SLAC-PUB-2806
August 1981
(M)

THE THRILL OF PROGRAMMING

THE AGONY OF DEBUGGING*

John R. Ehrman
SLAC Computing Services (Mail Bin 97)
Stanford Linear Accelerator Center
P. O. Box 4349
Stanford, California 94305

Hank Hamilton
Standard Oil of California
320 Market Street, Room 630
P. O. Box 3069
San Francisco, California 94119

Abstract:

All programs contain implicit assumptions about what we expect will be proper program behavior. Today's computing systems rarely provide facilities for verifying these assumptions. We propose that a variety of mechanisms be provided by the basic architecture of a computer's hardware and software systems that will facilitate writing programs whose assumptions can then be verified without the need for additional "assertions" beyond those already contained in the program.

Presented at Session A600 of SHARE 57, Chicago, Illinois, August 23-28, 1981.

---

* Work supported by the Department of Energy under Contract Number DE-AC03-76SF00515.

CONTENTS

FIGURES

This paper illustrates a viewpoint, not a list of specific suggestions for implementation. There are many examples, and they bear minimal relation to one another except for trying to show an idea in many different lights. We don't expect all of the examples to strike a resonant chord in your mind, but hope that a few of them will be close enough to your experiences to show you what we're getting at.

INTRODUCTION

The interface between human beings and computers makes programming and debugging on contemporary machines far more difficult than it should be. This is due mainly to constraints and oversights in system design, and not entirely to poor programming techniques or the weaknesses of programming languages. Whether implicitly or explicitly declared, much useful and necessary information is either not made available to, or is thrown away by, current machines and programming systems. By keeping and checking this information at both the hardware and software levels, many common problems can either be solved at the language level where they belong, or they can be eliminated entirely.

Past system design choices have leaned toward minimizing hardware costs at the expense of programming costs, in the belief that hardware speed was more important than programming convenience. This has had two results: first, the cost of system software has risen astronomically; and second, the end user of most computer systems -- the applications programmer -- has seen little progress in the kinds of work he can do with his skills. This is costly now, and will be even more so in the future. Because hardware costs will continue to decrease, it is imperative that new systems provide much greater support to the programmer than in the past. The costs of poor design, as reflected in the effort that must be spent in debugging, are passed on to the user by the supplier of the computing system. Thus, ease of debugging is of great concern to a computer user even if he does no programming at all, because he eventually pays for every system failure.

Because debugging has been so difficult on most machines, progress in the programming profession has been retarded because many programmers must stuff their heads with arcane tricks and trivia. It is wasteful for a programmer using what we call a "higher-level" language to have to know anything at all about machine language or control blocks, or to have to debug from a memory dump. It has been difficult to develop high-level languages with nontrivial expressive power, because simple flaws in a program produce diagnostic information that is unrelated to the form or intent of the original language. There has therefore been a tendency to divert attention from the real causes of programming

difficulties with exhortations such as "Minimize Bugs" or "the manual says you shouldn't do that".

Behavioral Assumptions

It is clear that all programs assume various consistencies and regularities of behavior on the part of the computing system. For example, the erroneous use of an uninitialized variable is due to a violation of the hardware's (untested) assumption that all variables have been given values by the program. We will see in the examples below that there are many such implicit assumptions of behavioral correctness in all programs, and that few attempts are made to verify these assumptions. It is important to observe the difference between assumptions of logical and behavioral correctness: the former is the basis of a difficult and important field of research, while the latter has been almost entirely overlooked.

The underlying viewpoint here is twofold. First, any computing system which does not provide an option for full checking of every step of a computation is careless, and therefore potentially dangerous to use. If the user chooses to trade some safety checks for faster performance, he should be allowed to make that choice and take his chances. Second, even though many inconsistencies can be detected by a translator, certain kinds of errors cannot be detected at compile time but will only be discovered through rigorous checking at run time. This is particularly true in languages allowing mixing or dynamic assignment of types.

Thus, the system should be able to verify at each step, no matter how large or small, that the expected progress of the program is indeed identical to its actual progress, and if it is not, how and why it is not. It is desirable that these checks can be performed whenever they seem to be justified: they may be for purposes of system security, data integrity, debugging, caution, or whatever else the programmer requires.

This idea of providing behavioral data to the programmer has occurred in other forms also. For example, with respect to the optimization of programs, Knuth is "convinced that all compilers written from now on should be designed to provide all programmers with feedback indicating what parts of their programs are costing the most; indeed, this feedback should be supplied automatically unless it is specifically turned off" (Computing Surveys 6, 268 (1974)).

