MAN-TO-MACHINE COMMUNICATION AND AUTOMATIC CODE TRANSLATION

A. W. Holt and W. J. Turanski*

Summary

The ACT system is a programmed adjunct to a general purpose computing machine (currently considered for the MORDIC) whose purpose is to facilitate the initial encoding and subsequent application of specific "code-to-code translation" procedures. The phrase "code-to-code translation" is primarily meant to cover the conversion of problem oriented pseudo-codes into machine code equivalents. The ACT system provides for the "housing" of many distinct translation procedures - carrying algebraic, data-processing, simulation languages, etc., into any of a variety of computer codes - within the bounds of a single controlling system.

The programmed components of the ACT system include:
1. An allocation interpreter - AI
2. A "core library" of basic translation functions
3. A general translation library whose content varies with use of the system.

The allocation interpreter permits the formation of large translation programs out of large (and small) pre-stored sub-programs. Without greatly impeding computation speed, AI determines internal and external storage allocations for all space-taking portions of information as part of problem computation. This facility is shown to be essential to the construction of ACT, and perhaps widely useful in other types of programs.

The core library contains translation functions which play a fundamental role in a wide range of code-to-code translations. Included are many functions which usually play an important part in machine assembly programs. Other functions provide for general library handling, code reordering, etc.

Also associated with ACT is a writing convention called "canonical form" which governs the presentation of data (i.e., programs in source code submitted for translation). Canonical form, while flexible enough to permit highly diverse problem codes, still standardizes all those signals which the basic ACT components decode.

The ideas presented herein were developed at the University of Pennsylvania under contract to the U.S. Army Signal Corps in connection with the FIELDATA family of data-processing equipment. Although reference is made to the FIELDATA family in the presentation below, the ideas should be applicable to other families of equipment. Because of the prevailing tendency towards modular design in computing systems, such families are rapidly becoming more numerous. 1,2,3

General Considerations

The Problem

The ACT system is designed to reduce the man-to-machine communication problem in operating the Army FIELDATA system. The fundamental assumptions with regard to this communication problem are:
1. that the FIELDATA system employs a variety of different digital computers;
2. that the computational problems to be solved with these machines belong to many different "problem areas" - e.g., trajectory computation, estimation of statistical parameters, information retrieval, language translation, etc.;
3. that the exigencies of time and circumstance may make it necessary to run any of the intended range of problems on almost any of the intended range of machines;
4. that the overwhelming majority of Army personnel concerned with these problems will not be computer programming experts for any (let alone all) of the intended computers; and
5. that the efficiency of the entire FIELDATA system hinges primarily on the speed of problem solution - i.e., the total "lapsed time" from problem statement till solution delivery. Therefore, the speed with which a problem can be correctly encoded for computer solution may matter a good deal more than the

* The work described in this paper was done for the U.S. Army Signal Research and Development Laboratory, Fort Monmouth, New Jersey, as part of a "multiple task" project at the University of Pennsylvania. The authors were engaged in this work by sub-contract of Remington Rand Univac, Philadelphia, Pa., to the "Common Programming Language" task of the above project.
11.1

efficiency of the machine program with which the computation is finally performed.

Existing Techniques

The man-to-machine communication problem has been widely recognized and seriously attack­ed in many of its special instances. Most of the present day "automatic programming systems" are the result of manufacturers' attempts to narrow the gap between "the problem", which ex­ists in the minds of technical specialists, and "the program" which "causes" the computer to compute a solution. The invention and con­struction of an automatic programming system normally involves two major steps:

1. The invention of a special problem code (often referred to as a "pseudo-code") which:

   (a) is compatible with already es­
   established habits of expression within some technical specialty, and
   (b) is sufficiently formal and complete to permit automatic conversion - or "transla­tion" - into a computer-coded program.

2. The construction of a special trans­

   lating routine which accepts as input data problems represented in problem code and pro­

   duces as output corresponding machine programs.

Most commonly, the translating routine is designed to accomplish its entire task before computer solution of the stated problem begins. Thus, bringing a problem to solution with the help of an automatic programming system is typically a two-stage process - translation followed by computation. (See figure 1).

