Multi-Lingual End-User Programming with XML

R Hague and P Robinson
University of Cambridge Computer Laboratory William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK

{Rob.Hague,Peter.Robinson}@cl.cam.ac.uk

Abstract

There is no ideal programing language. Each is better suited to some tasks rather than others. This suitability varies not only with the overall goal of the finished program, but also with different stages of development such as architectural design, detailed implementation, and maintenance. The situation is even more acute in the case of end-user programming languages, which cater for a much more varied user population. It would therefore be advantageous to allow the same program to be viewed, and edited, in a number of different languages. We have developed a system, Lingua Franca, that provides this facility for end-user programming languages in the setting of ubiquitous computing in the home.

Introduction

Since the invention of FORTRAN in 1957, high level programming languages have proliferated to the extent that it is no longer feasible to count them. These range from special-purpose, niche languages to the almost ubiquitous C and C++, and include approaches such as visual programming and programming by demonstration in addition to conventional textual languages.. This diversity would not survive if all languages were equal. It exists because any given language is better at some tasks than others. Of course, commercial concerns also play a part. While there is a significant degree of overlap and duplication amongst languages, the selection of the proper language for a given task renders the programmer's task significantly easier (and, conversely, the selection of the wrong language may render it difficult or impossible).

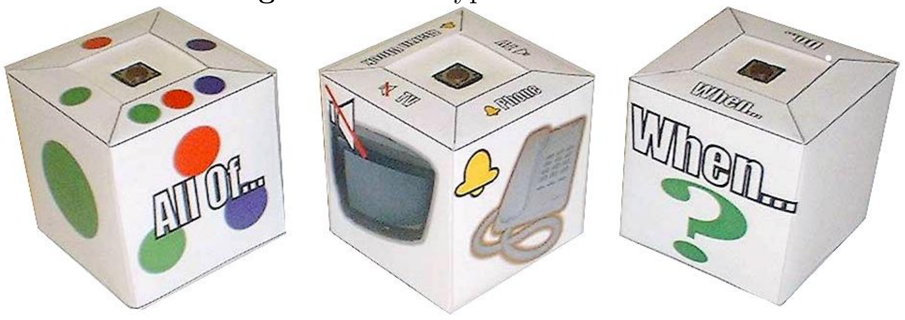
A similar disparity exists between different operations that a programmer may wish to perform on a program. The Cognitive Dimensions of Notation framework (Green, 1989) categorizes such tasks as *incrementation* (adding new information), *transcription* (from one notation to another), *modification* or *exploratory design* ("combining incrementation and modification, with the further characteristic that the desired end state may not be known in advance"). These four types of activity make different demands of languages, and so it would be reasonable to suppose that different languages are useful for different manipulations (and at different points in the development process).

This conjecture is borne out by numerous studies into the applicability to different representations of the same data to different problems based on that data (Whitley, 1997). Cox (1996) goes further, developing an environment that specifically supports the use of multiple notations for the representation of data relating to logic problems of the type found in GRE tests (used to assess applicants to graduate schools in the US). We have created an XML-based system that applies a similar idea to programming. A program may be created, viewed or edited in one of several languages, and changes are automatically visible in all representations.

Context

This work arises from observations made while designing an end-user programming language for use in the context of *ubiquitous computing* (technology that "disappears into the background" (Weiser, 1991)). Our work focuses on the networked home, in which domestic devices are augmented with communication and computation. Such a network allows devices to work in concert to do more than they can individually, but this is of little use if there is no way for

Figure 1: Prototype Media Cubes



the user to control this enhanced functionality. We believe that end-user programming may provide a means for such control.

One of the language designs considered was the *Media Cubes* (Blackwell and Hague, 2001), a tangible programming language in which programs are constructed by manipulating physical blocks as opposed to text or graphics (Figure 1). This language has numerous positive characteristics; in particular, it allows a smooth progression from direct manipulation to programming, and it provides a concrete representation for abstract concepts. This makes it particularly suited to users unfamiliar with programming. However, it has one major drawback: there is no external representation of the program once it is created. This makes it impossible to edit the program after it is created, or even to examine it to ensure it is correct.

Instead of redesigning the language to avoid this deficiency (and consequently losing many of its positive qualities), we sought to design an environment in which such languages may be usefully employed. By allowing a program created using the Media Cubes to be viewed and edited using another language, the deficiency becomes far less problematic.