We will begin by looking at some of the problems caused by a lack of such caution, and show how some of the problems could be avoided by applying the concept of "keeping information available"; in some places this kept information will be called "attributes". We will use a set of examples for a computer (hard-

ware and software) system which progress from very simple forms or uses of attributes to very complex ones. The names used for these forms, and the distinctions among them, are for convenience only: our purpose is to illuminate some classes of problems, and we have chosen this structure and terminology for only its illustrative uses.

ATTRIBUTES

At each level of complexity in a computing system, there are attributes implicitly or explicitly assigned to whatever computational component is of interest at that level. In most cases, an attribute can be thought of simply as an explicit behavioral assumption, or a declaration of intent: the object described by the attributes is intended to be used, or should behave, in a specific way. Most hardware and software systems now available do not retain information about these attributes, nor do they often test attributes that may have been provided.

Cells

The most basic component of storage on a computer could be called a cell: this might be a word, a bit, a byte, a register, or whatever basic unit of storage is most natural for the program being written or the problem being solved.

The most important attribute of a cell is simply whether or not it contains something. The contents of a cell can be an instruction or a datum, or its contents may be undefined. In most current systems, no provision is made for indicating that the contents of a cell is undefined: its contents may be brought from memory by a data fetch or by an instruction fetch, with no inquiry about whether or not a valid datum or instruction was previously placed into the cell. Similarly, a program might attempt to use the contents of an uninitialized register for arithmetic or addressing.

The results of fetching the contents of a memory cell whose contents are undefined are familiar. In the case of a data fetch, we have an "undefined-variable" problem. The following program will print senseless results:

```
C FORTRAN EXAMPLE
      A = B * C + D
      WRITE (6,'4E15.6') A,B,C,D
      STOP
      END
```

Figure 1: Use of Undefined Variables

In the case of an instruction fetch, we have branched into an uninitialized area of memory or to a "missing subroutine" (or have forgotten to initialize an instruction to be planted into the instruction stream while the program was running).

System support tools for detecting such errors are meager indeed. The original IBM 7040 implementation of WATFOR used the 7040's ability to initialize storage with bad parity to set an "undefined" flag in data areas. In the System/370 series, assorted simple-minded checks for data and opcode validity will occasionally trap such fetches from undefined cells. It is clear, however, that the 370's opcode checking is a rather feeble way to protect the programmer from his (or the system's, or a compiler's) errors. Programmers of machines in which all bit patterns represent valid opcodes can testify to the difficulty of debugging a program in which control is "lost" -- it usually stops only by performing an I/O select on a non-existent device.

The defined/undefined attribute of a cell, like a protect key, also has application to security problems: a program would not be able to read the "old" contents of memory if all unused cells had been initially flagged by the supervisor as "undefined". For example, if the appropriate System Generation parameter is selected, the System/370 OS/VS supervisor will zero all pages acquired via a GETMAIN so as to prevent programs from prying into the previous contents of an area of real memory. If attribute flags were available, the supervisor could simply flag all cells in the acquired area as "undefined". Indeed, dynamic assignment of such an attribute would be useful in other ways: a program might flag a variable as undefined so it could subsequently trap unexpected or invalid access.

Another important, but less well defined, attribute of a cell is the set of programs which may access it, and with what intention. For example, certain cells containing information such as the date should be readable by any program, but writeable only by the supervisor. Similarly, some routines (such as I/O interface routines) can be executed by any program, while certain others

are accessible only to a limited set of programs. (We are all familiar with programs which have branched into some random area of the Supervisor or system nucleus and have then stopped in some strange way.)

Cell Contents

At the next higher level of complexity, we are interested in the attributes of the contents of a cell. The contents of a cell will customarily be either an instruction or a datum. The failure to distinguish between these two types of contents results in another well-known class of programming errors: an instruction fetch of a datum is a "branch into the data", and a data fetch from a cell containing an instruction is a "using instructions as data" problem. The two following examples illustrate how these could happen; while nobody is likely to write such code, it is easy to hide what is happening under layers of what seems to be "correct" code.

```
C FORTRAN EXAMPLE
      CALL COMMON
C  (WHERE WE ASSUME "COMMON" IS
C  THE NAME OF A COMMON BLOCK)
      STOP
      END
```

Figure 2: Using Data as Instructions

```
C FORTRAN EXAMPLE
      REAL A(10) / 10 * 1.0 /
      WRITE (6,'E15.6') A(100)
      STOP
      END
```

Figure 3: Using Junk as Data

In common higher-level languages such as Fortran, these errors can be caused by running off the end of an array, treating scalar data as an array, inconsistent declarations among routines, etc. While some languages make software provisions for guarding against the simplest causes of such errors (for example, the SUBSCRIPTRANGE condition in PL/I, and the automatic subscript checking provided by WATFIV), there is relatively little help provided by the hardware in most machines. Some of the Burroughs machines (such as the B5500) and the IBM System/38 are exceptions to this rule.

