AD 731349

ARPA ORDER NO.: 189-1

R-622-ARPA

August 1971

On the Future of Computer Program Specification and Organization

R. M. Balzer

D D C
RECEIVED

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
Springfield, Va. 22151

Rand
SANTA MONICA, CA. 90406

20

MISSING PAGE NUMBERS ARE BLANK AND WERE NOT FILMED

ACCESSION for
CFSTI WHITE SECTION
DDC BUFF SECTION
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
DIST. AVAIL. and/or SPECIAL
A

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand or of ARPA.

DOCUMENT CONTROL DATA

1. ORIGINATING ACTIVITY The Rand Corporation	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

ON THE FUTURE OF COMPUTER PROGRAM SPECIFICATION AND ORGANIZATION

4. AUTHOR(S) (Last name, first name, initial)

Balzer, R. M.

5. REPORT DATE	6a. TOTAL NO. OF PAGES	6b. NO. OF REFS.
August 1971	23	-

7. CONTRACT OR GRANT NO.	8. ORIGINATOR'S REPORT NO.
DAHC15 67 C 0141	R-622-ARPA

9a. AVAILABILITY/LIMITATION NOTICES	9b. SPONSORING AGENCY
DDC-A	Advanced Research Projects Agency

10. ABSTRACT

Summarizes the currently available methods of organizing computer programs--subroutine pyramid, generators, co-routines, and passed subroutines--and presents an alternative concept, program integration, based on use of the total context rather than specific procedures. Most of a typical program is devoted to housekeeping data--subroutine save areas, parameter passing mechanisms, indices, pointers, tree and list structures, dictionaries --that have nothing to do with the specific problem but rather with its computer solution. Programs expressed entirely in problem-specific terms require implied rather than specified processing; logical process specifications not affected by data representation; dynamic linkage by the system of separate specifications, with dynamic adaptive modification at execution; and dynamic requesting of information as required from the current context. Steps in this direction include CORC, DWIM, VERS, question-answering systems, PL/I ON-UNITS, "Dataless Programming" (described in RM-5290) and Ports (described in R-605). The field is ripe for a breakthrough.

11. KEY WORDS

Computer Programming
File Structure and Management

ARPA ORDER NO.: 189-1

R-622-ARPA

August 1971

On the Future of Computer Program Specification and Organization

R. M. Balzer

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

PREFACE

This report subjectively summarizes the state of the art in computer program organization and specification. It suggests that evolutionary progress from this base is inappropriate and that a breakthrough in an alternative direction is imminent. The report characterizes this alternative direction through the current research efforts and describes its advantages.

As part of Rand's continuing effort in man-machine communication for the Advanced Research Projects Agency, this report should be of interest to those working in the areas of computer program organization and specification, and to policymakers directing such research.

SUMMARY

This report summarizes the currently available methods of program organization: subroutine pyramid, generators, co-routines, and passed subroutines. It suggests that the procedure concept upon which these methods are based inherently limits program flexibility and necessitates a level of detail artificial to the problem being solved.

The report then presents an alternative method of organization and specification, "program integration," based upon implied processing to eliminate housekeeping data, their representations, and maintenance; logical process specification to suppress representational dependencies; dynamic linkage of separate specifications as integrated by the system; dynamic adaptive modifications of such specifications; and dynamic requesting of information as required from the current "context." These capabilities are illustrated through current research efforts and the author suggests that the field is ripe for a breakthrough through what may be a synergistic combination of these capabilities.

CONTENTS

PREFACE iii

SUMMARY v

Section

I. INTRODUCTION 1

II. CURRENT STATUS 2
Pyramid Approach 2
Generator Approach 4
Co-Routines 4
Passed Subroutines 5

III. INTEGRATED PROGRAM SPECIFICATION AND ORGANIZATION: THE BEGINNINGS 7

IV. CONCLUSION 12

REFERENCES 15

I. INTRODUCTION

Since electronic computation began in the late 1940s, we have been faced with controlling and utilizing the ever-increasing computational power of computers. Efficiency considerations and lack of understanding of alternatives led to the development of the current family of computer languages. The highly structured nature of these languages and the detailed specification they require has largely obscured the alternatives.