Lingua Franca - Scripting in Many Languages

The system produced is based around *Lingua Franca*, an XML-based intermediate form common to all scripting languages in a system. Common intermediate forms have been used extensively to integrate components written in different languages, the most recent example being Microsoft's .NET Framework (Microsoft, 2004). Lingua Franca goes beyond conventional multiple-language systems in that it not only allows translation from various source languages to the intermediate form, but also allows translation from the intermediate form to the source languages. By combining these two processes, it is possible to effect translations between different source languages. This, in turn, allows a single component to be manipulated in multiple languaes at different times. In contrast, previous systems have allowed different language to be used for different components, but each component was always represented in the same language.

While it is possible to translate from traditional intermediate forms (which are almost exclusively designed for the purpose of being executed directly, or for further transformation into a form that may be exectuted), the results are generally imperfect reconstructions of the original source, and require significant effort to comprehend. Aside from the problems associated with recreating the structure of the code, secondary notation – elements that have meaning to the programmer, but do not affect execution, such as variable names, layout and comments – cannot be reproduced, as they were discarded during the translation into intermediate form. In contrast, Lingua Franca explicitly encodes all structure and secondary notation, allowing the original source code to be reconstructed exactly.

The two types of secondary notation that are most commonly discarded when translating a script from one form to another are point annotations, such as comments, and higher level structure, such as loops. Both of these may vary greatly from language to language. Lingua Franca allows multiple point annotation elements to be associated with a part of a script; each

Figure 2: A Simple Lingua Franca Program

```
<xax:group name="Shopping">
  <receive event="O">
    <nt>OutOf</nt>
    <receiveonce>
      <nt>ReorderFood</nt>
      <dispatch>
        <nt>Order</nt>
        <nts>
          <param event="O"
                 name="nts"/>
        </nts>
      </dispatch>
    </receiveonce>
  </receive>

  <receive>
    <nt>Day</nt>
    <nts>Sunday</nts>
    <dispatch>
      <nt>ReorderFood</nt>
    </dispatch>
  </receive>
</xax:group>
```

annotation is tagged with a notation type, to allow language environments to determine which (if any) to display. Higher level structure is represented by grouping; again, each group is tagged with a type (such as while loop), which may imply a particular structure, and language environments may use this to determine how to display the group's members. Unlike point annotation, any environment that can display Lingua Franca can display any group, as in the worst case it can simply display it as a grouped collection of primitive operations.

Different source languages are supported by the Lingua Franca architecture via "language environments" that translate between the source language and Lingua Franca. The most general class of language environments perform translation in both directions; these may be used to edit a script, by first translating from Lingua Franca to a "source" notation, modifying that representation, then finally translating it back to Lingua Franca.

Not all environments allow translation in both directions. Some language environments only translate from the source notation to Lingua Franca (and may only be used to create scripts). The Media Cubes are a prime example of such an environment. Others only translate from Lingua Franca to some other notation (and may only be used to display scripts).

Programs in Lingua Franca are held on a central database. All or part of the "corpus" of programs may be examined or modified over the network, via the HTTP protocol (Fielding et al., 1999). Lingua Franca is represented as XML. The use of standards representations and protocols allows implementors to take advantage of widely available tools and libraries.

The structure of Lingua Franca is broadly similar to that of the π-calculus (Milner, 1999), but uses asynchronous as opposed to synchronous events (several formal calculi based on asynchronous events exist, but non match the semantics of Lingua Franca). Asynchronous events were considered to be a more suitable for ubiquitous computingSaif and Greaves (2001), and are implemented by the GENA protocol (Cohen et al., 2000).

Figure 2 illustrates a simple Lingua Franca program, expressed using the XML representation. The overall effect of this program is to construct a shopping list of items as they run out, and then send out orders for them on Sunday. The program consists of a group element named “Shopping”. Within this group are two receive elements; these elements represent listening processes.

Events are specified using a notification type (nt) and a notification subtype (nts), and are written in the form “nt/nts”, with “nt/*” being shorthand for an event with the given notification type, but any notification subtype. receive and receiveonce elements react to any event with the appropriate notification type and optional subtype, specified by the nt and nts elements, by dispatching events specified by any dispatch subelements, and creating new processes corresponding to any receive or receiveonce subelements. receiveonce elements act in the same way, but as opposed to continually listening for events and reacting to each one, they react to a single event and are then removed. The event attribute may be used to bind an event to a name, allowing data from the event to be accessed using the param element; in the example above, this is used to produce an “Order” event with the same notification subtype as the corresponding “OutOf” event.

