Multilingual Programming: Coordinating Programs, User Interfaces, On-Line Help and Documentation

GARY PERLMAN

School of Information Technology
Wang Institute of Graduate Studies
Tyngsboro, Massachusetts

ABSTRACT

The high cost of software is not due to the difficulty of coding, but in recoding and redocumenting software. This can be better understood when one considers how many expressions of the same ideas must be constructed and coordinated. Program code and comments, user interface and on-line help, and a variety of off-line documents, all must be consistent. A solution to the coordination problem is presented in this paper. Multilingual programming is a method of developing software that uses a database of information to generate multiple target languages like commented program code, user interface languages, and text formatting languages.

The method begins with an analysis of a domain to determine key attributes. These are used to describe particular problems in the domain and the description is stored in a database. Attributes in the database are inserted in templates of idioms in a variety of target languages to generate solutions to the original problem. Because each of these solutions is based on the same source database of information, the solutions (documents, programs, etc.) are consistent. If the information changes, the change is made in the database and propagated to all solutions. Conversely, if the form of a solution must change, then only the templates change. In sum, the method saves much effort for updates of documents and programs that must be coordinated by designing for redesign.

CONTENTS

Problems with Documentation
Examples
    Experimental Design Specification
    Data Bases of Bibliographic References
    Data Analysis System Interface
    Option Parser Generator
    Electronic Survey System
Formalization
        Abstractions
        Idiomatic Templates
        Database of Attributes
        Multiple Target Languages
        Template Macros
    Properties of Multilingual Programs
        Generalization & Imagination
        Flexibility and Resilience to Change
        Accuracy and Consistency
        Economy of Expression
Discussion
        Choosing the Appropriate Focus
        Multiple Views of Programs
        Natural Language
    Cost/Benefit Analysis
References

PROBLEMS WITH DOCUMENTATION

There are many types of text associated with software: (a) the program code for the software, (b) the comments in the code, (c) the user interface specification, (d) on-line error messages, (e) on-line documentation (f) off-line reference materials, (g) off-line quick-reference sheets, and probably others. A major problem with software documentation is maintaining accuracy and consistency. Accuracy is the coordination of the program code with all other forms of documentation. ***Does the documentation accurately reflect the input/output behavior of the program?*** Consistency is the similarity of related document parts. ***Are examples, options, etc., shown in the consistent formats throughout?*** These are not easy problems to solve. Here is a typical scenario:

> ***A program is designed and written. Comments are inserted into the program. Preliminary documentation for the program is written, and users give feedback to the developers. New features are put in the program, and some, but not all, of the comments in the code are updated. Some prompts and error messages in the user interface are not changed to reflect the workings of the new program. New documentation is written, after which some user interface prompts are modified. The program is shipped to market.***

I contend that the problems of accuracy and consistency can be traced to the wasted dual efforts of programmers and documenters. Traditionally, documentation by programmers have been viewed as inefficient for several reasons: (a) Programmers do not think documentation is their problem. (b) Programmers are not interested in writing documentation. (c) Programmers do not have the time, or their time is considered too valuable, to be writing documentation. (d) Programmers do not know how to write good documentation. These are some of the same reasons why programmers are viewed as inefficient workers on user interfaces (Perlman, 1983). Consequently, programmers do not write documentation, except for program comments, and technical writers are hired to write user documentation. Two

groups of people work on specifying the same information in different formats for different audiences. Programmers write for compilers and programmers that might work on their code in the future, and documenters write for a variety of user populations. It is a waste of effort to have different people spend their time expressing the same idea in different languages.

This paper presents some practical solutions to the problem of accuracy and consistency of documentation. I will not talk about documentation separated from the issues of programming, user interfaces, or on-line help. These problems must be addressed in a coordinated fashion.

EXAMPLES

With the following examples, I hope to convey the diverse applications of multilingual programming by showing its use in a variety of domains. The technique can be summarized as follows: We begin with ***analysis*** of the problem domain, breaking it into small parts. Then we use this analysis to describe a particular problem. At that point, there is an abstract description of the problem. We then ***synthesize*** the description into a solution. Because we have a point at which a problem is described abstractly, we can synthesize several solutions.

While the above is abstract, it can be further summarized as analysis followed by multiple syntheses. In the following examples, this pattern is the one to watch for. The method will be formalized later.

