a running program to be translated into a text in the normal way. VSeq could exploit this fact to produce a graphical debugging facility that further enhances support for progressive evaluation.

A similar debugging environment could be produced for LFScript, either as a (substantial) extension to the hypothetical editor, or as a separate program that made use of the same notation.

Such support could be achieved by adding to the execution engine the facility to execute submitted code in a sandbox environment, with a specified set of input events, and the possibility of single-stepping through the code. In this case, it would be necessary to provide some interface to control and configure this testing environment. While a dedicated interface would be easiest to implement, it would not necessarily be a good match to all language environments. Hence, a better approach would be to extend the network protocol to support sandbox evaluation,

allowing individual language environments to introduce sub-devices to support progressive evaluation in an appropriate manner. Alternatively, LFCore could be extended to support evaluation, allowing these sub-devices to perform tests of the code without the need to communicate with the central server.

7.3.10 Implications

In general, the evaluation suggests that the VSeq language environment is good for *transcription* and *incrementation*. The lack of juxtaposability (and limited visibility in the case of large diagrams) is potentially harmful to *modification* and *exploration*, but any problems are likely to be manageable for small to medium sized programs. This broadly fits the aims of the language.

The qualities of LFScript make it suitable for most programming tasks, although in some cases another language may be more suitable (for example, in the case of small programs, VSeq may be better suited due to enhanced visibility). However, much of the usability of LFScript is contingent on the editor used. The hypothetical editor introduced provides a fair degree of support for the language; several suggestions for more advanced support have been made during the evaluation. Implementing such an editor would be relatively straightforward, either as customisations to an existing editor, or as a standalone application.

This assessment reinforces the initial conclusions drawn about the Media Cubes language, namely that the language is only suitable for the creation of small programs. In particular, it should be noted that *modification* and *exploration* are impossible; only *incrementation* and *transcription* are supported. The problem of premature commitment is a major issue for the language, compounding the restrictions on program size previously noted. Even taking into account the aims of the language, this is far from ideal, and should be addressed in future revisions. On a more positive note, the ability to add secondary notation in a natural and flexible way is an unexpected benefit of the tangible approach, and the good role-expressiveness and closeness of mapping contribute to making the language easier to learn.

Overall, the three language environments produced are acceptable as demonstrations of the concept of multi-language translation, but fall short of being useful languages in their own right. VSeq and LFScript are close to achieving this aim, but both have a number of relatively minor problems that need to be addressed. The Media Cubes language has more serious problems, and would require substantial reworking to become a useful tool. This is not to say that either tangible programming or write-only languages are necessarily without merit, but rather that the current instantiation is flawed.

Further possibilities for improvements in the language environments, and the Lingua Franca architecture itself, are discussed in Chapter 8.

Chapter 8
Conclusions & Future Work

This chapter summarises the work reported, and relates its contribution to the aims described in Chapter 1. Possible avenues for further investigation are discussed, and the work is placed in the context of current and future developments.

8.1 Summary

Chapter 3 presented a design for the Media Cubes language, a tangible programming language for use in the networked home. The design is based around a dynamic arrangement of components, in which no static model is constructed. While this approach has numerous advantages, it has several major disadvantages, most notably that, without an external representation of the program, it is impossible to view or edit programs. This led to the development of a system to support the representation of a single program in multiple, interchangeable, languages.

Chapter 4 discussed the problems associated with producing a reversible mapping between disparate programming languages. In particular, the problems of representing secondary notation features, and structures not present in all languages, were examined. A solution was proposed, based on a shared intermediate form that encoded secondary notation and structural information appropriate to all languages supported by the system in an extensible manner. This representation makes reversible mapping between disparate languages feasible.

Chapter 5 presented Lingua Franca, a system based on the ideas in Chapter 4, for supporting multi-language end-user programming in the networked home. An intermediate form was detailed, along with its external representation in XML, and its execution behaviour. A system architecture suitable for implementing these facilities in a networked environment also described.

Chapter 6 described the implemented components of the system. These consist of an execution engine that executes Lingua Franca code, and acts as a program database, plus several language environments that support individual programming languages. Three language environments — VSeq, LFScript and the Media Cubes — were presented.

Chapter 7 evaluated the work, examining both the correctness of reversible translation, and the usability of the languages produced.

