SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **3**:1–2 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

End-User Programming of Reconfigurable Systems

Rob Hague[1,*] and Peter Robinson[1]

[1] *University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK*

SUMMARY

There is no ideal programing language. Each is better suited to some tasks rather than others. This is not only true for complete programs, but also for different stages 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

In the five 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, 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

*Correspondence to: University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK

home networking; how can a householder exploit the potential that the network provides? In other words, how does the user control the automated reconfiguration of a pervasive system?

We seek to addresses one particular approach to usability in the pervasive home network – that of end-user programming. This may seem an unusual approach, 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 employ whichever language is best suited to the task at hand, and in particular choose a different language for each stage of program development. This approach was initially conceived as a result of work on a novel language, the Media Cubes, which is also presented.

We provide an overview of our approach to the problem, which is based on the use of a language-neutral intermediate form to effect translation between source languages. We then describe Lingua Franca, a system based on this approach, and VSeq, a visual programming language incorporated into the system.

Background

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 necessarily 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.

The importance of this transition is difficult to overstate. 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 machine. Of course, at this stage, programming was itself arcane and a difficult skill to master, 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 subsequent successors enabled specialists in a wide range of areas to harness the ever-increasing power of computers.

Throughout the early decades of computing, the terms "user" and "programmer" 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 necessity of programming. This lead to the concept of applications - programs that could be used in the form given to perform some

specific task. Technologies such as direct manipulation ([1]) meant that such applications could be made even easier to use.

Such improvements in ease of use, combined 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.

From User to Programmer

The development of computer user 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 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.

Another problem is that end-user applications typically provide little opportunity for automation. This fails to take advantage of a basic strength of computers:

> 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? [2]

The *Free Software* and *Open Source* movements may at first sight appear to offer a solution, and indeed do relieve many of the problems of proprietary software 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 accountants, who have computational needs and want to make serious use of computers, but who are not interested in becoming professional programmers. [3]

While open source development gives users 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 general-purpose 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 end-user programming system in use today is the automated spreadsheet, introduced with VisiCalc, and typified by Lotus 1-2-3 and Microsoft Excel. 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 Apple Computer's HyperCard. This system allows users to write 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 behaviour such as animation to be associated with a reader's actions. 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.

From Machine Room to Every Room

Weiser [4] and Norman [5] 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 [5], Raskin [6] and others. However, while this is 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 truly 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 Escritoire [7], and very small, as in mobile phones). A particular class of hardware that is far more important in ubiquitous computing than conventional computing are sensors. An accurate and timely picture of the environment in which activity is occurring is a major component of many ubiquitous computing systems. This gives rise to 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 room for a keyboard, a surface to rest on, or even a free hand. 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.

From One Language To Many

Most, if not all, computing tasks may be regarded as manipulation by the user of a set 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 [8] 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 cognitive psychology work relating to the use of notations, and frameworks for applying this work to the design of notations [9].

Several studies have shown that task performance may be influenced by the way in which data is represented ([10], [11]). 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.

Some conventional programming tools make limited, ad-hoc use of multiple representations; for example, systems such as JavaDoc produce hypertext documentation from source code. Similarly, code generators such as *yacc*, and the GUI design tools and code generation wizards of Integrated Development Environments (IDEs) such as Microsoft Visual Studio allow the creation of source code via a notation more suited to a specific task (such as Backus-Naur style context free grammars, or manipulations of graphical elements on a virtual canvas). However, modification of programs is usually still 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.

Meyers [12] describes one such system. This system is based on a canonical representation of programs, coupled with an open-ended set of views that may be applied to this representation. This system allows translation both to and from the canonical representation, but does not provide structured support for features present in some but not all views. This makes supporting very different languages problematic. Simonyi's "intentional programming" [13] takes a slightly different approach; instead of allowing the user to select a language with which to view the entire program, the system allows the user to construct the program from an ad hoc combination of language features. Each of these features may have multiple views. This affords increased flexibility, arguably with a corresponding increase in complexity for the user. In both cases, the systems are targetted at professional programmers; however, multi-language techniques may also be profitably employed in systems aimed at wider user communities.