The attributes of a cell's contents can be as simple as a differentiation between instructions and data, or far more detailed (for example, what does X'4040405C' represent?). Data can have attributes such as integer, decimal, character, etc., while code can have attributes such as problem-program instructions or system instructions. (The latter distinction would help in detecting problems such as branching into the operating system nucleus.) It would even be useful (on a machine having variable-length instructions, like System/370) to assign attributes like "start of instruction" and "middle of instruction"!

It is also helpful to be able to identify larger structures as being active (e.g., instructions) or passive (e.g., data), since these identifications can change with time and with levels of control. For example, a program's instruction stream is data to a compiler or loader, but is a set of instructions to whatever processor will later interpret them. Distinguishing between instructions and data also makes it simpler to construct re-entrant code (or "pure" procedures).

Dynamic assignment of attributes is also needed for the contents of cells. For example, the output of a conversion routine is constructed from objects of various types (characters, numeric constants, etc.), and is eventually stored with a specific type.

A more difficult problem arises when data is read from external devices. Either types must be assigned by the read routine (which may lead to errors if the input data does not match its assigned type), or the attributes of external data must be carried <u>with</u> the data on the external medium. This would allow for immediate checking of type conflicts before the data is used, preventing the many kinds of errors that occur when data is converted between external and internal forms.

```
/* PL/I EXAMPLE */
DCL 1 DATA, 2 NAME CHAR(40),
            2 SALARY FIXED DEC(11) ;
/* ... COMPUTE STRUCTURE'S VALUES */
   WRITE FILE (PAYFILE) FROM (DATA);
/* ETC. */
     /* NOW, IN ANOTHER PROCEDURE: */
DCL 1 INFO, 2 NAME CHAR (35),
    2 SS_NUM FIXED(9), SALARY FIXED(11);
   READ FILE (PAYFILE) INTO (INFO);
```

Figure 4: External Data Characteristics

Operations

The next higher level of complexity appears when the contents of a cell is being used by an elementary operation on the machine. For example, the contents of a cell may have been retrieved by a data fetch, and it may indeed be data. However, there may be a data type implied by the instruction (such as floating point) that is in conflict with the type of the datum (such as integer). If the attributes of the contents of a cell are known, it then becomes possible to detect conflicts between the desired and actual types of information fetched from a cell.

```
*  ASSEMBLER EXAMPLE
        LE    2,NUMBER
        ME    2,NUMBER
        - - -
NUMBER DC     F'10.0'
```

Figure 5: Invalid Data/Operation Combination

This behavioral assumption is easy to violate. Data from an external medium may be misused in many ways: binary data might be treated as characters, or vice versa.

```
/* PL/I EXAMPLE */
  DCL N FIXED BIN(31);
/* WE "KNOW" ABS(N) < 99999 */
  READ FILE (F) INTO (N);
/* VALUE READ IS X'41500000' */
```

Figure 6: Invalid Data Type in I/O

A Fortran programmer who uses an EQUIVALENCE statement to describe the same cell as containing either an integer or a real datum can make the mistake of using a name in an expression whose type does not correspond to the type of the datum currently residing in the cell. In PL/I, improper use of based variables can cause the same problem. Similarly, it is possible in Fortran to use floating-point data of differing lengths in such a way as to generate specification errors due to improper alignment of the data with respect to machine word boundaries. (Even worse, the System/370 Byte-Oriented Operand Feature can hide this error completely, with an increase in execution time as the only side effect.)

Another well-known error occurs when an attempt is made to call an undefined subroutine. The address constant which should have contained the address of the subroutine has not been relocated; thus its attribute should not be set to "valid branch address" until relocation (by the Loader or the Program Fetch routine, for example) has been completed.

```
/* PL/I EXAMPLE */
   CALL BOONDOCK;
/* WHICH TURNS OUT TO
   BE AN UNRESOLVED
   EXTERNAL REFERENCE */
```

Figure 7: Use of Uninitialized Code

This and similar "how-did-I-get-here?" errors could be more easily solved by assigning a "point-of-origin" attribute to branch

instructions, such that the location(s) of the most recently obeyed branch instruction(s) can be saved to provide flow-trace information.

