RADC-TR-68-341

PROGRAM TRANSFERABILITY STUDY

George H. Mealy
T. E. Cheatham, Jr. (Computer Assoc.)
David J. Farber (Rand)
Edward Morenoff (RADC)
Kirk Sattley (Computer Assoc.)

TECHNICAL REPORT NO. RADC-TR-68-341
November 1968

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

57

PROGRAM TRANSFERABILITY STUDY

George H. Mealy
T. E. Cheatham, Jr. (Computer Assoc.)
David J. Farber (Rand)
Edward Morenoff (RADC)
Kirk Sattley (Computer Assoc.)

MAINLINE 11—68

FOREWORD

This report, accomplished under Project 5581, Task 558102, is the result of a series of two-day meetings over a six-month period among a group of consultants under contract to Rome Air Development Center, a representative of the Rand Corporation, and a representative of RADC. The chairman of the group, George H. Mealy, is an independent consultant; T. E. Cheatham, Jr., and Kirk Sattley are with Computer Associates; David J. Farber is with the Rand Corporation; and Edward Morenoff is with the Information Processing Branch of RADC.

This group was formed at the request of the Department of Defense (DDR&E), Dr. James Ward, for the purpose of determining what action, if any, could be taken in the immediate and far future to reduce the problems associated with transferring computer programs among computers of different size and manufacture. The study devoted itself only to the software aspects of transferability. There is a hardware aspect which has yet to be addressed and where some promising approaches need investigation. This effort was motivated by the continued interest and encouragement of Dr. Ward.

The responsibility for the content of this report remains solely with the members of the study group and does not necessarily reflect the views of their respective organizations. The authors wish to acknowledge the early contributions of Joseph W. Smith and Stephen Warshall to the study group. Minor editing of the report was accomplished at RADC without specific concurrence of the authors.

This report was presented in June 1968 at a Department of Defense meeting attended by representatives of DDR&E, DCA, ARPA, Army, Navy, Air Force, Marines, Joint Chiefs of Staff, and the Intelligence Community.

This technical report has been reviewed by the RADC Foreign Disclosure Policy Office (EMLI) and the Office of Information (EMLS) and is releasable to the Clearinghouse for Federal Scientific and Technical Information.

This technical report has been reviewed and is approved.

Approved: Major James G McGinnis
for FRANK J. TOMAINI
Chief, Info Processing Branch
Intel & Info Processing Division

Approved:
JAMES J. DIMEL, Colonel, USAF
Chief, Intel and Info Processing Div

FOR THE COMMANDER: Irving J. Gabelman
IRVING J. GABELMAN
Chief, Advanced Studies Group

ABSTRACT

This report treats the problem of transferring programs from one operating environment to another with the expenditure of a small fraction of the initial programming development time and cost. Programs considered range from quite small ones, such as routines for evaluating arctangents, to large and complex systems, such as compilers, data management systems, or command and control systems. The initial and final environments may be slightly or highly dissimilar with respect to machines, machine configurations, or operating systems and languages used.

The problem stated above is not solved by the current technology, even when the initial program development makes use of a standard version of a single, higher level language. Current languages and operating systems are such that in order to adapt a program to a change in environment it is necessary to make changes in the initial form of the program itself rather than merely to recompile it. The two principal reasons for the necessity of making such changes, which may require very significant expenditures of time and programming effort, are:

- The current technology encourages, and usually forces, the programmer to make implicit in the form of his program many details of its initial environment, such as word length, character code, services performed by the operating system, and the like.

- The structure and representation of the data accessed by the program is describable in machine-processable form only to a small degree. Again, the programmer is encouraged or forced to make this description implicit in the form of his program.

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. Particularly in the case of larger, more complex programs, such changes can currently amount to an almost complete reprogramming effort.

Despite this situation, the current state-of-the-art permits an attack on the transferability problem on several levels:

- By the use of appropriate methods of administrative control of the programming process (including documentation), programs to be written using existing languages and systems can be transferred with materially less effort than is typically required at present.

- Based on the current "Third Generation" systems, languages, and programming technology, extensions can be developed in such a way that programs written for the resulting systems can be transferred with significantly less effort than would be the case using purely administrative control.

- By relaxing the above restriction requiring the use of the existing base of operating systems and languages, a programming and operating environment can be created which will substantially eliminate, for most cases of practical interest, the problem of program transfer.

This report makes recommendations on all three levels.

TABLE OF CONTENTS

I. THE PROBLEM

The problem addressed in this report is that of transferring programs from one computer system or operating environment to another at a fraction of the initial programming development time and cost.

The programs considered range from simple mathematical routines such as those for the elementary functions to application programs written in some higher level language like FORTRAN or COBOL, and on to complete sub-systems or operating systems such as compilers, data management systems, command and control systems, and the like.

The initial and target computer systems and operating environments might be slightly to highly dissimilar with respect to machines, machine configurations, or operating systems used. Thus the transfer of programs from some particular computer system to another within the same "family," which differs in such features as the amount of directly accessible high speed memory, speed of central processor, types of peripheral equipment, and so on will be considered. The report also will discuss the transfer of programs between completely different computer systems where, for example, even the means of representing information might be different (e.g., binary 36-bit "word" to binary 24-bit "word," or to decimal 10-digit "word," or to 8-bit "byte,"). *

There are other problems in the development of programs and programming systems which are directly related to the problem of transferring programs but which are of a somewhat smaller degree of difficulty. For example, the problems of modifying and extending some extant program, as well as the problems of maintaining and updating a program system, are very strongly related; facilities which will ease the cost of transferring programs should also significantly affect the cost of program maintenance and updating, as well as modification and extension. Updating, here, refers to both software and hardware updating. Software updating might include changes to algorithms or data representations to improve performance, adjustments to software to match hardware field modifications, and so on. Hardware updating might include replacement of components with faster or more reliable components, addition of more high speed memory, addition of new kinds of peripheral devices. It is also noted that facilities which provide for the inexpensive transfer of programs will also facilitate the initial design and development of program systems and should result in decreased time and cost for this activity as well, especially for large program systems.

*The principal factors which operate to impede transferability are listed in Appendix I.

There follows in the remainder of this section a discussion of a number of attempts to solve or at least alleviate the problem of program transfer; in the next section we will consider the basic inadequacies in the facilities available: the basic reasons why the current programming technology does not "solve" the problem.

There have been three basically different approaches to solving this problem:

- One approach is to attempt to transfer "ones and zeros," that is, to somehow "decompile" a binary, decimal, or even symbolic machine language program, and to generate coding for some new system either automatically or semi-automatically.

