Production of Large Computer Programs

HERBERT D. BENINGTON

The paper is adapted from a presentation at a symposium on advanced programming methods for digital computers sponsored by the Navy Mathematical Computing Advisory Panel and the Office of Naval Research in June 1956. The author describes the techniques used to produce the programs for the Semi-Automatic Ground Environment (SAGE) system.

*Categories and Subject Descriptors: K.2 [**History of Computing**]—SAGE, software, systems*
General Terms: Design, Management
Additional Key Words and Phrases: Lincoln Laboratory

Editor's Note

When we all began to work on SAGE, we believed our own myths about software—that one can do anything with software on a general-purpose computer; that software is easy to write, test, and maintain; that it is easily replicated, doesn't wear out, and is not subject to transient errors. We had a lot to learn.

As Herb Benington discusses in the following paper, we had already successfully written quite a lot of software for experimental purposes. We were misled by the success we had had with capable engineers writing programs that were small enough for an individual to understand fully. With SAGE, we were faced with programs that were too large for one person to grasp entirely and also with the need to hire and train large numbers of people to become programmers—after all, there were only a handful of trained programmers in the whole world. We were faced with organizing and managing a whole new art.

Bob Wieser (who led the software design and production effort at Lincoln) and his group decided with great wisdom to build the tools needed for such an endeavor instead of trying to do the whole job with the limited resources at hand. We paid a price—the schedule slipped by a year—but the organization that was established really got on top of the job and stayed on top.

Much of what Herb and others created for the SAGE job was forgotten and had to be relearned later by others when they faced similar problems. I confess to having a certain amount of purely human pleasure at watching other organizations suffer through the problems of building large programs—organizations that had been so critical of our own difficulties.

One thing not to forget is the challenge of putting so large and complex a program into a limited computer capacity. The FSQ-7 was the largest machine we felt able to build in the early 1950s; its capacity is trivial by today's standards. One might think that with today's technology, SAGE-like software would be easier to build. Unfortunately, this seems not to be so. There is a kind of Parkinson's Law for software: it is infinitely expandable and swells up to exceed whatever capacity is provided for it.

Foreword

The following paper is a description of the organization and techniques we used at MIT's Lincoln Laboratory in the mid-1950s to produce programs for the SAGE air-defense system. The paper appeared a year before the announcement of SAGE; no mention was made of the specific application other than to indicate that the program was used in a large control system. The programming effort was very large—eventually, close to half a million computer instructions. About one-quarter of these instructions supported actual operational air-defense missions. The remainder were used to help generate programs, to test systems, to document the entire process, and to support those other managerial and analytic chores so essential to producing a good computer program.

As far as I know, there was no comparable effort under way in the United States at the time, and none was started for several years. Highly complex programs were being written for a variety of mathematical, military, and intelligence applications, but these did not represent the concerted efforts of hundreds of people attempting to produce an integrated program with hundreds of thousands of instructions and highly related functionality. In a letter to me on April 23, 1981, Barry W. Boehm, director of software research and technology at TRW, says of the paper, "I wish I had known of it a couple of years ago when I wrote [a] paper indicating how many of today's software engineering hot topics had already been understood in 1961 in Bill Hosier's IRE article. Your paper predates much of that understanding by another five years."

By chance, the paper was presented in Washington, D.C., in June 1956 at a symposium on advanced programming methods for digital computers, sponsored by the Navy Mathematical Computing Advisory Panel and the Office of Naval Research. The paper was given there because Wes Melahn (soon to become president of System Development Corporation, and now at the MITRE Corporation) was deeply concerned with the programming of an air-defense system, as well as with the theory and mathematics of advanced digital computing at universities. All the other papers at the symposium were presented from the perspective of either universities or the nascent computing industry. The hot topics were machine organization, development of algorithms, and the development of higher-order languages. The common goal was to produce instructions that cost less than $1 per line. The audience was somewhat chilled to hear that we could not do better than $50 per instruction in our particular effort—and that we were talking about tens of thousands of pages of documentation.

I lost interest in the subject until several years ago, when I joined the MITRE Corporation and became interested in what had happened to data processing in the ensuing 20–25 years. I showed the paper to a number of colleagues, some of whom knew nothing of the SAGE development and some of whom had been deeply involved with it. Generally speaking, they were surprised that we had developed or used techniques with SAGE that today are considered essential to the effective production of large computer programs. (We did omit a number of important approaches, which I will say a little more about below.)

It is easy for me to single out the one factor that I think led to our relative success: we were all engineers and had been trained to organize our efforts along engineering lines. We had a need to rationalize the job; to define a system of documentation so that others would know what was being done; to define interfaces and police them carefully; to recognize that things would not work well the first, second, or third time, and therefore that much independent testing was needed in successive phases; to create development tools that would help build products and test tools and to make sure they worked; to keep a record of everything that really went wrong and to see whether it really got fixed; and, most important, to have a chief engineer who was cognizant of these activities and responsible for orchestrating their interplay. In other words, as engineers, anything other than structured programming or a top-down approach would have been foreign to us.

Between the early 1950s and the mid-1960s, thousands of computer programmers participated in the design, testing, installation, or maintenance of SAGE. They learned the system well, and as a result, the chances are reasonably high that on a large data-processing job in the 1970s you would find at least one person who had worked with the SAGE system. The initial SAGE prototype program slipped its initial schedule by about one year. After that, dozens of major modifications were installed at dozens of sites with slips of at most several weeks. The disciplined approach, which had started at MIT's Lincoln Laboratory, persisted for over 15 years at SDC. Why is it, then, that there are so many tales of computer-program projects whose schedule slippages were much greater than SAGE's and whose overruns are often horrendous? There are three major reasons.