Experimental Design Specification

UNIX¦STAT (Perlman, 1980) is a compact data analysis system developed at the University of California, San Diego, and at the Wang Institute of Graduate studies, running on the UNIX operating system (Ritchie & Thompson, 1974). **anova**, a UNIX¦STAT program, does a general analysis of variance. For non-statistically trained people, that means it is used primarily for analyzing data collected from experiments with controlled factors. Traditional ANOVA programs (Dixon, 1975; Nie *et al*, 1975) require that data be input as a matrix and the description of the experimental design information is in a special language separate from the data. In my experience, this method of experimental design specification leads to confusion and errors when used by inexperienced analysts. The **anova** program was designed to read self-documented input and from that, infer the structural relationships (the experimental design) in the data.

Each input line to **anova** contains one datum preceded by the names of the level of factors at which that datum was obtained. For example, suppose we have an experiment testing the effectiveness of two display formats, **B&W** and **color**, to two classes of readers, **young** and **old**. We present both formats to each reader, and measure comprehension on a percentage scale. Some of the data might look like this:

```
BamBam   B&W     young   52
BamBam   color   young   78
Fred     color   old     25
Fred     B&W     old     75
Pebbles  color   young   83
Pebbles  B&W     young   65
Wilma    B&W     old     93
Wilma    color   old     58
```

anova takes this analysis and infers the experimental design by synthesis. There are several points worth noting in the data. (1) The order of input lines to **anova** does not matter. (2) Each line is close to self explanatory; we know that **Fred** is **old** and what his scores are for the **B&W** and **color** format conditions. (3) **From the data,** we can see that every subject saw both format conditions (it is a ***within subjects factor***), but no subject was both **young** and **old** (age is a ***between subjects factor***). (4) There were four subjects.

The idea behind the **anova** program is to remove tedious and error prone tasks from data analysts by providing a synthesis of analysis. Given this design information, much of the data analysis process can be automated and verified (Perlman, 1982).

Data Bases of Bibliographic References

The references to this paper are stored in a simple database. The format for a record looks something like this:

```
author  = Perlman G
article = Multilingual Programming
journal = Asterisk
date    = in press
```

Records are extracted from a central database and sorted before being formatted for input to the **troff** text formatting system (Kernighan, Lesk, & Ossanna, 1974). There are several types of publication records in the database: books, journal articles, articles in edited books, technical reports, and so on. For each publication type, a different format is required. The references in this paper are printed in APA format (APA, 1983). Two properties of the formatting might change: the output format, or the text formatter. For example, the ACM uses a different format, and Scribe (Reid & Walker, 1980) and TEX (Knuth, 1979) are other text formatters. With my personal database system, it is a simple translation of one format to another, or of one formatter to another. Templates defining how the records (analysis) are formatted (synthesis) are simply redefined. Again stepping back for an overview, this is an example of analyzing a problem into simple parts, and synthesizing several different solutions.

Data Analysis System Interface

S is a system and language for data analysis (Becker & Chambers, 1984). While at Bell Labs, I developed a high-level user interface to the **S** language using the IFS (Vo, in press) user interface language. **S** is a large system, with over 300 functions, each with about 3-6 options. The system I built (Perlman, 1983) has a screen with a form and a menu for every **S** function; the menu controls the invocation of the function and the form allows users to supply options. There are over 100 menus arranged in a hierarchy to help users find the functions of interest. In all, there are close to 500 screens, each with menus or forms, and on-line help. In developing this system, I pushed the idea of multilingual programming to its limits, only to find out it was more powerful than I had anticipated.

It was clear to me that programming 500 screens by hand, even with a high level language like IFS, was going to present problems. User interface design is an iterative process, and if each iteration involved changing hundreds of files containing screen descriptions, then it would be impossible to make many changes. Early in the development, I decided to design a special purpose artificial language (Perlman, 1984) especially suited to designing screens in the IFS language. An artificial language is a special purpose notation for precise and concise communication within a limited domain. My goal was to be able to specify the screen designs with as little syntax as possible. In the words of Tufte (1983), I wanted to minimize the "ink to data" ratio and specify only the information that changed from screen to screen. I did not want to repeatedly specify the formatting information because it would have wasted my time and made it more difficult to maintain consistency.

