Systems for systems implementors—Some experiences from Bliss*

by WILLIAM A. WULF

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

The programming language Bliss was developed at Carnegie-Mellon University expressly for the purpose of writing software systems* and has been in use for over three years. A considerable number of systems have been written using it: compilers, interpreters, i/o systems, simulators, operating systems, etc. The language was designed and implemented in the conventional sense of an isolated language system, and relies on the file system, editors, debuggers, etc., provided by the manufacturer and/or other users. In this paper we shall not describe Bliss, that has been done elsewhere;[1,2] nor shall we attempt to justify the language design, that has also been done.[3,4] Rather, we shall attempt to analyze and evaluate the particular decision** to implement Bliss as an isolated language rather than as a piece of a more comprehensive system. Some comments are made on the implications of this analysis/evaluation on the shape that such a system might have.

A CHARACTERIZATION OF THE PROBLEM AREA

We shall restrict our discussion to the field of "systems programming." While there is no universally accepted definition of this term, it is useful to have some characterization of it against which to frame the discussion. In particular we can discern four properties of systems programs relevant to this discussion. They:

1. must be efficient on a particular machine;
2. are large, probably requiring several implementors;
3. are "real" in the sense that they are widely distributed and used frequently (perhaps continuously);
4. are rarely "finished," but rather are elements in a design/implementation feedback cycle.

These properties may be factored into two sets—technical issues (item 1), and program management issues, i.e., those that arise exclusively because the systems are large, real, and volatile (items 2, 3, and 4).

The technical issues relate primarily to efficiency of two types: local and global. In most cases software systems can at most tolerate moderate inefficiencies in their object code; in a few critical situations anything other than the most efficient possible machine code is unacceptable. Although the issue of efficiency is largely language/compiler related, it must be recognized that a more general statement applies: given the *logical* machine which a software implementation system (SIS) defines, and any discrepancy between that model and the hardware itself, that discrepancy may become critical in one of two ways: either the cumulative inefficiency due to the distributed effects of the discrepancy is significant, or the use of the construct which invokes the discrepancy produces intolerable inefficiencies in some local context. Although we cannot hope to design an SIS which eliminates the effect entirely, we must take some care with the conventions we adopt, both in the SIS itself and in the management tools given the user.

The managerial issues which arise in the construction of large, complex systems can be separated into two

* This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office for Scientific Research.

* Primarily for the PDP-10 although Bliss has now been implemented for several other machines.

** At the time, of course, the decision was made by default; the more ambitious alternative was not considered.

classes:

1. those which we presently believe to be solvable within the framework of a static, compilable language; and
2. those whose solution, at present, seems to require the construction of a "total" programming environment which, in addition to the language, includes editing, monitoring, etc.

Within these classes, the tools for program management come in three forms: those which help specify a global structure to a task (top down modularization), those which specify common elements of a fine structure (predictive bottom up modularization), and finally those related to the relatively mechanical aspects of file manipulation, editing, debugging, etc.

A VIEW OF THE PROBLEM

Most of the recent effort devoted to the design of languages and systems has been expended to improve the convenience with which a program may be written. While convenience is an important criterion, it should not be the only, or even the central, issue in the design of a system for implementing other systems. The notion that convenience in writing programs should be the central issue results from the naive view that software is simply designed and written. That view is fallacious in terms of the four properties listed above.

In particular, programming systems are never finished but are in a constant state of evolution. New features are added and old errors repaired. The more heavily a system is used, the more rapid the rate of evolution and repair. This situation seems inevitable so long as new application areas, all with slightly different requirements, continue to emerge. Thus, the central problem of devising a system for systems programming would appear to be that of providing mechanisms for enabling the programmer to cope with this evolution while satisfying technical constraints imposed by systems implementation in general.

The mechanisms by which programmers may cope with the evolution of a system are those which we have termed 'managerial' above. It is these mechanisms which are most prominently lacking in our current system implementation tools; the consequence of this lack is the introduction of peripheral modifications which subvert and distort the original structure of a system and lead to inefficient, "dirty" systems.

WHAT IS A "GOOD" MANAGEMENT TOOL?

If the central problem of systems programming is that of coping with the evolutionary nature of systems, then a good tool is one which creates an environment in which this is relatively easier to do. Moreover, given an existing system and the desire to modify it in some way, the difficulty of making that modification is directly related to the extent of its interaction with what already exists. Modifications whose effects are localized are easy to make. Modifications whose effects are global—whether due to a large number of textual or conceptual interactions—are difficult to make.