- A second approach is to insist on the use of some higher level language with the expectation that by moving a compiler onto the new system, other programs can be transferred to it merely by recompiling them.

- The third approach is the administrative or management solution which ranges from the insistence on replicating functionally identical hardware configurations to so constraining the behavior of programmers that their programs are easily transferable.

These will now be considered in somewhat more detail.

1. Direct Program Transfer

There have been a number of attempts to develop "recompilers," that is, programs which take a binary, decimal, or symbolic program for one computer system, "decompile" it, and then generate coding for some other system, either completely automatically or semi-automatically, with the expectation of programmer intervention to handle certain "hard" cases. With the exception of certain small or rather special programs (e.g., those generated by a compiler from a higher level language version in the first place) this attempted solution has not met with much success.

This, of course, is not surprising since, with the exception of certain highly restricted situations, this solution is theoretically impossible. The behavior of certain components which depend upon timing constraints or run-time data values cannot be determined a priori, thus clearly denying a complete solution.

Nor has this solution proved feasible in a practical sense. If one attempts to model the two computer systems at the gate level, the combinatorics get completely out of hand. Further, there is no effective theory of systems which might provide a host formalism permitting some practical or partial solution to this problem. Current formalisms do not effectively contend with such issues as timing, the finite nature of computers, hierarchies of storage, and so on.

2. Use of Higher Level Languages

This "solution" has enjoyed considerable appeal; however it has not lived up to the expectations advertised for it. Some difficulties have arisen from the fact that in most cases a completely standard or universal language (and language processor) is lacking. This, in turn, is related to the fact that there exists no effective theoretical framework to permit formalizing the structure (syntax) and meaning (semantics) of programming languages, thus denying the possibility of any precise standard language model. Other difficulties arise from the fact that each of the higher level languages available accomodates only a narrow band of the spectrum of programs to be transferred. Let us consider briefly three higher level languages which have been implemented for a number of computers and for which there are "standard" versions: COBOL, JOVIAL (J-3), and NSA-ALGOL.

COBOL

The development of the COBOL language and the development of processors for COBOL programs on virtually every major computer system must be viewed as a very significant achievement. COBOL was one of the first language systems to provide separate facilities for the description of data and data structures and for the manipulation of these data and structures. This separation is of significant importance in that it encourages the explicit description of data rather than the implicit description which is inherent in most programming languages. COBOL is still about the only language system which permits any kind of environment specification. It seems clear that the use of COBOL has permitted the development of a number of application programs and even a few program systems which do enjoy machine independence and which are thus transferable. The only real objection to COBOL is that it is relatively limited - the problems to which it is applicable fall into a rather narrow band of the spectrum: typically problems which trace their origins to unit record equipment. It must be remarked, of course, that even though the band is narrow there are a relatively larger number of actual applications for which COBOL is quite appropriate. No criticism of COBOL for its "narrowness" is intended. This was, after all, a specific and quite reasonable design goal. Indeed, COBOL would probably not have achieved the level of machine independence which it has, had its designers attempted to make the language more general than it is.

JOVIAL

JOVIAL has been offered as a language which is appropriate for a larger class of application programs than COBOL and, indeed, as a language appropriate for programming complete systems. JOVIAL also has facilities which permit the separation of data description and data manipulation, although these facilities are rather more awkward and in many ways less general than those available in COBOL. JOVIAL, however, has only a primitive facility for dealing with the environment in which a program might run, that is, one can specify the other programs and data which will always be available when a given program is to run. The standardization of JOVIAL, in the sense that it permits a truly machine-independent program specification, has not been as successful as has been the case with COBOL, primarily because its COMPOOL facility is not a part of standard (J-3) JOVIAL. It seems clear that part of the reason for this is that JOVIAL attempts to cover a wider spectrum of problems. It does in fact do this in the sense that there are more data types and a better ability to "manipulate bits"; it is exactly this generality which considerably weakens its ability to permit explicit specification of the environment. This specification thus tends to be implicit. Further, whereas COBOL does have a facility which amounts to providing for explicit storage management, JOVIAL has relatively little except for some rather primitive provisions for program and data overlay. It is often in this area, where the specification becomes essentially implicit, that the inability to transfer programs derives.

NSA-ALGOL

NSA-ALGOL is an interesting case, as it is a language which was developed specifically to be machine independent. Indeed, in the original implementation of the language there were two different machines, either of which could be chosen as the compiling machine for a program written in NSA-ALGOL and, at the time of compilation, any one of four quite dissimilar machines could be chosen as the target machine which is to execute the compiled program. The target machines vary in word size as well as in other characteristics.

Like COBOL, NSA-ALGOL was developed with a narrow band of the complete spectrum of programs in mind, in this case programs primarily concerned with numeric computation. As basic entities the language has only numeric and boolean quantities plus arrays of them. Experience has shown that for "ordinary" numeric computations NSA-ALGOL does indeed provide machine independence so long as the accuracy of the results is not critically dependent upon the word size of the target computer, the radix in which numbers are represented, or the technique employed for round-off.

NSA-ALGOL also has a fairly primitive data description facility through its provisions for "table" declarations. These include the packing of several fields of a table into one computer word. However, because the declaration facility is relatively weak, programmers resort to techniques such as defining what the compiler thinks are two different tables and then basing these two tables at the same location, hoping thereby to have an item in one of the tables really be an n-tuple of items of the other table. However, the table item allocation mechanisms employed for the four different target computers are of necessity somewhat different, and will often result in differences in the actual overlaying of items, thus inducing differences in the behavior of programs referencing overlayed tables. That is, the table declarations do not provide a true extension to the language which permits it to deal with n-tuples. Programming "tricks" which attempt to endow it with this ability often lead to machine dependence.

The preceding discussion of three languages illustrates their particular shortcomings from the point of view of transferability. Such defects also appear in other languages, e.g., NELIAC and PL/I. The next section will discuss more basic issues: It will develop that these cannot be resolved by mere addition of features to languages of the current varieties.

To summarize: The approach of using some higher level language and expecting automatic transfer of programs by simply compiling them for a new machine does not provide a solution to the problem. While there are several languages which permit relatively simple kinds of programs to be transferred, these languages do not cover a sufficiently wide band of the spectrum of programs.

3. Administrative and Management Solutions

One kind of management solution to the problem of permitting transferability of programs is that of insisting on duplicating, at least functionally, a given hardware configuration. Indeed, on the surface this seems to be a simple and complete solution. However, this "solution" has a number of subtle problems in addition to the rather obvious problem of favoring one manufacturer, an important factor in cases of large procurements.