Dreyfus' famous question of artificial intelligence, "Can we reach the moon by climbing up the branches on a tree?" [1] might validly be directed at the major effort in programming tool enhancement, the continued development of more powerful and general-purpose languages built upon the same data elements and organizational techniques.

Few of these enhancements can properly be called breakthroughs or the products of research. Instead, they are the technological outpouring characteristic of engineering disciplines. For this reason, the term "Software Engineering" has been coined for this activity. This is not to minimize past progress and achievements, but rather to suggest that the field is ripe for a breakthrough. This report 1) summarizes the current status, 2) indicates the possible beginnings of this breakthrough, and 3) proposes some breakthrough characteristics and methods of achievement. This report is naturally biased by the author's deep involvement in, and commitment to, the development of this area.

II. CURRENT STATUS

For almost every computer language, the basic unit of composition is the procedure or function. Almost universally, it is the basic organizational building block for programs. Although the form and internal composition of these units are highly language-dependent, they have certain broad characteristics in common:

1) They are composed of a linear sequence of statements or operations, some of which may affect the execution order of the sequence.
2) The domain of elements that the routine manipulates is composed of a set of background elements bound to the routine, and a set of arguments "passed" to the routine at the time of invocation.
3) The binding mechanisms are quite varied; they range from the static compile-time binding of FORTRAN [2], to the block-structured binding of ALGOL [3], to the dynamic-instance binding of LISP [4].
4) Except for constants, all the domain elements can normally be both read and written.
5) Upon its completion, the routine can (and in some languages must) return a value to the invoker.

This almost universal existence of the procedure or function as the basic organizational building block is one of programming's most striking characteristics--and one which must be radically altered if a specification and organizational breakthrough is to occur. Section III discusses this issue. First, we examine each of the four currently available organizational methods.

PYRAMID APPROACH

The subroutine pyramid approach is the most prevalent (and probably the earliest) form of program organization.

In this method, a main procedure invokes a subprocedure, which completes its operation before returning to its invoker. To perform its function, this subprocedure may invoke other subprocedures, which may, in turn, invoke further subprocedures, and so on. This hierarchical structuring typically involves the simpler operations at the lowest levels, with higher levels combining one or more lower-level routines. Programs can thus be written as a series of levels, each treating lower-level routines as basic operations that can be used to build more complex operations.

This structuring capability allows suppression of the details of the defined operations invoked (within the levels in which they are used); this keeps the specified processing manageable. This ability to build upon lower-level defined operations represents the main logical benefit of this approach. (It also reduces the memory space required for the program and saves programmer writing time.)

Because it tends to isolate functions into single centralized procedures, this method furthers program "modularity," i.e., programs that have reusable parts and that are easily modified. This modularity, which is critical to a flexible organizational structure, is limited by four factors:

1) The inflexible method of argument specification used in almost every language, i.e., positional correspondence between arguments and formal parameters, makes it awkward to use optional arguments or arguments that are only required in certain subroutine uses.
2) Possible side effects, i.e., the modification of data not local to the subroutine, prevent the subroutine from being handled as a basic unit because one must account for its interaction with other units using such data.

3) The sequential and highly interdependent nature of the statements within the subroutine produce local interactions, or "side effects," that limit the ease with which such subroutines can be internally modified.
4) This rigid compartmentalization hampers modification spanning subroutine boundaries.

The three other methods of program organization are modifications of this subroutine pyramid approach.

GENERATOR APPROACH

The generators common in list-processing languages, such as IPL-V [5], are subroutines that retain enough "context" from one invocation to the next to sequence through an explicit or dynamically computed set, returning a separate set element for each invocation. Generators thus represent an extension of the simple pyramid approach. However, they do not affect the hierarchical structure.

