CENTRAL FLOW CONTROL SOFTWARE DEVELOPMENT: A CASE STUDY OF THE EFFECTIVENESS OF SOFTWARE ENGINEERING TECHNIQUES

Peter C. Belford and Richard A. Berg
Computer Sciences Corporation
Silver Spring, Maryland, U.S.A.

Thomas L. Hannan
Federal Aviation Administration
Washington, D.C., U.S.A.

Abstract

The purpose of this paper is to present cost and error data collected during the development cycle of a large-scale software effort, to analyze this data in comparison with other available data from similar projects, and to evaluate the effectiveness of the techniques utilized on the project. The project being reported on is Computer Sciences Corporation's development of the Central Flow Control Software System for the Federal Aviation Administration's Air Traffic Control System Command Center. Analysis of the cost data provides insight not only into the added development costs associated with severely limiting module sizes, but also into the effectiveness of various cost estimation techniques. The error data analysis supports the usefulness of the software engineering techniques which were used on the project in conjunction with definitive module-level test requirements. The paper provides a foundation upon which to establish the development and data collection environment for future software systems.

Introduction

Major software development projects employing controlled software engineering techniques occur infrequently over the life of an organization. It is even more uncommon for these controlled projects to have the management support required to collect sufficient data to evaluate the techniques utilized. In order that subsequent developers may have the opportunity to select effective software engineering techniques, more project details need to be collected, evaluated, and the results stored for their use.

This paper describes a major software development project, the software engineering practices employed, the data collection procedures and results obtained, and conclusions about the effectiveness of the practices as implemented on the project. The experiences gained from the project were not of the controlled, psychometric variety; the budget and deadlines were real, and the various lapses in data and, perhaps, in resolution, reflect these facts. Taking this into consideration, the following material is presented, not as conclusive proof of the efficacy of a particular methodology, but as experiences resulting from a representative large-scale development project.

Project and Approach

The Federal Aviation Administration (FAA) operates an Air Traffic Control System Command Center (ATCSCC) whose function is balancing the flow of air traffic so that in-flight delays are minimized. The Central Flow Control (CFC) System provides automation support for this function. A computer complex in Jacksonville, Florida, is linked with the major FAA nationwide facilities to provide up-to-date information about proposed flights and in-flight movements. This demand information is fed into a data base along with airport capacity information, and is subsequently used by ATCSCC personnel in an on-line query environment. The overall system can be described as an on-line (inquiry), real-time (flight data), transaction-oriented (independent asynchronous activities) information system.

The project was initiated in late 1975. It was decided to establish a rigid software architectural requirement and functional (stimulus/response) specification to ensure that the system would be able to evolve along with the application. Maintainability with an emphasis on modifiability[1] was the prime objective prior to contract initiation. However, a more recent grouping of these factors indicates that flexibility tended to become the primary goal, with maintainability second, and reliability a weak third for the initial system.[2]

Toward this end, the FAA specified the functional requirements,[3] utilizing an existing hardware configuration, and provided a baseline operating system. Computer Sciences Corporation (CSC) was competitively awarded a contract to modify the operating system, develop data base management and applications software, and provide support software for system development, generation, test, and performance evaluation. The award was made in April 1977, and the system was delivered in January 1979, six months prior to the operational readiness date of July 1979.

The selection of a development methodology was based as much on management criteria as on technical criteria. It was decided to move stepwise through the

development process, checkpointing each phase by baselining the output. With the requirements definition phase and the functional specification phase completed and their results baselined, the remaining development effort was allocated to three major phases: (1) the system design phase; (2) the unit design, code, and test phase; and (3) the system test phase. The final two phases were performed four separate times. Each time a portion of the software (called a Build) was implemented to demonstrate a subset of the functional capabilities of the system.

Software Engineering Methods

It became clear that successful execution of the development phase (within both budget and schedule) would require careful front-end attention paid to: (1) product definition, (2) tool utilization, (3) practice standardization, and (4) organizational structure. An attempt was made (with varying degrees of success) to define software and documentation products so that they would evolve naturally, and so that as much of these products as possible would be in machine-readable form. A set of tools and practices were selected to assist in those areas which had been troublesome in the past, most notably control and communication. Finally, an organization was formed based upon a Work Breakdown Structure (WBS), which was to serve as an accounting, data collection, and referencing mechanism as well as a basis for scheduling.

The system design phase involved the least infusion of modern practices and yielded the least amount of software engineering data. The products of the phase were (1) the Program Design Specification (PDS), (2) the System Development Plan, and (3) the System Test Plan. The PDS was developed using Hierarchy plus Input-Process-Output (H-IPO) diagrams. The Hierarchy (H-) diagrams eventually evolved into execution diagrams, which gained reasonable support, but the IPO diagrams, which met with some initial success, were quickly supplanted by Program Design Language (PDL). The PDL allowed for six basic structured constructs. Data was also addressed hierarchically in the PDS, and the CODASYL Data Description Language was employed.[4] This complemented the PDL, and both were updated for inclusion in the final documentation.

The unit design, code, and test phase was the most amenable to incorporation of modern practices and was the source of the most useful data. Additionally, the organization was adjusted during this phase to provide both a Quality Control group and an Independent Data Base Administrator.

