Uniform Referents: An Essential Property for a Software Engineering Language

Douglas T. Ross
SOFTECH, INC.
WALTHAM, MASSACHUSETTS

I. Introduction

The term software engineering is new and has yet to achieve a well-defined meaning. Most present-day software has not been engineered at all, but instead is a product of the individual experience of its creators and primarily ad hoc choices among various alternatives. The effects of this unstructured approach to software are quite obvious in terms of poor performance, missed schedules, and uncontrolled costs. The need for a real and viable software engineering discipline is obvious, but we can expect no rapid resolution of these difficulties in spite of significant advances being made on many fronts. The problems of software engineering are among the most challenging facing mankind, due to the diversity of problem areas and the variety of machine and language techniques available for use. There is, however, evidence of an increased use of systematized approaches, and dawning recognition of many of the fundamental issues which clearly are central to a software engineering discipline.

This paper does not attempt to lay out a grand plan for a complete software engineering discipline. Instead, this paper presents a single, sharply-focused brief on what appears to be the most fundamental aspect for a software engineering discipline—one basic criterion which a general-purpose programming language must meet if it is to be used as the expressive vehicle for software engineering activities regardless of the style of practice of those activities: There must be a single, uniform method of referencing operands regardless of their detailed implementation.

II. Programming Language: The Math of Software

Any engineering discipline depends upon an underlying scientific foundation, for only with knowledge of how real-world phenomena behave can an

Appeared in
J. T. Tou, Software Engineering,
Academic Press (1969)

engineer design with any basis in fact. The scientific underpinnings of an engineering discipline invariably are understood and manipulated by an engineer in terms of mathematical formulations which capture the essential concepts in workable form. Formulas or geometric constructions expressed in this mathematics may be manipulated and computations may be performed as a part of the design process with confidence that the real world will match the interpreted results "to within engineering accuracy." Such mathematical manipulations and computations form the primary overt activity of the engineer as he designs a solution to a particular problem.

For mechanical, electrical, aerodynamic, and other engineering disciplines, the appropriate language for design is mathematics proper, of both algebraic and geometric forms. For software engineering, however, such ordinary mathematics must be augmented by such concepts as assignment of values to variables, iteration or computation steps, recursion, and other mixtures of logic, computation, and time sequence such as are found in programming languages. In fact, for software engineering, programming language is the "math" needed to capture the requisite real-world phenomena in the form needed for effective design.

Programming language features for software engineering must be carefully selected; not any old programming language features will do. An unstructured blur of assembly language depending in turn upon an ad hoc collection of machine hardware features which just happen to have been brought together for some particular computer design has a low probability of matching the real world in any profound way. Similarly, most "high-level" languages invariably will be either not complete or not concise enough to serve as a basis for software engineering, for they have a similar ad hoc ancestry with roots in scientific computation or business data processing areas, which omit many important aspects of general software construction.

Careful inspection and experimentation with various software engineering activities discloses a few fundamental features that somehow must be present in a software engineering language. These features can be informally derived directly from high-level considerations of the software engineering design process itself. The purpose of this paper is to indicate such a derivation of one basic principle without attempting an exhaustive enumeration of the consequences. It will be clear that there are many potential ways to realize this principle by specific features of specific programming languages. The thesis of the paper is, however, that *any* successful language for software engineering must in some manner speak directly to the points raised here.

III. Outside-In Problem Statement

Our primary thesis is that there can and must exist a single language for software engineering which is usable at all stages of design from the initial conception through to the final stages in which the last bit is solidly in place on some hardware computing system. That this thesis is itself not ad hoc and after the fact can be seen from the following quotes taken from the author's report written in September 1960 at the beginning of the Massachusetts Institute of Technology Computer-Aided Design Project. The ideas presented at that time have motivated directly many of the features of the AED (Automated Engineering Design) languages and systems developed by the author and his colleagues since that time [1].

We begin with a paradoxical twist. We have just finished pointing out the essential equivalence of design and problem-solving, which would seem to indicate that we were going to turn our attention to the solution of problems. Instead, however, we now declaim that our main objective is not to *solve* problems, but to *state* problems.