8.2 Contribution

In accordance with the aims set out in Section 1.6, the work described has produced the following:

- An approach for the integration of multiple languages, allowing a programmer to easily switch between languages depending on the task being performed, preserving secondary notation and structure
- An example architecture implementing this approach in a particular context, that of the networked home

- A number of novel programming languages within this architecture, targeting a variety of user populations and tasks
- An evaluation of the architecture and languages

8.3 Future Work

My work has aimed to demonstrate the feasibility of multi-language end-user programming based on reversible translation. While this has been achieved, there are many areas that the work could be extended and built upon.

8.3.1 Testing

As an end-user programming system, user testing would form an essential part of any further development of the Lingua Franca language environments. While the current prototypes suffice to demonstrate that such systems can be implemented, a useful end-user programming language cannot be developed without input from the intended user population.

Iterative design techniques would provide a good basis for further development, as these couple the development process tightly to the reactions of users in the population in question. Participatory design techniques take this further, involving the users directly in the design progress. The Lingua Franca architecture would be particularly amenable to such techniques, as variations in semantics may be implemented at a high level (in terms of Lingua Franca code), facilitating rapid development.

8.3.2 Additional Language Environments

The three language environments presented demonstrate the key features of the Lingua Franca architecture. However, additional environments would serve both to exercise the translation capabilities of the system, and to serve other user populations, usage contexts and programming tasks.

There are currently no output-only language environments implemented within the Lingua Franca architecture, but these are relatively simple to implement. In particular, if the target language is XML-based (such as HTML or SVG), the problem may be solved using an XSLT stylesheet, specifying a mapping between Lingua Franca and the target language. This stylesheet could be applied by a dedicated language environment that communicated with the execution engine in the normal way. Alternatively, if the execution engine was modified to add an appropriate processing instruction, any web browser could download the program and apply the stylesheet, obviating the need for a separate client. To support multiple output languages in this way, the execution engine could add a processing instruction for each and rely on the browser to provide a means of selection, or the desired language could be encoded in the URL, allowing the execution engine to provide only the processing instruction for the desired language.

Input-only languages are similarly straightforward to implement, as both the Lingua Franca format and the means of submitting it to the server (HTTP) are well defined and simple to use. Several methods of entering code into the systems have been suggested, such as natural language (including speech recognition), and inference from either usage patterns or explicit examples. These examples share the Media Cubes' lack of external representation, and hence would benefit from the facilities provided for Lingua Franca.

8.3.3 Improvements to Lingua Franca

The core Lingua Franca architecture is sufficient for the purpose of demonstrating the concept of multi-language translation. However, there are numerous possible ways in which the system could be improved.

The first is the efficiency of the execution engine. At present, the execution engine is implemented in Python, uses a standard data representation with significant overhead, and bases evaluation on a naive interpretation strategy. Significant gains in efficiency could be achieved by rewriting the execution engine in a systems programming language, hence reducing the run-time overhead that the implementation language imposes. Similarly, a dedicated data representation, such as that discussed in Section 6.2, may be of benefit. The support for progressive evaluations discussed in Section 7.3.9 would also be highly beneficial.

At present, events in Lingua Franca are defined as a set of name-value pairs, one of which is required ("nt") and another of which has defined semantics if it is present ("nts"). While this representation is sufficiently powerful to encode channel passing as in the π-calculus, it provides very limited facilities for Lingua Franca programs to manipulate data. While for many programs, routing events without changing their content is sufficient, the inability to manipulate date is nevertheless a serious problem.

A solution would be to adopt a more general structure for events. XML would be the obvious choice for this, as in addition to its suitability to the task, the facilities for manipulating it are already in place. Hence, an event would consist of a single `event` XML element with arbitrary children. To reference event data, a `param` node would not provide a component name. Instead, it would provide an XPath expression to be applied to the event data to obtain the result. Similarly, instead of having `nt` and `nts` children, `dispatch` nodes would have arbitrary children; these would become the children of the emitted event node.

While more powerful in terms of data manipulation, this event structure is less closely tied to the underlying GENA event mechanism. While GENA events may have an arbitrary XML payload, they must also have an NT, and optionally an NTS, by which subscriptions are organised. Hence, the new event structure must encode this data. One possibility would be to add `nt` and `nts`attributes to the `event` elements (and consequently the `dispatch` nodes). However, this has the disadvantages described in Section 5.4, and for these reasons it may be preferable to use child elements. It would also be necessary to consider the case where no NT is provided. Possibilities include disallowing such events, providing a default NT, or considering such events "internal" and not forwarding them to the external event system.