Becker and Chambers had already done much of my work by designing the **S** interface language using the **m4** macro processor (Kernighan & Ritchie, 1980). The **S** interface

language defines various attributes of **S** functions and their options. The most notable are the attributes of options including: (a) **name**: the name of the option, (b) **type**: the data type of the option value, (c) **size**: the dimensions of the option's value, (d) **default**: the default value, and (e) **requirement**: whether the option was required or not. Other information, such as the allowable range of options, is coded algorithmically. Becker and Chambers write this information in a dialect of **m4** and use **m4** macro definitions to generate RATFOR code for input to a compiler. The format of **m4** macros is simple: a macro name is followed by a parenthesized list of comma-separated arguments. For example, the following is an option to the **S plot** command.

```
ARG (main, OPTIONAL, CHAR)
```

In English, the main title of the plot is an optional character vector with no default value.

Missing from the information in the **S** *interface language* is on-line help about the purpose of the functions and options. I had to add this information from the **S** documentation by hand to build the high level interface. Once this was done, all the information about the S functions was parameterized (analyzed) and centralized.

The generation of the screens is straightforward, but contains many details. For each function, in an **S** interface language source file, there is a definition of its name, purpose, etc., and the attributes of its options. This is a relational database of information about each function and their options. From these, **m4** macros are defined to parse the information so that it is available to a code generator. The code generator takes this information and extracts what it needs for different parts of the screen design. The following parts are generated: (1) **declarations**: Options are represented as variables that have to be declared. (2) **titles**: Forms and menus have prompts based on the names and short descriptions of options. (3) **help**: On-line help is extracted from the **S** manual and coordinated with the screen designs. (4) **validation**: Inputs are validated based on the datatype and range of options, and required options must be supplied before a function is allowed to run. In short, each piece of information about each function is used several times in several contexts.

The code generation can be summarized as follows. Several sources of information are integrated into one consistent database. Information from this database is parsed with **m4** and used to fill in the blanks of idiomatic templates (ie. macro definitions) in the IFS language. The same information is used more than once, for help, validation, and for declarations, but each datum comes from one source, thus ensuring consistency.

The result of using **m4** macros to design and implement the IFS/S interface was beneficial many times over. (a) **Generalization**: As an abstract notation, it allowed me to see more relationships than otherwise would have been possible. (b) **Abbreviation**: As an abbreviated language, it saved much typing (for custom fixes) and reading time. (c) **Consistency**: As a single source for the design, it allowed extremely consistent development. If a change were made in the design, that change was centralized in the templates, and only a regeneration process was necessary. (d) **Flexibility**: By localizing all the IFS specific language in the macro definitions, much flexibility was gained. During the IFS/S interface development, IFS itself was under development, and several times the IFS language changed so that the whole system was corrupted. Only the macros had to be changed to reconfigure the system, not 500 screen designs. This would have been non-trivial because the generated screen designs contained an average of 400 lines of IFS code, or a total of about 200,000 lines. The screens were so long because additions to the design were centralized with a one time cost for each addition; each effort on a screen was repaid by being multiplied by several hundred screens.

Because the IFS code was separated from the description of the functions, macros could be written to generate text in other languages. This was used to create a variety of paper documents, using the **troff** text formatter, full and quick references, each in a few hours. There was no problem of the ***accuracy*** of these documents because they were generated from the same source as the user interface, which was generated from the program code. There was no problem with the ***consistency*** of these documents, again because they were all generated with the same macros. Such standardization is especially impressive with such a large system and such detailed documents (some were about 100 pages with indexes automatically generated).

Option Parser Generator

SETOPT is a code generator that produces a parser to handle UNIX program command line options (Perlman, 1985). UNIX program options are wildly inconsistent, and the efforts of Hemenway & Armitage (1984) to define a syntax standard were accompanied by the development of SETOPT to help develop compliant programs. In addition to ensuring a consistent syntax for command line options, SETOPT deals with on-line help, type checking, input conversions, and range checking. In short, SETOPT aids all aspects of programming command line options on UNIX.