The *manner* of stating problems is also of primary importance. You must be able to state problems from *outside in*, not *inside out*. The normal practice of being explicit about a problem is to build a description of the solution procedure in small steps, i.e. to *code* the problem . . . . Stating a problem step by step from the inside out in this manner is satisfactory if the problem is well understood to begin with. But there is an alternate procedure which is frequently used among people, but is virtually non-existent as regards computers. In this alternate procedure, which we call stating the problem from the outside in, we state the problem originally in general terms and then explain further and in more and more detail what we mean . . . . It is quite apparent that this procedure of stating a problem from the outside in by means of further and further refinements is the only feasible way to attack grand problems. The inside-out method of stating problems, which is normal practice with computers, cannot possibly be carried out until a problem is stated in sufficiently detailed form, and that statement itself can come only from an outside-in consideration of the problem. Normally this preliminary outside-in study of the problem must be carried out entirely by people. The principal aim of computer-aided design as we mean it is to permit the computer to play a part in the *scheming* portion of problem-solving by providing a mechanism for performing outside-in problem statement.

As far as the outside-in problem statement sequence is concerned, an abstract or idealized problem is treated in the same way as a detailed practical problem, and in fact is more pertinent to the study. We only arrive at explicit problems after a long series of hazy, incomplete, ambiguous refinements of the original goal statement, which taken together acquire precision, so that our main emphasis must be on the consideration of problems which are in a sense abstractions or idealizations of practical problems . . . . In the sense that these things do not accurately mirror the real world of our problem, they are abstract or ideal, but it must clearly be recognized that these traits are not arrived at by choice, but are forced upon us by circumstances. The very reality of our problems forces them to be idealized . . . .

Our goal at this point in this discussion is to devise a scheme for representing the elements out of which problems are composed. All substantive problems are internally structured. We recognize that they are made up of subproblems which in turn have internal structures of their own. The sequence of substructures is terminated finally in some elemental quantities which are intimately related to the particular aspect of "reality" with which the overall problem is concerned. We wish to devise a scheme (a mathematical model, if you will) which we will consider in terms of a computer structure, for manipulating problem elements. Problem elements may be of arbitrary form and our primary objective is to have a computer structure capable of expressing relationships between general objects in a natural non-artificial way . . . .

Under the new philosophy, successive stages of problem statements are greater and greater refinements of the original statement of the problem. Each stage is represented in and materialized by the language and computer structure. The end result is a sufficiently refined solution to the original goal achieved by a sequence of elaborations, modifications, tests, and evaluations all of which taken together constitute the evolution of an ever-clearer idea of just what the problem *is* that is to be solved . . . . Since the computer is able to work in partnership with the human at *all* levels of consideration on the problem, the process is truly computer-aided problem solving, or if some other term (such as software engineering) is used, the successful completion of even a rudimentary system based upon this philosophy will represent a significant advance in the utilization of the combined talents of men and machines.

IV. An Example

To illustrate the idea of the above ten-year-old ideas about outside-in software design, we take a simple, easily recognized example: a generalized model of an information processing system. The first step in outside-in design is to describe the *primary* modules of the system to provide the first layer of substructuring. In our case we will consider the information processing system in terms of three primary modules: the *memory module*, the *selection module*, and an *action module*. The next step is to give those terms meaning by providing more detail. This is done by defining the entities from which those modules are composed, by describing the properties of those entities and how values of those properties change according to the rules of behavior for each module and between the several modules. Figure 1 indicates such a breakdown for the modules of the information processing system model. We may give the rules of behavior for the modules by describing basic actions involving the properties of the entities. The natural mode of expression is the use of functional notation and assignment statements to indicate the type of value

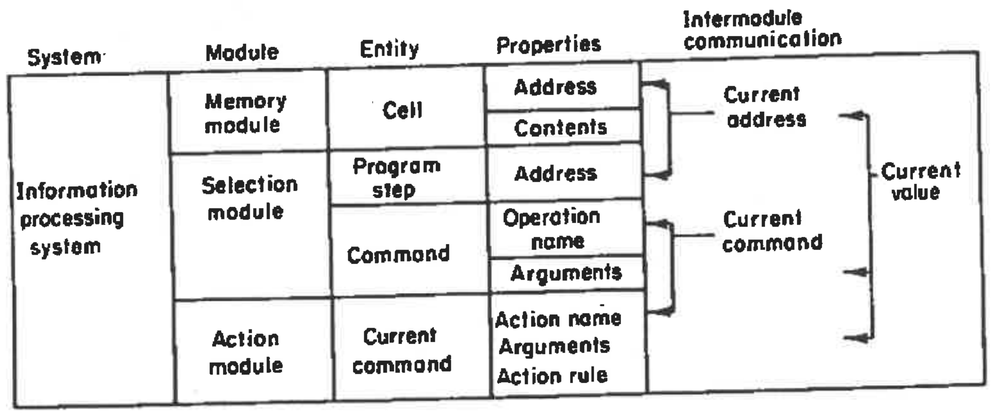

FIGURE 1. Information processing system model.

