A program for software quality control

by PAUL OLIVER

Department of the Navy
Washington, D.C.

INTRODUCTION

The Automatic Data Processing Equipment Selection Office (ADPESO) of the Department of the Navy is engaged in a development program for software to be used in the quantification of computer systems selection criteria, and the application of quality control procedures to selected software products.

That such a program be undertaken by this centralized ADPE Selection Office is both proper and important to the successful performance of our mission. This mission, briefly stated, is to evaluate and select, or review the selection of, commercially available automatic data processing equipment for approval by the Assistant Secretary of the Navy for Financial Management.

This development program is the responsibility of ADPESO's Software Development Division, and is concentrated in three areas. A COBOL Compiler Validation System has been designed, implemented, and is being used throughout the Federal Government to determine the degree to which COBOL compilers conform to the published standard. A system to facilitate the process of COBOL benchmark program conversion, evaluation, and implementation has been completed and is being field tested. Finally, an experiment in using a library of synthetic programs for system performance measurement is being conducted.

An evaluation of such a program requires a description of the projects, the identification of project controls which have been applied, and the resultant or expected payoffs. These will be discussed in turn.

Why a quality control program?

The problem we are attempting to alleviate is a financial one. During fiscal year 1973 ADPESO participated in 189 acquisition actions with a monthly rental value of $691,000. This does not include 173 reutilization actions. The scope of these actions is quite broad. Recent acquisitions have included 100 mini-computer configurations, 50 key-to-disk configurations, and a medical laboratory information system.

How do the above dollar figures relate to software? Precise measurements are difficult, but we estimate that the Department of the Navy's annual software expenditure is approximately three times that of hardware. Barry Boehm has cited a similar figure for the U. S. Air Force,[1] and we suspect this figure is fairly universal.

Our present work has as one of its principal purposes the lowering of software production and maintenance costs. These costs will of course vary with the nature of the system in question. A 1964 SDC report[2] suggested that approximately 19 man-months were required for the delivery of 1000 machine language instructions. The data were derived from 26 projects, and included program design, coding, and testing time. The incremental time per 1000 additional lines of code was 5 man-months. Corbato's data gathered from the Multics project[3] indicate that productivity can be vastly increased through the use of a higher level language, but software still remains an expensive product.

Much of the high software cost is the result of duplication and of conversion costs. Williams[4] has reported on a conversion project undertaken in 1964 by the Lockheed Missiles and Space Company. The 220 FORTRAN programs which were converted, from an IBM 7094 to a UNIVAC 1108, required five months for the job, at a cost of approximately $241,000. To alleviate this problem we need to ascertain the degree to which higher level language translators conform to published standards, and we could certainly use more in the way of conversion aids, particularly data conversion.

Finally, we have found that the entire competitive selection time can be disturbingly long—nine to 23 months in our experience. We say "disturbingly" because a long selection process is expensive for both buyer and vendor. We are interested in software which could perhaps be used in shortening the time span.

THE PROJECTS

The goals

The user of higher level languages in software development will reduce the cost of such development, principally by increasing programmer productivity. Two languages, FORTRAN and COBOL, have been standardized so as to increase their usefulness. Standardization efforts are also under way for BASIC and PL/1. If standardization is indeed to bear some advantages, commercial compliers must adhere, in their translation, to the published standard. The adherence to a standard must include language semantics (where unambiguous) as well as language syntax. Effective implementation of a standard requires a means of measuring the degree to which compilers conform to the standard. Thus, the development, use, and maintenance of a validation system for COBOL compilers has been an important effort on the part of the Software Development Division.

Portability is a measure of the ease of moving a computer program from one environment to another. Many factors affect a program's portability: the computer system, the language used, program design, and the application. At this time, we are specifically interested in COBOL program portability. A COBOL program would be completely portable if all non-standard functions (e.g., extensions to the language) could be reduced to standard functions, all implementor names could be resolved, data representation were standardized across computer systems, and no "implementor defined" language elements were used. Practically, this means that a completely portable COBOL program is a figment of the imagination! We can, however, greatly improve a program's portability by developing software which addresses itself to the above problems.

It is also important that we not sacrifice too much efficiency for the sake of portability. A recent study by International Computer Systems, Inc.[5] indicates that COBOL programs are generally easier to convert (to other COBOL dialects) than programs in FORTRAN or assembly languages. Unfortunately, the same study also indicates that the relative operating costs of converted COBOL programs are much higher than those for other languages. Our aim is to achieve significant portability at a modest cost in efficiency. Because such an aim is quite relevant to benchmark programs, we refer to the conversion system we are developing as the Benchmark Preparation System (BPS).