Automatic programming systems which func­tion as in Figure 1 depend upon three codes: S, V, T. They are usually produced by manufac­

        turers as adjuncts to particular computers.

Thus, "Computer V" and "Computer T" of Figure 1 are almost always one and the same. If, for example, the manufacturer is putting a "scientific machine" on the market, he will usually design an algebraic-mathematical pseudo­

code to go with it, and he will prepare the translating routine to produce programs for itself on that computer.

Past experience with the building of auto­

matic programming systems has brought to light several facts which have a strong bearing on the FIELDATA man-to-machine communication prob­

lem:

1. To produce an automatic programming system (S, V, T) normally requires major pro­

   gramming effort - measured in tens of man-years.

2. Such systems are usually very difficult to modify in response to changes in the specifi­

   cations for S, V or T. This difficulty arises because the translating routine, within which the specifications for S, V, and T are "frozen", is usually a large computer program from which it is impossible to 'factor' the components, S, V, T.

3. As a consequence of the immediately preceding point, it is further true that a translating routine which has been programmed for the triple (S1, V1, T1), does not give a substantial head start for producing the routine for (S2, V1, T2) or for (S1, V2, T1).

FIELDATA Requirements

The requirements for FIELDATA computing, as set forth at the beginning of this intro­

duction, make it clear that:

1. The operation of the FIELDATA comput­ing facility will require a large variety of source codes available for easy use in a rich variety of technical specialties.

2. A given source code will have to be translatable into any one of several target codes.

Some more detailed discussion of both points follows.

Problem Code Flexibility

It is a well known fact that each technical field has associated technical "jargon" - special vocabulary as well as special forms of expression - which belongs to it and it alone. This specialization of language resembles the specialization of computers for particular purposes. In general, special computer problem codes for statisticians, physicists, psycholo­

rists, business administrators, operations researchers, etc., have not been developed be­

cause the art of automatic programming is young and the building of new systems very expensive. If the cost of translating routines were not an issue, it would be desirable to provide each technical field with coding conventions which most nearly suit its established habits of expression.

Quite apart from making provisions for presently fixed habits of expression, the cre­

ator of a translating routine must also note that habits of expression change with time and circumstance. As the constellations of problems to be solved undergo change, so do the related language habits. Old words are put to new uses; new words are invented; abbrevi­

ations are created to refer to oft-repeated combinations; previously disjoint sets of nota­

tions must be combined, etc. Thus, expression forms change in response to changing needs. In many important cases the expressions lead to computable problems. The speed with which answers appear depends heavily on the amount and kind of hand "recoding" (or "encoding") of the original problem statement before the computer can "take over" and ultimately execute the proper se­quence of computer steps to achieve a prob­

lem solution.

Computers - Variety and Compatibility

No matter how similar two computers may seem, they always turn out to be distressingly different in relation to some problems. Consi­

der, for example, two computers, C and 2C, which differ from each other in no respects other than that 2C has twice the internal storage capacity of C. For all those problems easily accomplished.
by a single run on C, Computers C and 2C are fully equivalent. For those problems which involve more than one run on C - or perhaps the use of overlays - Computer 2C is equivalent to Computer C in only the sense that those originally programmed for C will also run on 2C. Thus, for such problems, a major task of conversion from one computer to another may be encountered if

1. the problem was originally programmed for 2C, or
2. the problem was originally programmed for C but must be made to run efficiently on 2C.

So-called "compatibility" between machines (in the sense of very similar or identical instruction codes) does not guarantee ease of program conversion from one computer to another. Only small problems will convert readily, while big problems will offer difficulties not much less formidable than when "incompatible" machines are considered. Although the FIELDATA computers are being designed with mutual compatibility in mind, significant differences between one translating routine \((S_1, V, T_1)\) and another \((S_2, V, T_2)\) must be expected.