First, the industry went through a phase where we decided that computer programming and the computer programmer were "different." They could not work and would not prosper under the rigid climate of engineering management. Just a few years ago, I heard with amazement the executive vice-president of one of our very largest information-system firms say, "Herb, you have to realize the programmers are different; they have got to get special treatment." I almost ran out to sell his stock short, but then I discovered that his more realistic middle management had realized the failure of this nostalgic view of the computer programmer.

Second, if anything, the pendulum has swung too far in the other direction. Many of our government-procurement documents act as if one produces software in the same way that one manufactures spacecraft or boots. When I got back into the computer programming business several years ago, I read a number of descriptions of top-down programming. The great majority seemed to espouse the following approach: we must write the initial top-down specification (for example, the A Spec), then the next one (typically, the B Spec), so we will know precisely what our objectives are before we produce one line of code. This attitude can be terribly misleading and dangerous. To stretch an analogy slightly, it is like saying that we must specify the characteristics of a rocket engine before measuring the burning properties of liquid hydrogen. Generally, software is the most complex component of a system. Twice as much software can improve the performance of a system by 1 percent or by 500 percent. The percentage can only be determined if a great deal of detailed analysis (including coding) is undertaken to understand the "burning properties" of software. I do not mention it in the attached paper, but we undertook the programming only after we had assembled an experimental prototype of 35,000 instructions of code that performed all of the bare-bone functions of air defense. Twenty people understood in detail the performance of those 35,000 instructions; they knew what each module would do, they understood the interfaces, and they understood the performance requirements. People should be very cautious about writing top-down specs without having this detailed knowledge, so that the decision-maker who has the "requirement" can make the proper trade-offs between performance, cost, and risk.

To underscore this point, the biggest mistake we made in producing the SAGE computer program was that we attempted to make too large a jump from the 35,000 instructions we had operating on the much simpler Whirlwind I computer to the more than 100,000 instructions on the much more powerful IBM SAGE computer. If I had it to do over again, I would have built a framework that would have enabled us to handle 250,000 instructions, but I would have transliterated almost directly only the 35,000 instructions we had in hand on this framework. Then I would have worked to test and evolve a system. I estimate that this evolving approach would have reduced our overall software development costs by 50 percent.

The third reason that we keep seeing missed schedules was pointed out to me by the editor of one of our best computing journals, who says he has concluded that producing large computer programs is like raising a family. You can observe your neighbors and see all of the successes and failures in their children. You can reflect on the experiences you had as one member of a large family. You can observe all the proper maxims of life and society. You can even study at length the experiences of many others who have raised families. In the final analysis, however, you have to start out and do it on your own, learn the unique options you have, see what unexpected problems arise, and, with reasonable luck, perform about as well as those who have been doing it forever.

The latter observation may be reassuring to the new program manager, but there have been numerous significant advances in the techniques for producing large computer programs since we did the SAGE job over 25 years ago. A few that strike me as most important are:

- We now use higher-order languages in virtually all situations.
- Almost all software development and unit testing are done interactively at consoles in a time-sharing mode.
- We have developed a large family of tools that allow us to do much precise design and flow analysis before coding. (I still say that we should use these techniques before we start finalizing our top-down requirements.)
- We have developed organizational approaches that improve or at least guarantee the quality of the systems much earlier in the game. These include some of the structured languages, code reviews, walk-throughs, etc.

For further progress, I would stress the following.

- Since the SAGE effort, we have talked about the need to invest in tools that help produce programs—that is, in tools for coding, editing, testing and debugging, configuration management, consistency checking, structural analysis, etc. I believe too little effort has been spent on thinking through such tools and standardizing them so that they can become analogous to the relatively few higher-order languages that we use with great facility.
- Finally, there remains a tremendous range and ability among computer programmers to do

different jobs. Some are good gem-cutters for any kind of stone. Some can play very special roles—for example, where fastidious approaches are needed. Some are brilliant and articulate conceptualizers and leaders. Some should not be allowed near a computer. We must learn to recognize these types, to use them in their right place, and to set higher standards for not using people even though the market seems insatiable.

—*Herbert D. Benington*

Introduction

At the 1955 Eastern Joint Computer Conference, Jay W. Forrester suggested that the evolution of electronic digital computers might be roughly divided into five-year periods, each period with its paramount significance.

1945–1950 was the period of electronic design. From 1950–1955, attention has been focused on the solution of scientific and engineering problems. 1955–1960 will encompass the upswing in the commercial data-processing applications.... 1960–1965 will probably mark the shift of major attention to the use of digital computers as the central elements in real-time control systems.

With respect to this last period, Forrester continues:

General purpose digital computers, as outlined in [recent news] releases, are to be the nerve centers for tying together the flow of information in our forthcoming new air defense system. This type of control system, we can assume, will develop further into a high-speed automatic control and regulation of future civilian air traffic....

[Or,] consider the chemical plants and oil refineries.... In the last 30 years the automatic controls in an oil refinery have risen ... to some 15 percent of the investment in a refinery [or often about] $15,000 worth of automatic controls. I believe we will see digital computers as controllers and monitors of operation in these plants to permit closer control, higher-speed chemical reactions, larger outputs, and a better product.