A significant factor contributing to delays in computer systems acquisition has been the preparation and processing time of user benchmarks. Some way of measuring minimal system throughput capability is required. For selection purposes, benchmarks are the accepted measurement tool in the Department of the Navy. The major problems with natural benchmarks (i.e., existing application programs) have been the following:

(a) Each time an agency selects a system a new set of benchmarks is prepared. This is wasteful.
(b) The benchmarks are often not debugged, and usually biased toward a given architecture.

The latter problem will be partially alleviated by the BPS. In order to reduce production duplication and costs we are developing a "reference benchmark program library." This is a set of task-oriented synthetic programs which can be used individually or in a mix, in conjunction with or instead of natural benchmarks.

Systems to fulfill the goals

The COBOL Compiler Validation System consists of audit routines, their related data, and an executive routine (VP-routine) which prepares the audit routines for compilation.[6] Each audit routine is a COBOL program which includes many tests, and supporting procedures indicating the results of the tests. The audit routines collectively contain the features of Standard COBOL (except for the Report Writer module). The executive routine automates the creation of a file containing the audit routines with implementor names inserted in the source code, and the operating system control cards required for compiling and executing each routine. The testing of a compiler in a particular hardware/operating system environment is accomplished by compiling and executing each audit routine. The output report produced by each routine indicates whether the compiler passed or failed (individually) the tests in the routine. If the compiler rejects some language element by terminating compilation (giving fatal diagnostic messages) or terminating execution abnormally, then the test containing the code the compiler was unable to process is deleted, and the audit routine compiled again. A test is deleted by inserting NOTE at the beginning of the test paragraph, thereby changing the source code in the test paragraph to comment statements. The output reports of the audit routines constitute the raw data from which the members of the Federal COBOL Compiler Testing Service (an activity of the Software Development Division) produce a Validation Summary Report, which provides a consolidated summary of the results obtained from the validation of a compiler.

The results of running the COBOL Compiler Validation System do not suggest the degree to which the compiler is usable (i.e., capable of data processing applications) but the degree to which individual language elements are usable. This will give an indication of conversions which will be necessary in order to utilize a source program from another system supporting the same language specifications/standard. Thus, the Validation System tests a COBOL Compiler's adherence to the standard language syntax, and, where unambiguous, language semantics. The latter of course is a more difficult area because of the lack of appropriate mechanisms for precise semantic specifications. The Validation System does not evaluate the implementation of a compiler (i.e., is it a text-in-core or compiler-in-core, etc.) nor its quantitative performance characteristics.

Additionally, the summary of a validation includes an indication of unspecified language semantics (i.e., where latitude is given for vendor implementation), and ambiguous language semantics. Finally, tables summarizing the running time and memory utilization of the audit routines, and a characterization of compiler hard copy output and diagnostics are included in Validation Summary Reports.

The benchmark preparation system performs conversion in the major areas affecting portability of application COBOL programs; nonstandard COBOL functions, implementor names, and data representation. A COBOL source program translator (NAVTRAN-C) takes native machine COBOL programs and converts them to machine independent COBOL (ANSI X3.23-1968 language specifications). Those functions in the native machine COBOL which are extensions to the ANSI language specifications (and therefore cannot be converted) are flagged by the translator. Implementor names in the benchmark programs are replaced with unique names in the machine independent source programs. These names are recognized and replaced by the VP-Routine when the programs are implemented on the target machine.

Input data files associated with the benchmark programs are translated by a series of COBOL programs. These data translation programs make use of data conversion subroutines inherent in the respective COBOL Compilers (native or target machine) in translating the machine dependent data to machine independent format and vice versa. Machine dependent data characteristics may include arithmetic sign, word boundary alignment, and certain internal representations. The COBOL data translation programs are created from the benchmark program file descriptions. The creation is performed by program generation. File descriptions in the data translation programs are those for the native machine file, machine independent file, and ANSI/target file. The native machine file description is used to read the native machine data files and build machine independent data files. All data in these will be in display or character mode with the signs of numeric data stored separately. Essentially, machine, dependent data are translated to a string of characters which may then be subject to straight character code translations for the appropriate machine.