Whenever an OutOf/* event occurs, the first receive element creates a new process, which consists of a receiveonce that reacts to a ReorderFood/* event (the subtype is not significant), and dispatches an event Order/O.nts, where O.nts is the notification subtype of the original OutOf/* event. On receipt of a Day/Sunday event, the second receive element dispatches a ReorderFood/- event, which causes those processes to dispatch the appropriate Order events, after which they are removed (as they correspond to receiveonce elements, as opposed to receive.)

A Menagerie of Programming Languages

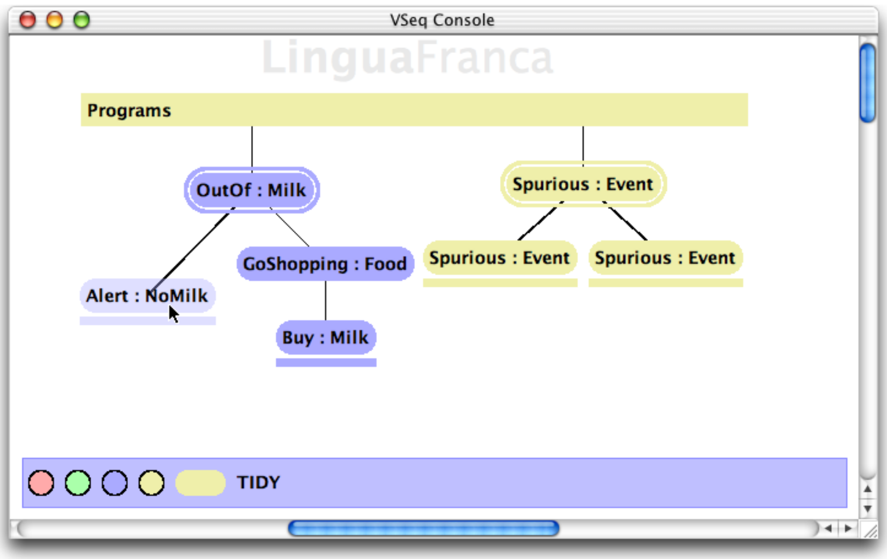
A variety of scripting languages are being developed in order to demonstrate the flexibility and range of the Lingua Franca architecture. These languages are designed to complement each other, in that they may be used to perform different manipulations on the same script with ease.

It is of course possible to manipulate Lingua Franca directly in XML form. However, this is needlessly difficult, as the programmer only has access to primitive operations, and would have to build up higher level operations from scratch. More importantly, manipulating the code directly in this way bypasses most of the integrity checks that would be performed by a bona fide language environment, and hence creates the risk of introducing malformed code into the database, or accidentally removing or modifying data associated with another language.

Hence, a textual language, LFScript, has been developed to provide an interface familiar to those with experience of conventional scripting languages, allowing access to low-level features of the Lingua Franca system without the problems associated with manipulating the XML directly. It is envisaged that this will be mostly useful for creating and editing substantial scripts, a task most likely to be undertaken by someone with at least some programming background. It would be possible to compile an existing imperative language such as C into Lingua Franca, as the event mechanism is sufficiently flexible to model imperative execution. However, programs encoded in this way would have a very different structure from those created using event-based languages, and as such it would be difficult to take full advantage of the facilities for providing two-way translation. Instead, the textual language is patterned more on declarative languages, and makes the Lingua Franca primitives available to the user, in addition to providing higher level constructions such as synchronous functions and sequencing.

The VSeq language environment allows the user to manipulate Lingua Franca programs in the form of event diagrams. These diagrams represent causal relationships via a tree of nodes.

Figure 3: A program rendered in the VSeq console, and a fragment of the corresponding Lingua Franca source



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.

A "console" application, shown in Figure 3, allows the user to edit the diagram interactively. Leaf nodes in the diagram (further distinguished by being underlined) represent events that will be emitted, and correspond to dispatch nodes. Unadorned nodes correspond to receiveonce nodes, while nodes with an outer border represent receive nodes. Rectangluar strips, such as the one labelled "Programs" in Figure 3, are used to represent groups.

The ordering implied by the tree corresponds to causality; the effect of a node is conditional on the event described by its parent. Nodes may be moved by dragging with the mouse. Horizontal ordering of siblings is reflected in the sequential ordering of elements in the corresponding Lingua Franca program. Dragging one node over another connects adds it as a child of that node, provided that the resulting tree would correspond to a valid Lingua Franca program.

The console ensures 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 oval allows new nodes to be created. The "Tidy" tool lays out a node and all of its children in an orderly fashion, without changing the ordering or containment properties.