The need for many source codes and the inclusion of many machines (and many configurations for each machine) in the FIELDATA system makes the normal automatic programming approach economically unfeasible. To go from a source code to a machine patently requires a prohibitive large number of translation routines - a prohibitively large number when the effort for each such translating routine is considered. Even worse, if one had gone to the labor of constructing these many translating routines, one would still be faced with an unmanageable chore in keeping them up-to-date in response to changing requirements.

**Code-to-Code Translation**

The study of many problem oriented source codes and their translation into computer code reveals that various translation processes have many important common elements. Often commonness between translation processes can be found by examining their respective constituent parts. For example the translation of \(S_1\) to \(T_1\) - briefly \((S_1, T_1)\) - and the translation \((S_2, T_2)\) may both employ the same assembly system for the terminal portion of the translation. Sorting is a frequent common sub-process; at a finer level of analysis, table look-up, list forming, subdividing a "sentence" into "words", economizing storage locations, etc., have frequent occurrence.

Another kind of commonness frequently found between two translation processes is the employment of a common control sequence with differences in regard to what is controlled. This is illustrated by generator systems (such as FLOWMATIC) in which the individual generators are properly regarded as sub-translators subject to an over-all controlling scheme. Two quite different pseudo-codes might be handled by the same controlling scheme with differences only in the library of generators being controlled.

These examples indicate that code-to-code translation is coherent area of computation for which programming aids may be prepared. These programming aids (whose nature and form will be discussed below) will:

1. Reduce by large factors the effort required to produce an operating program for the translation \((S, V, T)\).
2. Similarly reduce the effort required to modify existing translation programs in response to changed specification for \(S, V,\) or \(T\).

The ACT system is designed to be a programming aid such as discussed above. Once constructed, it should have a far reaching effect on the manner and effectiveness of FIELDATA utilization. In working on some particular class of problems - or even just one large problem - programmers will find it convenient to refine their tools of program expression as they go - by modifying or adding to some previously available set of conventions. Thus code or "language" development will go hand in hand with problem solution, just as is normally the case in technical or scientific work without computers. Furthermore the ACT system should greatly facilitate the re-translation of a particular source code, originally translated to one machine but now required for another.

**The ACT System**

The ACT System is a collection of programmed devices which loosely fit together to help in the implementation of specific translation aid.

Also associated with the ACT System are two sets of writing conventions; one set governing the presentation of translation procedures - to be called "ACT translation procedures" - and the other to govern the presentation of data (i.e., source-coded programs submitted for translation.) This is analogous to computer systems, for which data and programming conventions must always be specified. A complete description of ACT will therefore include:

* This is almost exactly the problem to which a proposal named UNCOL (sponsored by SHARE and described in ACM Communications, Vol. I Nos. 6, 9) is addressed. In our opinion the UNCOL proposal, while aimed in a valuable direction, is practically unworkable. For more details, see section on General Translation Library below, also a letter from George H. Mealy, Switching Systems Dept., Bell Tel. Lab., Inc., to the members of the UNCOL committee, dated Feb. 3, 1959.

** Some of these components have importance and usefulness above and beyond their function within ACT. This is exactly comparable to parts of a computer (such as memory units or tape transports) which can be integrated into many different systems.
Figure 2 is based on this conception.

When a call for a new phase is encountered, the calling phase is suspended and the called phase initiated. When the called phase terminates it may either be held for re-use or dropped, depending on the manner of call.

Aside from special AI macro instructions having to do with space allocation (including new phase call) EM code contains ordinary computer instructions with sequence and-segment-relative addresses. These addresses are converted by a program loader into fixed form when the phase is first encountered during program operation. If a phase is held for reuse and then re-activated, it is returned to memory to the same set of locations to which it was originally assigned. Between uses of the same phase AI will have taken care not to assign any newly developed information which this phase requires to locations causing conflict with old information already belonging to the phase.

Although phases of EM-coded programs will normally be functional units (corresponding to a box on some flow-chart description), the converse need not be expected. The only use which is made by AI of the knowledge that a program is subdivided into phases with particular interconnections is to temporarily dump some information in favor of other information which is now required and which could not otherwise be accommodated. Therefore functional units which in their coded form, occupy very little space in the various computer media are not worth treating as separate phases. The bookkeeping involved in their handling is not worth the space return which they can bring.