Figure 1. Prototype Media Cubes

End-User Programming In The Home

This work arises from observations made while designing an end-user programming language for use in the context of *ubiquitous computing*. The field of ubiquitous computing concerns technology that "disappears into the background" [10]. Our work focuses on the networked home, in which domestic devices are augmented with networking 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 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* [14], a tangible programming language in which programs are constructed by manipulating physical blocks as opposed to text or graphics. This language has several advantages. 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 determine its function, or ensure that it is correct.

Instead of redesigning the language to circumvent 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 significant.

Translation based on common 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 (for example, names, comments, and formatting such as indentation in a textual language). 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.

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.

A Multi-Language Programming Environment

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 or Eclipse; 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 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.

Issues of reversible translation

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 mapped onto that object.

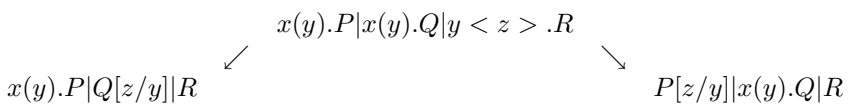
Another way of stating this requirement is to say that f composed with f^{-1} must be the identity on texts, and f^{-1} composed with f must be the identity mapping on programs. The first property requires that each text maps to a unique program. This may be achieved by ensuring that any redundancy in 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 was 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. 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.

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.

Figure 2. Non-determinism in the π-calculus



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 several steps, each of which consists of a term in the language being rewritten as another 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.

In the case of a deterministic system such as the λ-calculus, 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, see Figure 2).

One advantage of using a term rewriting approach in a multi-language setting is that is 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 of the languages supported by the system. This gives the user the same latitude of choice of language for debugging as for other tasks.

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 [15], § 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".

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 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 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 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. Given that these texts are equivalent in terms of execution, what distinguishes them from one another?

The term *Secondary Notation* is used to describe the features of a text that do not affect execution. 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. However, the notations 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 spatial 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 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 language 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.

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

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

Structure as Secondary Notation

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

The vast majority of structures in most programming languages are based on the notion 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.

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 [8] or Schemas [16], [17].

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.

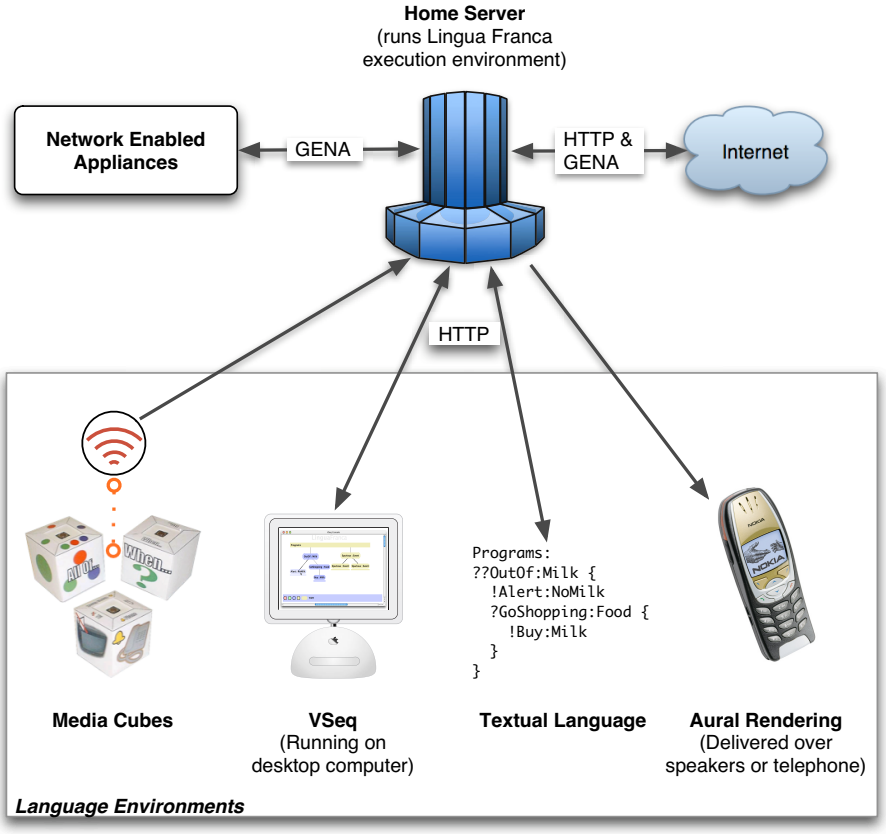
An Overview of the Lingua Franca architecture