During the past five years, we have seen developments in automatic programming where the emphasis has paralleled Forrester's first three periods. We can compare the electronic-design phase with the development of basic programming techniques of translation, compilation, and interpretive routines. Scientific and engineering calculations have been assisted by the PACT and A-2 compiling systems, and commercial data processing by BIOR and B-0 (to name but a few). More important, our colleagues who build computers have come to realize that a computer is not useful until it has been programmed, and that programming is an expensive job that requires both machine assistance and human sympathy.

This paper looks ahead at some programming problems that are likely to arise during Forrester's 1960–1965 period of real-time control applications. At first glance, these are problems that will result from the need for very large, very efficient programs, where one program (consisting of over 100,000 machine instructions) may be used in several machines during periods of months or years. On closer inspection, we realize that these are problems that must be faced whenever the need arises for the systematic preparation and operation of large, integrated programs, whether these programs are used for commercial processing, scientific calculation, or program preparation itself.

During the past several years at the Lincoln Laboratory, several system programs containing over 30,000 machine instructions each have been prepared. These programs are used for data processing and control in real-time systems. Production of these programs is briefly described here, particularly in terms of cost and organization. Four problem areas are stressed.

The first problem is *computer operation.* Computer time is at a premium when a large program is being prepared by relatively inexperienced programmers, when the machine and its terminal equipment are being shaken down, and when the machine-program system requires inordinate testing and debugging. The only answer is highly systematic, highly mechanized program preparation and computer operation. A Lincoln Utility System of service routines containing 40,000 instructions has been prepared to ease this problem.

The second problem is *program or system reliability.* Needless to say, a large program is distressingly prone to all types of design and coding errors, including some very subtle ones. In spite of this tendency, it must be extremely reliable if it is to control effectively a system involving extensive equipment or manpower. This is true not only in a real-time system, but also in commercial applications unless equipment engineers can outvote lawyers. Reliability is also a major factor in the preparation of ambitious automatic programming systems—how many unreliable programs have been produced with supposedly well-tested compilers?

Next, there is the problem of *supporting programs.* It has been the experience of the Lincoln Laboratory that a system of service programs equal in size to the main system program must be maintained to support preparation, testing, and maintenance of the latter.

Finally, there is the problem of *documentation.* In the early days of programming, you could call up the programmer if the machine stopped. You seldom mod-

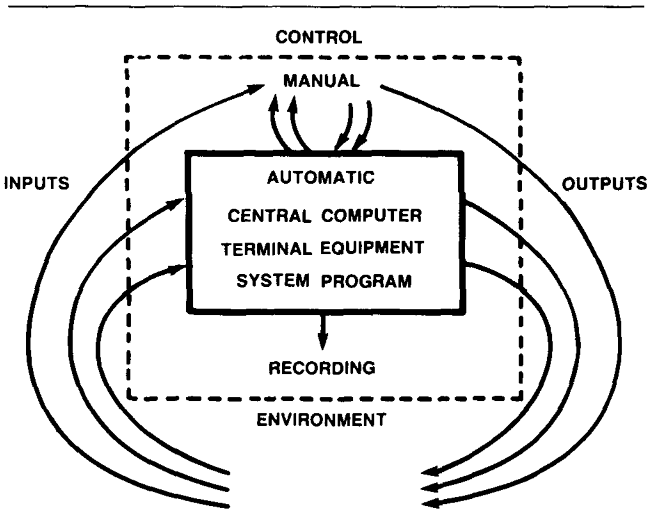

Figure 1. Typical control system. In general, a typical control system uses automatic and manual elements. The automatic portion consists of a centralized digital computer, terminal equipment communicating with the environment, and a computer program incorporating system memory and standard operational procedures.

ified another person's program—you wrote your own. Although present automatic programming technology has done much to make programs more communicable among programmers, there is a long way to go before we can take an integrated program of 100,000 instructions and make it "public property" for the user, the coder, the tester, the evaluator, and the on-site maintenance programmer. The only answer seems to be the documentation of the system on every level from sales brochures for management to instruction listings for maintenance engineers. Such documentation will require the development of new methods and new languages; more significantly, it will require a much more extensive use of the computer to assist in program production, documentation, and maintenance.

At the last ONR symposium on automatic programming held two years ago, the most popular theme was simplifying program input through the use of symbolic inputs, machine compilation and generation, algebraic translation, etc. Very little was said about checkout or debugging, training, or operation. I suspect that for many the past two years have been a period of realizing that automatic programming concepts must go beyond the input process into these other areas.

Large Programs for Control and Processing

Before considering these problems in more detail, consider some rudiments of large systems and large programs. Figure 1 represents a broad flowchart of a typical control and processing system such as might be used for air-traffic control, industrial-plant control, or commercial applications. The area inside the dashed line represents the control system; the area outside is the environment to be controlled. In general, control consists of a manual and an automatic component. Manual in-out data could use voice phones or radios, teletypes, meters, etc. Typical automatic inputs and outputs might be teletype data or high-bandwidth digital data from or to analog-to-digital converters.

The central control is a high-speed, general-purpose, digital machine that includes in-out terminal equipment and is controlled itself by the system program. Depending on the degree of system automation, manual control and processing might range anywhere from one half-awake computer operator (who will be awakened by an alarm) to a staff of several hundred operators and supervisors, each of whom must communicate directly with the computer. The machine can signal the man through indicator lights and alarms, cathode-ray displays, or printed data; the man can respond with digital keyboard inputs or a variety of analog-to-digital devices. Periodically, the computer records data for later analysis of system performance.

From the computer's point of view, then, the system consists of a wide variety of inputs and outputs, each with different data characteristics—peak rate, average rate, reliability, coding, etc. The system program must perform a wide variety of tasks.