Upon transfer of the data files to the target machine, the reverse operation occurs. The machine independent data are read according to the file descriptions, and written using the ANSI/target file descriptions. The data translation programs also provide the capability of validating the data files, e.g., numerically described fields which do not contain numeric data are identified. This can be done by a separate execution or in conjunction with creating the independent or target machine data files. The benchmark package (programs and data files) which is distributed is itself in machine independent form. Programs are in a source program library (Population File). The Population File contains the benchmark source programs, data translation programs and the VP-Routine. Prior to benchmark processing the VP-Routine selects the machine independent COBOL programs from the population, inserts the necessary COBOL implementor names and creates a job stream file for input into the computer system. The VP-Routine also provides the updating capability for the Population File. A summary of all changes made to the Population File and the job control language generated for the run stream file is part of the output created by the VP-Routine. This summary is used to determine the changes a vendor has to make in implementing the benchmark on his system.

The Reference Benchmark Programs Library has been used in performing an experiment to determine the suitability of synthetic programs in alleviating the problems created by natural benchmarks. Five processing tasks were selected as representing, in varying combinations, a large number of application tasks. These were sequential file processing, indexed sequential file processing, relative I/O processing, sorting, and computation. COBOL programs were written to perform each of these tasks, with each program controlled by a set of compile-time and execution-time parameters. The ability to vary automatically certain parameters at compile-time provides us with the flexibility to develop a fairly rich mix from just a few basic programs.

We have found, through our testing with these programs, that a small number of simple, task-oriented, synthetic modules can be combined into a versatile job mix. A relatively small number of parameters is sufficient to enable a single program to reflect the characteristics of a broad class of applications. Also, individual modules have proven useful in exercising isolated computer system features, such as I/O handling. Finally, if one accepts a "modest" workload characterization, aimed more at reflecting extremities and crucial areas rather than comprehensiveness, it is possible and reasonable to construct a benchmark from a set of synthetic modules.

PROJECT CONTROLS

Why

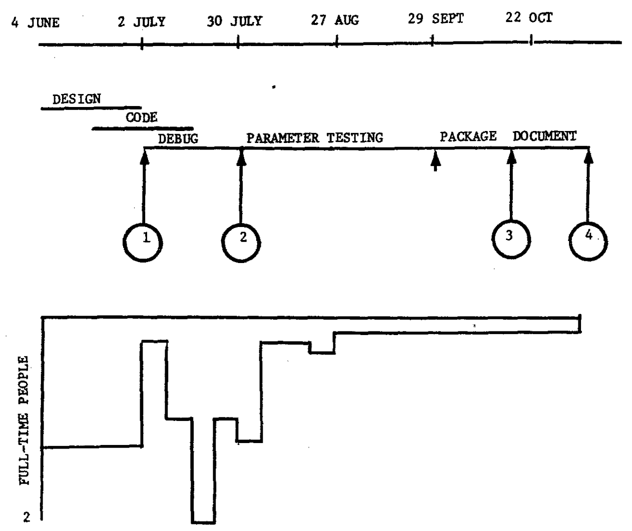
Boehm[1] has suggested that the phrase "software engineering" is a contradiction in terms because we have no data base to be used in measuring, in some way, what we produce and how well we produce it. Yet, his own studies indicate that we *do* have some data to work with. In order to obtain more, those of us whose business it is to develop software must keep records of our efforts, and thereby control them. This does not present an undue hardship in our case since the Department of the Navy strongly encourages that we be accountable for what we do, how we do it, and what it is worth.

How

Because we are a small organization, our controls are modest but, we think, effective.

Much has been made of structured and modular programming.[7] These concepts are gaining acceptance in Government and private industry. While we take no issue with their merits, we would suggest caution in their applicability. Modularity will often reduce some of a system's complexity, but may introduce additional complexity, particularly in the intermodule connections. The nature of the COBOL Validation System dictates that it be highly modular, but we have found that much of its complexity is due to its modularity. We have also found that GOTO-less programming can be awkward and, especially in COBOL, costly. We realize that deviations from the concepts are "allowed," but then we are back to what have for years been recognized as simply good programming practices.

We *do* follow modular programming concepts as design aids. This seems to have become a very common practice. A recent Hoskyns survey[8] for the British Government showed that 98 percent of modular programming practitioners did so in the design stage. A major benefit of this practice has been a lowering of maintenance costs.



Milestones:
1. Submit Narrative
2. Submit Parameter Specifications
3. Complete Population File
4. Complete Instruction Manual

Figure 1—Project history chart for synthetic benchmarks

We've considered the "lead programmer"[9] idea and discarded it as inapplicable to our environment. We are blessed with a surplus of "lead" programmers, and our projects, while sometimes large, as in the case of the Validation System, are not massive.