As mentioned, the support for higher level structures would benefit from a schema language, allowing their internal composition to be formally specified. A group corresponding to a structure would include a URL to the appropriate schema. This would allow language environments to preserve higher level structures not previously encountered, simply by ensuring that any edits made do not invalidate the group with respect to the referenced schema.

More generally, the execution engine provides an ideal platform for numerous program management facilities. One obvious extension would be a revision control system to manage updates to the corpus. In addition to the normal benefits of such systems (such as the ability to roll back to a previous version of a program), this would be obviate the need for language environments to store the program on which they were based, as mentioned in Section 6.3.3. Simi-

larly, the execution engine would be a sensible place to implement the policy and consistency checking described by Saif et al (2001).

8.3.4 Further Developments of Existing Language Environments

While sufficient to demonstrate the viability of the Lingua Franca architecture, the language environments implemented have significant room for improvement. A number of possible improvements were presented in Section 7.3; others are discussed here.

Development of LFScript

The LFScript language is the most similar to conventional programming languages, and is most likely to scale well. Hence, it is likely to be the primary language for the development of substantial programs within the environment. While LFScript already provides access to the full functionality of Lingua Franca, it lacks features to support large-scale program development.

A key feature to be added would be explicit support for the creation of reusable code fragments, akin to functions in conventional languages. Such code fragments may be readily created using `new` and `repreceive` nodes, using techniques similar to those described by Milner (1999); however, adding support for them in the language would make their use simpler and less error-prone.

A more comprehensive solution, and one more in keeping with the nature of the Lingua Franca architecture, would be the implementation of a hygienic macro system such as that found in Scheme (Kelsey et al 1998). This would allow LFScript programmers to add new syntactic forms to the language in a structured way. A sufficiently flexible system would allow abstractions such as that described above to be easily defined and extended by the user.

The use of `group`s to represent higher-level structures in LFScript (and in Lingua Franca in general) provides an established framework for integrating the defined structures with the system as a whole. Moreover, the retention of structural data that allows the defined syntax to be reconstructed would mean that debuggers may show the code as it was written, as opposed to the generated code, mitigating one of the problems traditional associated with macro systems.

Another improvement to the system would be to remove the identifier tags from the source code. However, to preserve the reversible translation between LFScript and Lingua Franca, it must be possible to reproduce secondary notation from other languages. The format of the texts does not provide any way to store such secondary notation, and so it must be retrieved from the original source program via the included reference.

One possible approach would be to compare a text generated from the original program to the modified text. A set of transformations could then be computed to transform the former to the latter. An equivalent set of translation from the original program to the modified program would effect the modifications while preserving the secondary notation.

While this would appear to be attractive in principle, there are obstacles to implementing it in practice. Chief amongst these is the fact that standard algorithms to compute differences between documents encode movement of code from one place to another as a deletion followed by an insertion. This is precisely the behaviour we do not want, as it does not preserve secondary notation. If a suitable algorithm could be found (perhaps taking advantage of the well-defined tree structure of programs), this approach would be a possibility, but extensive investigation and testing would be required to determine if it is in fact viable.

Development of VSeq

There are numerous minor improvements that could be made to the VSeq console application. For example, the drawing canvas is currently of a fixed, though large size. It would be preferable to make the canvas infinite in size, but this presents a problem for standard scroll bars, which display the size of the viewed area relative to a fixed area. A solution would be to have the canvas extend dynamically whenever a diagram component is moved close to the edge of the canvas. This would allow the entire diagram to be accommodated, no matter how large it grows, whilst retaining the standard method of navigating a large plane.

Other techniques may be profitably employed to manage large diagrams. A thumbnail version of the diagram may aid navigation. Generalising this idea, allowing the user to zoom in and out provides a flexible way to view overall structure and fine-grain details in a single interface. Alternatively, large-scale displays such as the Escritoire (Ashdown 2004) may be profitably employed. While a desk-style display with fine-grained control is unlikely to be present in the home, a large projected display may be present as part of a home cinema system. Adding a coarse-grained pointing device to such a setup would be useful in several contexts, and would provide a novel interface to VSeq.

As it stands, VSeq lacks higher-level structures. Graphical representations of such structures would be an obvious extension to the environment. The representation of existing structures, such as those generated by the Media Cubes' Connection cube, or the *until* clause in LFScript, would be a starting point. Higher-level structures specific to VSeq could also be developed.