1. It must remember the state of environment. Depending on the application, this may require from 100,000 to many billions of bits of information stored on drums, tapes, or photographic plates.

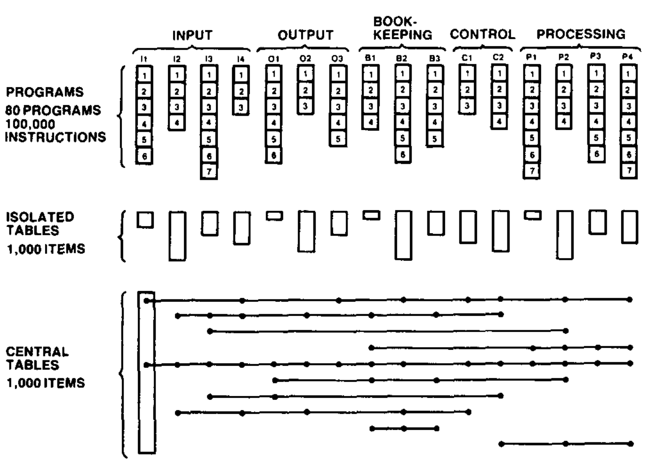

Figure 2. Static program organization. A system program of 100,000 instructions is organized into programming groups for input, output, etc. Each group contains several subprograms and requires both isolated and central tables.

2. It must sample each input either periodically or on demand, translate the data, test for reasonableness (usually in terms of the present state of the environment), and either revise its memory content accordingly or transmit the data for further processing.

3. It must, either periodically or on demand, calculate, monitor, correlate, predict, control, summarize, record, and decide.

4. It must encode and transmit outputs to all terminal devices.

5. Finally, the program must control the frequency and sequence with which it performs each input, output, processing, or bookkeeping task.

In order to give these features some physical meaning, let us attach rough numbers to a typical control problem. Figure 2 shows the organization of a typical 100,000-instruction program that contains 80 component subprograms. In other words, each subfunction requires a logically distinct subprogram containing an average of 1250 instructions. In the figure, each box (e.g., I12) represents a subprogram; they are grouped as follows.

1. There are four major *input* channels (e.g., punched cards, teletype, audio-bandwidth data link, and manual keyboards) designated by program groups I1 to I4. For each channel, several different types or sources of data are received by the control element. For example, I3 requires seven subprograms, I31 to I37.

2. There are four major *processing* functions, which require a total of 24 component subprograms. In an air-traffic-control application, a typical process might be: first, review all aircraft landing at all airports; next, monitor these with respect to airspace assignment and sudden trouble situations; finally, prepare a revised space assignment.

3. A third group of 15 subprograms are required for program *bookkeeping*. These programs coordinate communications between all other programs, monitor ystem load, and prepare summary data for output.

4. The *output* makeup programs use three channels—for example, cathode-ray display, audio-bandwidth data link, and teletype. Fourteen subprograms are required to scan the system memory and make up properly coded output messages.

5. Finally, seven *control* subprograms are required to control the timing, sequencing, and operation of all other subprograms.

The 100,000 instructions represent standing operational procedures for the system; they do not change as the system operates. The system memory, which is stored separately in system tables, can be broken down into two blocks: *isolated tables*, which store information required by one program group only (e.g., I2), and *central tables*, which store data shared by two or more program groups. In measuring the complexity of the table structure, the total table memory required by tables is not nearly so important as the number of items. In this sense, an item is defined as one unique type of information. A single item may be represented once in the tables (e.g., "process I42 is being performed"), or the item may be represented 1 million times (e.g., "customer account number").

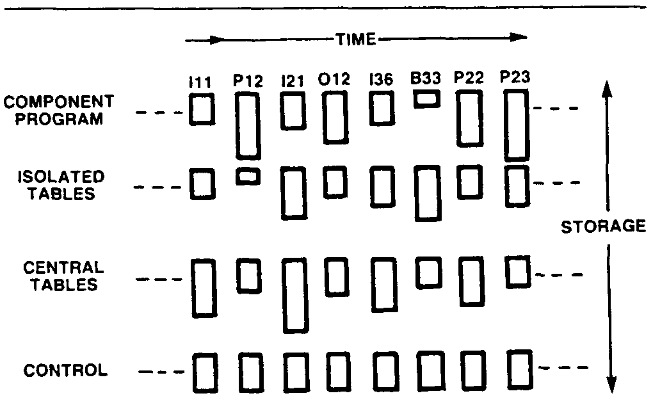

Figure 3. Dynamic program operation. Component subprograms (Figure 2) time-share the control computer. Each component program requires isolated and central tables; a control program, which remains permanently in storage, directs sequence and frequency of operation of component subprograms.

In the example given, 1000 items each are required for the isolated and central tables. For 10 of the central items, the program groups which set or use the item are shown; for example, the first item is used by I1, I4, O3, B2, C1, C2, P2, and P4. If 1000 such lines were drawn, the dot matrix would measure the communications (and complexity) within the program.

Figure 3 shows how the component subprograms time-share the machine to perform the control and processing functions (only a small portion of the complete program sequence is shown). Each component subprogram requires its isolated tables, pertinent portions of the control tables, and certain control subprograms. Eighty programs must time-share the machine. In general, some subprograms will operate unconditionally in a fixed sequence but at different frequencies; other programs will operate on demand.