First, it turns out to be surprisingly difficult to obtain and maintain truly identical configurations. Field modifications, slight differences in behavior due to variations in timing, differences in mechanical devices, and the like all induce differences in a given configuration. These can usually be gotten around by careful administrative control and an insistence on programming techniques which do not "push the equipment to the limit," but depend only upon expected or average performance.

A more serious difficulty is that many installations have a certain percentage of their problems which are "standard" with the remainder being problems which are unique to them. These unique problems in turn often require rather different hardware configurations than those required for the "standard" problems. Indeed, something as simple as adding one peripheral device to a "standard" configuration can induce considerably more in the way of change to that configuration than is obvious on the surface. For example, to deal with the new device the operating system will require extra core memory space for the "books" it must keep relative to the device. This, in turn, induces differences in the operating system's allocation to memory, which, in turn, induces differences in the memory actually available to user programs.

The notion of a "family" of computers with models ranging over a couple of orders of magnitude in price and somewhat more in performance, but with "upward" or "downward" or "complete" compatibility has also proved to be less of a panacea than was expected a few years ago. Things like basic cycle time, basic core sizes, availability of drums, discs, or just tapes as back-up stores, and such, have, with conventional programming techniques, led to radically different strategies of program construction (particularly for "system programs"). The resulting differences render programs non-transferable in any practical sense. For example, a data management system using tapes only will very probably have a strategy of employing secondary storage which would be highly inefficient if utilized directly with a drum and discs replacing the tapes.

Another kind of management solution which has often been attempted is that of setting standard specifications on programs so constraining the behavior of programmers that their products will be completely modular with very precise functional specifications and clean interfaces. The successful application of these standards is clearly useful in increasing the ease with which programs can be transferred. It does not, however, solve the problem due to a number of reasons. The successful application of a complete set of standards implies that they be documented and rigidly enforced throughout the program development, debugging, and improvement. It should be realized that even if such standards are enforced, the difficulty of transferring will still be serious.

II. THE INADEQUACIES OF THE CONVENTIONAL APPROACHES

In the previous section the problem was discussed and several specific approaches which have been tried in the past were described. In this section, in general terms, the basic reasons why the current programming technology does not permit a "solution" to the problem will be presented. We will discuss four basic kinds of difficulties and suggest, in a general way, the kinds of language and/or system design, language and/or system development, and in some cases, research needed to overcome them. The detailed proposals which form the conclusion of this report appear in later sections. The four areas of difficulty are: (1) the lack of facilities for explicit specification; (2) the lack of constraints which would serve to regularize the behavior of programmers; (3) the level and degree of representation commitments and binding required by current languages and systems; and, (4) the lack of specific elements of programming technology or failure to properly utilize the available technology.

1. Explicit versus Implicit Specification

The fact that the specification of many entities is implicit rather than explicit is one of the most important single factors making transfer of programs either difficult or impossible. For example, using machine language to program a function concerned with manipulating the elements of some data set requiring only a few bits for their representation, in the interests of storage efficiency such items are often packed several per word, with the result that the specification of these items ends up being implicit in the shift constants and masks used to fetch or store them. This kind of implicit specification of the allocation and accessing of data, even with excellent program documentation, generally produces a nightmare if modification of item sizes or transferal of the program to another computer is desired later.

Of course there are many languages and/or systems which have explicit data declaration facilities and which handle the fetching or storing of data values via an assembler or compiler generating the detailed coding from explicit data descriptions. With such facilities, changes in data descriptions can be reflected in programs by simple re-assembly or re-compilation. However, even with respect to data, the extant facilities in languages and systems such as COBOL, JOVIAL, PL/I, NSA-ALGOL, CL-I, CL-II, and the like do not provide anything approaching a complete solution.

First, there is the question of the generality of data structures which can be handled. COBOL really only knows about files, records, and fields of records; JOVIAL and NSA-ALGOL only know about "tables" and "arrays"; PL/I has the definition of a data structure within a program rather than as a separate component.

Second, there are many other "facts" about data required by certain programs which almost always end up being handled implicitly. For example, a given data set generally has many different representations. It might appear on one or more input media: cards, tapes, teletype, and so on. It probably has several internal representations: high speed core, back-up store, disc file, and so on. Also, it probably has a number of output representations: printer report, various forms of dumps, a graphical display, and the like. All these forms must be specified explicitly if all the processors which transact with the data set in its various forms are to be transferable. For each representation of a data set there may be a variety of methods of access to a given element. However, present data description facilities generally permit only one access method; the programmer needing (for reasons of some actual or imagined efficiency) other methods of access must arrange for these implicitly using tricks like that of overlaying "two" tables as discussed before with NSA-ALGOL. There is also the question of the data description itself. Clearly, to a data input mechanism, or to a compiler, or to a file maintenance mechanism, this description is data and thus requires a description.

The purpose here is not to write an essay on data and data descriptions. The intention has been to indicate, using data as an example, what is meant by implicit versus explicit description. Indeed, there are numerous other facilities and entities which must enjoy explicit specification. For example, the specification of programs must be explicit so that the compiling of references to them and the linking of programs to other programs and data can be specified and handled. There are also such things as "messages" which are passed between programs, these must be identified and handled explicitly. These "messages" include interrupt signals, various communications with the monitor or the user, program or data status information, and so on.

Summarizing, provision for explicit specification of data, programs, actions, messages, linkage, and the like is a prerequisite for reasonable program transferability. Languages and systems must be constructed in such a way that such explicit specification is possible, but no current languages and systems are entirely successful in meeting this requirement. However, current technology could host such facilities, if the proper point of view or strategy were employed. The betterment of current languages and systems in this regard is primarily a development project. While research in this area is needed, particularly in gaining a better understanding of explicit handling specification of such things as program status, interrupt signals, simultaneous or parallel program execution and control, much could be accomplished within the current state-of-the-art.

2. Regularizing the Behavior of Programmers

The second area for discussion pertains to the excessive complexity and generality of present languages and, particularly, operating systems. It should be noted that this is really more another point of view regarding the first area, that of providing for explicit specification, than a different topic, but exploration of the issue in this way could be fruitful.

In a very real sense, a programmer setting out to prepare a complete program or some part of a larger program has entirely too much freedom. In particular, if a higher level language has been chosen for his use, he is constrained to some degree, since, for many programming tasks, each language induces a "style" to which the programmer might be expected to adhere. In other words, the programs resulting from two different programmers given the same task should be reasonably similar. However, for those parts of the problem for which the language has no explicit facilities or for which the explicit facilities are too general, different programmers often produce entirely different programmatic solutions to the same problem. This difference is strongly correlated with the difficulty of transferring the programs.