The validity checking imposed on instructions is usually simple but stringent. (There has been some improvement since the days of the IBM 704, when assorted undocumented bit combinations could be used for making up interesting instructions.) On System/370, an opcode which has passed the level of validity checking can still produce all sorts of nonsensical actions: one can with alarming ease destroy the contents of base registers, branch to invalid addresses, over-write code, modify control blocks accidentally, etc.

Current systems rarely handle such operator-operand conflicts, and those that do are usually partially or fully interpretive. Three examples are the PL/I Checker, and the student-oriented language processors WATFIV (for Fortran programs), and SPASM (for Assembler Language programs). WATFIV checks some operand types as the program executes, whereas SPASM simply flags potential type conflicts when the program is assembled.

Compound Operations

A compound operation is a sequence of operations that performs a reasonably complete language function, such as the evaluation of an expression, the movement of a piece of data, a call of a subroutine, etc. While the distinction between compound operations (as defined here) and operations (as defined in the preceding section) is not very precise, some examples will illustrate the kinds of problems that can arise.

A statement may contain a reference to a subscripted variable, such as AA(J). It is implicitly assumed that the subscript J will not take on values exceeding the declared bounds of the array AA. Because this implicit behavioral assumption is so frequently and so easily violated, many computing systems now allow the programmer to state the assumption explicitly, in the form of a request for subscript checking.

Suppose a datum is to be moved from one place in memory to another. It is important that the types of the source and destination cells correspond: they should either be identical before the movement, or the destination type should be forced to match the source type when the movement has been completed if such a type change is explicitly requested. By disallowing some kinds of movements when type inconsistencies appear, we could prevent such common errors as over-writing programs with data, storing integers into floating-point variables, storing variable results on top of constants or addresses needed for subroutine calls, etc. The following figure illustrates one such error.

```
C  FORTRAN EXAMPLE
      COMMON /BBB/ NUMBER
      NUMBER = 13
      CALL FREAKY
C  ETC., ETC.

      SUBROUTINE FREAKY
      COMMON /BBB/ PI
      PI = 3.14159265
      RETURN
      END
```

Figure 8: Data Type Conflict

Fortunately, some help can be provided by compilers in detecting potential errors at this level. The PL/I language attempts to deal with such type inconsistencies by defining an exhaustive set of conversion rules covering all possible valid operand pairs. This can of course lead to unsuspected program behavior: the programmer should be warned, instead of being provided with lengthy and costly (and often unwanted) hidden conversions.

From a system programmer's point of view, one of the most important attributes of a compound operation is its "divisibility" attribute: what kinds of "interruptions" are tolerable while that operation is being performed. It would be useful to flag compound operations as "disabled" for certain interruption conditions; implementation of semaphores, status switching routines, and critical sections would be simplified.

From the point of view of a high-level language program, evaluation of a square root is usually considered to be indivisible. However, if that program wants to receive control to handle error conditions such as a negative SQRT argument (as in the IBM Fortran Extended Error Handler), the process must be divisible. From the viewpoint of hardware interrupts, the square root routine is arbitrarily divisible at the level of instruction sequencing.

Procedures

A procedure can be understood in the usual sense of a Fortran FUNCTION or SUBROUTINE, a PL/I PROCEDURE, as a block of a program, etc. The important characteristics of a procedure for our purposes are that it has a well-defined and small number of entry and exit points, manipulates a well-defined set of input data,

and produces a well-defined result. An important characteristic of procedures is that they can be separately compiled.

To give some illustrations of the incredible variety of errors that can occur at this level, consider a simple function-type subroutine. The user may pass it fewer arguments than it expects, or more. The types of the arguments may not correspond correctly between the calling and called routines. The routine may store into its arguments, possibly over-writing a constant. (One XDS system stores constants under a separate protect key to prevent over-writing.)

```
C  FORTRAN EXAMPLE
      CALL PSYCHO(170)
      STOP
      END
      SUBROUTINE PSYCHO(X)
      X = 5.678
      RETURN
      END
```