Large-Program Systems—Centralized versus Decentralized

At this stage, we can consider the effect of program size and integration on the design, testing, and operation of the program. To date, there have been several programming systems of over 50,000 machine instruc-

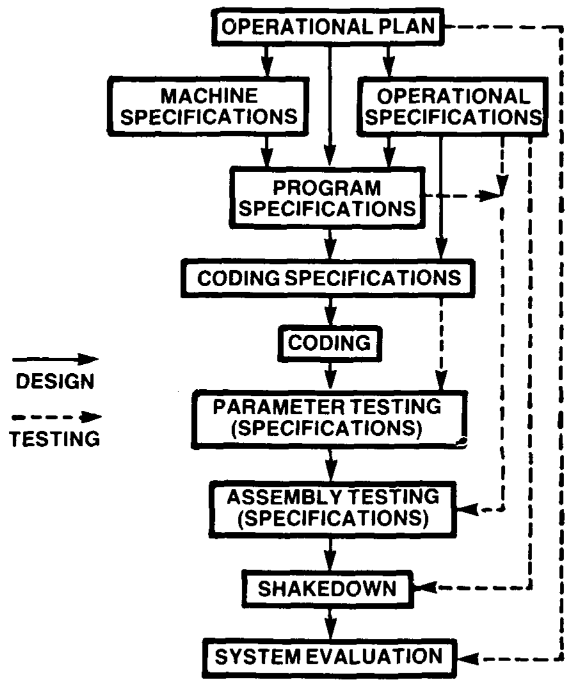

Figure 4. Program production. Production of a large-program system proceeds from a general operational plan through system evaluation; for example, assembly testing verifies operational and program specifications.

tions prepared for business and scientific applications. For the most part, however, these programs have been what might be called large *decentralized* programs; that is, the data-processing function has been divided into a dozen or so parts, and the communication between these parts has used blocks of data stored on magnetic tape or punched cards.

Usually, the format and coding (i.e., the structure) of these blocks can be unequivocally defined with relative ease. This considerably simplifies the design problem; after the blocks have been documented, groups of programmers can be assigned to each part with the assurance that little communication between these programmers will be necessary. If the fullest decentralization is desired, the component programs will not share machine storage or machine time. (In some applications, even different machines are used.)

Control of data processing in a decentralized system is primarily manual. Tape reels and programs are changed by computer operators (and even shipped to remote locations). If an unexpected result develops, an engineer or accountant or supervisor can print out intermediate data and decide after the fact what course should be taken. Efficient use of computer time need not be closely monitored, since there are no real-time constraints.

In testing or debugging one part of the system, data produced by other parts are not required until the very last moment that the system is put into operation. (Probably many of the decentralized systems currently in operation still contain many minor errors which are being compensated for daily by users who have become accustomed to these minor idiosyncrasies.)

The important point is that one can write a large programming system and still maintain a high degree of decentralization. Like most decentralizations, this course produces a system that contains semantic inconsistencies, ambiguities, and errors; operating inefficiencies result from duplication and wasted motion.

Real-time control systems have presented the first computer application where a very large program is required to perform all assigned functions, and yet where the disadvantages of decentralization cannot be tolerated. Success or failure of the system usually depends on efficient use of computer operating time. Internal control of the real-time program must be highly organized if efficient time and storage allocation are to be achieved, if the many in-out devices are to be adequately sampled, and if automatic decisions are to be made when unusual conditions develop within the program or from the external environment.

The control program must be *centralized.* This complicates design and coding since communication between component subprograms must have a high bandwidth. The use of each of the thousands of central table items must be coordinated between 100 or so component subprograms. Organized, readable specifications for the design and coding phase accomplish part of this task. Even then, only the most thorough testing of the entire program ensures that system threads have been carefully worked out, that incompatibilities are discovered, and that all contingencies are accounted for.

Preparation of a System Program

Figure 4 indicates the nine phases used at the Lincoln Laboratory in preparing a large system program. First, an *operational plan* defines broad design requirements for the complete control system consisting of the machine, the operator, and the system program. This plan must be prepared jointly by the computer systems engineers and the eventual user of the system.

From this plan, detailed *operational specifications* are prepared that precisely define the "transfer function" of the control system. In this representation, the computer, its terminal equipment, and the system program are treated as a black box. On the other hand, this description is sufficiently detailed that programmers can later prepare the system program using only machine and operational specifications. The operational specifications correspond to the equations the scientist gives a programmer; numerical analysis has yet to be performed.

Program specifications outline implementation of the operational black box by the system program. These specifications organize the program into component subprograms and tables, indicate main channels of program intracommunication, and specify time- and storage-sharing of the machine by each subprogram. Continuing the analogy, program specifications correspond to a broad flowchart of the solution.

After the operational and program specifications have been completed, detailed *coding specifications* are prepared that define the transfer function of each component subprogram in terms of the processing of central and isolated items. From these specifications, it is possible to predict precisely the output of the subprogram for any configuration of input items. The coding specifications also describe all storage tables.

Each component subprogram is *coded* using the coding specifications. Ideally, this phase would be a simple mechanical translation; actually, detailed coding uncovers inconsistencies that require revisions in the coding specifications (and occasionally in the operational specifications).

After coding, each component subprogram is *parameter tested* on the machine by itself. This testing phase uses an environment that simulates pertinent portions of the system program. Each test performed during this phase is documented in a set of *test specifications* that detail the environment used and the outputs obtained. In the figure, the dashed line indicates that parameter testing is guided by the coding specifications instead of by the coded program; in other words, a programmer must prove that he satisfied his specifications, not that his program will perform as coded. (Actually, test specifications for one subprogram can be prepared in parallel with the coding.)