In general the "goodness" of a tool appears, then, to be directly related to the degree to which its use permits and encourages decoupling, isolating, decisions and hence localizing their effect. Thus, to pick two trite examples, subroutines and macros are good tools precisely because they permit isolation of a computational representation (a particular encoding) from the intended effect of that computation.

MANAGEMENT FUNCTIONS PROVIDED BY LANGUAGE

One view of the recent history of the development of programming systems holds that it has been a search for panaceas. According to this view the development of large 'shell' languages (e.g., PL/I), extensible languages, time-sharing, etc., have each in turn been sponsored, in part, because they promised to be *the* solution to providing more convenient, accessible, and cost/effective computing. Whether this view has complete validity or not, we do not want to fall into the trap of looking to the mystic word 'system' to remedy the ills of past software development projects. Therefore we will first discuss some of the management facilities which can and should be provided at the language level.

The decision to use any higher-level language represents a good program management decision to the extent that the structuring facilities of the language are used in the implementation. It represents a sound technical decision to the extent that they are usable. For example, Bliss chose to include Algol block-structure, scope and extent of variables, functions, boolean and arithmetic infix operators (with precedence rules), and many of the elements of the Algol control structure (with *goto* specifically excluded). These were chosen as representatives of good management tools from the realm of general purpose languages. The PDP-10 hardware model accepts these constructs with very little overhead which makes them sound technical tools as well. The remainder of Bliss is composed of operators, control structures, data structures, etc., which although not entirely unique, are somewhat different from those in other languages because: (1) of the structure of the PDP-10, (2) of the efficiency

problems imposed by implementation languages in general (that is, a concerted effort was made to minimize the discrepancy between the logical Bliss machine and the physical PDP-10), and (3) no suitable models for certain management tools could be found in existing languages.

As stated in the introduction this paper is not intended to be a definitive description of Bliss. However, two aspects of Bliss related to management issues are discussed below to illustrate how these may manifest themselves in a language design:

(1) Control Structures: Other than subroutines and co-routines, the control structures of Bliss are a consequence of the decision to eliminate the *goto* (see References 4, 5, 6 for a discussion of the reasons behind this decision). In Reference 4 the author analyzes the forms of control flow which are not easily realized in a simple goto-less language and uses this analysis to motivate the facilities in Bliss. Here we shall merely list some of the results of that analysis as they manifest themselves in Bliss.
   (a) A collection of 'conventional' control structures: Many of the inconveniences of a simple goto-less language are eliminated by simply providing a fairly large collection of more-or-less 'conventional' control structures. In particular, for example, Bliss includes: conditionals (both *if-then-else* and *case* forms), several looping constructs (including *while-do*, *do-while*, and stepping forms), potentially recursive procedures, and co-routines. While anything in addition to the *goto* and a conditional branch may be considered "syntactic sugar" in most languages, these additional forms are essential to convenient programming in Bliss (although they are not all theoretically needed for completeness, see Reference 6).
   (b) Expression Language: Every construct in Bliss, including those which manifest explicit control, are expressions and have defined values. There are no 'statements' in the sense of Algol or PL/I. It may be shown[6] that one mechanism for expressing algorithms in goto-less form is through the introduction of at least one additional variable. The value of this variable serves to encode the state of the computation and direct subsequent flow. This is a common programming practice used even in languages in which the *goto* is present (e.g., the FORTRAN 'computed *goto*'). The expression character of Bliss is relevant in that the value of an expression is a convenient implicit carrier of this state information.
   (c) Escape Mechanism: Analysis of real programs strongly suggests that one of the most common 'good' uses of a *goto* is to prematurely terminate execution of a control environment—for example, to exit from the middle of a loop before the usual termination condition is satisfied. To accommodate this form of control, Bliss allows any expression (control environment) to be labeled; an expression of the form "*leave* ⟨label⟩ *with* ⟨expression⟩" may be executed within the scope of this labeled environment. When a *leave* expression is executed two things happen: (1) control immediately passes to the end of the control environment (expression) named in the *leave*, and (2) the value of the named environment is set to that of the ⟨expression⟩ following the *with*.

