Editor's Note: The following article will be published in two parts. Part 2 will appear in a later issue of the COMMUNICATIONS.

THE PROBLEM OF PROGRAMMING COMMUNICATION WITH CHANGING MACHINES

A PROPOSED SOLUTION

Report of the Share Ad-Hoc Committee on Universal Languages

By J. STRONG, North American Aviation
J. WEGSTEIN, Bureau of Standards
A. TRITTER, Lincoln Laboratory
J. OLSZTYN, General Motors Corp.
O. MOCK, North American Aviation
T. STEEL, Systems Development Corp.

I. *BASIC ASSUMPTIONS.*

One of the fundamental problems facing the computer profession today is the considerable length of time required to develop an effective method of communication with the machine. Moreover, it seems that the ability to communicate easily is no sooner acquired than the language changes, and the problem is renewed, usually at a higher level of complexity.

A. *Obsolescence of Machines.* Experience would indicate that there are few machine users who do not obtain new and different machines every three to five years—a change generally prompted by technical obsolescence rather than by the decaying ability of the machine. In most cases, the appearance of a new machine coincides with the necessity to expand computing capacity. Although the programming cost involved in changing machines is high, the pressure to advance to these new machines has been sufficiently great to justify their acquisition.

B. *Growing Sophistication of Machine Languages.* So far, each advance in machine design has been accompanied by an increased complexity in the structure of its language, making programming in machine-like language progressively more costly in both dollars and elapsed time.

C. *Existing Compilers.* Current compilers alleviate the problem somewhat by translating some particular "easy-to-code" language into a specific machine code. The principal shortcoming in this approach is the considerable lapse of time between the initial conception of the "easy-to-code" language and its general acceptance. This time is of the same order of magnitude as the machine replacement cycle, resulting in the ever-present danger of having a good compiler available for the old machine just after it has been replaced.

II. *APPROACH TO A SOLUTION.*

Let us examine a broad set of specifications which might describe a "system" capable of solving this problem.

A. *Ideal Specifications.*

1. The individual with a problem would describe its method of solution in the language most natural to his way of thinking about the problem.
2. Once his problem has been coded in this language, the solution might be obtained by processing the program with acceptable efficiency on any present or future computer. (Efficiency of operation in a proposed system is equal to the lowest possible total cost, including programming, divided by actual total cost.)
3. A minimum of "system programming" should be required to produce the system initially and to maintain it. Most installations should not be required to do any system programming at all.

B. *Practical Specifications.* The ideal specifications listed above need some interpretation. In their ultimate implications, they may be incapable of realization in this century. A few qualifications are in order.

1. Saying that the man with the problem should speak in the "most natural" language is misleading. His native tongue, English, is capable of almost infinite shades of meaning. Few people can use English precisely and unambiguously. Consequently, the most natural method of expression will have to be compromised in the interest of precision. This is not unusual; almost every scientific discipline, and most trades and professions, have their own unique, fairly precise language.
2. The ideal of processing every problem in every language on every computer and producing efficient results can probably be realistically approached without too much compromise by permitting the following conditions:
   a. The coder can achieve higher efficiency if he knows which specific computer his routine will be processed on.
   b. Any routine written for a small computer may be processed on a larger one. However, a routine written for a larger computer might be capable of execution on only a few small computers and certainly not on a computer several orders of magnitude smaller. For example, 709 routines would not normally be translated into E101 language since they might run for many years on the smaller machine.
   c. A large computer is available for development of the system. A run on a large computer would prepare the system for all subsequent use on a small computer.
   d. Although the existence of the system may influence computer design, the system should not be dependent on this factor. Moreover, it would be undesirable if the system should in any way inhibit the development of more powerful computers by restricting the complexity of their basic languages.
3. With regard to the time required for system programming, one would hope that some useful results of the system would be available in from three to five years. The design of the system should not rely on a major "breakthrough" in programming art. The system, for example, should not stand or fall upon its ability to develop a routine which is so general that it can transform every type of language into every other type.