The latest round of operating systems is to be particularly noted in this regard. Such systems as OS/360, EXEC-8, and the like are extremely powerful systems, and utilized properly, permit rather complete control of a computer's resources in a large variety of ways. Unfortunately this generality is also attended by complexity, and the result is a system beyond the capacity of all but very knowledgeable programmers to maintain or modify. The casual user, including almost all programmers, typically contents himself with those few facilities with which he is familiar, bending them in whatever way he can to "fit" his current problem. If the subset he "knows" differs from the subset another programmer "knows," programs produced are likely to be quite different. In some way operating systems or detailed data representation and manipulation must be arranged so that there is an explicit "way" to accomplish any particular task.

3. Degree of Representational Commitment and Other Kinds of Binding

Current programming languages mirror very closely many of the facilities available in actual hardware. In one sense this is good, since it might then be expected that programs would "fit" the hardware and enjoy the attendant efficiencies. However, there are also certain bad effects. One is that current languages demand a commitment to data representation before one can even start programming. Once the programmer has decided on a particular representation, be it integers, floating numbers, characters, or arrays of characters, he then has a small repertoire of operations with which he can manipulate these quantities. However, a large portion of many programs have nothing to do with the kinds of quantities which are "natural" to most computers. For example, one fairly common kind of program is best thought of as dealing with "objects," properties of objects, and relations among objects or properties. The "objects" might be programs with "properties" like name, length, status, and such, and "relations" like a sub-program of, calls as a sub-program, and references the data set, for instance. A program concerned with these might be expected to transact with the properties and relations, establishing new relations, inquiring about properties, and so on. If the programmer has to use integers or characters to model these objects, properties, and relations, and to use the ordinary arithmetic and logical operators to specify his transactions, he might be at a serious disadvantage. The necessity of making a commitment to some particular representation, like integers, before he can start "programming" may seriously affect the quality or even the "do-ability" of a program. If the programmer (unusually but wisely) rejects the use of conventional languages and systems and attempts to initially "solve" his problem in terms of the units of data and unit transactions natural to it, he is still eventually faced with the programming task. His "solution" can't be run and must be considered as, at most, a guide to the "real" solution.

An alternative approach might be to supply a programmer with data and operations not so closely tied to computer representations, such things as "objects, properties, and relations," or "set quantities and operations," etc., and permit him to "breadboard" a solution. Once he had a demonstrable solution he could then attend to the problem of "packaging" his solution, contending with the need for a "real computer" as the basic host medium. The results of his "tuning" and "tailoring" operations could also be run and evaluated. In a sense he could view the task of taking what is agreed to be a correct solution and make the "solution" efficient, confident that at all stages he could verify he still had a solution. Parenthetically it should be noted that languages such as AMBIT/G* and language features such as Floyd's Non-Deterministic Algorithms**, if implemented appropriately, might provide such a facility for certain kinds of problems. AMBIT/G permits description of various objects and their properties and relations and prescription of manipulations of them. The specific means of computer representation is deferred. Floyd's Non-Deterministic Algorithms provide a facility which permits deferring the detail of how some particular choice (or move in the play of some "game," of I/O unit, etc.) is to be made. If a particular choice leads to "failure," the program will automatically "backup" and a different choice may lead to "success."

Along with the question of the representational commitment which most current languages and systems demand, there is the related question of the kinds of binding allowed and the times at which binding is permitted.

Typically, most languages and systems restrict quite rigidly the time that various kinds of binding of meaning or value to various entities is permitted. For example, the "meaning" of a procedure to be called (i.e., the number and types of arguments and the number and types of results returned), must be bound at "compile-time" while the "value" (i.e., the actual coding plus any required linkage) is bound at "load-time" or "block-entry" time. Similarly the meaning of a variable (its type, dimension, means of access) is bound at compile time and its value is subject to change dynamically. Getting around the restrictions imposed by rigid bind times is a significant source of "implicitness" in many programs. This rigidity typically derives from some imagined hierarchy of "function" in a system. If the notions inherent in the Extensible Machine*** concept were employed properly, such artificial restrictions would disappear.

* Christensen, Carlos, "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language," Computer Associates, Inc., Wakefield, Mass., CA-6711-1511, 15 November 1967. (To appear in the Proceedings of the Symposium on Interactive Systems for Experimental Applied Mathematics, held 26–28 Aug. 67).

** Floyd, Robert W., "Non-Deterministic Algorithms," J.ACM, Vol. 14, No. 4, Oct. 1967, pp. 636-644.

*** Goodroe, John R., and Leonard, Gene F., "An Environment for an Operating System," Proc. ACM 17th Nat. Conf. 1964.
Leonard Gene F., and Goodroe, John R., "More on Extensible Machines," Comm. ACM, Vol. 9, No. 3, March 1966, pp. 183-188.

4. Programming Technology

Programming, both as a science and as an engineering discipline, is in its infancy. There are a reasonably large number of elements of programming technology which have developed, but there is a general lack of any unified theory or even unified treatment of these existing elements of the technology. A great deal of this is, of course, due to the fact that programming is a relatively new field. Also its rather strong interdisciplinary nature, that is the lines between programming and mathematics, operations research, control theory, logic, and so on, are not well understood. It is most revealing that there are few, if any, manufacturers of computer systems which have a department of programming research. They very likely have departments of research devoted to mathematics, electronics, and solid state physics, and, in some cases, even such specialized fields as cryogenics and magnetohydrodynamics. However, programming research is not yet fashionable, it is typically a sub-department of mathematics. This general situation also carries over into universities and government laboratories and government-supported computer activities. It is only very recently that doctoral programs in programming as such have been established; it is still the case that a far higher proportion of research and development funding is devoted to "hardware" than to "software." This is in spite of the fact that there is now a general understanding that the cost of the software for any particular system is likely to be higher than the cost of the hardware.

It is not within the scope of this report to consider this aspect of the problem in any depth. But the lack of a unified technology for programming is doubtless a principal cause of the difficulties and this situation is not improving very rapidly.

There are a few specific areas where research and development would have a very direct influence on the problem of program transferability. These include:

- The optimization of programs
- Formal models for the syntactic and semantic specification of programming languages
- A theory of systems
- Extensible programming languages
- Extensible operating systems

BLANK PAGE

III. PROPOSED TECHNICAL APPROACH

The current state-of-the-art permits an attack on the transferability problem at three different levels:

A. Appropriate methods of administrative control and documentation for programming development using current languages and systems.