TABLE I. Rules for Modules

```
Memory module:
                              contents:= read(address)
                                         store(address, contents)
Action module:
                              result:= perform(action, arguments)
                              action:= get.action(action.name)
Selection module:
                         argument.address: get.argument(value)
                         new.address:= next(address)
System cycle:
                   repeat begin perform(get.action(get.action.name(read(address));
                                read(get.argument(read(address)))));
                                address:= next(address)
                                end;
```

yielded by various functions. This is shown in Table I. Thus for example, the notation "contents:= read (address)" specifies that a basic action of the memory module is given by a "read" function which yields the contents at a specified address. Notice that the intermodule behavior is given by the basic system cycle algorithm which calls on the various action functions of the several modules.

Clearly, a simple stored-program computer is an instance of such an information processing system. The memory module is the storage, the action module is the arithmetic element, and the selection module is the control element of the computer. Clearly, also, we can continue the process by giving more details. For example, the memory module can be modeled in more detail by introducing further entities describing how the memory is organized, either as a serial memory or parallel, with fixed or variable word length, etc. Similarly the control element can have indexed or nonindexed instructions, etc. When such elaborations have been made, still further divisions can also be constructed in the same manner to an arbitrary level of detail.

The information processing system model also is applicable to a form of batch processing operating system as well. In this case, the memory module can be disk or tape units, the action module is the loader, and the selection module is the executive for the operating system. Again such an operating system instance can further be subdivided to make various explicit elaborations in many ways. The design elaborations can be stopped at any stage, and functions simulating finer detail may be supplied to test the operating characteristics under various statistical assumptions before proceeding further with the design. (Such a scheme for operating system design has been described by Randell [2].) The outside-in method is a viable and useful technique for organizing in an orderly way any activity of software design.

V. A Graph Model Representation

The structuring which results from outside-in design may be visualized in one of two dual ways. The submodule idea corresponds to a combination of the nested layers of an onion and the overlapping regions of a Venn diagram as shown in Figure 2. Such a method of diagramming the structure

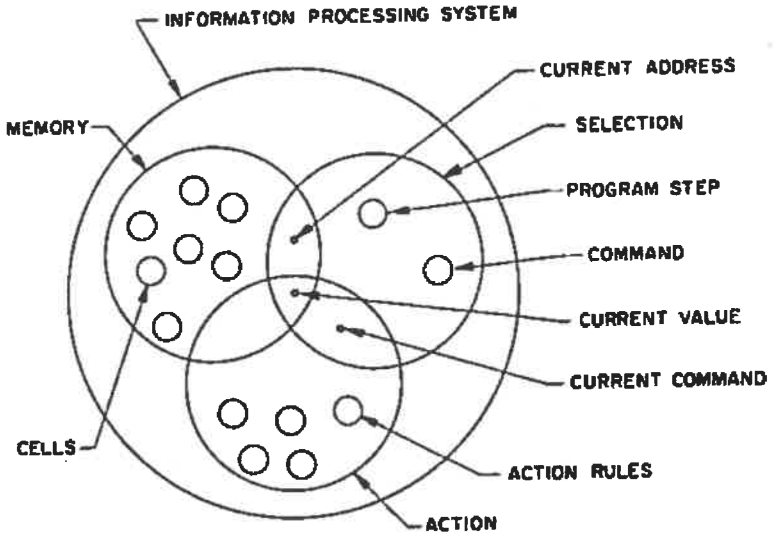

FIGURE 2. Nested-module view.

becomes cumbersome, however, and it is easier to visualize the equivalent information by taking the dual of such a diagram according to the following rules: Convert each region into a node, and each boundary between regions into an arc joining the corresponding nodes; see Figure 3.

The manner in which two modules are related is a property of the arc connecting the respective nodes. Figure 4 indicates various ways of visualizing the process of giving more and more detail in outside-in design. If the modules actually interchange information of some sort, this is indicated by the fact that the corresponding regions overlap, in which case the common intersection becomes an additional node, splitting the original arc into two separate arcs. As Figure 4 also shows, if more detail about the intersection is appropriate, the left end, middle, and right end of the connection between the regions can be elaborated further.