III. *FURTHER ASSUMPTIONS.*

The proposed solution to the problem will be presented in some detail in paragraph 5 below. Since some of the assumptions underlying this solution may be considered controversial, a preliminary discussion of them here may help clarify the final proposal.

A. *Languages.*

1. It is impossible to agree on one universal POL. Since there are many varieties of problems, any attempt at universality of problem-oriented languages will result either in inadequacy (such as an attempt to use algebraic language for a logical problem) or such extensiveness as to become useless. In the latter case, the "universal" POL is really the sum of all possible POL's and is never truly universal for long, since the language must grow to cope with the new classes of problems that arise.
2. Machine languages will continue to grow in complexity and will become increasingly difficult to code in. Everyone looks with dread at the possible computers of the next decade, which will be simultaneously executing multiple asynchronous stored programs. There is little reason to expect a reversal of this trend.
3. The present status of the compiler art requires a rather difficult generative routine to transform each POL formulated into each ML desired. Moreover, additional routines must be written whenever it is necessary to produce a different ML from the one belonging to the machine on which the translation routine is executed. The number of individual compilers of the current type needed can only increase as it becomes desirable for POL's to multiply, for machines to be replaced, and for one organization to have several types of machines. The time is fast approaching when the need for these routines will exceed any possible supply.

B. *Programming.*
   1. At the present state of the programming art, a reasonably efficient routine can be written to transform any one specific language into any other specific language.
   2. The complexities of the language transformed determine the size of the machine needed for an acceptable degree of efficiency in both the transformation process and the subsequent execution of the ML routine.

C. *Machines.* There will be available for the system programmers a machine at least as complex as the IBM 709 and machine language programming tools at least as versatile as the SHARE 709 programming system.

IV. *SOLUTION—THE THREE-LEVEL CONCEPT ("UNCOL").*

A. *History.* This concept is not particularly new or original. It has been discussed by many independent persons as long ago as 1954. It might not be difficult to prove that "this was well-known to Babbage," so no effort has been made to give credit to the originator, if indeed there was a unique originator.

B. *Outline.* The system is composed of three levels of language, as shown in the schematic diagram, Appendix B.
   1. ML Level. The lowest level (closest to the bits in the machine hardware) is composed of all the current or future ML's.
   2. POL Level. The highest level (furthest from the machine) is composed of all the current and future POL's.
   3. UNCOL Level. The center level is a single language "UNCOL," the Universal Computer Oriented Language.
   4. Generators. Generators are those routines which perform the transformation from the POL's to UNCOL. They are analogous to present generative compilers except that they produce, not a number of ML's, but only UNCOL. As with present compilers, one of these would be needed for each POL used with a given machine.
   5. Translators. These are routines which perform the transformation from UNCOL to ML, and are like present compilers in that they are one-time preprocessors before execution of the customer's program. However, they are not so complex nor so difficult to write as the "generators" described above, since UNCOL, being computer oriented, has many things in common with each of the ML's. For each machine only one translator need ever be produced, regardless of the number of POL's formulated or used.