With SETOPT, each option is described with a list of attributes in a format convenient for input to **m4** (Kernighan & Ritchie, 1980), a macro processor. For example, a simple text formatting program might take options to control the width of lines, whether lines are numbered, and page headers. With SETOPT, the following could be used to specify these options.

```
OPT (w, width, Line Width, INT, 0, 72, value>0)
OPT (n, number, Line Numbering, LGL, 0, FALSE)
OPT (h, header, Page Header, STRING, 0, "",
    length(value) < optval(width))
```

This analysis of the options states that the **width** option is an **integer** of dimension **0** (a scalar) whose default value is **72** and whose minimum value is **0**. It is set with the **-w** flag, and its purpose is to set the **line width**. Note in the previous English explanation how the parameters of the **OPT** macro can be plugged into a **troff** (Kernighan, Lesk & Ossanna, 1978) template to provide detail. The same information is used by SETOPT to generate a **C** language (Kernighan & Ritchie, 1979) parser for handling all aspects of the users interface: (a) parsing the options on the command line, (b) validating options and providing standardized error messages, (c) allowing access to on-line help, (d) allowing interactive setting of options, and several other capabilities. Like the IFS/S interface, much effort can be expended because the SETOPT tool can be used with hundreds of UNIX programs.

Again, the process is the same as with the other examples. A domain of application is chosen and analyzed so that the problems in the domain are parameterized. This is the analysis stage. This information is in a database from which several solutions can be synthesized. The synthesis is done by plugging information from the database into templates in different languages: with SETOPT, **troff** macros to generate UNIX manual entries, and **C** program code to produce a user interface.

The manual entries generated by SETOPT are not complete, nor what I would call great prose. SETOPT *provides a simple scheme to insert explanatory text in different* parts of the generated document. It is difficult, but not impossible, to generate smoothly flowing text. Computer program documentation, especially that on program option attributes, does not need to read like great prose.

Electronic Survey System

Surveys for gathering information can be described with a simple grammar. In an electronic survey system (Perlman, in press), survey questions are represented as having four basic attributes: (1) **variable**: a variable that is set by answering a question, (2) **prompt**: a prompt that is presented to a respondent, (3) **help**: more detailed information, available on request, about the requirements for the answer, and (4) **type**: the type of survey question (e.g., multiple choice, rating scale, etc). Based on the question type, other parameters might also be supplied. For example, a minimum and maximum value might be supplied for a Thurstone scale question of the form:

Rate on a scale from *minimum* **to** *maximum*...

Based on these parameters, a question database is constructed, and from it, **C** program code (Kernighan & Ritchie, 1979) is generated to administer the survey. By changing the templates from which the program code is generated, **troff** text formatting commands are used instead to generate a paper survey. Work is underway to generate a form based survey system using the Rapid/USE prototyping tool (Wasserman, 1979). In summary, several different synthetic solutions to problems are formed from the same analysis.

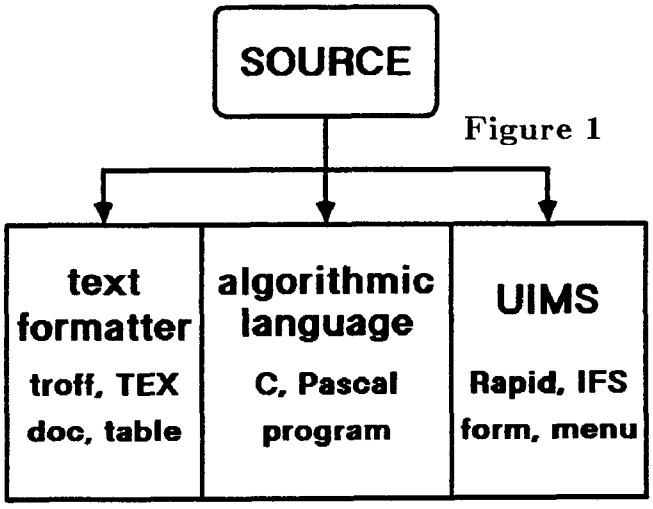

Figure 1

FORMALIZATION

Each of the previous five examples show the same process, depicted in Figure 1. First, an abstraction of a domain is used to analyze a problem. This analysis results in a source database of information representing the problem from which solutions can be constructed by synthesis. The information is plugged into idiomatic templates to generate instances in several classes of target languages: text formatters, programming languages, and user interface management systems; hence the name ***multilingual programming***. For each class of target language, there are several possible specific languages. The results of the syntheses can include program code, program comments, user interface code, on-line documentation, and off-line documentation. In this section, I will attempt to describe multilingual programming more formally.