Figure 9: Argument Type Inconsistency

In IBM Fortran, the value of the function may be returned in the wrong register if the caller declares its type incorrectly, and the calling or called routine may attempt to use alternate returns which are ignored by the other.

```
/* PL/I EXAMPLE */
DCL UGH ENTRY(CHAR(*),BIT,FIXED(3));
  CALL UGH('CHAR','1'B,15);
      /* AND SO FORTH */

/* THE FOLLOWING IS COMPILED
   SEPARATELY, OF COURSE */

UGH: PROC(NOW,FOR,SOME,FUN);
  DCL NOW FLOAT, FOR CHAR(100),
      SOME LABEL, FUN ENTRY;
  FOR = NOW || FUN(SOME);
  RETURN;
END UGH;
```

Figure 10: Assorted Procedure Type Conflicts

In such cases, some important information is being ignored or lost: a procedure has attributes, and they should be checked for correct correspondence with the attributes of any other procedures with which it communicates. To illustrate, consider the list of errors in the previous paragraph.

1. One of the attributes of a procedure is the number of arguments it expects.

2. The types of the arguments form another set of attributes.

3. It is important to know which arguments are passed by name, by value, by address, or by some other convention. That is, the dynamic behavior of an argument is an attribute of a procedure: the called procedure knows how it will access its arguments and whether or not it will modify them.

4. The linkage conventions between calling and called routines are an important attribute of a procedure. For instance, the subroutine linkage conventions implemented by the SAVE macro-instruction under OS could save the caller's registers, and then set the attributes of registers 2-12 to "undefined".

5. The special-case and error-handling characteristics of a procedure are also attributes. (See CACM 19, 642 (1976).)

It is hard to overestimate the frequency and variety of the programming horrors caused by the lack of properly used attribute

information at this level, as the above examples indicate. However, some help may be available. If a large program is compiled in a single unit, the inter-procedure relationships can be checked at compile time if the translator is willing and able; PASCAL is a good example. The PFORT Verifier (see Software Practice & Experience 4, 359 (1974)) checks static inter-module relationships where possible. WATFIV and the PL/I Checker can perform some dynamic checks as well. In most cases, separately compiled procedures containing mistakes (or intentional misstatements) still have almost no defenses against one another.

Modules

For our purposes, a module may be bigger than a procedure and may be smaller than a complete program. A module is a collection of one or more procedures written in the same language, along with their associated support routines that provide the necessary run-time environment interface. All the procedures use common data structures and representations, storage conventions, linkage conventions, etc. Conceptually, we can consider a module to be that level of program complexity at which side-effects can first appear.

There are many familiar problems at this level. In Fortran, it is possible to make inconsistent declarations of a COMMON block area in different procedures, or declare a subroutine and a COMMON block with the same name.

```
C  FORTRAN EXAMPLE
      SUBROUTINE VENI (VIDI, VINCI)
      COMMON /MAIN/ LOTSA, GRIEF(1000)
      CALL WHONOS(WHAT,WILL,HAPPEN)
C  AND SO FORTH .....
      BLOCK DATA
      COMMON /WHONOS/ JUNK
      DATA JUNK / 469 698 558 /
      END
```

Figure 11: Inconsistent Uses of External Names

In PL/I, external data can have inconsistent declarations, and data and procedures can be given the same names. In almost any language, it is possible for some distant routine to modify a

global variable, and thereby cause a side-effect that may or may not have been desired or expected.

Just as for the "internal" data types described in many of the preceding examples, external names and data should have attributes as well. The presence of such attributes would allow a compiler or loader or link editor to check for consistency among the various procedures and data areas in a module. (This could eliminate the Linkage Editor's annoying habit of never telling where it found which copies of the things it kept, or which things it threw away from what.) Furthermore, if the library routines supporting the compiled procedures use the same linkage conventions as the procedures themselves, then the same tests can be applied uniformly to all components of the module.

Programs

A program can be described as a collection of one or more modules (which may have been written in more than one language), which can perform a well-defined computing activity without needing the presence of other user-written routines. A program needs only the support of its host operating system.

The most obvious example of problems to be solved at this level occurs when modules written in different languages are to be combined. The linkage conventions may be different, data types and representations vary from language to language, the runtime environments are mutually immiscible, the interfaces to the operating system are inconsistent or incompatible, and so forth. PL/I can help with the "FORTRAN" attribute for external procedures, but if there are lots of environment switching SVC's and data transformations, then the cost of mixing languages can be very high.

The presence of attributes would make it possible for modules to specify which data representations are used, what characteristics are expected of the runtime environment, what special features are needed from the operating system interface, etc. At this level, current systems use attribute information only in simple ways, such as file attributes detected by the operating system when a data set is opened.