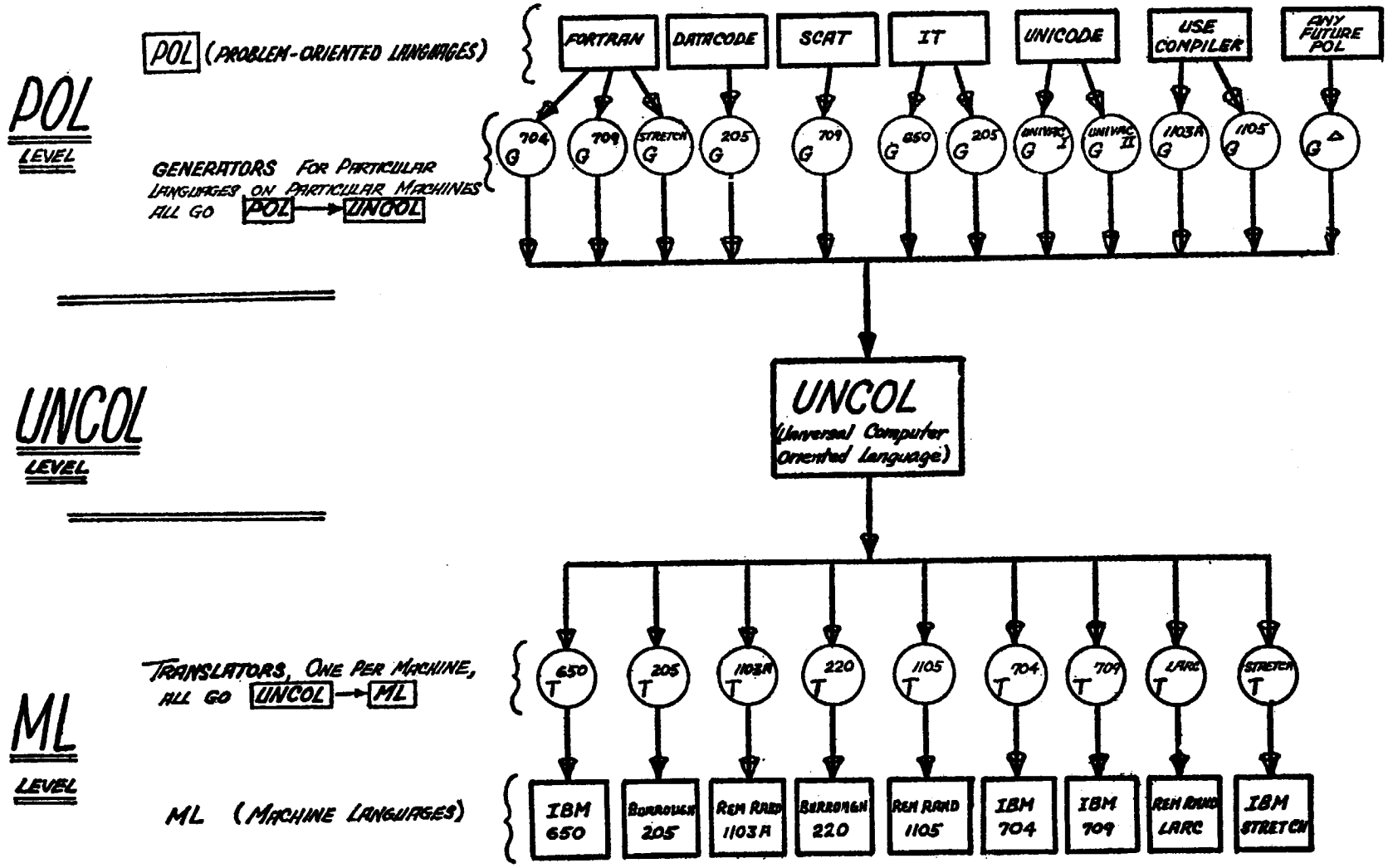
V. *CONCLUSIONS.*

It might seem that the UNCOL three-level system does nothing but complicate the current method, whereby compilers enable one to proceed with direct simplicity from a chosen POL to a specific ML. Such is not the case. The versatility of this system is almost unlimited. Because of its complexity detailed discussion of UNCOL has been found to require a special flow chart notation (described in Appendix B). With the addition of this notation, illustrative examples of how this system might function are presented in Appendix D. Conclusions are summarized below.

A. *Definite Immediate Benefits.*
   1. As each new machine is produced, all that would be necessary in the way of system programming is a single translator to convert UNCOL into the new ML. The machine manufacturer would probably write several successive versions for each of his machines in an attempt to exploit to the fullest the outstanding features of each machine.
   2. As each POL is invented, a system programmer must write a generator to translate his POL into UNCOL. *Thereafter, his POL will never become obsolete.* Routines written in it can be executed on any machine at any time in the future.
   3. The UNCOL system will enable an installation to use its current programs on any machine without additional programming cost.
   4. The UNCOL system can be designed, programmed, and installed within three years. The

THE 3-LEVEL CONCEPT

2-28-58



above advantages can be achieved without requiring a major breakthrough in programming techniques.