Figure 2 is a graphical representation of the process of multilingual programming. At the top of the Figure are two shapes representing instances in a specific subject domain. An analysis of the instances shows that each has five key concepts in the domain. This pattern is formalized and the information from those five key concepts is extracted and parameterized in a relational database (depicted in the center of Figure 2) to form one source of the information. From this database,

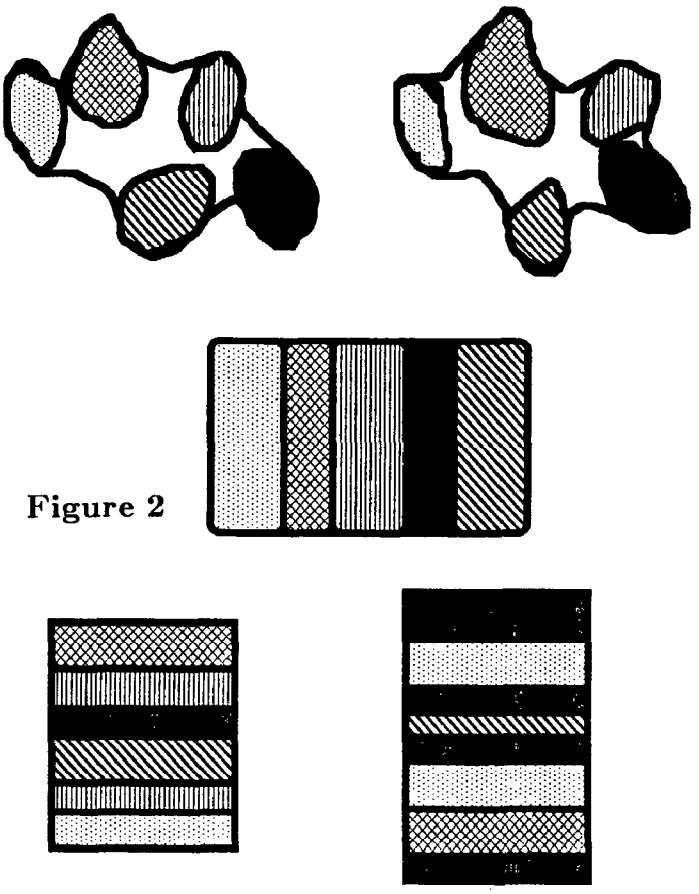

Figure 2

several different ***views*** or ***solutions*** are possible, each being a synthesis of the information in the database, shown at the bottom of Figure 2.

It is not necessary that ***all*** information in the database be used in forming a synthetic ***view***. In declaring variables in a programming language, a help string is not necessary, although it is customary to put that information in comments next to the code that is generated. The synthesis on the lower right of Figure 2 does not contain the information shaded with vertical lines.

Is is possible to use the same information (always from the same source) more than once. In generating printed documentation, it is a good idea to provide several levels of detail: (1) quick reference, (2) a table of attributes, and (3) detailed information. The same information might go into each of these, although more would go into the detailed documentation.

When real systems are being developed, these views evolve through an iterative elaboration and refinement process. Consider the development of a user interface system. The templates might begin by scavenging an existing piece of code, parameterizing some parts. A first generation user interface might not check ranges of input values. A second generation user interface might check ranges, but not provide diagnostic error messages. The flexibility of the method of multilingual programming allows developers to address unanticipated needs flexibly and gradually work toward a better system. Note that all the while, the consistency of the system is maintained by generating text based on a single database with the same templates. Change is localized in the templates, thus minimizing effort.

Abstractions

In describing Figure 2, I did not tell how one would notice that several instances share common concepts. I do not know how this can be done in general, except by experience. It was only after writing the **troff** text commands to format hundreds of references that I noticed I was wasting my time doing the same action repeatedly and that changes in format would be difficult. With experience with similar tasks, a person's performance improves, which is a hint that repeated actions can be automated.