The following example illustrates the conflicts that can be produced by using each language's "natural" data representations in a mixed-language program.

```
/* PL/I EXAMPLE */
DCL NAME CHAR(100) VARYING,
    SALARY FIXED DEC(12,2),
    RESULT FIXED DEC(5,2) ;
/* COMPUTE SOMETHING LIKE A TAX */
   CALL TAXES(NAME, SALARY, RESULT);
/* ETC. CALLS A FORTRAN SUBROUTINE */

      SUBROUTINE TAXES(NAME, AMOUNT, ANSWER)
      LOGICAL*1 NAME(100)
      REAL*8 AMOUNT
      REAL*4 ANSWER
C ... COMPUTE SOMETHING
      RETURN
```

Figure 12: Conflicts Between "Natural" Data Types

Operating Systems

The operating system is comprised of those routines which provide interfaces such as access to I/O devices, error detection facilities, program services, etc., between programs and the machine's facilities.

As a simple example of a problem caused by a lack of attribute information at the interface between programs and systems, suppose we are given an object deck of a complete and thoroughly debugged program, and try to run it under OS/MVT, OS/VS1, MVS, or CMS. The program is loaded into memory and begins execution, only to run into peculiar difficulties. Only then do we discover that the program was composed on the assumption that it would be run under DOS, which has an entirely different set of interface conventions. The following figure illustrates one of the differences in the ways two operating systems treat identical language constructs:

```
 * ISSUE DOS VERSION OF TTIMER
       TTIMER   ,
+      SR    0,0
+      SVC   52
 * ISSUE  OS VERSION OF TTIMER
       TTIMER   ,
+      SR    1,1
+      SVC   46
```

Figure 13: Operating System Incompatibilities

Even if we isolate a program's operating system dependencies into a single system-interface routine, we will still have problems with that routine.

It is reasonable to require that a program should always have sufficient attribute information to indicate initially which of a set of standard conventions it will choose, and the system should be able to adapt itself to whatever interface conventions the program then wishes to use (or it should politely refuse). Thus, DOS services might automatically be made available in any more general system.

The system interface is called upon to provide a variety of services; the requests are usually checked for minor error conditions (such as boundary misalignments of SVC parameters), but rarely for consistency in any larger sense. For example, it is common under OS to find that an attempt to do I/O with an unopened DCB causes an undesired branch somewhere into the low end of memory.

Many system integrity and security failures occur at this level. Operating systems must do a lot of checking to prevent errors, but the lack of attribute information allows for checking only on an ad-hoc and case-by-case basis. To take a simple example, if "control block xxx" attributes could be assigned, much of the checking and many of the sources of error could be eliminated.

Architecture and Configuration

Architecture represents the machine characteristics imposed on the operating system, such as the instruction set. The configuration is the total set of resources and facilities available to the system and its supporting architecture, such as I/O devices, hardware facilities, etc.

We can give several examples of problems that arise at the level of the machine's architecture. The Decimal Instruction Simulator on the 360/91 and 370/195 is provided so that support can be given to an attribute of a program ("it uses decimal instructions") which was not provided by the original architecture. Similarly, the Extended Precision Floating Point Simulator allows the system to support another class of instructions whose execution it would otherwise be forced to disallow. Both simulators dynamically detect an "attribute" of a program (it produces certain types of program checks) in such a way that an apparent mismatch with the architecture can be avoided. Such instruction-set attribute conflicts are usually discovered the hard way, when a program terminates abnormally. The set of instructions used by a program is an obvious attribute of that program.

Other problems arise when programs written to optimize their performance on one architecture are run on another. Thus, a SORT may run much more efficiently in a real memory environment than in a virtual storage. Such environmental assumptions embodied in the design of a program are also attributes.

To further illustrate the need for attributes at the configuration level, suppose we have a program which reads from two tapes and merges their contents on a third. If we want to run the program on a machine which has only two tape units, it should be possible to allow the "output" tape to be simulated by any other appropriate storage device, so that the "spooled" output is later written on a real tape without having to write additional JCL or job steps. (In fact, this "data staging" technique is regularly used on some CDC machines.)

The Job Control Language provides a means for externally specifying some of the attributes of a program, and for configuring the "system" under which it will run. Much of the frustration in using JCL comes from the fact that the program's true attributes may be very different from the attributes "hypothesized" in the user's JCL statements, and because the only way to detect the mismatch is to wade through a morass of manuals and dumps. (Consider the many forms of the 013 abend, for example!) Time-sharing systems allow some attributes to be specified at the time the program is run, thus achieving a small degree of configuration independence.