B. Development of extensions to the current language and system base.

C. Creation of a new programming and operating environment.

The payoff at each of the three levels, in terms of cost savings in program transfer, can be estimated only qualitatively - certainly it is least at level A and greatest at level C. Indeed, at level C the problem of program transfer can probably be substantially eliminated in most practical cases.

The next three subsections describe the three levels of attack, followed by a discussion of the level of effort and time frame in which work might be accomplished at each level.

1. Attack Level A: Administrative Control

Since no immediate panacea exists for the problem of program transferability, it is necessary to consider, in this section, what can be done administratively with present systems and languages. The heart of the conclusion reached is to exercise strict control through the adoption of standard practices. It is realized that many of the practices advocated will increase to some degree the cost of producing and running programs. The extent to which these practices minimize transferability cost, balanced against the cost of applying them, will determine which practices merit use in a particular programming endeavor.

These standard practices include such areas as the control and quality of the documentation effort, and the setting of specific and standardized communication conventions between programs, and, for that matter, between programmers. Administrative control must be applied to guarantee that the documentation is both complete and meaningful, and that standardized conventions are observed.

There is a set of administrative constraints that can be applied to the interaction of the program and the operating system. These constraints deal with features of the operating systems which are not available throughout the spectrum of systems on which the program is to run, and might include elaborate filing structures and exotic dynamic loaders or access methods. One technique of enforcing such constraints is to delete from the system manuals all references to the undesirable features and, further, to guarantee that the system report any programs which use such features. Again it is realized that such prohibitions may be costly in both programming time and efficiency. However, it is felt that such techniques would result in appeals to management by programmers who wish to use these "forbidden" features, thus serving to identify and flag these potentially problem uses.

A somewhat similar constraint can be applied to programming languages. Control must be established over the use of language features which are either non-standard or non-transferable. The latter includes assumptions about word size, character representation, and numeric abnormalities. The operation must select that subset of the programming language which is common to the set of machines he wishes to run on, ensuring, however, that this subset represents a useful programming language. Again, the economic pros and cons of including features which are not universally implemented must be considered. In addition, a set of features must be isolated which should be avoided because of either machine-dependent characteristics or non-uniformity among implementations. A prototype of such a list has been prepared for the COBOL language.* (A note of warning: Because a standard exists for a programming language, it should not be assumed that a similar standard exists for the semantics of the language, or for that matter, that specific compilers conform to the standard or are "bug-free.")

Another area which must be controlled is that dealing with the description of the data used by programs. Data description conventions, adequate for the majority of areas, must be developed or borrowed and must be adhered to in all programs which are candidates for transferability. In the future this documented data description will serve as part of the documentation body and, in addition, may form the basis for "automated transferability."

Many of the numerical algorithms used in programming are sensitive to such things as precision, round-off, and truncation, and thereby give rise to a major source of transferability difficulty. Further difficulties arise from the fact that the standard mathematical functions are not uniformally implemented among operating systems. The union of these two problem areas causes much difficulty in the transference of complex numerical procedures.

One way of alleviating this problem is to establish standards for the basic mathematical functions such as SIN, COS, and so forth. These standards should impose constraints on precisions and the behavior of these routines in limiting cases.

With respect to the precision, truncation, and round-off problems a study should be undertaken to categorize, catalogue, and even develop algorithms which are insensitive to these problems within broad bounds. Such a study could result in

* Westinghouse Electric Corporation.
COBOL Programming Tips.
Westinghouse Electric Corporation Management Systems Department, April, 1967.

recommended approaches to particular algorithms for given ranges of precision. In some fields of mathematics there already exist known algorithms which exhibit such insensitivity. Further research should also be undertaken to investigate the effects of round-off on the classic numerical procedures.

This is not necessarily an exhaustive list of controls which can be applied. However, they are the most essential ones. It is important to emphasize that to be useful, such controls must be unconditionally and continuously applied, they must be adhered to even in the midst of last-minute schedule panics, and must be continuously updated during the debugging and improvement stages of the program's development.

2. Attack Level B: Extensions to the Existing System Base

Several operating systems currently, or soon to be, in use can be extended in such a way that ease of program transfer is significantly enhanced. Among these systems are: CL-II (IBM 7090 and 7094), AOSP (Burroughs D825 and D830), MCP (Burroughs 5500 and 8500), MULTICS (GE 645), and OS/360 (IBM/360). The details of extension, and the amount of development effort and time required, will depend strongly on whether one or more and which of the existing systems are used as a base, or whether an operating system is to be developed from scratch. In any case, the resulting system* includes the following facilities and features:

a. Stored Data Descriptions

Data sets stored in the filing system, as well as data existing in storage only during execution of a program, conform to data descriptions which exist in the filing system and are accessible to any program executed by the system, at any time. The description facilities combine description at the data item or field level (as in the CL-II system or in COBOL, but further elaborated), the record level, and at the data set level (as in OS/360). **

b. Derived Descriptive Information

While the stored descriptive information will be explicit to a much higher degree in current systems, this need not place an increased burden on the users of the systems. To the greatest extent practicable, descriptive information will be derived by the system from information normally in source programs and by extensive provision of facilities for declaration by default (that is, standard assumptions will normally be made by the system in the absence of explicit declarations by the user). ***

* Where the word "system" is used below, it usually refers to the common characteristics of a number of environments between which programs may be transferred with relative impunity.

** See Appendix II for a more detailed discussion of data description.

*** Description of the data descriptions themselves — "data meta-description" — is a feature to be treated at level C only. This subject is discussed in Appendix III.

c. Use of Data Descriptions by Compilers

The compilers in the system will use the stored data descriptions (either by use of a preprocessor or by extensions to the existing compiler, as appropriate) rather than require that descriptive information be submitted as part of the program to be compiled.

d. Other Uses of Data Descriptions

Since the data descriptions are stored in the filing system, they are available for use at other than compile time. The system contains programs which use the data descriptions for converting data between its external and internal forms, and between different internal or different external forms. These programs provide, or can be extended to provide, a rather general data conversion and reporting facility.

e. Mechanized Facilities for Administrative Control

Descriptive facilities within the filing system satisfy the requirements for administrative control of programming development efforts at the level of the best current practice. This is based on the ability to store, process, and retrieve information used both for project control and for documentation purposes.

f. Program and Text Editing

Facilities will be provided for both context and line editing of programs and arbitrary character text (e. g. , program documentation). Line editing refers to the ability to insert, add, delete, or replace one or more lines (or records) using line number references for control of the scope of each action. Context editing has more recently appeared in interactive systems; control of the scope of an action is exercised by searching the text for occurrences of some fragment of text (e. g. , insert following the first occurrence of the word "apothegm").

g. Operating System Interface

The functional capabilities of the language included will imply, in turn, that certain functions be provided by the operating system. The operating systems used must be made uniform in this respect by definition of a standard functional interface and extension of the systems to meet that interface where necessary. A representative set of functions is contained in Appendix IV.

h. Minimum Hardware Standard

At least in the area of amount of main storage and direct access storage, minimum hardware configuration standards must be set to permit transferability at acceptable performance levels.