The power of EM may be said to derive from the following fact. In the case of EM programs the total space requirements of the program (for external and internal space) have been broken down into space requirements for individual program phases (where different phases may overlap by requiring the same information). At any one moment in computation time only one phase is operating and hence only its space requirements must be satisfied. Any parts of other phases which do not overlap with the active phase can be dumped and later restored if necessary. But the greater the amount of dumping and later restoring the slower the program operates. If, therefore, there is enough memory space to permit the assignment of phases side-by-side the program will run correspondingly faster. In this way EM is able automatically to trade time for space depending on the total available storage of the computer and the sizes of various data lists which may have grown in course of computation.

The Core Library

The core library consists of a collection of sub-programs - called translation procedures - which have near-universal importance in translating pseudocodes into machine codes. (Many of these functions derive from a single fact about the relation of source to target: namely that the information in the source code is packaged by time of generation, while in the target code...
it must be packaged by time of use).

The writing conventions - called "Canonical Form", abbreviated "CF" - governing source-coded programs submitted for translation, are entirely related to the core library. All of the signals for which CF provides are decoded by one or several procedures in the core library. Conversely, all those translation procedures which only deal with signals specified by CF belong in the core library. The following is a partial list of functions fulfilled by the core library procedures.

Coding Item Handling - Each CF item of pseudo-code (such as a FORTRAN formula, an assembly instruction, a FLOWMATIC sentence, etc.) has a formal structure which provides for standard item features such as: (a) an item identification number (b) a symbolic item name (c) item type indications. In addition, after some preliminary editing, the body of the item may be recognized to consist of a sequence of symbols. The core library provides CF item handling functions which allow specific translation procedures to gain access to constituent portions of items by means of "item dissecting" commands. Similarly, if a translation procedure produces CF coding items as output, they may be assembled from their constituents by use of "item assembling" commands.

Patch Interpretation - CF coding items may be generated (by programmer or generating routine) in some order wholly different from the order in which they will be next required. Item rearrangement may be coded into the original item sequence by use of the item identification numbers as well as by use of special "patch items". Core library functions will cause item re-ordering, thus interpreting the "patch" intentions expressed in the original item sequence.

Symbol Interpretation - By "symbol" is meant the smallest meaningful constituent of a coding item. Thus, the computer instruction "CLA B 5J2I" is composed of three symbols. Symbols may be of various types from the point of view of what interpretation they require during translation. Some symbols, for example, are absolute and require no interpretation; other symbols are "dummy" and require replacement by specified expressions; yet other symbols refer to named coding items - i.e., are "addresses" - and will require address interpretation. The core library contains a variety of procedures to effect symbol interpretation.

Type Interpretation - CF coding items may contain marks to indicate their specific type. Thus, for example, a FORTRAN DO-statement and a dimension statement might be distinguished as to type; or a statement to be interpreted by FORTRAN as opposed to a statement to be interpreted by SAP may be distinguished by type. In general, type indications always mean that the items thus categorized, are to be respectively delivered to different translation procedures for treatment. Core procedures are used to deliver CF coded items to appropriate translating routines in proper priority sequence, according to type indication.

Call Interpretation - One of the frequent mechanisms employed in code-to-code translation is the adjoining of one or more batches of coding items, taken from a library of such "batches", to the coding items which already belong to the source coded routine being interpreted. A good example of this is the selection of subroutines from a library. After the subroutines have been picked up and (with some adjustments) adjoined to the routine undergoing translation, this latter routine has been 'expanded' by the inclusion of new coding items. Special items in the source coded routine, called "call items" tell what particular batches are to be selected from the library for inclusion.