We keep records of our work. An "initial project form" is used to identify the project requestor, the purpose and nature of the project, time requirements, resources required, and expected payoff. We generally cannot afford the luxury of continuing if resources required exceed payoff, *in dollars.* We then prepare a work plan. This includes a schedule, checkpoints, milestones, and manpower requirement distribution. Milestones are distinguished from checkpoints in that the former require a concrete action or document to be taken or produced, while the latter may simply consist of an indication that "parameter testing is complete". Figure 1 shows the work plan for our synthetic program library project. It is important that we indicate the distribution of manpower over the project lifespan, since this enables us to coordinate manpower requirements for several projects. We review the workplans whenever we feel it is necessary (but at least at the checkpoints and milestones). If we fall behind we revise the workplan. Thus far, we've successfully resisted the temptation to add manpower or adopt unreasonable catch-up schedules when we fall behind. The necessity for such resistance has been well documented by Brooks[10] and others.

We maintained a log of compiler errors (computation, sequence control, input-output, etc.) but we abandoned this because we did not find it overly useful. Over a sample period of five months we produced approximately 10,000 lines of COBOL code; and 42 compile-time errors. Approximately half of these were "clerical" errors (bad keypunching, sloppy printing, etc.). Recognizing the smallness of the sample, we would still make the generalization that any overly extensive effort in beefing up a compiler's syntax diagnostics capability may be a waste of time.

A test log is kept for all projects. The log indicates which program or module is being tested, aims of the test, whether these were achieved, and resources used. The same five month sample showed that we achieved our aims in just under 60 percent of the tests, and that new problems were discovered in some 30 percent of the tests. Also, the average test run used less than three memory minutes of UNIVAC-1108 time. All our work is done in a remote job entry (batch) mode. Yet, the above figures seem to imply that our testing habits are more consistent with what would be expected in an interactive program development environment. Sackman[11] and others have suggested that on-line programming improves efficiency. It appears that, additionally, experienced programmers tend to behave as if they were in an on-line environment, even if they are not.

Boehm's[1] statistics indicate that 45-50 percent of software efforts are devoted to checkout and testing, and that only about 20 percent of the time is spent in coding. The data base for these figures was derived from large systems projects, such as the OS/360 development. Ours are much more modest projects, and our results are both different and more variable. About 50 percent of our time in the synthetic library project was devoted to coding, and less than 25 percent of the time was spent on integration and testing. The Benchmark Preparation System figures are quite different. Coding has taken up less than 25 percent of our time, with integration and testing using up some 60 percent of the time. The resulting low figure for analysis and design (15 percent) is due to the simple fact that much is already known about the portability problem areas in COBOL.

Packaging and distribution of all our software products follow fairly simple guidelines. The programs, in machine independent form (all implementor names are parameterized, as are machine dependent features such as precision and size of numerical fields), are placed on a standard magnetic tape reel, together with a copy of the VP-Routine. The latter is used for parameter substitution (to a form acceptable to any specific system), "library management", and job control statements generation. Accompanying the tape is a user guide, brief narrative description of the system, and, where applicable experimental results. The programs are self-documented, so that we can avoid excessive external documentation. While we recognize the importance of *adequate* documentation, we have found that excessive documentation, such as detailed flow charts can be a hindrance to proper documentation. Distribution is through the National Technical Information Service.

A few words of caution about these and other published statistics and practices. First of all, they reflect a very specific environment. We have a small (eight people) staff with very homogeneous backgrounds. Our systems are modest in size, and "utility" oriented. All our work *must* be portable, since we are currently using UNIVAC-1108, IBM 360/65, and HIS 6050 systems for product development. Furthermore, our COBOL compiler validation responsibilities have recently required us to use our software on a Burroughs 6700, HIS 437, and IBM 370/155 system. Thus, portability is truly a necessity for us.

Secondly, even for a similar environment, the statistics should be viewed as "guidelines". They are simply products of our experience which we hope to learn from but do not expect to be bound by.

THE PAYOFF

What has it all cost us?

Total cost for the synthetic programs library, including machine time, clerical support, and salaries was under $6,000. This benchmark preparation/conversion package has cost us about $8,000. The COBOL Compiler Validation System was originated in 1969 by the U.S. Navy Programming Languages Group under the direction of Capt. Grace M. Hopper, USNR. A reasonable estimate of its initial cost is not possible, but we do have an expected cost for the audit routines we are preparing in anticipation of the revised COBOL standard. Our schedule calls for completion of the project by November, 1974. Total calendar time for the project will be 15 months and we anticipate to expend 36 man-months on the effort. Total cost for the new Validation System should be in the neighborhood of $75,000. The new system will be approximately twice the size of the current one, which is comprised of about 130 programs, or 100,000 lines of COBOL code. The implication here is that we expect our productivity to be about 33,000 lines of COBOL code per man-year; a remarkably high figure (Corbato[3] has reported a number in the neighborhood of 1200 PL/1 lines of code per man-year on the Multics project). This is due almost entirely to the fact that we are "borrowing" most of the design work from the present Validation System. We know the modules we will require since the standard is defined for us. The VP-Routine is already available. Many of the audit routines will be extensions of current ones. Thus, our time will be spent primarily in identifying tests, coding, and testing.