We use the term "Lingua Franca" to describe both an intermediate form supporting bidirectional translation, and the software architecture using this intermediate form. In this section, we 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. This software is responsible for 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 connects Lingua Franca code to other devices and software in the home network.

The external representation of Lingua Franca, used when transferring code between pieces of software, is a dialect of XML. Modification of the corpus is performed via HTTP [18], using standard request types. The interface is comparable to, but simpler than, WebDAV [19]. 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 [20], allowing the requester to specify any point in the document tree. XPath specifications represent a path (or multiple paths) from the document root to a node, 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 this case.

Figure 4. The Lingua Franca architecture



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 4.

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. 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. 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.

In Lingua Franca, paths specify a node in the corpus, using XPath language. 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, effecting editing of programs.

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 easily added to the program database.

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. Enhanced versioning 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 explicitly add new content 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.

The Lingua Franca Execution Model

The Lingua Franca execution model is loosely based on the π-calculus [21], with certain important differences. These stem from the decision to model events in a similar way to GENA [22], the underlying event mechanism for 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 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.

The External Representation of Lingua Franca

We have chosen XML [8] 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. Why have we made this choice?

XML is a widely-used standard for encoding structured data. It was originally designed for the encoding of documents, and is a derivative of SGML, an older and more complex standard serving broadly the same purpose.

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 [23], [24] and Chemical Markup Language[25], While other standards, most notably ASN.1 [26], provide generic mechanism for the interchange of structured data, they are notoriously difficult to implement. 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 processor that complies fully with standards. 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 by 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 the case of Lingua Franca, interoperability with other components of the home network is

Figure 5. XML representation of the execution features of Lingua Franca

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

essential, while the rate of transactions is likely to be low. Hence, XML is a good choice of interchange format.

In addition, several programming languages have used XML as a concrete syntax. The most notable of these is XSLT [27], 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 of 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.

Lingua Franca in XML

The representation of Lingua Franca in XML is based upon a straightforward translation of the expressions used to describe the execution model. Figure 5 shows the XML representation of a simple Lingua Franca process. 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 terminates.

The XML representation of Lingua Franca makes use of XML Namespaces [28]. 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.

The `receive` element is used to specify a process that listens for events with a certain notification type and subtype. When such an event is received, a new process is created, and events are emitted, based on the immediate children of the element. The `receiveonce` behaves similarly, but is removed after it receives a matching event. The `event` attribute of these elements is used to bind any received event to a specific name. This name may be dereferenced in the resulting processes using a `param` tag, allowing these processes to vary based on the specific event that triggered them.

The `dispatch` element is used to specify that an event should be emitted. `nt` and `nts` are used in `receive`, `receiveonce` and `dispatch` elements to specify notification type and subtype. If `nts` is omitted in a `receive` or `receiveonce`, the containing expression matches events with any NTS. `dispatch` elements a required to specify both NT and NTS. Other children of `receive` and `receiveonce` elements specify the resulting process.