As parameter testing of component subprograms is completed, the system program is gradually *assembled and tested* using first simulated inputs and then live data. For each test performed during this period, *assembly test specifications* are prepared that indicate test inputs and recorded outputs. Assembly testing indicates that a system program satisfies the operational and program specifications.

When the completed program has been assembled, it is tested in its operational environment during *shakedown*. At the completion of this phase, the program is ready for operation and evaluation.

Figure 5 indicates reasonable production costs that might be expected in preparing a system program of 100,000 instructions. Considering the present technology of program preparation, our experience does

PHASE	ENGINEERING MANPOWER (MAN-YEARS)	COMPUTER TIME (HR)	PAPER OUTPUT (PG)
Operational Plan	?	0	500
Operational Specs	30	0	2,500
Program Specs	10	0	500
Coding Specs	30	0	5,000
Coding	10	0	3,000
Parameter Testing	20	1,000	2,000
Assembly Testing	30	2,000	1,500
Shakedown	?	?	?
Evaluation	?	?	?
	130	3,000	15,000

Minimum Production Time = 18 Months

Figure 5. Production cost. Using present techniques, the production cost for a 100,000-instruction program can easily require $55 per instruction.

not indicate that these are at all overly pessimistic estimates. The estimates shown do not include training of programmers, preparation of ancillary programs, development of control-systems techniques, or overhead supporting activity. They include only engineering manpower required to produce the system program. Let us assume an overhead factor of 100 percent (for supporting programs, management, etc.), a cost of $15,000 per engineering man-year (including overhead), and a cost of $500 per hour of computer time (this is probably low since a control computer contains considerable terminal equipment). Assuming these factors, the cost of producing a 100,000-instruction system program comes to about $5,500,000 or $55 per machine instruction. *In other words, the time and cost required to prepare a system program are comparable with the time and cost of building the computer itself.*

The Lincoln Utility System

In order to simplify the preparation and operation of all programs, the Lincoln Laboratory has prepared a set of service routines called the Lincoln Utility System. This system was designed to assist all programmers in using the machine; its present size—40,000 machine instructions—is indicative of the importance attached to its role. The Lincoln system does not provide automatic-coding facilities in the conventional sense. Compared with systems that have been developed at computing centers where scientific and engineering calculations predominate, the Lincoln system has concentrated more on systematizing computer operation and program debugging than on developing automatic translation of programmer language into machine language. Design of the system followed these ground rules.

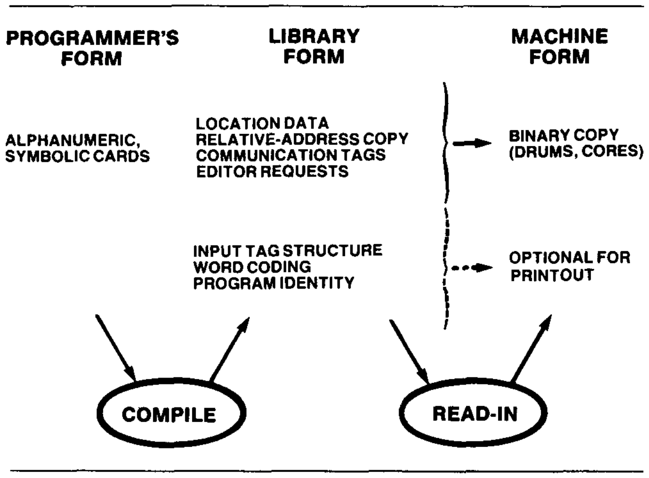

Figure 6. Program input process. With the Lincoln Utility System, compiled programs are stored with the programmer's full input structure; at read-in time, the program is finally converted to machine binary language. Even at this time the symbolic input structure is available to other service routines.

1. At the Lincoln Laboratory, most programs are prepared by relatively inexperienced programmers. As many features as possible were included to help them, yet no features were used that were so complicated that only experienced programmers could use them with facility. Also, programmers do not operate the machine during debugging; they are required to plan and document their operations beforehand.

2. Computer time for parameter testing, assembly testing, and system shakedown is scarce. A large effort has been devoted to systematizing and mechanizing computer operations in order to use minimum computer time.

3. The Lincoln Utility System includes several features that assist programmers in communication and documentation problems encountered during the design and testing phases of system program production.

4. The Lincoln Utility System contains extensive debugging features including facilities for remote, flexible card control of the computer and programs to be tested.

5. Programs are prepared in machine language because automatic coding techniques developed to date do not guarantee the efficient programming required for a real-time system. (In retrospect, this ground rule seems very shaky.)

6. The Lincoln Utility System, which is quite large, has not been so centralized that its initial production was delayed or that its revision and improvement are difficult.

With the Lincoln Utility System, programmers code in floating address using some subroutine requests, particularly for card input and printed outputs. When programs are compiled, they are stored on a magnetic-tape library with their full input structure; that is, the library copy contains program identity, a relative-address binary copy, assigned memory locations, a floating-address tag table, subroutine requests, etc. Storage in this form has several advantages. First, modifications to a program can be expressed in the floating-address input structure; for recompilation, the compiler does not require a complete program copy. Second, all postmortems during and after program operation are retranslated into input language; programmers do not write programs in symbolic form and receive fixed-address outputs. Third, major modifications in storage addresses and locations can be made to a checked-out program at the time the program is read into the machine because system design parameters are stored in a central communication pool (see Figure 6).

