A panel session—Software transferability

Program transferability

by JAMES A. WARD, Chairman of Session
Department of Defense
Washington, D. C.

During the early development of higher order languages in 1959, we were told that programs written in such languages could be run on almost any computer. Now, ten years later, we find that programs so written not only are non-transferable from one manufacturer’s computer to that of another, but, in some instances, cannot be run on two computers of the same make and model with memories of different sizes. In the intervening years, computers have become much faster and the cost per operation has become much cheaper, while the cost of programming has become relatively far more expensive. Still, we do not have program transferability and millions of dollars are spent each year on the un-inspiring task of reprogramming routines on additional computers.

The objection to transferability of software is that it is impractical: programs transferred would operate very inefficiently because of different internal machine organizations. Data is particularly difficult to transfer onto machines of different word length. If such transfers are made there is loss of efficiency. A system that would provide for such transfer would unduly restrict the freedom of the programmer and introduce inefficiencies.

This is why in many military systems all the computers are required to be identical, or in the same family, so that programs written for one computer can be used on all and that programs and data can be transferred. The current procurement for World-Wide Command and Control is just such a system. The lack of transferability also makes the upgrading of a computer a traumatic ordeal for almost any computer installation in government or industry. A number of people who have recently gone through this to obtain third generation equipment seem to think that the government (or DoD) should freeze present hardware designs to prevent reprogramming for the fourth generation of computers. They seem to forget that this is a perennial request. I was asked to join such a movement over six years ago and this request has been repeated every year thereafter. Had this movement been successful, the current third generation would not have come about.

The design of digital computer hardware has made tremendous strides during the past ten years. Further advancements are forthcoming throughout the field, particularly in military computers and scientific computers designed for large government problems. I feel it is criminal to stifle advancement; therefore, I do what I can to encourage new designs and internal organizations.

But we still need the compatibility among different machines that will permit the efficient transfer of programs, data and programmers from computer to computer without reprogramming. Since the hardware design should not be unduly restricted, the compatibility necessary for transfer must in large part be provided by software. I feel certain that this will eventually evolve. In fact, I am happy to report that progress toward solution is already on the way:

a. The Air Force at Rome Air Development Center has initiated an R and D program to determine requirements for software transferability.
b. Dr. Hopper, in the Navy, has developed a standard COBOL by which programs in that language have been compiled and executed on many different machines.
c. Mitre Corporation, under DOD contract, is studying the problem of transferability of data.
d. There are also several laboratories with different manufacturer's machines in which any program written can be compiled and executed on any machine.
I am convinced that such transferability of programs, data, and programmers is within the present state-of-the-art. This panel from government and industry has been assembled to tell what has been done and what is planned.

Let us all cooperate to hasten the day when most programs written in any higher order language can be compiled and executed on most existing computers. The time, money and manpower saved by eliminating reprogramming can then be used to solve other more interesting and useful problems.

Program transferability

by ROBERT W. BEMER
General Electric Company
Phoenix, Arizona

General

The problem of program transfer is such that most people think they understand the process better than they do. Optimism is rampant; success is elusive. I have some tenets which I believe to be sine qua non:

- Program transfer is complicated by each element which is different—user, CPU, configuration, operating system, etc.
- Programs must be planned for transfer. "After-the-fact" is virtually useless, like post-classification for information retrieval. The information loss is too high in the transfer from programmer to code. If everyone wrote and documented his program as a connectable black box, only the connecting process would need to be under the control of the user.
- In twelve years of hearing proponents discuss it, I have not yet seen successful mechanical translation of machine language programs. There are the processes which a translator:
  a. Thinks it can do and can.
  b. Thinks it can't do and says so, for human rework.
  c. Thinks it can do and can't, and therefore doesn't say so!
- Transfer should always be made on a source program basis. Recompilation is a trivial expense.
- To the highest possible degree, the documentation of the program should be self-contained in the source program itself (rather than in the auxiliary documentation), and in a standard format and placement so that mechanized program tools know where to find the machine-readable information for extraction and use.
- Production of identical answers is (particularly for scientific problems) an additional requirement which must be specified and paid for. Differences may be due in part to differing internal arithmetic modes, but more often they are due to the overlooking of imprecision in method. On balance, obtaining different answers must be considered a healthy phenomenon.
- The criterion which a software module/component must meet in order to be self-documented adequately is:

  "Can it be dropped into a program/data base for problem brokerage, whereupon a completely anonymous user may make a mechanical search to his requirements, find and use the module in his problem, and pay automatically a brokerage fee upon successful usage?"