A feature not shown in the example is parallel composition of processes; this is achieved simply by placing the processes as siblings. In addition, the `new` element provides access to names guaranteed to be fresh, and the `sum` element provides summation akin to that in the π-calculus.

The external representation is 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 to not 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 the general secondary notation elements of the kind described in *Translation based on common intermediate form*. The `group` element is used to express grouping. The `note` element is used to add secondary notation. It contains an application-specific encoding of the secondary notation item; the attributes of the element may be used to determine this encoding.

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 neccesary 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, an 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 compiled languages) are not of concern for the development of the prototype.

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 [29]. However, as the design of the Lingua Franca architecture stabilised, the flexibility of generic technologies became less important, and the extra work involved in applying them to the specific problem more so. 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 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 is 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. However, in common with many other situations, the node types in Lingua Franca are unlikely to change (while the definition of Lingua Franca remains the same), each language environment is likely to add additional operations.

The *visitor pattern* [30] describes an alternative structure. The `visitor` interface contains an `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. However, adding new node types is difficult, as it requires a change to the visitor interface, and consequently to all classes implementing it. LFCore partially avoids this problem by providing a `defaultVisit` method that is called by the default implementation of each `visit` method. Java allows additional methods to be added to a supertype without recompilation of subtypes, and hence 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 as such may employ any of the numerous XML parsers that implements 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 parse 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 its data to and from its XML representation in order to parse the data provided.

VSeq

The most mature language environment created for the Lingua Franca system is VSeq ("Visual Sequences"). This language environment 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.

VSeq represents a Lingua Franca program as a tree diagram broadly mirroring the structure of the underlying Lingua Franca code. 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 receiveonce nodes. Nodes with an outer border represent receive 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.

Groups 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. Note nodes are not directly represented in the diagram. Instead, they are used to determine properties of the graphical representation of their parent nodes, such as background colour.

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 use 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 notation (distinct from explicit secondary notation represented by group and note elements).

The layout algorithm is implemented via `LayoutVisitor`, a subclass of `LFVisitor` (see visitor pattern). `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 determining the absolute position of the node requires traversal of the tree. 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 the diagram is modified by the user).

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 to determine the background colour. The latter is one of a set of common `NoteNode` subclasses used to represent secondary notation properties that may be used by multiple language environments. The background colour of an element's representation is a concept that makes sense in many contexts; in comparison, 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.

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

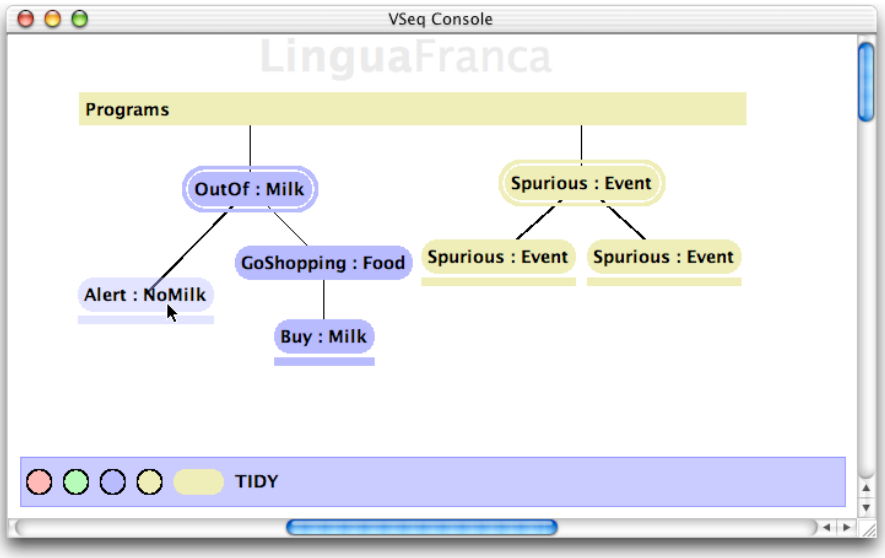
A "console" application, shown in Figure 6, allows the user to edit the diagram interactively. This is a standard desktop application, implemented in Java using the LFCore toolkit. It runs on any platform that provides Java 2 Standard Edition and the Swing user interface toolkit. The console may be used quite 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.