In order to debug programs, a "checker" facility is used. This is a service program of 10,000 instructions that allows the program to be tested—the checkee—to be operated either interpretively or noninterpretively under control of a pseudoprogram of executive instructions. When the checkee is operated in the interpretive mode, the checker automatically detects loops, arithmetic alarms, illegal in-out sequences, and illegal instructions. It stores a history of program operation including branches, change-registers, and in-out transfers. In the interpretive mode, the checkee cannot cause a machine halt; when alarm conditions are detected, the checker automatically generates special outputs and moves on to another job. The checker provides a wide variety of outputs including instruction-by-instruction printouts, dynamic change-register printouts, and alarm printouts. Using the executive instructions, a programmer can set machine registers or memory registers to test values; he can start and stop the checkee at selected locations; he can request different outputs for different regions of the program; he can request alarm outputs if the checkee transfers control outside a fixed region or if a loop of more than n cycles is performed; he can indicate the use of different executive subprograms depending on the results of checkee operation; he can indicate which portions of his program are to be performed noninterpretively. From a programmer's point of view, the checker is a special-purpose, checkout computer; it is a stored-program machine with highly flexible input, output, and control sections. (See Figure 7 for a sample executive program.)

All utility programs are controlled by utility control cards. Before a machine run, a deck of binary cards, checker executive cards, etc., is prepared. The operator places the cards in the reader, pushes one button, and the rest of the computer operation is automatic.

A final feature of the utility system is the use of a large communication pool of numerical parameters shared by all programmers. Each programmer can specify that constants or addresses in his program should be taken from the pool. Numbers in this pool are expressed symbolically by the programmer in both his coding specifications and his coded copy; the machine supplies proper numerical values at read-in time. These values may be unknown to the programmer and even changed from day to day. For example, communication tags are used for extracting information (usually table items) that is packed into a full word. The programmer need not know the exact location of the word in memory, nor the position of the information bits within the word. Communication tags are even used to indicate the location in memory of the program itself. A program-design group assigns specific numerical values to the tag pool from day to day, in some cases long after component subprograms have been debugged. Since numerical values are assigned only when the program is read into the machine, it is possible for system designers to move programs and tables within drum and core memory merely by changing constants in this pool. Only one central document needs to be revised, and minimum testing on the computer is required. Figure 8 indicates the allocation of the 40,000 instructions in the utility system.

PROGRAM	LENGTH
Compiler	10,500
Read-in	1,300
Library Merge-Output	4,700
Checker	7,500
Master Tape Load	2,000
In-Out Editors	2,400
Communication Pool	4,100
Utility Control	3,000
Numeric Subroutines	1,000
Miscellaneous	4,000
	40,500

Figure 8. Utility system. The Lincoln Utility System requires over 40,000 instructions as indicated.

Testing

It is debatable whether a program of 100,000 instructions can ever be thoroughly tested—that is, whether the program can be shown to satisfy its specifications under all operating conditions. Considering the size and complexity of a system program, it is certain that the program will never be subjected to all possible input conditions during its lifetime. For this reason, one must accept the fact that testing will be sampling only.

On the other hand, many sad experiences have shown that the program-testing effort is seldom adequate. When the program is delivered for operation, its performance must be highly reliable because the control system is a critical part of a much larger environment of men and machines. One error per 100,000 operations of the entire program can easily be intolerable.

As a result of facing this problem for some time at the Lincoln Laboratory, the following principles have evolved to govern our testing.

First, parameter testing (i.e., testing of individual component subprograms in a simulated environment) cannot be too thorough. This phase must discover all errors internal to the program and its individual coding specifications. Even if parameter testing were perfect (which it never is!), many errors in system design would remain to be discovered during subsequent assembly testing.

Second, initial assembly testing should be performed using completely simulated inputs. There are several reasons. First, only in this way can all test inputs be carefully controlled and all tests be reproducible. Second, when errors are discovered with a new program using live inputs, there will always be a question whether the program or the machine is at fault. Integration of the system program with terminal equipment should not be attempted until the assembled program has been well tested.

A third principle is that the testing facility used during the assembly test phase must contain extensive, flexible facilities for recording both system

```
CHECKER     CARDS/DELAYED
01  NI  11A   11R
02  AL  07
03  LP  25
04  LR  12   13
05  TR  12   13
06  BG  12A   13Z+6
07  LP  4
08  LR  14   15   16
09  BG  14A   16L+5
10  CC
11  QT
```

Figure 7. Sample executive program. The Lincoln checker is controlled by pseudoinstructions. The executive program shown indicates regions of the checkee to be performed noninterpretively (01 NI), alternate executive instructions in case of checkee alarm (02 AL), maximum-length loops (03 LP), legal regions of checkee operation (04 LR), checkee output mode (05 TR), etc.

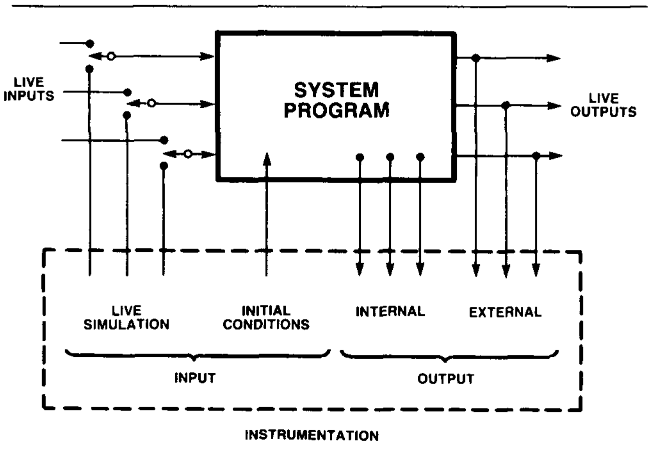

Figure 9. Test instrumentation. Proper testing of a control system requires an automatic facility for simulating inputs and monitoring outputs. With this facility, extensive testing can be performed and outputs produced for either diagnosis of system errors or verification of proper system performance.

outputs and intermediate outputs (i.e., subprogram intercommunications). Without this facility, rapid and reliable diagnosis of system errors is impossible. After a test has been conducted and errors found, it should be possible to correct the error before the program is put on the machine again.