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.

We are also considering the feasibility of a purely presentational form, and cannot be used to create or edit scripts, but only to display them. This allows it to be specialized in order to facilitate searching and comprehension of scripts. This form may be interactive, in the sense that it may include user-initiated animation to aid navigation, but the user cannot make any changes that are propagated back to the Lingua Franca corpus.

Perhaps the most unusual of the language environments being developed for use with Lingua Franca is the Media Cubes language, mentioned previously. This is a "tangible" programming language. In other words, it is a language where programs are constructed by manipulating physical objects—in this case, cubes augmented such that they can determine when they are close to one another.

The faces of the cube signify a variety of concepts, and the user creates a script by placing appropriate faces together; for example, to construct a simple radio alarm clock, the "Do" face of a cube representing a conditional expression would be placed against a representation of the act of switching on a radio, and the "When" face against a representation of the desired time. In an appropriately instrumented house, the representation can often be an existing, familiar item, or even the object itself. In the above example, a time could be represented using an instrumented clock face, and turning the radio on could be represented by the radio itself, or its on switch. One significant difference between the Media Cubes language and other tangible computing interfaces is that the user does not construct a physical model of the program, but performs actions with the cubes that cause parts of the program to be created or combined.

The Media Cubes language is intended to be easy for those unfamiliar to programming, and as such would provide a low-impact path from direct manipulation to programming. However, as mentioned, the language as it stands is unusual in one very significant respect - scripts do not have any external representation. This means that it is only feasible to construct small scripts, and that, once created, scripts may not be viewed, and hence may not be modified. In most circumstances, this would render the language all but useless, but the Lingua Franca makes it feasible to include niche languages such as the Media Cubes in a system without sacrificing functionality.

Status and Further Work

A working prototype of the Lingua Franca execution engine has been implemented. This comprises of an interpreter for the intermediate form, an HTTP server to allow clients to add, remove and replace code in the corpus over the network, and an interface to the GENA protocol to allow scripts to interact with the outside world.

The VSeq language environment has been implemented, and the other language environments mentioned are under development. Working prototypes of the Media Cubes have been developed, but currently use infra-red to communicate with a base station. The requirement for line-of-sight that this imposes makes the current prototypes unsuited to user testing; a version based on radio communication has been planned. The Media Cubes language environment has been developed as a simulator, in the using a simulator, combined with "Wizard of Oz" techniques for user studies (Wilson and Rosenberg, 1988).

A core requirement of Lingua Franca language environments that implement bidirectional translation is that that translation must be reversible. For example, if a program is translated from VSeq into Lingua Franca, and then back into VSeq, the result should be identical to the original program. This goal has been achieved in the language environments to date. As components of the system may run on devices with limited processing power, runtime efficiency is a factor. Consequently, we have moved from using a scripting language (Python) and a generic tree representation (DOM) to a systems language (Java) and a custom representation in more recent versions.

Once the prototype language environments have been produced, a user study in which participants apply the languages to various tasks will be conducted. This will allow us to establish the relationships between tasks and languages, and will inform the subsequent refinement of those languages.

An important point about the work is that it limits its scope to a specific domain, that of end-user programming in the domestic environment. This is intentional, as it reduces the problem of translating between arbitrary, general purpose programming languages to the far more tractable problem of translating between closely related languages. This is not only useful in its own right, but may well produce results that will inform the design of more generally applicable systems.

References

Alan F Blackwell and Rob Hague. Autohan: An architecture for programming the home. In *the IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 150–157, 2001.

J Cohen, S Aggarwal, and Y Y Goland. General event notification architecture base: Client to arbiter, 2000. Internet draft `http://www.upnp.org/download/draft-cohen-gena-client-01.txt`.

Richard Cox. *Analytical Reasoning with multiple external representations.* PhD thesis, University of Edinburgh, 1996.

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

T RG Green. Cognitive dimensions of notation. *People and Computers V*, pages 443–460, 1989.

Microsoft. *Microsoft .NET Framework.* 2004. `http://www.msdn.microsoft.com/netframework/`.

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

U Saif and D Greaves. Communication primitives for ubiquitous systems or rpc considered harmful. In *ICDCS International Workshop on Smart Appliances and Wearable Computing*, 2001.

Mark Weiser. The computer for the 21st century. *Scientific American*, pages 94–110, 1991.

K N Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, (1):109–142, 1997.

J Wilson and D Rosenberg. *Prototyping for User Interface Design.* North-Holland, New York, 1988.