CO-ROUTINES

Co-routines [6], unlike generators, do affect the hierarchical structure. Not only do they retain context between invocations, they also resume execution from the point at which it was suspended on the previous invocation. Thus, a RETURN from co-routine B to co-routine A, which called it, has the same effect as a CALL of co-routine A from B. In either case, A continues from the statement following the invocation of B with its context restored. In addition, the co-routine invocation allows arguments to be passed back and forth between the two routines. Because the routines invoke each other, no hierarchical relationship exists between them. Each uses the other to obtain or provide elements for processing, a simpler and more natural organization in pipeline or multi-pass applications.

PASSED SUBROUTINES

More than the others, this approach fulfills the requirements for more flexibility, i.e., for modifying the behavior of a subroutine upon the determination of its invoker. Therefore, it offers a better basis for organizational improvements. The method is based on the realization that much flexibility can be obtained by having the subroutine invoke a routine, specified by its invoker, at an appropriate spot or spots within its operation. For example, this method of passed subroutines can be used to add specific processing to a basic tree-walking program to effect search, copy, or destroy algorithms. The ON-UNITS of PL/1 [7] are a small extension to this approach that allow the specified processing to be invoked whenever their associated special conditions occur, without explicitly being passed down to each routine. This represents a twofold advantage:

1) Because the specified processing is set up once and not explicitly passed down to lower levels, these routines need not be concerned with or aware of any "special condition" processing that might occur while they are running.
2) These ON-UNITS are not invoked at any particular spot in the lower levels; they are invoked whenever their special condition arises.

This approach also has two limitations. The special conditions are either limited to hardware-detectable conditions, such as zero-divide or end-of-file, or must be explicitly signaled by the program. Secondly (and this is really the crux of the matter), the specified processing is "pointwise applicable," that is, either the subroutine or the specified processing is running. The ON-UNIT is a separate entity that is invoked at a point and has a very limited ability to affect the processing of the interrupted

routine once it has completed. Our technology has not provided any means of combining the two into an integrated, coordinated unit in which each can affect the global rather than pointwise behavior of the other. This global integration between separate units is the goal of our research in program organization; it is treated in Sec. III.

III. INTEGRATED PROGRAM SPECIFICATION AND ORGANIZATION: THE BEGINNINGS

Although the passed subroutine approach is limited, it appears to represent the best approach to program organization for which language facilities are generally available. However, there are some special systems that have gone beyond these techniques and that represent the evidence of our effort and the start of our progress in an area of program organization best described as "integrated." This report uses these systems to help characterize and define this area, which the author feels represents the future of program specification and organization.

The simplest efforts, Cornell University's CORC system and Bolt, Beranek and Newman's DWIM (Do What I Mean) system, correct spelling and keyboard entry mistakes [8-9]. These systems are interesting because they attempt to correct certain discovered errors on the basis of the objects under discussion, i.e., the context. They represent the simplest instances of systems that resolve local difficulties through the global context; that is, they amalgamate all global information rather than using a particular item of global information, for example, the declared attributes of a given variable.

This use of the total picture for resolving difficulties is *the* basic characteristic of program integration. The absence of this capability characterizes, and inherently limits, the program organization methods discussed in Sec. II.

"Dataless Programming" [10] and Jay Earley's VERS [11] represent a separate line of development. Both attempt to express programs by the logical processing required rather than as dictated by a particular data representation. A separate program part contains the processing specification needed to particularize the logical processing for a chosen representation. Both systems use a common syntactic form,

which can express any particular representation, to specify the logical processing. Extension capabilities are provided by allowing the programmer to specify the manipulatory routines the system needs to perform the logical processing specified. This allows some "data items" to be calculated as needed rather than being explicitly present. Clearly, future systems in this area will automatically determine an appropriate representation based on the operations required by the logical processing.

Design of the Dataless Programming and VERS systems was motivated by the desire to simplify programming by removing representation details from the specification and by determining appropriate representations after all the logical requirements were known. Although unimplemented, these systems clearly satisfy these requirements. But much more is required for program integration. Programming can be further simplified by completely eliminating "housekeeping" data and their representations rather than merely removing representational details. "Housekeeping" data refer to those data that are not part of the problem but that facilitate the algorithmic solution specified. Unfortunately, these include most of the data items in typical state-of-the-art programs, e.g., subroutine save areas and parameter passing mechanisms, indices to arrays, pointer variables, most tree and list structures, and dictionaries.