This would be one standard that nobody would argue about—if he got found money at the end of the month, for conforming. Perhaps this might be a better solution than patenting software. Only thus can the non-specialist take advantage of computer utilities.

Some information required to transfer (run) a program

- Program name (number)
- Program function
- Descriptors, classification (computing reviews)
- Original computer system
- Original configuration, subset of required configuration, options used/available
- Other systems/configurations verified to run on
- Operating system, requirement, linkages, interfaces
- Running instructions
- Store requirements (resident program, nonresident program, data, tables, segmentation, overlay sequences)
- Source language (standard, dialect)
- Input/output data
- Data structures
- Data types
- Data elements, collating sequence

(1) To complete while producing the program.
• Interfaces (other units called, libraries)
Connections (via jumps, switches, natural flow)
Languages/processors equipped to call this program
• Method, average runtime (for interactive simulators)
Restrictions, constraints, degenerate cases, idiosyncrasies
Range, accuracy, precision
Changes occurring in conditions, status, original input

Optional
Information specific to program transfer
Default options—referring to international/national standards
Responsible organization
Grade of program (thoroughness of testing)
Test cases and answers (possible autoverification and answer match)
Bibliography, references
Copyright, price, etc.
Source/object program listing, number of instructions/statements

Mechanical tools for conversion

• Combinatorial path exercisers through a program
• Programs which page the source code for the programmer and mechanically force him to be up-to-date
• Programs which mechanically check the linkage of units of a software system to provide a directed graph for flow verification, ensuring that any software unit will not interface with other software units to which it should not be connected.
• Mechanical determination of valid paths in the reverse direction of flow, as a diagnostic tool for finding “How did we get here from there?”
• Mechanical verification of successful meeting of interface requirements when passing from one software unit to another in a forward direction.
• Mechanical re-verification of linkage and interface requirements for any revisions.
• Code acceptance filters.
• A patch defense (correct/change in source code only)
• (De-)flowcharters

Mechanical capture of facilitating information

The source-to-object program translation process yields information. Much of this is lost, but needn’t be. Some of this information concerns elements which are not themselves standardized, but can be part of a standard list of measurements useful to program transfer.

Therefore a language processor should be constructed:

• To be self-descriptive of its characteristics (i.e., features contained, added or missing; dialects or differences).
• To affix to the original source program, as a certification of a kind, either an identification of, or its actual characteristics. It may also strike characteristics or features which were unnecessary for that source program.
• To inspect transferred programs for a match to its own characteristics.

If the transferred program is processed successfully:

• The identification of the new processor is also affixed to the source program.
• In any area where the new processor has lesser requirements (i.e., a smaller table worked successfully; a missing feature was not required), the affixed information is modified to show the lesser requirement.

Thus a source program, once processed, contains information on:

• The minimum known characteristics required for successful processing.
• All processors (with operating systems) which treat the source program successfully.

Software compatibility

by JOHN A. GOSDEN
The Mitre Corporation
McLean, Virginia

Data Exchange

There is a growing need for data exchange, particularly the passing of files of data between programs that were produced independently. This will be needed in the development of computer networks and data bases; for example, a head office installation collecting files from various divisions of a corporation to build a data base. Both the development of formal and informal computer networks as well as the economic feasibility of large
data bases are favoring the development of arrangements for a considerable volume of data exchange, whether directly over communication systems or by the dispatch of reels of tape and boxes of cards. These are very significant areas of growth that are just beginning to emerge in commercial EDP and are already creating problems within the Federal government.