In this graph model of outside-in design, action or meaning resides ultimately in the arcs. Whenever an arc represents a concept that is insufficiently

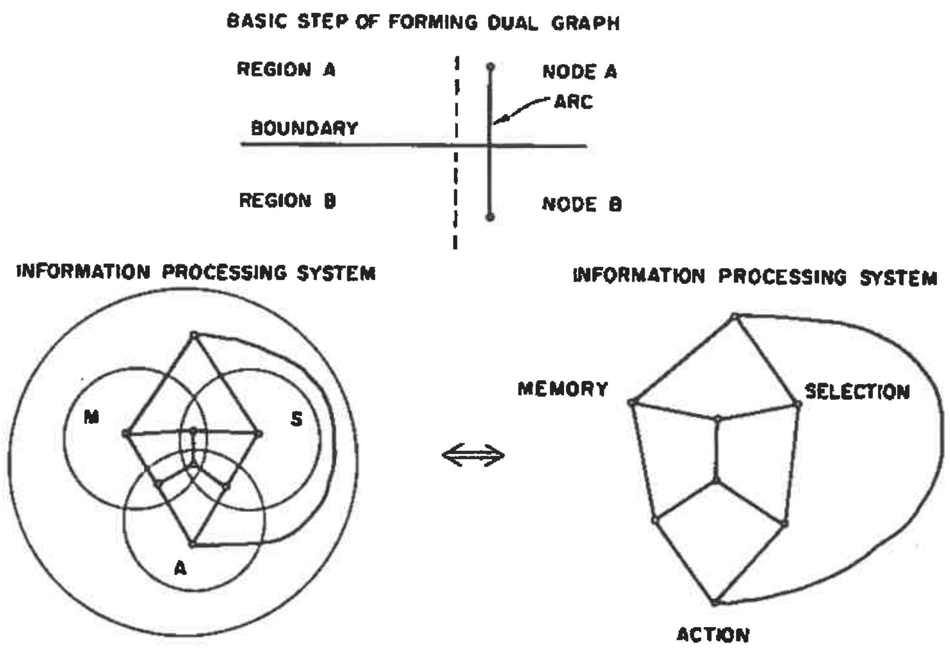
BASIC STEP OF FORMING DUAL GRAPH
REGION A
NODE A
ARC
BOUNDARY
REGION B
NODE B
INFORMATION PROCESSING SYSTEM
M
S
A
INFORMATION PROCESSING SYSTEM
MEMORY
SELECTION
ACTION

FIGURE 3. Graph model dual of modular structure.

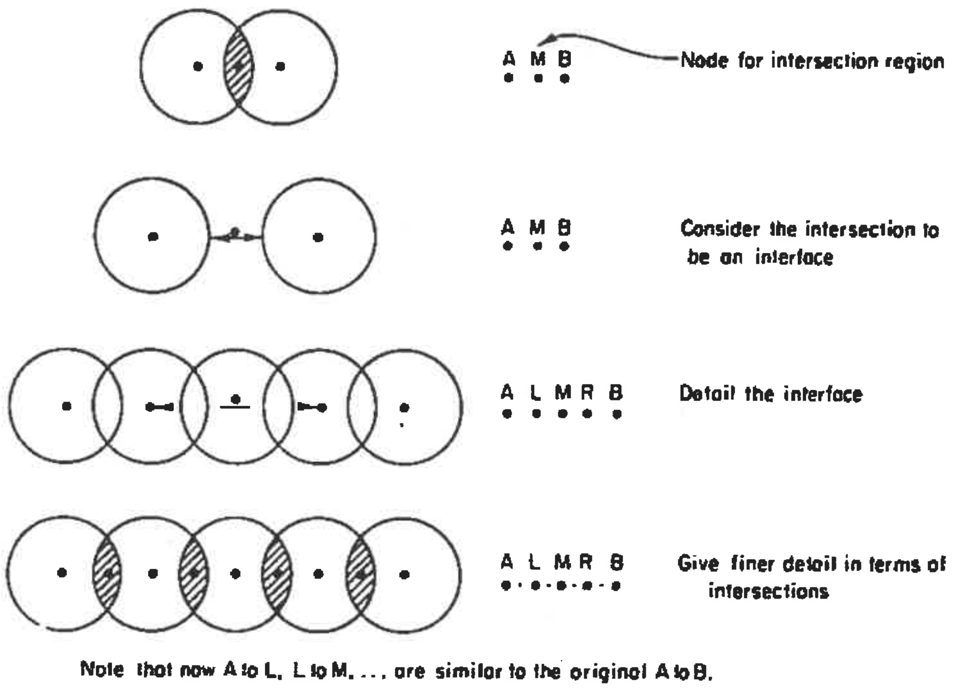
A M B
Node for intersection region
A M B
Consider the intersection to be an interface
A L M R B
Detail the interface
A L M R B
Give finer detail in terms of intersections
Note that now A to L, L to M, . . . are similar to the original A to B.