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 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,

Figure 6. A program rendered in the VSeq console, and a fragment of the corresponding Lingua Franca source




```
<corpus xmlns:xax="urn:xax" xmlns="urn:linguafranca">
  <xax:note role="rgh22.linguafranca.vsec.VisualPropertiesNode"
        creator="VSeq"> 0,0,58,27 </xax:note>
  <xax:group name="Programs">
    <xax:note role="rgh22.linguafranca.vsec.VisualPropertiesNode"
          creator="VSeq"> -72,60,239,27 </xax:note>
    <receive>
      <nt> OutOf </nt> <nts> Milk </nts>

      <xax:note role="rgh22.linguafranca.vsec.VisualPropertiesNode"
            creator="VSeq"> 0,60,100,27 </xax:note>
      <xax:note role="rgh22.linguafranca.common.BGColorNode"
            creator="VSeq"> #aaaaff </xax:note>

      <dispatch>
        <nt> Alert </nt> <nts> NoMilk </nts>
        <xax:note role="rgh22.linguafranca.vsec.VisualPropertiesNode"
              creator="VSeq"> -99,76,110,27 </xax:note>
      </dispatch>
```

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. This is achieved using `LayoutVisitor`. A parameter is passed to modify the visitors behaviour such that it will overwrite any existing `VisualPropertiesNode`s.

Other Language Environments.

In addition to VSeq, two other prototype language environments have been implemented. Firstly, a software simulation of the Media Cubes language environment has been produced. This consists of a textual console allowing interactions between media cubes to be entered interactively. As these interactions are received, a program is built up within the language environment. When the interaction signalling completion of the program is received, the program is submitted to the execution engine. Within this simulation, the core Media Cubes language has been implemented, along with several extensions.

Secondly, a textual language environment, LFScript, has been implemented. This takes the form of a conventional command line tools that provide bidirectional translation between LFScript and Lingua Franca. As with traditional languages, LFScript source code is stored in plain text files. In order to retain non-representable secondary notation, a reference to original Lingua Franca program is included in the LFScript representation. Wherever non-representable notation occurs, it is replaced by a tag containing the index of theappropraite element in the original program. This allows the element to be retrieved when translating the modified LFScript source back to Lingua Franca. The LFScript language consists of a consise notation that directly mirrors the Lingua Franca primitives, along with higher level constucts. A small number of higher-level constructs are present in the current prototype, and additional constructs may be readily added.

Conclusions

This paper has introduced a scheme for end-users to program reconfigurable domestic systems.

Representative parts of the Lingua Franca architecture (including the execution engine and program database, and a visual language environment) have been implemented. While still prototypes, the components implement the full functionality, including communications protocols. This allows them to be connected to other systems supporting these protocols (in particular, GENA). Performance is more than adequate; in particular, the VSeq console operates at acceptable frame rates even on relatively modest hardware. A more detailed description and evaluation of the work may be found in [31].

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.

As systems become more complex, automation and reconfiguration become increasingly important. The Lingua Franca architecture provides a flexible mechanism for integrating a wide variety of programming languages into a cohesive system. These languages can be tailored to suit any user population, in any situation, allowing the broadest possible access to the reconfigurability of the system, and the power that provides.