5. System programmers will probably work most of the time in UNCOL with only occasional descents to the ML level. A large percent of the customer's problems will be coded in one or another POL. If, however, no POL is suitable, the problem can be coded in UNCOL directly.

B. *Growth Potential.* Although the following potentialities exist, they are independent of the advantages of the UNCOL system listed above. That UNCOL provides for growth in this direction is an important extra dividend.

1. "Boot-strapping" is a distinct possibility. By boot-strapping is meant the ability to adapt the system to new POL's, machines (or even new versions of UNCOL) with a minimum of human programming. That is to say, the system can be to a large extent self-renewing, with the system routines capable of producing newer and better system routines.
2. Simple "boot-strapping" will probably be available almost immediately. System programmers writing in UNCOL can use an existing translator to produce their ML system programs.
3. It is quite possible that within five years programming techniques may permit writing a "general translator" in UNCOL which, given the characteristics of machine "A", would produce (on other machines) a translator that would convert from UNCOL into the ML of machine "A".
4. The last and least probable step would be the development of the "general generator". This routine, if given the characteristics of the POL and the machine it was to run on, would produce the generator to translate the POL into UNCOL.

C. *UNCOL Itself.*

1. The first step must be the development of UNCOL and its acceptance as a universal standard by some significant part of the computing profession.
2. Since UNCOL would be computer-oriented, any POL could eventually be expressed in it.
3. The effectiveness of UNCOL will depend upon how easy it will be to translate from it into each ML, while exploiting at the same time the advantages of the new machine concerned.
4. If the scheme is successful, UNCOL I should have a life expectancy of ten to fifteen years before any large revision is necessary. UNCOL II could be devised with the "general generator" in mind. Any transition to UNCOL II could be accomplished everywhere by boot-strapping techniques. Only one routine need be written.

APPENDIX B
UNCOL SYSTEM NOTATION

LANGUAGES refer to the symbols and rules for using them *as understood by a human being.* This is independent of the particular vehicle (and its local code) used to carry them (e.g. punched cards, magnetic tape, etc.).

POL: Any *P*roblem-*O*riented *L*anguage, e.g. FORTRAN.

ML: Basic *M*achine *L*anguage for any particular machine. If accompanied by appropriate "input translators", ML can be extended to include "machine-like" languages unique to a particular machine, e.g. SAP symbolic language for the 704.

UNCOL: *UN*iversal *C*omputer-*O*riented *L*anguage. This is a standard language oriented to the requirements of general purpose digital computers. It must have at least their characteristics of being able to express any computable problem.
(In the general schematic diagram, Appendix B, and there only, languages are surrounded by a rectangle □).

ROUTINES are sets of instructions arranged in proper sequence. They can exist in any language, and usually are transformed at least once into another language. They are designated by capital letters, thus:

P: *Problems.* Those routines which express the solution to a problem. Normally

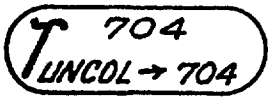
these are "customer's jobs" (e.g. payroll, aircraft performance, etc.) and can most easily be written in a POL.

G: *Generators.* Those routines which operate on routines existing in a POL and perform the transformation from one specific POL to UNCOL.

T: *Translators.* Those routines which operate on a routine existing in UNCOL and perform the transformation from UNCOL to one specific ML.

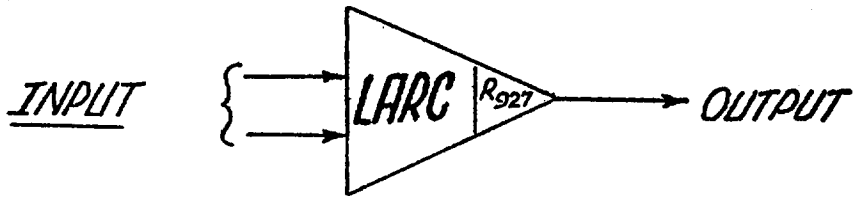
SUPERSCRIPTS The symbol for a routine carries a superscript, indicating the specific language in which the routine exists, e.g. FORTRAN (meaning that particular POL), UNCOL (which is unique), or 704 (meaning that particular ML).