This effort will produce an actual operational system which offers a workable, though partial, solution to the problems of transferability. But it will not afford a complete solution, nor a long-term one. Given the inevitable growth in size and complexity of computing systems and the problems assigned to them, the replacement of present systems as they become obsolete will still be a major task.

3. Attack Level C: Advanced Transferability Environment

Thoughts in the preceding sections have been bound by consideration of using existing operating and /or language systems as a base. In this section this restriction is eliminated.

An environment is considered herein in which attention is focused on first principles needed to guide the design and development of operating environments directly conducive to eliminating transferability as a significant problem. This essentially comprises adherence to the software technology and its application in an engineering sense to this problem.

To mount a long-term attack on the problems of transferability, the development of an Advanced Transferability Environment is proposed, to produce a running prototype during the life-time of the operational systems discussed in the previous sections, and to eventuate in the production of new operational systems.

In essence, the Advanced Transferability Environment will be available for generalizing, extending, and combining the best of today's thinking of its component parts (executive programs, filing systems, language processors, etc.). The requirements for transferability are not very different from the present tendencies of advanced programming systems. Transferability requires no great new inventions (they will be welcome, if they occur), but rather a thoroughgoing, consistent application of good design principles.

It follows from this position that transferability itself is not expensive when approached in this fashion. Those things which abet transferability: programming system facilities, styles of programming, universes of discourse, each have value in other aspects of the whole programming process.

The precise structure of the Advanced Transferability Environment would be determined during a design study. This effort should be started concurrently with, or within a few months of, the design work of the previous phase. The Advanced Transferability Environment designers should maintain an alert awareness of the history and results of the previous levels. This sub-section discusses some of the general properties of this environment.

The Advanced Transferability Environment will be manageable in size, i.e., "small." It will not be designed as a test bed for every possible software idea, but will strive to be an elegant configuration of the selected designs. Though it will include multi-programming and interactive access as a matter of course, the design will not be deformed by any necessity to service a large number of terminals with severe time constraints.

It will make extensive use of descriptions. To every object in the system there will correspond a description also in the system. When appropriate, these descriptions will be produced automatically as a by-product of input processing, but where necessary or desirable, the user will be required to produce them. The role of thoroughgoing explicit descriptions in transferability has already been discussed. The previous sections have mentioned the uses of data descriptions in compiling, I/O operations, run-time linkages, symbolic debugging, and the like. Thus any "overhead" associated with the maintenance of descriptions is chargeable only in small part to the transferability requirement.

Further, the system will be especially intended to develop and experiment with techniques of meta-description. (See Appendix III for a more technical discussion of the use of descriptions in transferability). Within a system using descriptions, the description of an object includes information about the object's type, allocation requirements, binding requirements, accessing procedures, and so forth; but the way in which the description contains - encodes - this information remains implicit in the logic of the processes which employ the descriptions. The purpose of meta-descriptions is to make explicit the rules for interpreting descriptions in line with the general principle that the way to program transferability lies in the direction of rendering explicit and processable many aspects of programming which today remain hidden. It would be impractical and undesirable to require all systems to use identical object description formats and conventions, it is much more feasible to allow any description technique to be used, provided that it can itself be described in a standard fashion.

It will be highly articulated: the system's armamentarium will not consist of a few large self-contained pieces (compiler, filing system, I/O handler, etc.) which live in different worlds, but rather of many smaller, better-defined packages which will be brought together as required to meet a particular requirement.

The structure of the executive will follow the design principles of the Extensible Machine*, consisting of an irreducible core surrounded by an ever-growing number of service functions.

The language processors will "know" about the executive services and how to use them. They will no longer have to "compile around," or duplicate the functions of the executive.

* Goodroe, John R., and Leonard, Gene F., "An Environment for an Operating System," Proc. ACM 17th Nat. Conf. 1964.
Leonard, Gene F., and Goodroe, John R., "More on Extensible Machines," Comm. ACM Vol. 9, No. 3, March 1966, pp. 183-188.

A "compiler" will be essentially a driver routine, summoning whatever constellation of task-handlers is required to process the particular source language.

All languages employed within the system (e.g. , for job control, problem statement, console transaction, etc.) will be derived from a common base language by a process of explicitly-defined extensions. *

Again, these features are in the direct line of current development, and involve no excessive overhead attributable to transferability.

* Cheatham, T.E., Jr., "On the Basis for ELF – An Extensible Language Facility," Computer Associates, Inc., Wakefield, Mass. (Submitted to 1968 Fall JCC).
Fischer, Alice E., and Jorrand, Philippe, "BASEL: The Base Language for an Extensible Language Facility," Computer Associates, Inc., Wakefield, Mass. (Submitted to 1968 Fall JCC).

IV. SCOPE OF PROPOSED EFFORT

In the preceeding section, the approaches recommended at levels A, B, and C have been discussed. In this section, the corresponding manpower and cost estimates are presented.

The objective of work at level A is to generate administrative control procedures and doctrines which must be strictly adhered to by all people associated with designing and implementing a program or programming system. These controls must be enforced by a czar who has final approval rights over the contracts under which the programming is sponsored.

A group of from six to twelve people working for a total of twelve months can produce such a set of control procedures for three different computers and operating systems and two different higher level language systems. Nine of these twelve months are required for design specification. Six months are required for review and revision of the ensuing specifications although three months may be overlapped with the design period. Within an additional three-month period, the services of three people would result in the formulation of a pattern which could be applied to other similar problem environment.

COBOL and JOVIAL are the recommended languages for the attack at level A. The IBM 360/50 using OS/360, the Burroughs D830 using AOSP, and the UNIVAC 1108 using EXEC8 are the recommended host systems. In the event that more specific guidance could be provided about particular military operational environments other equivalent machine and/or languages could be substituted.

The objective of the work at level B is to generate a transferability environment through the extension of the existing operating system/language system base by identifying deficiencies with respect to particular existing systems, defining how these deficiencies may be eliminated, and then implementing the resulting changes.