What are the benefits?

We expect the returns on our investment to be substantial. The best COBOL compiler we have tested to date ("best" in its conformance to the standard) has had some 30 areas of non-conformance. This not only impacts portability, but can have serious side effects. Many data base management systems are COBOL-based. Errors in a compiler can easily result in "dirty" data getting into the data base. We have, for example, identified some four different treatments of arithmetic statements, each producing different results! The validation of a compiler tells us where the danger areas lie. Furthermore, vendors are required to correct discrepancies once these have been identified. Thus, our validation of COBOL compilers enables us to reap the benefits of standardization. Without such a measurement tool standardization is a fruitless endeavor.

The high costs of processing benchmarks has already been mentioned. We know of a recent selection where the total award was for approximately $5 million. Included in that figure were some $500,000 which the vendor spent in processing the benchmark. Both the benchmark preparation system and our synthetic programs library would pay for itself if even a small portion of these potential savings in vendor expenditures are passed back to the Navy.

FUTURE EFFORTS

The benefits derived from validating COBOL compilers would also accrue in the validation of compilers for other languages. FORTRAN, BASIC, and, later, PL/1 are natural candidates.

Compiler efficiency, in terms of object code execution speed and storage utilization, has a significant impact on an installation's throughput. This in turn affects the timing of selections and therefore of expenditures. We believe more efficient compilers mean fewer dollars spent, or more work done for the same dollar. Thus, we are planning a set of test routines to determine the *relative* worth of a given compiler. That is, we want to measure how much room there is for improvement in execution speed and storage required. This project is in the design phase and will restrict itself, initially, to FORTRAN compilers, principally because it is easier to measure efficiency of FORTRAN compilers than those for most other languages. Knuth's work[12] suggests that our efforts may prove fruitful.

Finally, we believe that serious thought must be given to validating generalized data base management systems. Specifically, we are interested in finding ways of ascertaining that the data base one builds with these systems does indeed contain what we wish it to, that in retrieving data we get all that is proper, and *only* what is proper, and that use of such systems does not impact the integrity of the data base. We also plan to develop simple analytical models to be used in evaluating different types of data organizations. The possible ongoing contamination of these data bases by inconsistent object code has already been commented on.

CONCLUSION

Software to be used in improving or measuring the quality of other software is neither difficult nor expensive to produce. Our efforts are concentrated in the system selection area. We believe, however, that the benefits to be derived from such efforts have a broader scope, and are substantial enough to warrant persual by any data processing organization.

REFERENCES

1. Boehm, B. W., "Software and its Impact: A Quantitative Assessment," *Datamation*, May, 1973.
2. System Development Corporation, *Estimation of Computer Programming Costs*, SP-1747, September, 1964.

3. Corbato, F. J., "PL/1 as a Tool for System Programming," *Datamation*, May, 1969.
4. Williams, D. A., "Conversion Case Study and Experiences," American Management Association, Administrative Services Briefing Session #6379-02, *"A Hard Look at Software"*.
5. International Computer Systems, Inc., *Programming for Transferability*, AD-750 897, National Technical Information Service, 1972.
6. Baird, G. N., "The DOD COBOL Compiler Validation System," *Proceedings FJCC*, 1972.
7. Liskov, B. H. and E. Toster, *The Proof of Correctness Approach to Reliable Systems*, The MITRE Corporation, ESD-TR-71-222, Bedford, Massachusetts, 1971.
8. Rhodes, John, "Tackle Software with Modular Programming," *Computer Decisions*, October, 1973.
9. Baker, F. T., "Chief Programmer Team," *IBM Systems Journal*, Volume II, No. 1, 1972.
10. Brooks, F. P. Jr., "Why is the Software Late?," *Data Management* August, 1971.
11. Sackman, A., *Man-Computer Problem Solving*, Auerback Publishers, Inc., 1970.
12. Knuth, D. E., "An Empirical Study of FORTRAN Programs," *Software-Practice and Experience*, Volume 1, John Wiley and Sons, 1971.