(2) Functional Decomposition: An effective program management technique is to insist on functional decomposition and isolation of tasks. Technical issues suggest several alternatives for constructs all of which can be considered "function like": full-blown Algol functions (with display mechanism), Bliss "routine" (without display mechanism), co-routines, macros and the (Bliss) data structure mechanism.
   (a) Functions and routines are defined and called in Bliss in a manner similar to that in Algol, except that there are no specifications and all parameters are implicitly call-by-value. Functions and routines are examples of choosing well-known and admired managerial tools and adapting them to satisfy the technical requirements of a system implementation language.
   (b) Co-routines are often used (unwittingly) by programmers in any language; their essential nature is that they preserve some sort of "status" information upon exit and continue execution upon recall based on that status. If the status becomes arbitrarily complex, the only way to retain it is to remember essentially everything which pertains to the Bliss model in the machine for a running program, i.e., the stack, declarable registers, and program counter. Such information is best dealt with by the compiler (i.e., a minor implementation change might have drastic effects if everyone using co-routines of this complexity were saving

status information differently); thus the construct was included in the language.

(c) The Bliss structure mechanism allows the user to define an accessing algorithm—that is, the algorithm to be used to obtain the address of an item in the structure. In fact, there are no "built-in" data structures; the user must define the representation of every data structure by supplying an accessing algorithm for it. Once an accessing algorithm has been defined, it may be associated with a variable name and will be automatically invoked when that name is referenced. Thus, the user may choose the most appropriate (efficient) representation and may change the representation as the use of the data structure evolves.

With 20/20 hindsight it is obvious that it is the managerial issues, and not the technical ones which are the most costly, provide the most compelling reasons for adopting an implementation system, and hence are the primary ones to which such a system must respond. Moreover, a language can only make technical responses to these issues, and cannot respond to the entire spectrum of managerial issues. Conversely there are a set of issues to which the most appropriate response is at the language level.

The technical responses made in Bliss, such as the structure mechanism and removing the *goto*, are, for example, both good and made at the appropriate level. We consider the Bliss structure mechanism, for example, to be a "good" management tool because it decouples those decisions concerning the representation of a data structure from those decisions concerning the manipulation of the information contained in the structure. (In this context we consider the data structuring mechanisms of most languages to be "bad" management tools in that the representation decisions are made at a totally inappropriate time—namely, when the language is implemented.)

MANAGEMENT FUNCTIONS PROVIDED BY A 'TOTAL' SYSTEM

We wish to distinguish between two notions which the term 'system' might connote; for want of better terminology we shall refer to them as *internal* and *external*. By *internal* we mean those facilities which must be provided by a system coextant with that written by the programmer. Conversely, by *external* we mean those facilities which are never coextant with the user's program. Dynamic storage management and virtual memory systems are examples of the internal variety; editors, loaders, and linkage editors are generally of the external variety. Of course, there are numerous examples—the TSS dynamic loader, for instance—which cross this boundary.

External facilities

In some ways these appear to be the most mundane of those facilities which might be provided by a 'total' system. Editors, file systems, loaders, etc., are familiar to us all and that familiarity is indeed likely to breed a certain level of contempt—or at least a strong temptation to "make do" with the facilities that happen to be available.

However, measured against the definition of a 'good' management tool given above, most of the editors, etc., with which we are familiar are inadequate. Moreover, they are unlikely to become adequate unless invested with more specific knowledge of the structure of the items with which they deal. In particular, the notion of decoupling decisions carries the collateral notion of distributed definition and use (related definitions are grouped rather than related uses). Present editors, for example, simply do not cope with such structures—particularly if definition and use are in separate files. Fortunately, there is an excellent extant example of a system with many of these properties designed by Englebart, et al.[7]

Internal facilities

The class of facilities we have called 'internal'—those which require coextant support—are certainly more glamorous than the external ones. Our experience using Bliss strongly suggests that some of these mechanisms would be very valuable, in particular: incremental compilation, debugging at the source level (as with conversational languages), execution of incomplete programs, virtual memory, etc.

All these mechanisms represent 'good' management tools, when described at this level, in that they permit certain classes of decisions to be decoupled. The ability to execute incomplete programs, for example, is an attractive facility for permitting parallel construction of systems by several implementors.

Unfortunately there are no extant examples of systems which provide wholly 'good' tools from either the technical or managerial standpoint; the existing systems fail for two reasons:

1. They are inefficient in specific cases. To date these systems have generally been interpretive;

while a technical solution to this exists,[8] it is not clear that the residual distributed effect of this flexibility can be totally eliminated.