The inclusion of such housekeeping data critically limits contemporary programming and severely complicates the specifications of the logical processing desired. To facilitate the logical processing specified, the system should imply such housekeeping data, their representation, and maintenance. For instance, if the user specifies that the largest unfilled order should be scheduled first, the system should organize and maintain a housekeeping representation that allows it to execute this command when necessary. Several possible representations exist: a table of all

orders that could be searched for the largest unfilled one; a list of unfilled orders ordered by size, from which the largest can easily be selected; or simply a reference to the largest unfilled order. As new orders come into the system and existing ones get filled, each representation must be maintained to indicate the current status. But the system--not the user--should provide this maintenance. Because it introduces an artificial level into the solution specification, the inclusion of such housekeeping data, their representation, and maintenance in contemporary programs makes programming a professional rather than a lay activity.

Three advances should be noted in the area of housekeeping elimination: 1) Present higher-level languages have eliminated the subroutine save area and parameter passing mechanisms from the problem specification domain. 2) Carl Hervitt's PLANNER system eliminates the specification of tree-walking and backup mechanisms for searching through a goal-oriented problem space [12]. 3) The whole area of question-answering, characterized by such systems as Fred Thompson's REL and Stanford Research Institute's QA4, also eliminates search mechanisms from the problem specifications [13-14].

These question-answering systems also represent the best efforts in another, related area of program integration--obtaining information upon demand from a data base. Currently, when a subroutine requires a context-dependent data item, the invoking routine explicitly passes the item to the subroutine. This fixed-information interface over-restricts the range of environments in which the subroutine can be easily used. This is especially true for applications in which the required types and amounts of information differ from invocation to invocation. Furthermore, each invoker of the subroutine must explicitly be aware of the subroutine's information requirements. Using the techniques being developed by question-answering systems,

the subroutine should request information, as needed, from the current context. When the same information is always required, the system could specify it at the point of invocation. The user could then write specifications that eliminate another housekeeping function--the passing of required contextual information. The system could then provide this information as needed by the subroutine.

This leads us to the final program integration area, dynamic program modification. We have considered routines as separate closed units, invokable in a pointwise manner and having a well-defined interface active only at the invocation and return points. Dynamically requesting information broadens the interface between the two routines, and eliminating housekeeping data broadens the interaction between the processing requirements of individual statements. We now seek to similarly broaden the interaction between routines.

Instead of separate, closed units, routines should be the specification of a service invoked by the system when needed. This specification should first be integrated with the conditions and restrictions existing at the time of invocation to provide an adapted service coordinated with the total process performed. For example, the system should be capable of dynamically modifying a SORT routine to 1) sort ascendingly or descendingly, 2) sort specified objects by a specified attribute, 3) break ties by a specified function, and 4) sort only the largest or smallest N objects rather than the entire set.

Three efforts represent the progress in this area:

1) A concept called "Ports" [15] generalizes the binding between routines. Ports evolved from "Dataless Programming." The concept involves a co-routine linkage, but the routine invoked is

remotely determined, i.e., the invoking routine does not know to what routine it is bound. The invoking routine knows a particular service it expects performed and is invoking that service rather than a particular routine. A separate ISPL command, CONNECT, interconnects a port with another port, a file, or a terminal.

2) For hardware conditions, the ON-UNITS of PL/1 (see p. 5) allow remote processing specification.

3) Finally, Warren Titleman's ADVISE system allows a routine to be dynamically inserted before, after, or instead of the invocation of a specified function [16]. In LISP, this capability permits fairly general and extensive program modifications because almost all useful work is associated with a function invocation. Impressive as this system is, in our context it must be regarded merely as a sophisticated text-editing system, explicitly driven by the user and relying on the structure of the LISP language, rather than as a system capable of integrating conditions, modifications, and separate procedure specifications into a coordinated whole.