The development of data interchange is straightforward when the correspondents have agreed on the format, but where there has been no prior agreement, conversion usually involves considerable manual intervention. Some typical problems are that:

a. The sender's format may not be specified rigorously and an informal description may have to be debugged.
b. The sender's format may not be expressible in the receiver's system.
c. The sender's format descriptions may be embedded in the program.
d. The format in the sender's system may vary from record to record and be embedded in the data.

Any of these problems may arise when either an existing application is converted to a new system, or a new member of a cooperating network has a system different from that of any existing member.

There are two basic problems:

a. Few existing systems have any ability to deal with a new format automatically, and those that do are limited to data described in the same system.
b. The number of different, and often incompatible, ways of describing data is increasing; e.g., Format statements in FORTRAN, Data Description in COBOL, COMPOOL in JOVIAL, FFT's in FFS.

Any solution to this problem should not restrict participants in the use, within their own local system, any internal data structure they like or any programming or query language they like. Therefore we need a standard data description language for data exchange. It is expected that systems should interface with a single way of describing data for interchange and provide conversion processes in their interfaces. If a suitable interface is to be developed, we will not want to standardize formats, which would be absurd, but we would want to standardize ways to describe formats. We also want to attach the data descriptions to the data, so that the transmission of both data and its description can be performed without manual intervention.

A data description language for data interchange does not have to be read and understood principally by humans. It can be thought of as a complicated coding to be generated and interpreted by the interface modules of systems in a network. In a well-designed system a user would describe data in the form provided for local use, and the system would translate to data interchange conventions. Therefore, the data description language should be generally compatible with data descriptions in current programming languages. Later, developments in programming languages may be influenced by a desire to remain compatible with data interchange conventions.

Standardization of high-level languages

by GRACE MURRAY HOPPER

Director, Navy Programming Languages Group
Washington, D. C.

The terms "compatibility," commonality," and "transferability" are used in discussing the mobility of programs and programmers. The common element essential to such mobility is the establishment of standards for programs, programmers, and documentation. The adoption of high-level programming languages such as COBOL, FORTRAN, ALGOL, and JOVIAL is a required element of such standards. The high-level languages are innately self-documenting—an essential for transferability. Thus, their use provides assistance in the transfer of programs among activities; the conversion from one computer generation to another; the conversion from one computer manufacturer to another; and the transfer of programs for back-up and readiness.

Further, the programmers, themselves, need be trained but once and retraining upon transfer to a new system is virtually eliminated. Programmers become capable of greater productivity per unit of time resulting in less cost per unit of program. However, such application of the programming languages implies that they themselves be standardized. A standardized language must be commonly acceptable, competitive, developable, maintainable, and useful. The standard language must be developed, defined, approved, adopted, and installed.

The installation of a standard programming language, in varying environments with varied requirements,
requires of management the normal functions of promul­
gation, installation, control, evaluation, monitoring,
and provision for maintenance and changes. The execu­
tion of these functions can be assisted by the use of pro­
grammed tools such as validation routines, translators,
preprocessors, flowcharters, debugging aids, and by
standard manuals and instruction courses.

The discussion includes a survey of the growth of the
standard languages with implications and suggestions
for future developments. Information supporting the
need for the application of the languages and the re­
sultant time and cost savings is introduced. The neces­
sary components of an installation package are defined
and their implementation discussed. The need for
management interest, concern, support, and action is
stressed.

The transferability of computer programs
and the data on which they operate

by EDWARD MORENOFF

Rome Air Development Center
Griffiss AFB, New York

INTRODUCTION

Software transferability involves the transfer of pro­
grams and the data on which they operate from one
arbitrary operating environment to another, with the
expenditure of only a small fraction of the initial pro­
gramming development time and cost. The programs
can range from small routines for evaluating trigonomet­
ric functions, to large and complex systems such as com­
pilers, data management systems, or command and
control systems. The environments in which these pro­
grams are to be executed may be either slightly or
highly dissimilar with respect to machines, machine
configurations, or operating systems and languages
used.