As with LFScript, it may be desirable to allow users to add new types of higher-level structure to the system. One way in which this may be accomplished would be to introduce a system of "stencils", as found in structured drawing programs such as Visio[21]. These allow groups of objects to be stored in a palette, and copied into the diagram when required. Analogously, VSeq could provide a facility to save a subtree of the diagram to the toolbox. Thereafter, a copy of this subtree could be inserted into the diagram in the same way that other tools are used; namely, by dragging the tool from the toolbox to the drawing area.

Such stencils would provide a mechanism to insert complex structures into the diagram easily. It would be necessary to provide facilities for adding and editing stencils; while the former is straightforward, the latter would need attention. In addition, stencils as described are not parameterised. It could be argued that it is possible to encode parameterisation using Lingua Franca primitives, but as mentioned previously this is complex and error prone. Hence, it would be useful to provide a mechanism to parameterise stencils over certain elements. The code templates provided by many IDEs, in which editing a component also changes related components, until the template is "fixed", provide a model for one way in which this may be achieved.

Development of the Media Cubes

Before the Media Cubes are tested on users, it would be necessary to improve the prototype devices. As mentioned, infrared communication has proved to be unsuitable for the application to which the devices have been put; specifically, the directionality of infrared means that the cubes must be carefully oriented relative to the base station, making them awkward to use. In addition, the

[21] http://office.microsoft.com/visio

relative unreliability of infrared channels, and their susceptibility to interference from both natural and fluorescent lighting make them unsuitable for use in a home networking context. Replacing the infrared communication with a short range, low power radio link would make the prototypes suitable for use by end users in a testing context.

8.4 Ubiquitous End-User Programming

As ubiquitous computing matures, and becomes a part of everyday life, the way in which users control it will become increasingly important. While direct or implicit control may suffice for many applications, the sheer diversity of situations in which ubiquitous technology may be used means that users will, at some point, want to tailor the technology to their own specific needs. Multi-language end-user programming systems such as Lingua Franca provide a compelling approach to allowing the widest possible variety of users to do this, thus fully harnessing the potential of ubiquitous computing technology.

Appendix A
Schema for XML Lingua Franca

The following schema defines the XML representation of Lingua Franca in the RELAX NG schema language (Clark 2001).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
     ns="urn:linguafranca"
     datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <start>
    <element name="lf">
      <element name="eventqueue">
        <zeroOrMore>
          <ref name="dispatch.element"/>
        </zeroOrMore>
      </element>
      <element name="corpus">
        <zeroOrMore>
          <ref name="process.category"/>
        </zeroOrMore>
      </element>
    </element>
  </start>

  <define name="process.category">
    <choice>
      <ref name="dispatch.element"/>
      <ref name="repreceive.element"/>
      <ref name="receive.element"/>
      <ref name="group.element"/>
      <ref name="note.element"/>
      <ref name="new.element"/>
      <ref name="sum.element"/>
    </choice>
```

```
</define>

<define name="eventspec">
  <ref name="nt.element"/>
  <optional><ref name="nts.element"/></optional>
  <zeroOrMore><ref name="payload.element"/></zeroOrMore>
</define>

<define name="dispatch.element">
  <element name="dispatch">
    <ref name="eventspec"/>
  </element>
</define>

<define name="repreceive.element">
  <element name="repreceive">
    <optional>
      <attribute name="bindevent"><text/></attribute>
    </optional>
    <ref name="eventspec"/>
    <zeroOrMore><ref name="process.category"/></zeroOrMore>
  </element>
</define>

<define name="receive.element">
  <element name="receive">
    <optional>
      <attribute name="bindevent"><text/></attribute>
    </optional>
    <ref name="eventspec"/>
    <zeroOrMore><ref name="process.category"/></zeroOrMore>
  </element>
</define>

<define name="nt.element">
  <element name="nt">
    <choice>
```

```
      <text/>
      <ref name="param.element"/>
    </choice>
  </element>
</define>

<define name="nts.element">
  <element name="nts">
    <choice>
      <text/>
      <ref name="param.element"/>
    </choice>
  </element>
</define>

<define name="param.element">
  <element name="param">
    <attribute name="event"><text/></attribute>
    <attribute name="name"><text/></attribute>
  </element>
</define>

<define name="new.element">
  <element name="new">
    <attribute name="bindevent"><text/></attribute>
    <zeroOrMore><ref name="process.category"/></zeroOrMore>
  </element>
</define>