Manufacturers

Computing systems are intended to provide a desired set of facilities and functions, which are implemented in various ways by various manufacturers.

The most obvious incompatibility at this level is also one of the most trivial: there is a variety of character encodings, and

a variety of associated collating sequences. Even a simple data base cannot be moved among machines if the assumed orderings of alphanumeric keys within the data base are not invariant. It should be possible to design systems so that a collating sequence is specifiable as an attribute of the appropriate data, programs, systems, and hardware devices, and so that it may be varied to suit the needs of each usage.

Higher-level languages such as Fortran and COBOL were intended to provide some degree of manufacturer independence, and standards for such languages have arisen in the hope that program transferability might be enhanced. Unfortunately, other problems at this level have arisen because compiler writers have actually designed non-standard or machine-dependent features into the languages their compilers will be translating (e.g., REAL*8). Thus, the presence of such features is an attribute of the programs and of the systems and architecture the features will utilize. The only way to determine if a program's choice of language attributes will be tolerable to a compiler is to run it and hope for the best; it is a rare compiler that lets you specify the subset of the language it should accept.

However, there are even worse problems that can arise for a person trying to write transportable code, even for standards-conforming programs. These problems can often be traced to a lack of attribute information concerning such seemingly minor matters as (1) the maximum value that may be attained by a variable, (2) the representation used for integers, (3) the base and precision of floating arithmetic, (4) the behavior of the program and the arithmetic results when a computation exceeds some numeric bound imposed by the hardware, (5) the internal representation and packing scheme used for characters, and so forth. The availability of this kind of attribute information is necessary to be able to write programs with even a small hope of portability.

To illustrate, a PL/I program may declare a variable to be BIN FLOAT(15). On IBM System/370 systems, this gives 24 fraction bits, where other manufacturers provide exactly the requested 15 fraction bits. Thus, a program developed on an IBM system may give a false indication of the precision the program can actually provide.

Summary

Much of the trouble we encounter in trying to use a computer is caused by having thrown away too much useful information. In common computing terminology, we have been binding too many objects much too early. The preceding examples have tried to show that the discarded information could be kept and used in many helpful ways.

IMPLEMENTING ATTRIBUTES

Implementing a full set of attributes will have a number of costs. Some of the attribute information can obviously be defined in "hardware", and some in "software". Because much of what we call hardware is micro-programmed anyway, the micro-programs could support a full range of attribute handling. At the higher "software" levels, extra information can be attached to programs, data, systems, mechanical devices, etc. The necessity of attribute information attached to data on external devices has long been recognized, as many Data Base and Data Dictionary committees can testify.

It is important that the use of attribute information should be optional and controllable: a programmer may want to run his program under very complete checking and monitoring facilities at one time, and with complete freedom the next. At present it is very difficult to vary the modes in which a program may execute, so a programmer cannot make effective use of his prior knowledge of how the program will behave.

The ability to modify dynamically the uses made of attribute information is also important. For example, a program which performs a long, iterative calculation on a large array may wish to verify the correct definition and type of each element on the first pass over the array, and then test only for range conditions during the rest of the computation.

The idea of choosing the "style" of execution of a program is not unusual, particularly when we consider how widely it applies to other activities. Thus, the owner of a car can buy as simple or as fancy a machine as he likes, and can drive it any way he wants within whatever limits of social responsibility are imposed on him. A programmer should similarly be able to choose between speed, safety, and convenience, and the environment in which his programs run should be similarly prepared to impose limits of behavioral responsibility so as to protect both that environment and its other inhabitants. Regrettably, most programs are required to execute with a maximum of speed and a minimum of caution; it is small wonder, therefore, that we characterize the common result as a "crash", and display the results in a "dump".

Using Attributes

One implication of attaching attribute tags to such elementary objects as storage cells and data or instruction elements is that their sizes (in bits) will vary, possibly differing from user to user, or even from run to run for the same program. That is, one user may want to enforce the maximum possible checking, so that each cell has a large number of bits in its attribute tag; another user may want maximum speed or memory density, and therefore

will dispense with attribute tags entirely. In simpler forms, such tags have been used on many machines for a long time; for example, each word on the Burroughs B6500 contains 3 tag bits to indicate whether the word contains instructions or data, to identify descriptors, to provide memory protection, etc.

At the lowest levels, any of the attributes described above could be implemented by running programs in an interpretive or semi-interpretive mode. We should not object to interpreters on the ground that they perform slowly compared to compiled code, since the compiled code is itself being interpreted on another machine, via micro-programming. The availability of a writable control store means that the lowest level of interpretation should be tailored to the needs of the code it will be executing. The Burroughs B1700 provides this facility, whereby a "machine language" can be tailored to the needs of the program being interpreted.