The interest of the Rome Air Development Center in
this area extends over a period of several years. During
this time, both hardware and software research and
development programs have been initiated which either
directly or indirectly contribute to the solution of the
problem. The utility of adopting standards, and the
form these standards might assume, were defined in
1967.1

Typical of the efforts in the hardware area was a
study leading to the definition of a microprogrammed
computer main frame capable of efficiently executing a
number of different machine language instruction
sets.2 Typical of the efforts in the software area was the
work leading to the design of a generalized data
management system,3,4 the development of modular
programming design tools,5,6 and the Information
Processing Code.7

In January 1968, the Program Transferability Study
Group* was established. Its principal objective was to
examine the whole area of software transferability
formally, and see what, if anything, could be done to
eliminate the problems associated with transferring
programs and data between arbitrary environments.
The preliminary findings of the study group were
released in June 1968.8 The Group found the main
obstacles to software transferability to be loose specifi­
cation of data structures, lack of programming stand­
ardization, and lack of freedom when higher level
languages were used. Possible solutions to these prob­
lems are: (1) Administrative control of programming
and documentation (2) Extensions to current languages
(3) Use of a new programming environment which
would eliminate the constraints of the older system.

The problem

The study group concluded that the problem of trans­
ferring a program between arbitrary operating environ­
ments was not solved by the current technology, even
when the initial program development made use of a
standard version of a single higher level language.
Current practice is such as to require changes to the
initial form of the program itself rather than simply a
recompilation, in order to adapt the program to a
change in environment.

One of the principal factors necessitating such
changes is the lack of adequate facilities for the explicit
specification of data, programs, actions, messages,
linkage, and the like. The programmer is instead en­
couraged, if not actually forced, to make implicit in the
form of his program many of details of its initial en­
vironment and the data to be operated on. In order to
transfer a program from one environment to another,
reprogramming is required to the extent that differences
in the two environments must be reflected in changes to
the program.

Closely related to the lack of adequate explicit spe­
cification facilities is the lack of constraints which would
serve to regularize the behavior of the programmer. As
a consequence of the excessive generality and complexity

*The Program Transferability Study Group was chaired by
George H. Mealy. Other members included T. E. Chestam, Jr.,
David J. Farber, Edward Morenoff and Kirk Sattley.
in present operating systems and languages, a programmer setting out to prepare a complete program has entirely too much freedom. In the absence of explicit facilities or when such facilities are too general, different programmers produce entirely different programmatic solutions to the same problem. A strong relationship is believed to exist between this difference and the difficulty in transferring programs.

An example of a language which has enjoyed a measure of success in the development of transferrable programs is COBOL. COBOL encourages the explicit description of data rather than the implicit description inherent in most other languages. It is about the only language system which permits any kind of environment specification. It is limited in scope, however, to problems which trace their origin to unit record equipment. This narrowness of scope appears to be one of the principal reasons for its success in allowing programs written in it to be transferred between environments.

In addition to the use of higher level languages, the study group found two other basic approaches to solving the transferability problem, both largely unsuccessful. The first approach involves "decompiling" a binary, decimal or symbolic program written for one operating environment, and then generating new code for some other operating environment. The second approach is by the application of a set of management solutions. These solutions range from the insistence on replicating functionally identical hardware configurations to setting standard specifications on programs so they are completely modular with very precise functional specifications and clear interfaces.

Attacks on the problem

The second major conclusion of the study group was that the current state of the computer technology had reached a level of development at which something could be done about the problem of transferability today. Indeed, three complementary levels of attack were proposed.

The first level of attack, designated as level A, deals with the definition and application of suitable administrative controls, both to the programming and documentation processes, using languages and operating systems as they exist today. The resulting standards would form a common subset or intersection of the capabilities of the operating systems and languages to be involved in the family of transferrable environments. To work at all, these standards must be enforced by a czar with final approval over all programming work.