<define name="sum.element">
  <element name="sum">
    <oneOrMore><ref name="receive.element"/></oneOrMore>
  </element>
</define>

<define name="payload.element">
  <element name="payload">
```

```
      <attribute name="name"><text/></attribute>
      <choice>
        <text/>
        <ref name="param.element"/>
      </choice>
    </element>
  </define>

  <define name="group.element">
    <element name="group">
      <attribute name="name"><text/></attribute>
      <zeroOrMore><ref name="process.category"/></zeroOrMore>
    </element>
  </define>

  <define name="note.element">
    <element name="note">
      <text/>
    </element>
  </define>

</grammar>
```

Appendix B
User Questionnaire

The following is the questionnaire given to subjects in the study described in Section 7.2. It has been reformatted to fit the page, but is otherwise unaltered.

	Media Cubes	***VSeq***
Do you feel you understood the language as presented?		
Was the language adequate to express the program given?		
How confident are you that the programs created did what they were intended to?		
Would you use the language to control a home network? (Please state reasons)		
Was the correspondence between the programs in the two languages clear?		
Was it useful to have multiple representations of the program?		
Do you have any other comments on the system in general?		

Bibliography

Addlesee, M., Curwen, R., Hodges, S., Newman, J., Steggles, P., Ward, A. and Hopper, A. Implementing a Sentient Computing System. *IEEE Computer*, 34(8) (2001), 50–56

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S. and Trickovic, I. *Business Process Execution Language for Web Services Version 1.1.* (2003) Available at `http://www-106.ibm.com/developerworks/library/ws-bpel/`

Arnold, K. and Gosling, J. *The Java Programming Language.* (Addison-Wesley, 1996)

Ashdown, M. *Personal Projected Displays.* (PhD Thesis, 2004)

Barringer, H., Fellows, D., Gough, G., Jinks, P. and Williams, A. Multi-View Design of Asynchronous Micorpipeline Systems using Rainbow. In *VLSI '97* (1997)

Bierman, G. M. and Sewell, P. Iota: A concurrent, XML scripting language with applications to Home-Area Networks. *Technical Report 557 (ISSN 1476-2986)* (University of Cambridge, 2003)

Birrel, A. D. and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1) (1984), 39–59

Blackwell, A. Cognitive Dimensions of Tangible Programming Languages. In *First Joint Conference of EASE and PPIG* (2003), 391–405

Blackwell, A. F. First Steps in Programming: A Rationale for Attention Investment Models. In *IEE Symposium on Human-Centric Computing Languages and Environments* (2002), 2–10

Blackwell, A. F. Metacognitive Theories of Visual Programming: What do we think we are doing? In *Proceedings IEEE Symposium on Visual Languages* (1996), 240–246

Blackwell, A. F. *Using physical blocks to define interactions between electronic devices in a network.* (UK Patent GB2358726, 2000)

Blackwell, A. F. and Green, T. R. G. Does Metaphor Increase Visual Language Usability? In *Proceedings IEEE Symposium on Visual Languages* (1999), 246–253

Blackwell, A. F. and Hague, R. AutoHAN: An Architecture for Programming the Home. In *the IEEE Symposia on Human-Centric Computing Languages and Environments* (2001), 150–157

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C. and Orchard, D. (Eds). *Web Services Architecture.* (World Wide Web Consortium (W3C), 2004) Available at `http://www.w3.org/TR/ws-arch/`

Boudol, G. *Asynchrony and the π-calculus.* (INRIA Sophia-Antipolis , 1992)

Bray, T., Hollander, D. and Layman, A. (Eds). *Namespaces in XML.* (World Wide Web Consortium (W3C), 1999) Available at `http://www.w3.org/TR/1999/REC-xml-names-19990114`

Bray, T., Paoli, J., Sperberg-McQueen, C. M. and Maler, E. (Eds). *Extensible Markup Language (XML) 1.0 (Second Edition).* (World Wide Web Consortium (W3C), 2000) Available at `http://www.w3.org/TR/2000/REC-xml-20001006`

Brooks, F. P., Jr. *The Mythical Man-Month*: *Essays on Software Engineering.* (Anniversary edition, Addison-Wesley, 1995)