SUBSCRIPTS are used to designate the operation the routine performs. Normally there is no need to use a subscript with P. However, G and T *must* have a subscript, denoting the transformation that the routine performs. For example; the notation:



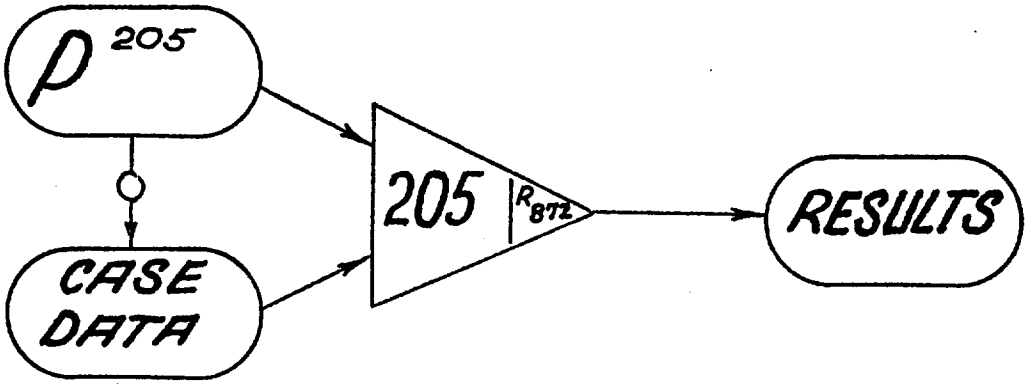
denotes a translator which transforms a routine in UNCOL into the same routine in 704 machine language. The translation routine itself is in 704 language.

MACHINES are denoted by a triangle, thus:



The number, preceded by R, at the apex denotes a specific machine run.

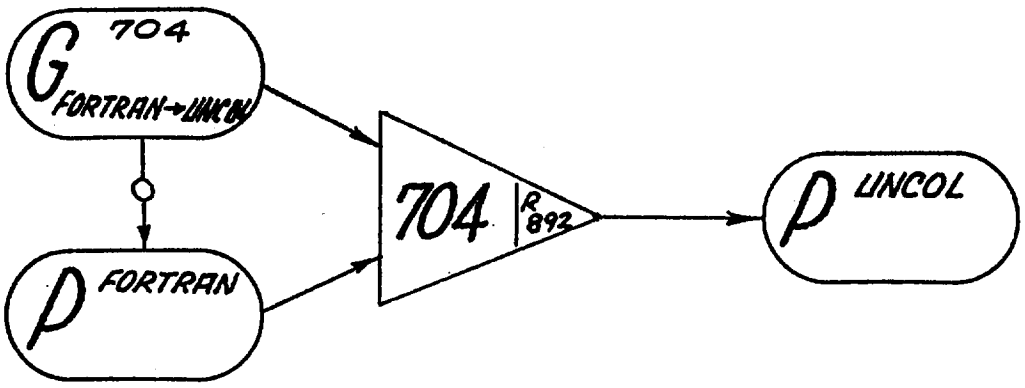
FLOW CHARTS indicate a machine run. All necessary routines and data are shown as input. The output is also defined. Routines are shown inside the oval symbol: ⬭. For example, a conventional production run, #872, on an Electrodata 205 is shown as:



The symbol ⬭ means "operating on". Therefore, the above chart means:

"A Problem routine in 205 language and its CASE DATA go into the 205 and appropriate RESULTS are produced.

Generators and translators operate on other routines (usually Problem routines) and produce the routine in another language as output; for example, a 704 run #892 might be:



This means: "The Generator in 704 language operates (in the 704) on the Problem in FORTRAN language and produces a Problem routine in UNCOL language as output."
NOTE several rules which are useful:

(a) The routine which is being executed must have a *superscript* the same as the machine being used for the run.
(b) The routine being operated upon must have a *superscript* the same as the *first half of the subscript* of the G or T which is being executed.
(c) Output is the routine that was operated upon with its *superscript* the same as the *last half of the subscript* of the G or T which is being executed.