Idiomatic Templates

A *template* is an abstraction of an idiomatic pattern of text that frequently occurs in a specific target language like a text formatting or programming language. Templates have slots where variables are inserted to form instances in the target language. For example, in the **C** programming language, a programmer might define the square root function like this:

```
/* sqrt: square root */
double sqrt (x)
double x; /* must be non-negative */
```

The documentation for **sqrt** might look like this:

```
                    TYPE    COMMENT
FUNCTION    sqrt    double  square root
ARGUMENTS
            x       double  must be non-negative
```

and could be based on some **troff** formatting macros (that would be defined elsewhere) like these:

```
.FN "sqrt" "x" "double" "square root"
.AG "x" "double" "must be non-negative"
```

The idiomatic templates for each language abstract the parts that remain constant across uses. Note that they contain the same information plugged into different, but corresponding slots.

```
C:
        /* purpose */
        type function (arguments)
        type argument; /* comment */
troff:
        .FN "function" "arguments" "type" "purpose"
        .AG "argument" "type" "comment"
```

Database of Attributes

The information from the previous example can be parameterized by analysis using a set of attributes:

```
function    = sqrt
purpose     = square root
type        = double
argument    = x
type        = double
comment     = must be non-negative
```

and put into a database with two relations, one for functions and one for arguments. This information is target-language independent, somewhat ***object oriented***, implying that a person does not need to know the syntax of *any* language to program or write documentation when programming multilingually. Information needed for code generation or documentation can be extracted and plugged into slots in templates. Language specific syntax information is held in the templates.

It can be difficult to write text, especially phrases, like the **purpose** and **comment** above, because the same information will have to *fit* into many templates. There is some virtue in the difficulty, because it forces using consistent formats (e.g., the tense and voice must agree).

Multiple Target Languages

Once information is in a database, many views of the database are possible. It is only by changing the definitions of the views, by modifying or substituting the templates, that different target languages can be generated. Each of these target languages is based on the same source of information, and so is consistent with the others.

Template Macros

It is not mandatory that macros be used to implement templates. There are several reasons why macros are preferable to more common language extensions like functions. (1) When macros are used, it does not matter if the target language has a function definition capability. Macros can be used to extend any language. (2) Macros do not need to adhere to the syntactic rules of the target language. Default values can be inserted to function calls, and variable names and values can be combined with string operations. (3) Macros are easier to write than more complex text generators like compilers. (4) General macro processors, like the **m4** macro processor (Kernighan & Ritchie, 1980) offer all or most of the capabilities needed for building templates. **m4** supports macro definition, parsing of parameters, string manipulation, condition testing, iteration through recursive macros, and arithmetic.

Macro processors like **m4** are not without their problems.

The Quoting Problem. The recursive evaluation of macros makes the quoting problem difficult to master. It can take a long time to learn how to get nested recursive macros substituted (to avoid quoting) and how to delay or stop the substitution (to use quoting).

Pretty Printing. The output from macro substitutions contain everything in the definitions of the macros. This includes any white space used to make the macro definitions more readable. So unlike most programming languages, structured macro writing style conflicts with functionality, especially for templates of text formatting languages for later human viewing. The solution seems to be to use a post-processor, a prettyprinter, to reformat the macro processor output for input to a target language processing system. Luckily, this usually involves just stripping off leading space on lines and removing blank lines.

Properties of Multilingual Programs

Generalization & Imagination

The following quote of Whitehead (1911) leads into one advantage of multilingual programming.

> ***By the aid of symbolism, we can make transitions in reasoning almost mechanically bye the eye, which otherwise would call upon higher faculties of the brain. By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems.*** (Chapter 5)

By parameterizing problems by analysis, a notation is established, and our ability to see new relationships and form new syntheses is enhanced.

Flexibility and Resilience to Change

A small change in a program, such as changing the type of a variable from an integer to a real should not require a huge effort. Most current practice requires many changes: (1) the declaration, (2) the program comment, (3) the user interface to read the variable, (4) on-line help and error messages, and (5) user manuals. It is not surprising that most of the cost of software is in maintenance and modifications to

working software. Multilingual programming provides a method for making software more flexible, allowing people to **design for redesign**.

Multilingual programming is resilient to changes of standards and software tools. Personal experience taught me this well. While working on a system written in a user interface language, the definition of the user interface language changed, leaving me with hundreds of thousands of lines of unworking code. Because I had generated the user interface language from a database, I replaced many hours of work by some minor changes to some templates.