2. They imply binding certain decisions at a very early stage, namely, when the supportive system is written. This is by far the more serious problem. Internal facilities are efficacious to the extent to which they can presume a particular structure in the system they support. (While this is also true of external facilities, in the latter case the assumptions are purely formal.) These presumptions are inviolate and their presence clashes with our definition of a good management tool.

The position taken by Bliss with respect to internal system facilities is the extreme one, and the original rationale for it is probably fallacious: the code produced by the Bliss compiler requires *no* run time support. The rationale for this position was that some systems would be written in Bliss could not presume such support—notably the lowest levels of an operating system. While this is indeed true, the fallacy is that the majority of programs written in Bliss have not been operating systems, nor will they be.

It is possible, of course, to write one's own support in Bliss, and a fair variety of these packages have been written—one of which is worth special mention.

The "timing package"[9] is a set of Bliss routines which may be loaded with any Bliss program. Using its knowledge of the run-time structure of Bliss programs the package can intercept control at "interesting" points, notably routine entry/exit, and record various information. In particular, the usual information gathered is the frequency and duration of routine executions and the memory reference pattern.

The timing package is a "good" management tool in the sense of the definition above in that it permits postponing (a programmer's) concern over specific local efficiency until there is evidence that local efficiency has global significance. Moreover, the timing package is a good technical tool in that its presence is not presumed and there is no distributed (or local) inefficiency implied by its potential use. Most of the systems written in Bliss have been "tuned" using this facility and the results are much as one would expect: a very small portion (less than 5 percent) of a program usually accounts for most of its execution time, the programmer is usually surprised by which portion of the program is taking the most time, and improvements by a factor of two in execution speed by relatively simple modifications are common.

The example of the timing package points out both an important distinction and a language requirement not discussed previously. The distinction is between those supportative facilities whose presence and form is requisite and presumed, and those facilities which, if available, may be exploited. The language requirement is that the link to these (optional) facilities should be 'natural.' Thus, for example, we consider dynamic storage management to be an inappropriate *presumed* facility, at least in the context of a SIS, because:

1. it undoubtedly implies a distributed overhead which is intolerable in specific cases,
2. it implies binding a decision, namely a particular storage discipline, which will be inappropriate in specific cases.

Yet, the ability to define and use a dynamic storage management system, and to do so 'naturally' in the language, is totally appropriate.

SUMMARY

The results of our experiences in using Bliss for over three years for a number of large software projects reinforces our view that the major problems of software development are what we have termed 'managerial' in nature, and not technical. The use of any higher-level language can alleviate certain of these problems, a careful language design can alleviate more and there are certain features which must be provided at the language level, but there are limitations to what can be done in any statically compilable language.

Many, if not all, of the issues which cannot be addressed by a compilable language may be addressed by a comprehensive system of which a language is only one part. Some of the facilities of such a system would deal primarily with various external representations of a program. Although these facilities need to be carefully integrated, they would presumably be related to familiar facilities. Moreover, such facilities are relatively 'safe' in that they deal primarily with the formal (syntactic) aspects of a program. We regret not having paid more attention to these facilities at an earlier stage of the Bliss effort.

Another class of facilities which might be provided by such a system relate primarily to internal representations of the program and must coexist with this representation. This class is at once more glamorous, potentially more useful, and more dangerous. The utility of such facilities is directly related to their specific knowledge of the internal structure of a program. To the extent to which the presence of such facilities forces a specific set of representations, whether of data or computation, they can magnify the problems they were meant to solve.

REFERENCES

1 W WULF et al
Bliss reference manual
Computer Science Department Report Carnegie-Mellon University Pittsburgh Pennsylvania 1970

2 W WULF D RUSSELL A HABERMANN
Bliss; A language for systems programming
Communications of the ACM 14 12 December 1971

3 W WULF et al
Reflections on a systems programming language
SIGPLAN Symposium on System Implementation Languages Purdue University October 1971

4 W WULF
Programming without the Goto
Proceedings of the IFIP Congress 1971

5 E DIJKSTRA
GOTO statement considered harmful
Communications of the ACM (letter to the editor) 11 3 March 1968

6 W WULF
A case against the Goto
Proceedings of the ACM National Conference 1972

7 D ENGLEBART W ENGLISH
A research center for augmenting human intellect
FJCC 1968

8 J MITCHELL
The design and construction of flexible and efficient interactive programming systems
PhD Thesis Carnegie-Mellon University 1970

9 J NEWCOMER
Private communication