IV. CONCLUSION

The previous sections described the current status of program organization and, through some promising efforts, characterized a goal organization: a programming system based on individual actions expressed entirely in problem-specific terms integrated by the system to perform the desired actions. Also discussed were the key capabilities required to achieve such a goal:

1) Implied processing to eliminate housekeeping data representations and their maintenance;
2) Logical processing specification to supress representational dependencies in the specification;
3) Dynamic linkage of separate specifications as integrated by the system rather than explicitly specified by the user;
4) Dynamic modification of such specifications by the system to fit or adapt them into the desired environment, conditions, and restrictions;
5) Dynamic requests of information to enable processes to obtain information as required from the current context through search or discourse if necessary, rather than relying only on the information provided by the invoker.

Although it is clearly related to our goal, the term "non-procedural language" was studiously avoided because of the inconsistency and lack of precision with which it is used. Also, the syntax used for program specification was not discussed because it falls outside the bounds of this report and because the ultimate syntax, "natural language input," necessitates the prior existence of a system such as the one described herein.

The goal of program integration is distant enough from our current capabilities that the correct path is not evident.

However, the author holds two unsupported but strong beliefs. First, that such a system is highly synergistic; as such, it is easier to tackle the entire problem than to work on one aspect and try to implement it within an otherwise contemporary system. Second, although it is evident that the system must contain a problem-solver, it is the flexibility rather than the problem-solving power of this unit that will make or break the system.

REFERENCES

1. Dreyfus, H. L., *Alchemy and Artificial Intelligence*, The Rand Corporation, P-3244, December 1965.

2. *USA Standard FORTRAN*, United States of America Standards Institute, USAS X3.0-1966, New York, March 1966.

3. Naur, P. (ed.), "Revised Report on the Algorithmic Language ALGOL 60," *Comm. ACM*, Vol. 6, No. 1, January 1963, pp. 1-17.

4. Berkeley, E. C., and G. D. Bobrow (eds.), *The Programming Language LISP--Its Operation and Applications*, MIT Press, Cambridge, Massachusetts, 1966.

5. Newell, A., *et al.* (eds.), *Information Processing Language-V Manual*, 2nd ed., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1965.

6. Conway, M., "Design of a Separable Transition-Diagram Compiler," *Comm. ACM*, Vol. 6, No. 7, July 1963, pp. 396-398.

7. *IBM System/360 PL/1 Reference Manual*, IBM Corp., C28-8201-0, Data Processing Division, White Plains, New York, 1967.

8. Freeman, D. N., "Error Correction in CORC, The Cornell Computing Language," *Proc. FJCC*, Vol. 26, Pt. 1, 1964, pp. 15-34.

9. Bobrow, D. G., D. L. Murphy, and W. Titleman, *The BBN LISP System*, Bolt, Beranek and Newman, Inc., April 1969.

10. Balzer, R. M., *Dataless Programming*, The Rand Corporation, RM-5290-ARPA, July 1967. (Also published in *AFIPS Conference Proceedings*, Vol. 31, Thompson Books, Washington, D. C. 1967, pp. 535-545.)

11. Earley, Jay, *Toward an Understanding of Data Structure*, Internal Computer Science Department paper, University of California, Berkeley.

12. Hewitt, Carl, *PLANNER: A Language for Manipulating Model and Proving Theorems in a Robot*, Massachusetts Institute of Technology, Project MAC, Artificial Intelligence Memo-168, Revised August 1970.

13. Thompson, F. B., P. C. Lockermann, B. Dostert, and R. S. Deverill, "REL: A Rapidly Extensible Language System," *Proceedings of 24th National Conference, Association for Computing Machinery*, ACM Publication P-69, pp. 399-417.

PRECEDING PAGE BLANK

14. Rulifson, John F., Richard J. Waldinger, and Jan Derksen, *A Problem Solving Language*, Technical Note 48, Stanford Research Institute, November 1970.

15. Balzer, R. M., *Ports--A Method for Dynamic Interprogram Communication and Job Control*, The Rand Corporation, R-605-ARPA, August 1971.

16. Titleman, Warren, "Toward a Programming Laboratory," *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, D. C., May 1969, pp. 1-9.