Summary

At every interface, the two participants must pass not only data, but descriptions of the data as well, so the descriptions can be checked for consistency. At every operator-operand interface, similar consistency checks must be possible. At the levels of calls among routines, the attribute information can be verified and discarded if it is known that the attributes can undergo no dynamic modification. For example, a compiler could perform one level of checking, the Linkage Editor could perform a second, the Program Fetch routine could perform a third, and various run-time support functions in hardware or software could provide dynamic checks on the remaining attributes which have not yet been bound.

There are many decisions to be made about where and how attribute checking should be done. However, the intent of this paper is to show that attributes are needed. Once the need is widely recognized, it will be easier to decide what functions to implement, and how.

FINAL REMARKS

The difficulties described in this paper have happened partly because we have allowed ourselves to use a measure of program efficiency which excludes people: we should be considering questions of programming efficiency, in which the wasted time and pointless mistakes of the programmer are important factors. On such a scale, the efficiency and utility of modern computers is extremely low. Too much effort has been invested in the technical aspects of efficiency, such as hardware or compilers. As Knuth

remarked in discussing programs, "premature optimization is the root of all evil" (Computing Surveys 6, 268 (1974)); his comment describes almost the full spectrum of computing activities.

As the above discussion shows, the computing community has paid a terrible price for these tools: whatever supposed "efficiencies" may have been gained are too often outweighed by the costs of poorly thought out systems on which we must run our clumsy, error-prone, obscure, and unnecessarily complicated programs. There has been too little progress over the past fifteen years in the methods a programmer must use to write and debug a program; the fact that a major diagnostic tool is still the memory dump is a damning indictment of our collective ignorance. (It may be appropriate to quote Dijkstra's remark in his ACM Turing Lecture: "The electronic industry has not solved a single problem, it has only created them.")

Many other technologies that people depend on are learning the social and economic costs of expecting human beings to be "reasonable". Cars with potentially dangerous design flaws are recalled for repair at the manufacturer's expense; airplanes contain triply redundant control systems; electric appliances contain fuses, double insulation, and other forms of "fool-proofing"; hospitals are liable to lawsuits for deviations from "accepted practice", and so on. Computer programs, however, are still fatally vulnerable to the tiniest and most trivial of errors, and we continue to believe that it is therefore the responsibility of programmers to write ever more perfect programs. As one frustrated programmer said, "we don't write software, we write brittleware!"

As you can see from the views expressed here, we believe that the eventual solutions to our need for safe and reliable computing lie primarily in neither of two currently popular directions: not in teaching (or insisting on) proper programming techniques using precisely-defined languages, nor in the application of greater computing power to make up for the complexity and intractibility of the problems people want to solve. Human beings will continue to muddle ahead, our problems will usually be ill-posed, our solutions will frequently be incomplete and badly thought out, and our individual limitations will continue to make the gap between our best intentions and the actual results impossible to close. And if we plan to submit ever more of our lives to the keeping of computer systems, then it will be better if the small matters of inconsistency and inaccuracy described here are caught and exposed before they grow into large, unmanageable, and intolerant systems.

Where Next?

The members of the SHARE-GUIDE Language Futures Task Force have observed many of the "accidents" and "traps" described in this paper in past and present programs. The folklore of almost every programming shop contains at least one story of the insidious bug that hid in a system for days, months, or even years, only reaching out to smite the user at the most awkward possible time.

In the thirty or so years of our industry, very little has been provided to protect us and our users from these admittedly common problems. A few processors have been produced that can do rather comprehensive checking for many of the problems we described, but they are usually labeled "student processors" and are shunned by the "professional" or "industrial" community. A worse reaction is that a shop which says it cannot afford the "inefficiencies" of really thorough diagnostic hardware or software will often prefer to trade system down-time and customer dissatisfaction for that truly costly "efficiency".

The Language Futures Task Force believes that even with the best of efforts and intentions, the problems illustrated above will continue to appear. We therefore believe that many shops would trade a little processor speed for increased program reliability and greater assurance that today's applications are running correctly today and will continue to run correctly tomorrow.

The Task Force identified four areas where checking should be provided as a matter of course: at all interfaces between routines, between routines and data, between routines and devices, and between data and devices. This checking should be done as close to (and, if possible, at) "production" execution time.

We invite interested people to participate in further discussions of these subjects.