The second level of attack, designated as level B, deals with the development of extensions to the current language and system base. This involves identifying the deficiencies with respect to particular existing systems, defining how these deficiencies may be eliminated, and then implementing the resulting changes to equalize the capability of all the operating systems and languages in the family of transferrable environments.

Part of the design problem at level B is to identify which extensions can be incorporated into the existing systems with a high probability of success within a two year period from the time they have been defined. Extensions not falling within this category would be deferred for consideration as part of attack level C.

The third level of attack, designated as level C, relaxes the requirement of using the existing base of operating systems and languages to allow consideration of a wholly new programming and operating environment which would substantially eliminate, for most practical cases of interest, the problem of program transferability. Within this environment the higher risk and/or longer term concepts would be tested and evaluated.

The life of the study group has been extended. It is currently continuing its investigations of selected aspects of the software transferability problem.

BIBLIOGRAPHY

1 E MORENOFF J B McLEAN
   An approach to standardizing computer systems
   Proc 22nd National ACM Conference Thompson Book Co August 1967 327-356
2 DECISION SYSTEMS INC
   Interim Status Report No 1
   Contract No F30602-68-0223 August 11 1968
3 P J DIXON J D SABLE
   DM-1 a generalized data management system
   Proc SJ C Thompson Book Co April 1967 185-198
4 E MORENOFF J B McLEAN
   On the design of a general purpose data management system
   Proc Fourth Annual Colloquium on Information Retrieval Int Information Inc May 1967 19-30
5 E MORENOFF J B McLEAN
   Inter-program communication program string structures and buffer files
   Proc SJ C Thompson Book Co April 1967 175-184
6 E MORENOFF J B McLEAN
   Program string structures and modular programming
   Proc National Symposium on Modular Programming Information and Systems Press June 1968 176-186
7 E MORENOFF J B McLEAN
   A code for non-numeric processing applications in on-line systems
   Communications of the ACM Vol 10 No 1 January 1967 19-22
8 G H MEALY et al
   Transferability study group
Transferability of data and programs between computer systems

by JEROME D. SABLE
Auerbach Corporation
Philadelphia, Pennsylvania

INTRODUCTION

The cost of problem analysis and programming dominates by far the cost of computer utilization. This dominance will increase in the future as hardware elements become more powerful and economical and problems become more complex and demanding of highly skilled analysis. As this trend continues, the sophisticated user will become increasingly intolerant of a situation which prevents economic transfer of programs and data from one installation or hardware type to another. A standard machine independent environment which provides data management services at several levels to programmers and task programs should be defined. Programs would then interface with a set of virtual machines, or standard program environments, which are independent of the particular hardware configuration of the installation.

The traditional approaches to data and program transferability have been through (a) the use of compatible hardware types which have presented “equivalent” hardware interpreters for data and program, and (b) the use of standard higher level procedure-oriented languages and compilers to translate programs to a particular machine type. The first approach guarantees complete interchangeability only as long as the program’s support software is duplicated but removes the possibility of matching special hardware types to problem areas for which they may be particularly appropriate. This restriction is unnecessarily severe in many cases. The second approach avoids this restriction but there are many problem areas for which standard procedure-oriented languages have not been adopted. Indeed, many systems will continue to be implemented in assembler and macro level languages.

I would like to suggest that the problem of data and program transferability can be approached most generally by extending approach “(a)” to include software as well as hardware interpreters. Software interpreters, or simulators, have been used in the past to transfer a machine language program from machine A to machine B. However, these have not been entirely successful or widely used. The success of this approach hinges on the ability to write programs for one of several standard environments and to describe in a standard, yet general, way the data structures which are to be transmitted and interpreted. To permit general applicability from machine level programs and data, through to higher level language programs and other character stream messages, requires that a wide range of data structures and languages be describable in a standard way. The language description standards must include the lexicographic, syntactic, and semantic levels.