FIGURE 4. Evolution of detail of an interface.

defined, that arc is further subdivided. New nodes arising in the middle of arcs may be connected to other nodes by new arcs, so the graph becomes rich with relationships. This subdivision process continues until the designer senses that a natural level of detail has been reached, providing a natural stopping point for the process. At this level, the concepts represented by the arcs may still be very complex and in general it will be clear that much further refinement by subdivision into further detail *could* be made, but still the design level is frozen at that point. By stopping the refinement, the designer has selected a certain set of arcs (or more precisely the concepts they represent) as *primitives* for that *level* of design. For a properly balanced level, all primitives are of essentially the same level of detail.

With a level thus defined, the designer next will invariably proceed to finer levels. This takes the form of expanding a single node of the current level into a new module, in which further subdivision takes place within the boundary of the module, as is indicated in Figure 5. (If an arc is to be elaborated, a new

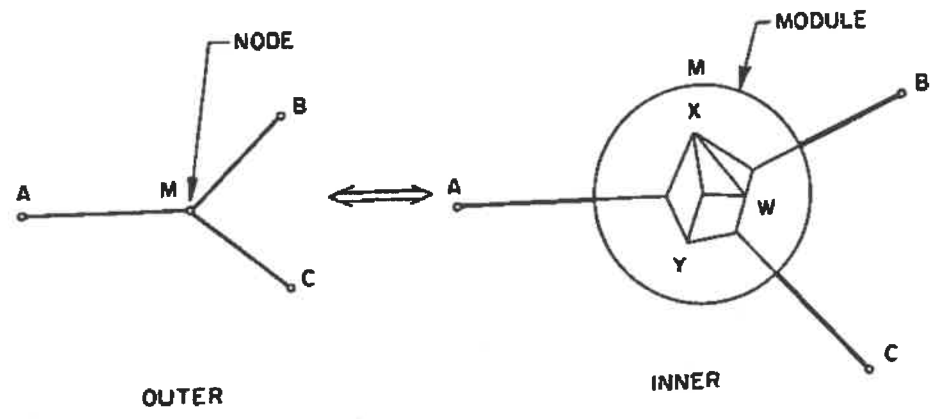

FIGURE 5. Interlevel communication through arguments.

node is interjected on the arc to be expanded into a module.) Thus, level-to-level considerations involve the identifications of modules at one (inner) level with *nodes* at the other (outer) level, as in Figure 5. Thus the module of the inner level shows detail of the corresponding node of the outer level.

VI. Interlevel Connection by Interfaces

As Figure 5 indicates, to apply the graph model method to the description of interlevel communication, it is necessary that certain arcs cross the boundary between successive levels, connecting a node of the finer level to a node of the higher level. Because an arc can connect only one node of the inner level to the outer level, only certain particular features of the inner level are known to the outer level. The process is analogous to subroutines with arguments. In Figure 5 the connections to nodes A, B, and C correspond to the arguments for the subroutine analogous to node M. The nodes x, y, and w, within M correspond to internal variables within the subroutine. The detailed structuring of the inner level is completely immaterial as long as the relevant argument connections can be made.

Each arc that crosses a module boundary corresponds to an interface between the two levels separated by the boundary. At the outer level, the node which terminates an interface arc has properties determined by the primitives of the outer level. Inside the module, the node which terminates the other end of the interface arc has properties determined by the primitives of the inner level. The interface arc itself mediates between the two. If the primitive properties of either end are changed, the interface must change correspondingly.

The crucial feature about outside-in design as the underlying methodology for software engineering is that because the interface properties *must* change when *either* end conditions change, the converse is also true. Namely, proper treatment of interface mechanisms will *allow the higher level to remain unchanged regardless of changes in the details of the inner level.* In other words all of the variations can be accommodated in the interface changes, so that the outer level is completely unaffected.