The need for comprehensive simulated inputs and recorded outputs can be satisfied only if the basic design of the system program includes an instrumentation facility. In the same way that marginal-checking equipment has become an integral part of some large computers, test instrumentation should be considered a permanent facility in a large program.

Figure 9 illustrates the role of test instrumentation in a system program. Each of the live inputs can be individually simulated; this allows simultaneous testing with both live and simulated data. In addition, the input instrumentation allows easy setting of initial conditions for system memory; this feature is performed by a special-purpose translation program that converts alphanumeric card data into system tables.

System Program	100,000 Instructions
Utility Programs	40,000
Special Programs	10,000
Test Instrumentation	20,000
Operational Instrumentation	30,000
	200,000 Instructions

Figure 10. Production of a system program. Supporting programs whose total size equals the system program may be required to simplify production and testing of the system program.

The output instrumentation "probes" both internal data (for diagnosis) and external data (for simpler verification).

One final principle should govern system-program testing: *All successful parameter and assembly tests must be reproducible throughout the life of the system program.* These tests must be documented in test specifications that detail the reasons for the tests, required inputs, operating procedures, and expected outputs.

The original reason for this requirement stemmed from the problem of revising the program once it was operational. The slightest modification to a program can be successful under limited testing conditions and yet still cause critical errors for other operations. Since it is desirable to retest the program thoroughly after each modification, a large battery of test inputs must be available. We have discovered two other incidental advantages of detailed test documentation. First, a programmer's tests tend to be more organized and more exhaustive if he must document them. Second, if machine-versus-program reliability is ever questioned, retesting is possible. If a known program and a known test fail, the machine is at fault.

Supporting Programs

The utility and test-instrumentation programs discussed are only part of the complete set of supporting programs. In addition, special programs, which assist preparation of the system program, are used to generate routine data blocks, perform special translation of alphanumeric data into parameter tables, assemble program-sequence and timing parameters, etc.

Operational instrumentation programs are used during system shakedown and evaluation. They contain simulation and recording facilities that are far more realistic and operationally oriented than the test instrumentation. System recorded data are analyzed with a battery of data-reduction programs (Figure 10).

Documentation—Design and Revision

As indicated earlier, documentation of the system program is an immense, expensive job. The output will run to tens of thousands of pages of specifications, charts, and listings. At the Lincoln Laboratory, these currently include the following.

Operational specifications
Program specifications
Coding specifications
Detailed flowcharts
Coded program listings
Parameter test specifications

Assembly test specifications
System operating manuals
Program operating manuals

The need for this battery of documents is obvious. The system and its program must be learned and used by management, operational-design engineers, system-operating personnel, training personnel, program-design engineers, programmers, program-test engineers, evaluation personnel, and, if more than one system is maintained, on-site maintenance programmers. Each of these users has very different needs.

Consider the problem of revising the system once the program is operational in the field. A minor change in the operational specifications is proposed. First, the cost and effects of this change must be evaluated in terms of the program, the operators, and, often, the machine. In order to make the change, several hundred revisions may be required in the specifications. If the change is approved, these documents must be changed, operating manuals revised, and the program modified and thoroughly tested. The wave of changes must be coordinated smoothly.

Digital computers are often sold to management on the basis of their programmed flexibility. We have said, "If your doctrine or procedure changes, no messy, expensive, time-consuming equipment changes will be required." In reality, this is not true today. The cost of the documentation mentioned is only a symptom of the design-coordination problem in large systems.

How can we reduce this cost? Obviously, as we have done already, by more extensive use of the computer. (At the laboratory, we have partially gone in this direction through the use of punched cards for storing all central design data. Decks are easily revised, fed into the system program, or listed for the user.) We must systematize design, production, and documentation both in the small and in the large. By "in the small," I mean what is already being done in automatic programming. Instead of an algebraic translator, we need a unified "bookkeeping–logical-processing–algebraic translator." Before we get this, we will surely need much more research on coding languages and representations. Eventually, programming should become a two-way conversation between the imprecise human language and the precise, if unimaginative, machine. The programmer will say, "Do this," and the machine will answer, "OK, but what happens if . . . ?" The smallest gain of such a system would be the elimination of the coding, parameter testing, and parameter test-specification phases. Unfortunately, these phases represent only one quarter of the system cost.

Documentation "in the large" poses a bigger challenge.

1. What integrated set of documents are required to design and describe a large system?
2. What language should these documents use?
3. How should they be cross-referenced?
4. Can we eventually store them on magnetic tape and let the computer analyze, print, and code?

Summary

The techniques that have been developed for automatic programming over the past five years have mostly aimed at simplifying the part of programming that, at first glance, seems toughest—program input, or conversion from programmer language to machine code. As a result of progress in this area (and a growing number of experienced programmers), we find that large programs can now be produced; unfortunately, they are difficult to test and document. If the newest very-high-speed, large-memory computers are to be fully utilized, we must develop automatic programming procedures so that they allow cheap production of highly reliable, easily revised, well-documented system programs.