Accuracy and Consistency

Much of the documentation and many program comments I read are inaccurate. This could be attributed to carelessness, but I think that would avoid confronting the problem. Text (comments and manuals) written about other text (program code), by hand, is going to lag behind, and updates can be forgotten. Also, text written about other text, by hand, can be inaccurate because people make different inferences from the same information. The method of multilingual programming promotes accuracy by automating the updates and removing chances for misinterpretation.

Once a document (user interface) exists, it meets or sets a standard format for related documents (software). The format of related documents (user interfaces) should be consistent so that people can learn based on their experience, not in spite of it.

Economy of Expression

Finally, multilingual programming supports abbreviation. Information in a database is about as abbreviated as possible, this information is ***crossed***, in the set theoretical sense, with templates for each language, thereby multiplying productivity.

DISCUSSION

Choosing the Appropriate Focus

Hester, Parnas, & Utter (1981) suggest that documentation of systems should precede any development, and others have suggested that user interfaces should be designed first. The motivation for writing documentation first is to write correct code efficiently, and the motivation for writing user interface specifications first is to ensure that programs are easy to use. These are good motivations, but show how good ideas can compete for attention. The solution is to work on both problems at the same time by analyzing the problem so that documentation, user interfaces, code, and so on, are treated as equally important parts of software products that require coordination.

There are problems with choosing a target language, documentation, programming, user interface, or whatever, as the source of information for other target languages. For example, writing documentation from program code is error prone and expensive. When target languages are used as source databases, they are almost always strained to accommodate the other languages. For example, the writing style tools of the Writer's Workbench (Frase, 1983; Macdonald, 1983) use **troff** text formatting macros as a text structuring language and try to infer structure based on formatting instructions. This is the opposite of the desired process, that format should reflect content. Much of the time, this works well, especially if writers use a high level set of macros developed at Bell Labs, but sometimes writers find themselves trying to fool the analysis tools.

It does not make sense to put one part of a programming system over another. Neither a good program with poor documentation nor a bad program with good documentation are acceptable. The implementation of programs, the development of user interfaces, the writing of documentation, all must be coordinated.

Multiple Views of Programs

Knuth (1982) developed the WEB system that combines program code with structured program comments so that both can be extracted for input to his TEX formatter (Knuth, 1979) or just the program code can be extracted for the compiler. It is a system for printing beautiful program listings with minimum programmer effort. While this process is similar to the one described here, the WEB system does not use analysis of problem domains to the same extent, nor does it allow for the use of parameterized information for domains outside programming, like documentation and user interfaces.

Natural Language

Natural language systems such as those of Schank (1979) are able to generate paraphrases of their inputs in several languages. Although this is an impressive feat, the hard part, according to Schank, is to understand the original input and represent that information in a data structure. Once that is done, the generation of paraphrases works on the same principle as in this paper. In the examples described in this paper, the problems of parsing the input is trivial compared to those faced by cognitive scientists.

Cost/Benefit Analysis

In this final section, I try to answer the question *When does multilingual programming pay off?* Multilingual programming requires planning on a larger scale than is customary. To implement that plan, there is the overhead of learning about generating templates. To offset that cost, there have to be benefits. Multilingual programming is especially suited to large projects or ones where a consistent solution is desired. Suppose that in a domain we have **D** documents (bottom, Figure 2) like program text, manuals, etc., that contain a total of **A** attributes (middle, Figure 2) to describe **P** problems (top, Figure 2). If any of these is large, then multilingual programming is more economic, but for different reasons. The total number of solutions generated is **P*D**, each of which has a size roughly proportional to **A**, making the size of a multilingual solution proportional to **P*D*A**. The cost of this is **D** times the cost of developing templates for each document type, plus **P** times the cost of describing the attributes of each problem. If **P** or **D** is small, then multilingual programming may not be worth the trouble of learning and enforcing the method. If **P**, the number of problems, is large, then multilingual programming aids flexibility for change and abbreviation. If **A**, the number of attributes, is large, then multilingual programming aids possibilities for generalization, flexibility for change, and accuracy. If **D**, the number of documents, is large, then we aid the accuracy of the documents, and help reduce human effort by abbreviation.