In order to give workable substance to this observation, it is necessary next to talk about the implementation of the outside-in scheme of software design.

VII. The Requirement for Uniform Referents

The graph model upon which the discussion has been based is an abstraction of the actual practice of software engineering. Actually every step is carried out by manipulations in the chosen programming language. We now are in a position to specify a powerful basic criterion which that language must satisfy if it is to serve throughout as the expressive vehicle for outside-in design.

The criterion is this: *A single uniform referent mechanism must be used for all references to operands.* We have already noted that in the graph model the primitives for a given level are represented by arcs. These arcs in turn are expressed (implemented) as programs in the language, which are combinations of the operators of the language with operands drawn from the nodes terminating the arcs. In order for the programs of an outer level to remain completely unchanged as interface programs change to accommodate various forms of inter-level detail, the manner of referencing these operands must be the same for all possible forms.

Once a programming language is provided with a uniform referent structure, the corollary property of *separable declarations* follows naturally. Any specific realization of a program must specify *some* choice of mechanism for each operand needed in the program. This choice is specified by *declarations* in the program. Given the uniform referent form, *any* consistent set of declarations will yield a working program and the statements of the program proper remain unchanged as desired.

An example of uniform referent notation and separable declarations is given by the reference mechanism of the AED-0 language. Table II shows the

TABLE II. AED-0 LANGUAGE DECLARATION CHOICES

The notation $A(B)$ in any syntactic context always means

Property A of thing B

AED-0 declarations allow choices:

A:	B:
ARRAY	INDEX
COMPONENT	POINTER
FUNCTION	ARGUMENT
MACRO	ITEM-STRING

With the .INSERT statement:

Program file	Declaration files
BEGIN	7094 version
.INSERT DECL $	360 version
BODY STATEMENTS $	1108 version
END FINI	

The program file never changes when any declaration is used

various choices of declaration for the symbols A and B used in an *atomic referent* $A(B)$ referring to "property A of thing B." The table also indicates the trivial but nonetheless important statement type, .INSERT which is used to supply various declaration choices without any change whatsoever in the source program. The statement .INSERT DECL $, in a program being compiled causes the compiler to search the active file system for a file named DECL which then is considered to have been physically inserted at that point in the source program. Thus, control of which of several DECL files is active during a compilation can result in drastically different results from a single source program. Because all of the various forms share the same referencing mechanism, the body of the source program need not be changed.

In the light of the preceding discussion regarding graph models of modular programs, it is instructive to point out that declarations such as are exhibited by the example of the AED-0 language correspond to specifying the nature of the interface arc penetrating the boundary between inner and outer levels. The act of declaration does not, however, supply the detail of the low-level module which in general is supplied in a separate operation as part of some kind of "loading" operation. At the time of loading, the desired low-level module is "bound" to the high-level program through the declared mechanism. In the case of the AED-0 language, for array and component references, the compiler itself supplies the definition and performs the binding; for procedures and macros, in general the loading and binding take place in a separate operation performed by the operating system.

VIII. Conclusion

The requirements of outside-in problem statements have shown that one feature of programming language design is central to the practice of an organized software engineering discipline: the use of a uniform referencing notation which is applicable to an arbitrary variety of detailed implementations. In a language having this feature the software engineer can iterate any aspects of the design while still maintaining the successively refined goals which the system being designed is intended to meet.

Software engineering involves a rich variety of pros and cons of *how* to use such manipulations to achieve given ends and how to establish the most general and useful high-level constructs (such as the information processing system example above) as well as the most useful low-level or primitive atomic levels which can be used as software component building blocks in many designs. We can expect rapid evolution of both high- and low-level software components as the field of software engineering matures. At every stage, the uniform referent feature of the underlying programming language being used as expressive vehicle for the design process will play a crucial and determining role.

REFERENCES

1. Ross, D. T., "Computer-Aided Design: A Statement of Objectives." Tech. Mem. 8436-TM-4, Defense Documentation Center No. AD252060. M.I.T. Electron. Systems Lab., Cambridge, Massachusetts, 1960.
2. Randell, B., "Toward a Methodology of Computer System Design." *Software Eng. Conf.* (sponsored by the NATO Sci. Comm.), *October 1968*, pp. 204–208. (Available from Sci. Affairs Div., NATO, Brussels.)