Burnett, M. Scaling Up Visual Programming Languages. *IEEE Computer*, 28(3) (1995), 45–54

Card, S. K., Morgan, T. P. and Newell, A. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7) (1980), 396–410

Carlisle, D., Ion, P., Miner, R. and Poppelier, N. (Eds). *Mathematical Markup Language (MathML) Version 2.0 (Second Edition).* (World Wide Web Consortium (W3C), 2003) Available at `http://www.w3.org/TR/2003/REC-MathML2-20031021/`

Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S. *Web Services Description Language (WSDL) 1.1.* (World Wide Web Consortium (W3C), 2001) Available at `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`

Clark, J. (Ed). *XSL Transformations (XSLT) Version 1.0.* (World Wide Web Consortium (W3C), 1999) Available at `http://www.w3.org/TR/1999/REC-xslt-19991116`

Clark, J. (Ed). *RELAX NG Specification.* (The Organization for the Advancement of Structured Information Standards [OASIS] , 2001) Available at `http://www.relaxng.org/spec-20011203.html`

Clark, J. and DeRose, S. (Eds). *XML Path Language (XPath) Version 1.0.* (World Wide Web Consortium (W3C), 1999) Available at `http://www.w3.org/TR/1999/REC-xpath-19991116`

Cohen, J., Aggarwal, S. and Goland, Y. Y. *General Event Notification Architecture Base*: *Client to Arbiter.* (Internet draft, 2000) Available at `http://www.upnp.org/download/draft-cohen-gena-client-01.txt`

Cox, R. *Analytical Reasoning with multiple external representations* . (PhD Thesis, University of Edinburgh, 1996)

Cypher, A. (Ed). *Watch What I Do*: *Programming By Demonstration.* (MIT Press, 1993)

Day, R. S. Alternative representations. *The Psychology of Learning and Motivation*, 22 (Academic Press, 1988), 261–305

DiBona, C., Ockman, S. and Stone, M. (Eds). *Open Sources*: *Voices of the Open Source Revolution.* (O'Reilly and Associates, 1999)

Droms, R. *Dynamic Host Configuration Protocol (DHCP).* (Request For Comments (RFC) 1531, 1993)

Dubuisson, O. *ASN.1 - Communication between heterogeneous systems.* (Morgan Kaufmann, 2000)

Elrod, S., Pier, K., Tang, J., Welch, B., Bruce, R., Gold, R., Goldberg, D., Halasz, F., Janssen, W., Lee, D., McCall, K. and Pedersen, E. Liveboard: a large interactive display supporting group meetings, presentations, and remote collaboration. In *SIGCHI conference on Human factors in computing systems* (1992), 599 – 607

Ene, C. and Muntean, T. A Broadcast-based Calculus for Communicating Systems. In *6th International Workshop on Formal Methods for Parallel Programming: Theory and Applications* (2001)

Fallside, D. C. (Ed). *XML Schema Part 0: Primer.* (World Wide Web Consortium (W3C), 2001) Available at `http://www.w3.org/TR/xmlschema-0/`

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. *Hypertext Transfer Protocol - HTTP/1.1.* (Request For Comments (RFC) 2616, 1999)

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Software Design.* (Addison-Wesley, 1994)

Gentner, D. and Nielsen, J. The Anti-Mac interface. *Communications of the ACM*, 39(8) (1996), 70–82

Golland, Y., Whitehead, E. J., Jr, Faizi, A., Carter, S. and Jensen, D. *HTTP Extensions for distributed authoring.* (Request For Comments (RFC) 2518, 1999)

Graham, P. *Hackers and Painters.* (O'Reilly and Associates, 2004)

Grant, C. A. M. Software visualization in Prolog. *Technical Report 511 (ISSN 1476-2986)* (PhD Thesis, University of Cambridge, 1999)

Green, T. R. G. and Petre, M. When Visual Programs are Harder to Read than Textual Programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)* (1992)

Green, T. R. G. Cognitive Dimensions Of Notation. In *People and Computers V*, Sutcliffe, A. and Macaulay, L. (Eds). (Cambridge University Press, 1989), 443–460

Green, T. R. G. and Blackwell, A. F. *Design for usability using Cognitive Dimensions.* (Tutorial at BCS conference on Human Computer Interaction, 1998) Available at `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf`