In the case of subroutine selection, call interpretation may be carried out through a hierarchy of levels. The subroutines selected for adjoining to the original item sequence, themselves contain further call items requiring (for their interpretation) the selection of yet other library routines for inclusion. Thus the original source coded routine contains "First level" calls; the subroutines thus selected may contain "second level" calls, and so forth, for nth level calls. 1

Call interpretation may be usefully applied to a variety of cases which have nothing to do with subroutine selection. For example, a routine coded in FLOWMATIC might be said to "call" on file descriptions from a library. The interpretation of such calls has much in common with subroutine selection. To go one step further: if there were a number of different files which all involved the same item description, then there might even be "second level" calls - the routines calling on file descriptions, and the file descriptions calling on item descriptions. Core procedures are used to carry out hierarchic fetching and adjusting of library information.

The General Translation Library

The contents of the general translation library are governed by the particular source codes and target codes for which translation procedures are desired. When stored in the translation library, these procedures are represented in EM code - hence contain special macro-instructions having to do with allocation. They also contain macro-instructions referring to translation procedures stored in the core library. In general, the time at which various core functions are brought into play in course of a translation is dependent upon features of source and target codes beyond those which are governed by canonical form. Hence the highest level control of a translation procedure always resides in a program belonging to the general translation library. With ACT, as presently conceived, there must be a distinct high-level controlling program for each source-to-target pair (S, T) desired. Of course these controlling programs will, in general, be very short, the bulk of the work being encoded in sub-programs...
11.1

belonging to general or core libraries.*

One class of translation procedures expected in the library deserve special comment. In course of many code-to-code translations one reaches a penultimate code form which may be roughly described as "assembly code" and a final step in translation called "assembly". There is considerable variation between present-day translation systems (such as FORTRAN, FLOWMATIC, IT, etc.) in regard to the complexity of this final assembly phase. In the FLOWMATIC compiler for example, only the most rudimentary assembly phase, consisting of nothing more than relative address interpretation is employed, while FORTRAN translation produces SAP assembly code with symbolic addressing, library reference ability, and many other features.

In the development of ACT a special analysis was undertaken of the terminal translation steps to determine whether a widely applicable 'end game' could be prepared for inclusion in many different translation procedures. This analysis resulted in a set of specifications for assembly-system and code-which includes many features normally included in such systems but also some noteworthy additions. (Description of these lies beyond the scope of this paper.) Assembly procedures conforming to these specifications make extensive use of the core functions previously described as well as some additional ones. Since, however, such procedures always depend, in part, on the target computer, they are themselves represented in the general translation library.

The success of the ACT system conception hinges critically on the feasibility of general translation library building, and we will now try to show that past experience with code-to-code translation justifies the expectations on which the over-all design of ACT is based.

It is well known that, in normal code translation procedures, the source code is made to pass through some number of intermediate forms until, through final processing it becomes target code.

In (1) the subscripts, 11, 12, etc., are meant to suggest successive Intermediate forms, while Al suggests Assembly form (A2, A3, etc., being possible Intermediate forms through which the code passes during assembly). In FORTRAN translation, for example, the initial formula statements pass through several intermediate forms until they reach "triple" forms; they are then translated further into SAP assembly code, etc. Generally speaking, as the code progresses from form to form, it becomes, step-wise, less and less conditioned by the specifications of the source code. Thus, for example, in FLOWMATIC, by the time the code has reached Op. file (1) form, it is largely independent of the detailed rules for FLOWMATIC statement writing. One could substitute many new conventions for FLOWMATIC sentence presentation, and correspondingly only alter the action of the so-called "glossaries", which (together with some general controlling code) determine the first intermediate code form. To put this another way: many details of FLOWMATIC code form are interpreted and done with in the first stages $C_0 \rightarrow C_{11}$. All such details could therefore be altered without requiring any revision in the compiler, save a change in the early phases.

Exactly parallel observations apply to the terminal stages of code translation with regard to the details of the target code. Again choosing FLOWMATIC as an example: one can alter the target computer (hence the target code) with little or no effect on the early stages of translation which are source-code determined.