REFERENCES

1. Sutherland I. *SketchPad: A Man-Machine Graphical Communication System*. Massachusetts Institute of Technology , 1963. (PhD Thesis)
2. Cypher A (ed). *Watch What I Do: Programming By Demonstration*. MIT Press, 1993.
3. Nardi BA. *A Small Matter of Programming*. MIT Press, 1993.
4. Weiser M. The Computer for the 21st Century. *Scientific American*. September 1991; pp. 94-110.
5. Norman D. *The Invisible Computer*. MIT Press, 1999.
6. Raskin J. *The Humane Interface*. Addison-Wesley, 2000.
7. Ashdown M. Personal Projected Displays. (PhD Thesis) *Technical Report 585 (ISSN 1476-2986)* University of Cambridge, 2004
8. Bray T, Paoli J, Sperberg-McQueen C and Maler E (eds). *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium (W3C), 6 October 2000.
9. Green TRG and Blackwell AF. *Design for usability using Cognitive Dimensions*. 1998. (Tutorial at BCS conference on Human Computer Interaction)
10. Cox R. *Analytical Reasoning with multiple external representations* . University of Edinburgh, 1996. (PhD Thesis)
11. Whitley KN. Visual Programming Languages and the Empirical Evidence For and Against . *Journal of Visual Languages and Computing*. 1997; 1 pp. 109-142.
12. Meyers SD. Representing Software Systems in Multiple-View Development Environments. (PhD Thesis) *Technical Report CS-93-18* Brown University, May 1993
13. Simonyi C. The Death Of Computer Languages, The Birth Of Intentional Programming. *Technical Report MSR-TR-95-52* Microsoft, 1995
14. Blackwell AF and Hague R. *AutoHAN: An Architecture for Programming the Home*. 2001. In proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments pp. 150-157
15. Lyons J. *Language and Linguistics: An Introduction*. Cambridge University Press, 1981.
16. Fallside D (ed). *XML Schema Part 0: Primer*. World Wide Web Consortium (W3C), 2 May 2001.
17. Clark J (ed). *RELAX NG Specification*. The Organization for the Advancement of Structured Information Standards [OASIS] , 2001.
18. Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P and Berners-Lee T. *Hypertext Transfer Protocol - HTTP/1.1*. 1999. (Request For Comments (RFC) 2616)
19. Golland Y, Whitehead EJ Jr, Faizi A, Carter S and Jensen D. *HTTP Extensions for distributed authoring*. 1999. (Request For Comments (RFC) 2518)
20. Clark J and DeRose S (eds). *XML Path Language (XPath) Version 1.0*. World Wide Web Consortium (W3C), 16 November 1999.
21. Milner R. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
22. Cohen J, Aggarwal S and Goland YY. *General Event Notification Architecture Base: Client to Arbiter*. 2000. (Internet draft)
23. Gudgin M, Hadley M, Mendelsohn N, Moreau J and Nielsen H (eds). *SOAP Version 1.2 Part 1: Messaging Framework*. World Wide Web Consortium (W3C), 24 June 2003.
24. Carlisle D, Ion P, Miner R and Poppelier N (eds). *Mathematical Markup Language (MathML) Version 2.0 (Second Edition)*. World Wide Web Consortium (W3C), 21 October 2003.
25. Murray-Rust P, Rzepa HS, Wright M and Zara S. A Universal approach to Web-based Chemistry using XML and CML. *Chemical Communications*. 2000; 16 pp. 1471-1472.
26. Dubuisson O. *ASN.1 - Communication between heterogeneous systems*. Morgan Kaufmann, September 2000.
27. Clark J (ed). *XSL Transformations (XSLT) Version 1.0*. World Wide Web Consortium (W3C), 16 November 1999.

28. Bray T, Hollander D and Layman A (eds). *Namespaces in XML*. World Wide Web Consortium (W3C), 14 January 1999.
29. 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), 13th November 2003.
30. Gamma E, Helm R, Johnson R and Vlissides J. *Design Patterns*: *Elements of Reusable Software Design*. Addison-Wesley, 1994.
31. Hague RG. End-User Programming In Multiple Languages. (PhD Thesis) *Technical Report 651 (ISSN 1476-2986)* University of Cambridge, 2005