Part of the design problem at this level is to identify which extensions can be incorporated into any system with a high probability of success within a two-year period from the time they have been defined. Extensions not falling within this category should be deferred for consideration to level C. Considering the same three host machines and two higher level language systems as at level A, the design phase will require the services of three people for three months in identifying the deficiencies to be corrected, and of six people for six months to define the necessary extensions by which the corrections may be effected in particular systems. The cost of implementing the extensions to a particular system can be reliably estimated only at the conclusion of the design phase. This is primarily a result of the very considerable variations in the extent to which any existing operating systems already supports the level B features.

The objective of the work at level C is the development of an operating prototype of the Advanced Transferability Environment which will be exercised in a test environment to evaluate the success of higher risk and/or longer term concepts and ideas required to eliminate the problem of program transferability. The services of ten people working for a total of 36 months are required for the development of the operating prototype: four people for the design phase and six people for the implementation phase of the effort.

A suitable machine facility must be provided exclusively for purposes of the Advanced Transferability Environment. This facility must be a clean, contemporary computer system, including peripheral equipment. The estimated cost of a system suitable for this work is $350,000.

Co-ordination and direct control of the efforts at levels A, B, and C should be effected by a small steering committee of people with nationally recognized systems and/or language expertise.

V. CONCLUSIONS AND RECOMMENDATIONS

The current state-of-the-art in programming technology has advanced to the point at which it is feasible to successfully attack the problem of program transferability. Three levels at which such transferability can be realized have been identified.

The degree of possible transferability is related to the restrictions that one imposes, that is, restrictions to:

A. current operating systems,

B. extensions to the current system and language base,

C. re-engineering of operating systems and languages.

Effort should be started at each of these three levels as discussed in the previous section of this report.

Work at level A is of particular significance only when: (a) a specific environment is identified in which the energies of the available people with the proper talent can be focused on the details of the defined problem; and (b) the time frame is such as to require interim gains before the results of work at levels B or C can be realized.

The importance of undertaking work at both levels B and C should be emphasized. The extended system envisaged at level B contains features which have proven value and which are well within the state-of-the-art to implement. Hence, this system would, in effect, remain available for continued production use as extended facilities are added to the existing operating environment. Work at level C, on the other hand, should be undertaken in order to provide a vehicle for making a significant departure from the current system base. This would lead to an operating prototype rather than a production system. Experience in its use would result in a production system for replacement of the level B system. Hence level C provides a longer term operational solution to the problem of program transferability.

APPENDIX I

FACTORS INFLUENCING DIFFICULTY IN TRANSFERABILITY

There are many differences in environment that can impede program transfer. The principal difficulties may be grouped into the following broad categories:

1. Differences in amount of main and auxiliary storage available with different systems and in different installations.

2. Differences between representations of different generic data types (e.g., floating scaie representations of real numbers).

3. Differences between monitor services and interfaces.

4. Differences in library content (availability of different programs and differences in algorithm, e.g., elementary mathematical function evaluation routines).

5. Existence of local system extensions and modifications.

6. Differences in organization and representation of data sets intended for use in inter-system communication.

7. Other machine differences affecting data organization and representation, e.g., the data aiignment restrictions in IBM System/360 and word length differences.

BLANK PAGE

APPENDIX II

DATA DESCRIPTION

Data descriptions are in use now, but in a less sweeping way than we contemplate, e.g., the COBOL Data Division, the PL/I DECLARE statement, the JOVIAL COMPOOL, data descriptions in CL-I (OSO) and CL-II (ODES) and in OS/360 (DSCB), as well as elsewhere. In particular, CL-II has a quite detailed description below the logical record level while OS/360 has no detail here but quite a bit above this level.

The study group's discussions concerning data description are summarized in the following apothegms:

- What are transferred are sets of programs and data (not just programs). Source programs have information which is consumed by the compiler and does not appear in their object form. In addition, there is yet other information which is not explicit in the source form.

- Query: Will there always be, even in principle, an undescribed, implicit level of structure (and/or semantic content)? The group feels that this is the case.

- A good example of explicit vs. implicit data description: The data is divided by the constant 2688. Implicit in the program is the knowledge that it is being converted from furlongs per fortnight to miles per hour. An explicit description would have stated the units or a scale factor.

- How compulsory should it be for a programmer to use explicit description, when there is a choice? The principle to be followed here seems to be that the system should make it easy for him to do the "right" thing, rather than impossible to do otherwise. This implies, of course, a substantial amount of explicit description supplied by the system by default.

- Future languages and systems should make much more use of explicit data descriptions, stored separately from the data. This follows from the tenor of the discussion at the group's meetings, and amounts to adopting the philosophy of the CL-I and CL-II systems.

- Data descriptions must be transferable. This follows from the first apothegm.

- Query: Should data descriptions be described? Yes, and this follows from the points developed below.

- A data description is itself data.

- There must be different representations for a data description. For instance, tape labels on 7 and 9 track tape and disk pack labels will be different representations. There may also be distinct external and internal representations of a given data description, e.g., the DD statement, DCB macro and DSCB in OS/360, the latter being the internal form.

- It will be important that a system have an encoded, internal representation for reasons of efficiency. The representation should be, however, standard within a given system.

- The proper route toward transferability of data descriptions is via a single standard meta-description (that is, the data description of data descriptions). (See Appendix III). The two possibilities of an implicit meta-description or of a single standard data description were dismissed on the basis of previous propositions.

APPENDIX III

DATA META-DESCRIPTIONS

The notation adopted here is that of CL-I; the notation "A/B" is the name of instance B of data set A. All instances of A have the same description, and this is the instance A of data set OSO (for "Object Specification Object"). OSO/A is the data description of data set A. In particular, OSO/OSO is the meta-description, or description of data set OSO.

Suppose there are two systems, OS_1 and OS_2, and two instances of HENRY produced on OS_1. Then,

$OSO_1/HENRY_1$

$HENRY_1/1$

and $HENRY_1/2$

exist. The following types of transfer can be contemplated:

1. Transfer $OSO_1/HENRY_1$ to OS_2 in order to produce future instances of HENRY - i.e., $HENRY_2/3$ ff.

2. Transfer current instances of HENRY to OS_2 for input.

In the first case a way is required to get $OSO_2/HENRY_2$, and this involves the use of OSO/OSO_1, according to our doctrine calling for a standard meta-description, together with OSO/OSO_2 and a program run on some system which understands the standard meta-description OSO/OSO.

In the second case, there are two possible routes:

a. Translate (e.g.) $HENRY_1/1$ to $HENRY_2/1$, or

b. Use $HENRY_1/1$ interpretatively on OS_2.