In the following paragraphs a hierarchy of data structure types will be described which range from machine and storage-oriented structures to “logical” data structures transmittable as character strings independent of physical representation. They are offered as one possible approach to a comprehensive set of data structures.

Data structures

The cell

The most primitive concept to be considered is that of an address space. This is viewed as a region of atomic elements or cells which are addressable with some address word. There may be a hierarchy of cells such that higher level cells form an address space for a lower level cell. The cell is viewed as a region in which data can be stored and accessed rather than the data themselves. Several address spaces, or stores, may be involved in a system or network.

A given computer will have a system of primary, secondary, tertiary, etc., stores associated with it. A cell in any store is addressable with an address or a simple transformation of an address.

The truck and train

The most primitive relocatable data structure is the truck. This is a module of data which can be stored in a cell. A sequence of trucks, called a train, may be defined and transmitted from one store to another. Any relocatable data structure (the term as used here includes programs) is ultimately handled as a train whose trucks are bytes, words, or pages.

The bead, strip, and plex

An element (e.g., train) of fixed length and defined field structure will be called a bead. A sequence of beads which address other beads then a network of beads, called a plex, is formed.*

*The terms plex and bead have been used by D. T. Ross in his work on the AED systems.
The stream

A list or list structure of byte trains of various lengths will be called a stream. Consider the data structures necessary for representing a process, or active job. These include a stream containing procedures and subroutines, a stream for the dynamic working data (or activation records) of each procedure, and a stream for the control stack which provides subroutine control. Collectively, then the process can be considered to be a stream list.

The item

Finally, a data structure may be defined in terms of named entities, called items, which bear a defined relationship to each other. These items are fields, records, files, and statements and are independent of any storage structure or address space. Items may be independent messages or form a highly structured hierarchy of nested elements, files, etc.

Specification languages

The success of the approach to data and program transferability being proposed here depends on the adoption of a comprehensive data description language (DDL). The DDL must meet a number of requirements:

1. It must be able to handle at least the range of data types listed above.
2. The requirement to handle item structures and source language messages means that it should be capable of expressing the symbols and syntax of context-free languages.
3. It should be capable of expressing the semantics of translation or interpretation of the source language strings, including procedure calls to other processors.
4. It should be representable in a graphical format that exhibits the structure of the language or data being described so that it represents an effective tool for human communication.
5. It should also be representable by linear strings amenable to transmission and computer input, and it should be easy to translate from the graphical to the linear version, and vice versa.

CONCLUSION

The problem of transferring data from one computer to another, and interpreting the data correctly, can be approached as a problem of constructing an adequate range of data structure types and devising a standard way of describing these types. No one level of data description is adequate. Rather, there must be a range of structures which go from the highly machine oriented cell structure to the user-information oriented structure. It is felt that a hierarchy of data structure types which is adequate for this task can be devised. The data types to be included have been listed above.

The problem of data description, however, is but a special case of language specification and special attention must be paid to the lexicographic, syntactic, and semantic aspects of specification. Some recently developed language specification languages and generalized language processors can be brought to bear on this problem. These processors can be employed as a standard interface in a network of different computer types. When furnished with the description of the structure (languages) to be accepted, they can carry out the appropriate translation and interpretation.

The stratification of data management services into a number of standard levels would make it appear to the programmer, that at any one moment, he is interfacing with one of a number of virtual machines which form an upward compatible hierarchy. These services may be, in fact, provided by a mix of hardware and software modules which depends on the particular system implementation, and the hardware types being used in a given instance. Thus, each program has, as its interpreter, a virtual machine whose interface with the program is known but whose composition may vary and is irrelevant except for timing considerations. A given installation, because of hardware and software modules used, may provide interpretive virtual machines only up to a given level, requiring programs written for a higher level virtual machine to undergo a translation process down to the appropriate level before interpretation can take place.

That the goals outlined above will be difficult to achieve in today's technology and marketplace is well recognized. However, I feel it is the responsibility of the industry to its users to embark on a comprehensive project to ensure software transferability across a wide range of problem areas and hardware types without limiting the development of new languages and machines.