Figure 3 shows a rough break-down of code-translation into stages which are successively more target-code determined and less source-code determined. The division into gross stages as in Figure 3 is neither unique nor optimal. Its sole purpose is to show how the ACT translation library may be used to good advantage. Suppose for example that a certain translation $(S_j, T_i)$ is coded with exactly six intermediate stages, shown in Figure 3. The over-all ACT program would read:

(1) Do stage 1
(2) Do stage 2
(3) Do stage 3
(4) Do stage 4
(5) Do stage 5
(6) Do stage 6

This program would be stored in the ACT library together with all six sub-programs which are called upon. Now suppose that one wishes to produce a translation for $(S_j, T_i)$ where $S_j$ differs from $S_i$ only in its rules for operation and operand presentation (such as parenthesis notation for algebra replaced by Polish notation; or the rules for file and item descriptions changed from "explicit" to "pictorial" form, etc.). The new translation may be obtained by writing a new controlling program

(1) Do stage 1
(2) Do stage 2
(3) Do stage 3
(4) Do stage 4
(5) Do stage 5
(6) Do stage 6

* It is projected that a special component of ACT can be prepared, to be called the ACT "translation control" program. Highest level control of translation would then always be vested in this program, and the total number of individual translation procedures required to handle a large class of $(S, T)$ would be greatly reduced. Further study is required to test the feasibility of this.

** This observation is the basis for the so-called AIMACO system which translates FLOWMATIC code into IBM target code.

*** The fact that some computer effects enter the translation procedure almost from the beginning is what makes the UNCOL proposal - namely to accomplish all translations in two stages, the first source determined and the second target determined - unfeasible.
and also coding a new sub-program corresponding to \( T_1 \). If, on the other hand, one wishes to produce a translation for \( (S_j, T_j) \), one will have to consider how radically \( T_j \) differs from \( T_1 \). The more radical the difference, the greater the number of intermediate stages which must be replaced.

Consider next, the problem of producing a translation for \( (S_k, T_i) \) where \( S_k \) is a source code which allows parts to be written in \( S_i \) and other parts in \( S_j \) with "type" labels to show which sections are written in accordance with which conventions. Suppose further that both translations \( (S_k, T_i) \) as well as \( (S_j, T_i) \) were already available, and both coded in six stages as in Figure 3. The new control program would cause some number of initial stages for \( S_i \) to be performed, and also some number of initial stages for \( S_j \). As soon as both streams of coding have achieved a common form, they are, from there on, submitted to the same terminal stages. How many stages would be involved before the two streams converge is a function of how profoundly the two codes, \( S_i \) and \( S_j \), differ in their features as analyzed in Figure 3.

These greatly over-simplified examples are meant to show that, because of the partial "factoring" of source from target effects which one may realize in code-to-code translations, one will have ample opportunity for the re-use of major translation sub-programs to be stored in the translation library. As already discussed previously, the extended machine base seems indispensable for the combining of such major sub-portions into entire operative programs.

Final Summary

The primary aim of the ACT system is to reduce to a minimum the labor required to implement a given translation \( (S_i, V_i, T_i) \). The reduction of labor is achieved through:

1. Storing translational functions of almost universal applicability in the core library and providing for their incorporation in new translation programs.
2. Making it possible for the translation library to grow in such a way that new programs \( (S_j, T_i) \) may use, as sub-programs, previously coded procedures already stored in the translation library.

Regarding item (2) above, its realization is entirely dependent on EM as the fundamental base on which ACT is built. The effectiveness of item (1), on the other hand, is dependent on the analysis of coding functions which leads to the definition of canonical form.

(Bibliography

STAGE I

PROGRAM I
(SPECIAL TRANSLATION ROUTINE WRITTEN IN MACHINE CODE V)

STAGE II

PROGRAM II

RESULT I
(PROBLEM REPRESENTED IN MACHINE CODE T)

RESULT II

PROGRAM DATA

COMPUTER T

PROBLEM

DATA II

COMPUTER V

PROBLEM

DATA I

S - Source
V - Vehicle for Translation
T - Target

Figure 1
TRANSLATION AND COMPUTATION
Figure 2
EXTENDED MACHINE
Figure 3
ANALYSIS OF CODE TRANSLATION
Figure 4
THE ACT SYSTEM