2(a) might be a direct translation or a double translation via a "standard" form of the data, described by OSO/HENRY. 2(b) would imply interpretative use, on OS_2, of the description OSO_2/HENRY. This description, as was the case with the data instance in 2(a), might go through a double translation:

$OSO_1/HENRY_1$ to $OSO/HENRY_1$ to $OSO_2/HENRY_1$

Which one of the above methods of transfer should be used depends, no doubt, on individual circumstances.

OSO_1 and OSO_2 have more information that OSO - in terms of our previous discussions, they have made commitments to specific representations on the two systems, whereas OSO has no such commitment (except to a pseudo-computer which "understands" the standard form of descriptions and data). Thus, where a piece of data is described simply as

```
ITEM INTEGER;
```

in OSO/HENRY, additional description of "INTEGER" is required for various systems. For instance, we might see descriptions like:

```
   INTEGER TYPE RADIX(2) SIGN(COMP, 2) LEFT(31) RIGHT(0);

   INTEGER TYPE RADIX(2) SIGN(MAGN) LEFT(35) RIGHT(0);

or INTEGER TYPE RADIX(10) SIGN(COMP, 9) LEFT(11) RIGHT(0)
     DIGITS(0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100);
```

for the System/360, 7090, and UNIVAC I, respectively. That is, even in the "standard" meta-description, there is some level of description which is implicit (although not necessarily that level implied by this example) in the design of the interpreter of that description. We believe this to be inescapable, and no disaster at all.

Another note to be made is that descriptions will, in general, require procedural elements - for instance, to describe various storage mapping functions.

APPENDIX IV

OPERATING SYSTEM INTERFACE

This Appendix lists a set of operating system interface functions obtained by comparison with those in the Burroughs MCP (B5500), the IBM OS/360 (System/360), and the RCA TDOS (Spectra 70). This is intended to be a representative list, and not necessarily precisely the set that would appear in the attack at level B.

- CALL is the function which causes a procedure to be executed against a given argument list. Two options exist: TASK causes execution as a subtask of the task invoking CALL, while ON requests deferred execution, in the style of the PL/I ON statement or the OS/360 "type 3 exits."

- RETURN causes exit from a procedure.

- TERMINATE causes termination of the referenced task.

- SIGNAL signals the occurrence of an event, thus causing triggering of a CALL with the ON option.

- WAIT causes the task issuing it to suspend execution until some specified event, or combination of events, has occurred.

- POST specifies that a given event has occurred, releasing any WAITS which are thus satisfied. (Query: Are SIGNAL and POST not essentially the same function?).

- PRIORITY changes the scheduling code for the specified task. (Simple priority scheduling is not necessarily implied here).

- PURGE causes execution to be cancelled at all levels of control up to (but not including) the one which set the stated ON condition.

- ENQUEUE and DEQUEUE force sequential, non-current access (by tasks) to a given system resource, usually data, by suitable use of the WAIT and POST functions.

- CHECKPOINT and RESTART allow a task, or group* of tasks, to be checkpointed for possible later restart.

* Many systems, including CL-II, MULTICS, and OS/360, allocate system resources to a group of tasks rather than to each task individually.

- ALLOCATE is a storage (main or auxiliary) space request, matched by FREE.

- ASSIGN and UNASSIGN allocate and free input/output devices and request the operator to do volume mounting and dismounting.

- CATALOG and SCRATCH enter and release data sets into or from the filing system.

- DATE and TIME report system data to the calling task. TIME may be absolute or elapsed execution time.

- INTERVAL sets a timing interval, either real time or relative to task execution time, as an ON condition for the invoking task.

- OPEN and CLOSE are the primitive form of the functions preparing for use of a data set on auxiliary storage and later releasing it from use.

- INOUT is the primitive form of an input/output request; its completion is signalled by a POST issued by the operating system.

The input/output functions should be specified at a higher level than INOUT; the latter typically is extremely machine-dependent in its detail. Many systems, in fact, establish interfaces at both the physical block and logical record levels as well as at the INOUT level. It may further be noted that INOUT is merely a disguised form of CALL with the TASK option. The difference is essentially that a different processor class (input/output channel rather than CPU) is being invoked.

The functions allowing the filing of data set descriptions and access to them are included, in a disguised form, in the CATALOG/SCRATCH, OPEN/CLOSE, and INOUT functions above.

Finally, it must be noted that although the above lists operating system functions in an essentially environment-independent manner, their exact expression in any given system will be at least somewhat different from that in another system. In order to provide transferability at the operating system interface level, it will be necessary to absorb such variations in the way code is compiled for each environment.

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Rome Air Development Center (EMIIF)
Griffiss Air Force Base, New York 13440

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
N/A

3. REPORT TITLE

PROGRAM TRANSFERABILITY STUDY

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
In-House

5. AUTHOR(S) *(First name, middle initial, last name)*

George H. Mealy
T.E. Cheatham, Jr. (Computer Assoc.)
David J. Farber (Rand)
Edward Morenoff (RADC)
Kirk Sattley (Computer Assoc.)

6. REPORT DATE
November 1968

7a. TOTAL NO. OF PAGES
32

7b. NO. OF REFS
6

8a. CONTRACT OR GRANT NO.
N/A

b. PROJECT NO.
5581

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
RADC-TR-68--341

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Rome Air Development Center (EMIIF)
Griffiss Air Force Base, New York 13440

13. ABSTRACT

This report treats the problem of transferring programs from one operating environment to another with the expenditure of a small fraction of the initial programming development time and cost. Programs considered range from quite small ones, such as routines for evaluating arctangents, to large and complex systems, such as compilers, data management systems, or command and control systems. The initial and final environments may be slightly or highly dissimilar with respect to machines, machine configuration, or operating systems and languages used.

The report is the result of a series of two-day meetings over a six-month period among a group of consultants under contract to RADC, a representative of the Rand Corporation, and a representative of RADC. This group was formed at the request of the Department of Defense (DDR&E), Dr. James Ward, for the purpose of determining what action, if any, could be taken in the immediate and far future to reduce the problems associated with transferring computer programs among computers of different size and manufacture. The study devoted itself only to the software aspects of transferability. There is a hardware aspect which has yet to be addressed and where some promising approaches need investigation. This report was presented in June 1968 at a Department of Defense meeting attended by representatives of DDR&E, DCA, ARPA, Army, Navy, Air Force, Marines, Joint Chiefs of Staff, and the Intelligence Community.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
COMPUTER SYSTEMS PROGRAMS						
PROGRAMMING LANGUAGES						
TRANSFERRING						