REFERENCES

APA, (1983) *APA Publication Manual.* (3rd Edition). Washington, DC: American Psychological Association.

Becker, R. A., & Chambers, J. M. (1984) Design of the S System for Data Analysis. *Communications of the Association for Computing Machinery*, **27:5**, 486-495.

Dixon, W. J. (1975) *BMD-P Biomedical Computer Programs.* Berkeley, CA: University of California Press.

Frase, L. T. (1983) The UNIX Writer's Workbench Software: Philosophy. *Bell System Technical Journal*, **62**, 1883-1890.

Hemenway, K., & Armitage, H. (1984) Proposed Syntax Standard for UNIX System Commands. In *Proceedings of the 1984 Summer Usenix Conference, Washington, DC.* El Cerito, CA: Usenix Association.

Hester, S. D., Parnas, D. L., & Utter, D. F. (1981) Using Documentation as a Software Design Medium. *Bell System Technical Journal*, **60:8**, 1941-1977.

Kernighan, B. W., Lesk, M. E., & Ossanna, J. F. (1978) Document Preparation. *Bell System Technical Journal*, **57:6.2**, 2115-2136.

Kernighan, B. W., & Ritchie, D. M. (1979) *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall.

Kernighan, B. W., & Ritchie, D. M. (1980) The m4 Macro Processor. Murray Hill, NJ: Bell Laboratories.

Knuth, D. E. (1979) *TEX and METAFONT: New Directions in Typesetting.* Bedford, MA: Digital Press.

Knuth, D. E. (1982) Literate Programming. Stanford, California: Stanford University. (Report No. STAN-CS-82-981.)

Macdonald, A. H. (1983) The UNIX Writer's Workbench Software: Rationale and Design. *Bell System Technical Journal*, **62**, 1991-2008.

Nie, H. H., Jenkins, J. G., Steinbrenner, K., & Bent, D. H. (1975) *SPSS: Statistical Package for the Social Sciences.* New York: McGraw-Hill.

Perlman, G. (1980) Data Analysis Programs for the UNIX Operating System. *Behavior Research Methods & Instrumentation*, **12:5**, 554-558.

Perlman, G. (1982) Data Analysis in the UNIX Environment: Techniques for Automated Experimental Design Specification. In K. W. Heiner, R. S. Sacher, & J. W. Wilkinson (Eds.), *Computer Science and Statistics: Proceedings of the 14th Symposium on the Interface.*

Perlman, G. (1983) The Interface Arsenal: Software Tools for User-Program Interface Development. In *Proceedings of the 1983 Summer Usenix Conference, Dallas, TX.* El Cerito, CA: Usenix Association.

Perlman, G. (1983) The Design of an Interface to a Statistical System. Murray Hill, NJ: Bell Laboratories.

Perlman, G. (1984) Natural Artificial Languages: Low Level Processes. *International Journal of Man-Machine Studies*, **20**, 373-419.

Perlman, G. (1985) An Overview of the SETOPT Command Line Option Parser Generator. In *Proceedings of the 1985 Winter Usenix Conference.* El Cerito, CA: Usenix Association.

Perlman, G. (in press) Electronic Surveys. *Behavior Research Methods, Instruments, & Computers.*

Reid, B. K., & Walker, J. H. (1980) *Scribe: Introductory User's Manual.* Pittsburgh, PA: Unilogic.

Ritchie, D. M., & Thompson, K. (1974) The UNIX Time-Sharing System. *Communications of the Association for Computing Machinery*, **17:7**, 365-375.

Schank, R. (1979) Presentation at the Second Annual Cognitive Science Society Meeting.

Tufte, E. R. (1983) *The Visual Display of Quantitative Information.* Cheshire, Connecticut: Graphics Press.

Vo, K. P. (in press) Integration, Interaction: The IFS Approach. *AT&T Bell Laboratories Technical Journal.*

Wasserman, A. I. (1979) USE: A Methodology for the Design and Development of Interactive Information Systems. In H. J. Schneider (Ed.), *Formal Models and Practical Tools for Information System Design.* Amsterdam, The Netherlands: North-Holland. pp. 31-50.

Whitehead, A. N. (1911) *An Introduction to Mathematics.* London: Oxford University Press.

Keywords:
Automatic Program Generation
Automatic Documentation
User Interface
Language Design