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J. and Nielsen, H. F. (Eds). *SOAP Version 1.2 Part 1: Messaging Framework.* (World Wide Web Consortium (W3C), 2003) Available at `http://www.w3.org/TR/soap12-part1/`

Hightower, J. and Borriello, G. Location Systems for Ubiquitous Computing. *IEEE Computer*, 34(8) (2001), 57–66

Horstmann, M. and Kirtland, M. *DCOM Architecture.* (1997) Available at `http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.asp`

Ishii, H. and Ullmer, B. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In *CHI '97* (ACM, 1997)

Kahn, K. ToonTalk - An Animated Programming Environment G for Children. *Journal of Visual Languages and Computing*, (1996)

Kelsey, R., Clinger, W. and Rees, J. (Eds). Abelson, H., Adams, N. I., IV, Bartley, D. H., Brooks, G., Dybvig, R. K., Friedman, D. P., Halstead, R., Hanson, C., Haynes, C. T., Kohlbecker, E., Oxley, D., Pitman, K. M., Rozas, G. J., Steele, G. L., Jr, Sussman, G. J. and Ward, M. *The Revised^5 Report on The Algorithmic Language Scheme.* (1998) Available at `http://www.schemers.org/Documents/Standards/R5RS/`

Kernighan, B. W. and Ritchie, D. M. *The C Programming Language.* (Second Edition, Prentice-Hall, 1988)

Klemmer, S. R., Li, J., Lin, J. and Landay, J. A. Papier-Mâché: Toolkit Support for Tangible Input. In *CHI 2004: Conference on Human Factors in Computer Systems* (ACM Press, 2004)

Le Hors, A., Le Hégaret, P., Wood, L., Nicol, G., Robie, J., Champion, M. and Byme, S. (Eds). *Document Object Model (DOM) Level 2 Core Specification.* (World Wide Web Consortium (W3C), 2003) Available at `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/`

Lee, P. M. *Bayesian Statistics: An Introduction.* (Second Edition, Oxford University Press, 1997)

Lesk, M. E. *Lex - A Lexical Analyzer Generator.* (Comp. Sci. Tech Rep. No 39, Bell Laboratories, 1975)

Lutz, M. *Programming Python.* (Second Edition, O'Reilly and Associates, 2001)

Lyons, J. *Language and Linguistics: An Introduction.* (Cambridge University Press, 1981)

Mackay, W. E. and Pagani, D. S. Video Mosaic: Laying out time in a physical space. In *ACM Multimedia '94* (ACM Press, 1994)

Marriot, K., Meyer, B. and Wittenburg, K. B. A Survey of Visual Language Specification and Recognition. *Visual Language Theory*, (Springer-Verlag New York Inc., 1998), 5–85

McGuinness, C. Problem representation: The effects of spatial arrays. *Memory and Cognition*, 13(3) (1986), 270–280

Meyers, S. D. Representing Software Systems in Multiple-View Development Environments. *Technical Report (ISSN CS-93-18)* (PhD Thesis, Brown University, 1993)

Milner, R. *Communicating and Mobile Systems: the π-Calculus.* (Cambridge University Press, 1999)

Murray-Rust, P., Rzepa, H. S., Wright, M. and Zara, S. A Universal approach to Web-based Chemistry using XML and CML. *Chemical Communications*, (16) (2000), 1471–1472

Nardi, B. A. *A Small Matter of Programming.* (MIT Press, 1993)

Negroponte, N. *Being Digital.* (Hodder and Stoughton, 1995)

Rees, G. (Ed). Nelson, G. *The Inform Designer's Manual.* (4th Edition, Interactive Fiction Library, 2001)

Newman, W. M., Eldridge, M. A. and Lamming, M. G. PEPYS: Generating Autobiographies by Automatic Tracking. In *ECSCW '91* (1991), 175–188

Norman, D. *The Invisible Computer.* (MIT Press, 1999)

Oberlander, J. Grice for graphics: pragmatic implicature in network diagrams.. *Information Design Journal*, 8 (1996), 163–179

Ostrovský, K., Prasad, K. V. S. and Taha, W. Towards a Primitive Higher Order Calculus of Broadcasting Systems. In *PPDP '02* (2002)

Ousterhout, J. K. Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer*, 31(3) (1998), 23–30

Petre, M. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6) (1995), 70–82

Peyton-Jones, S. (Ed). *Haskell 98 Language and Libraries: The Revised Report.* (A special issue of the Journal of Functional Programming, Cambridge University Press, 2003)

Prasad, K. V. S. A Calculus of Broadcasting Systems. *Science of Computer Programming*, (25) (1995)

Raggett, D., Le Hors, A. and Jacobs, I. (Eds). *HTML 4.01 Specification.* (World Wide Web Consortium (W3C), 1999) Available at `http://www.w3.org/TR/html401/`

Raskin, J. *The Humane Interface.* (Addison-Wesley, 2000)

Reiss, S. P. PECAN: program development systems that support multiple views. *IEEE Transactions on Software Engineering*, 11(3) (IEEE Press, 1985), 276–285

Rieman, J., Franzke, M. and Redmiles, D. Usability evaluation with the cognitive walkthrough. In *Conference on Human Factors in Computing Systems* (1995), 387–388

Roast, C. R., Khazaei, B. and Siddiqi, J. I. Formal Comparisons of Program Modification. In *IEEE International Symposium on Visual Languages (VL'00)* (2000), 165–171

Rodden, T., Crabtree, A., Hemmings, T. and Benford, S. Finding a place for UbiComp in the home. In *The Fifth International Conference Ubiquitous Computing* (2003)

Rodden, T., Crabtree, A., Hemmings, T., Koleva, B., Humble, J., Åkesson, K. and Hansson, P. Between the dazzle of a new building and its eventual corpse: assembling the ubiquitous home. In *Proceedings of the 2004 conference on Designing interactive systems: processes, practices, methods, and techniques* (ACM Press, 2004), 71–80

Rode, J. A., Toye, E. F. and Blackwell, A. F. The Fuzzy Felt Ethnography - understanding the programming patterns of domestic appliances. In *2nd International Conference on Appliance Design* (2004), 1–22

Rothermel, G., Li, L., DuPuis, C. and Burnett, M. What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs. In *1988 International Conference on Software Engineering* (1998), 198–207

Rumbaugh, J., Jacobson, I. and Bouch, G. *The Unified Modelling Language Reference Manual.* (Addison-Wesley, 1999)

Saif, U., Gordon, D. and Greaves, D. Internet Access to A Home Area Network. *IEEE Internet Computing*, 5(1) (2001), 54–63

Saif, U. and Greaves, D. Communication Primitives for Ubiquitous Systems or RPC Considered Harmful. In *ICDCS International Workshop on Smart Appliances and Wearable Computing* (2001)

Salber, D., Dey, A. K. and Abowd, G. D. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *CHI '99* (ACM Press, 1999)

Schneiderman, B. Direct Manipulation: A step beyond programming languages. *IEEE Computer*, 16(8) (1983), 57–69

Shalit, A. *The Dylan Reference Manual.* (Addison-Wesley Developers Press, 1996)

Steele, G. *Common LISP: The Language.* (2nd, Digital Press, 1984)

Sterling, L. and Shapiro, E. *The Art Of Prolog.* (Second Edition, MIT Press, 1994)

Sutherland, I. *SketchPad: A Man-Machine Graphical Communication System.* (PhD Thesis, Massachusetts Institute of Technology , 1963)

Suzuki, H. and Kato, H. Interaction-Level Support for Collaborative Learning: AlgoBlock - An Open Programming Language. In *Computer Support for Collaborative Learning '95* (1995)

Truong, K. N., Huang, E. M. and Abowd, G. D. CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home. In *Proceedings of UbiComp 2004, the Sixth International Conference on Ubiquitous Computing* (2004)

Turner, D. N. *The Polymorphic π-calculus: Theory and Implementation* . (University of Edinburgh, 1995)

Ullmer, B., Ishii, H. and Glas, D. mediaBlocks: Physical Containers, Transports, and Controls for Online Media. In *SIGGRAPH'98* (1998), 379–386

Walsh, N. and Muellner, L. *DocBook: The Definitive Guide.* (O'Reilly and Associates, 1999)

Weiser, M. The Computer for the 21st Century. *Scientific American*, (1991), 94–110

Whitley, K. N. Visual Programming Languages and the Empirical Evidence For and Against . *Journal of Visual Languages and Computing*, 8(1) (1997), 109–142

Williams, A. *Requirements for Automatic Configuration of IP Hosts.* (Internet Draft, 2002)

Williams, S. and Kindel, C. *The Component Object Model: A Technical Overview.* (1994) Available at `http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_comppr.asp`

Woo, M., Neider, J., Davis, T. and Shreiner, D. *OpenGL programming guide: the official guide to learning OpenGL.* (Addison Wesley, 1999)

Yang, Z. and Duddy, K. CORBA: A Platform for Distributed Object Computing. *Operating Systems Review*, 30(2) (1996), 4–31