Unit design was accomplished using PDL, and the unit test specification was generated by automated analysis of the PDL. This tool also verified the PDL for compliance with project quality standards. The test specifications were for unit testing based upon the decision-to-decision (DD) path structure of the design. The DD paths were determined from the constructs utilized in the PDL and were a relationship of the number of possible branch paths between constructs. The DD paths eventually demonstrated some highly advantageous properties (discussed subsequently in the section entitled "Presentation of Reduced Data"). Units, or modules, were constrained to single entry, single exit, and single function. They were documented at this stage by a machine-readable Prologue, which contained identification, operational-characteristic, cross-reference, data-definition, and processing-logic information.[5] Prologues were also automatically analyzed for completeness. Modules were subjected to Walkthroughs,[6] and the resultant error data, together with weekly resource expenditure sheets, the module PDL, Prologue, and Test Specifications were incorporated into individual Software Engineering Notebooks.

Unit coding and testing was based on the design in the PDL and Test Specifications. Structured code (in JOVIAL)[7] was derived from the PDL, and Test Procedures were developed based upon the Test Specifications; both of these items were then incorporated into the notebooks. Data interfaces were controlled by use of the JOVIAL data-description facility, COMPOOL, which was regulated by the Data Base Administrator. Resource utilization and error data were collected as they were in the previous phase; this data is presented and evaluated later in the paper.

The system test phase was carried out for each build by an Independent Test Team composed of both developer and user personnel. System Test Specifications and Procedures, in contrast to those at the module level, were for functional testing, and were traceable back to the requirements definition phase through the functional specifications. Errors were recorded on Test Team Trouble Reports, which were included in Build Test Reports.

Automated documentation tools were employed at the system level. The JOVIAL Automated Verification System (JAVS)[8] was modified for this project, and was used to produce program hierarchy (calling tree), program structure (DD paths), and cross-reference information, as well as to support the degree of test case coverage obtained. Intermediate JAVS outputs were also scanned by a specially developed software tool, which produced a Data Item Dictionary.

CFC Software Data Collection

Three general categories of data were collected on the CFC Project: activity data, software module structural data, and software error data. The activity data collected was man-hours expended by project personnel at the software subsystem level. Software module structural data included counts of the total number of executable source statements plus a count of the number of DD paths from the design for each module. Software error data included both walkthrough and execution time

errors. Walkthrough errors were recorded by quality control at the design walkthrough, and execution errors were recorded by the responsible programmers or the Independent Test Team, depending on the level of testing. Error data was collected both at the module (or unit) level and at the system level.

The procedures for data collection were based on the types of data and the mechanism used to collect each type. The data types and the corresponding basic collection devices were as follows:

- Activity data
  - Personnel: time accounting given at the subsystem level
- Module structural data
  - Physical structure: module source code
  - Logical structure: module PDL and test procedures
- Software error data
  - Module: programmer error log
  - System: test team trouble reports

Control of data collection was maintained within the development departments and monitored by the quality control staff on a continuing basis. Organization and delivery of the final data package was performed by the quality control staff. The following subsections describe in detail the forms and procedures used to collect and summarize the three data types.

Activity Data Collection

Time data in man-hours was collected for all personnel on the project at the functional subsystem level. This time data was collected by means of a CFC Project Data Collection Form that was distributed weekly and filled out along with weekly time cards by each person on the project. The number of man-hours expended in each subsystem for design, code, test, library maintenance, throwaway code development, management, and quality assurance functions performed within the subsystem was compiled from this data.

The data was collected for each of the four builds of the system. Within each build, data was presented for the Application/Simulation (APS) Subsystem, the Data Assembler (DA) Subsystem, the Data Base (DB) Subsystem, and the Data Reduction and Analysis (RA) Subsystem. These subsystems represented the major functions performed by the CFC System that were programmed in JOVIAL. The time spent in functional design and system level testing was not included in the build man-hour results.

Software Structural Data Collection

Two types of software structural data were obtained from the CFC Development Project. The data was acquired from examination of the module source code and PDL. The module source code data was an estimate of the physical size of a module measured by a count of the total number of executable source statements. The PDL data defined the logical structure of the design of the module, obtained by a count of the total number of DD paths in the design. Another measure of logical structure utilized was a count of the number of test cases used to exercise all the DD paths identified within a module during the design phase. The test case data was obtained from the test procedures in the Software Engineering Notebooks.

Software Error Collection

Software error data on the CFC Project was collected during the design and testing phases. Errors detected in the design phase were measured by the total number of design errors discovered during the design walkthrough of a given module. These error counts were collected by the quality control staff during each walkthrough.

Errors encountered during module or unit level testing were collected by the responsible programmer and summarized for each build. System level errors were collected by the Independent Test Team for each unsuccessful run. The failure information was derived from an analysis of the program output. If a failure was caused by more than one error, all errors were listed. Errors were also recorded during system acceptance testing. While not on a build basis, these errors provided information about problems encountered during the integration of the final system.

Presentation of the Data

This section presents the raw data collected on the CFC Project. Data is presented in the three categories described in the preceding section, and is presented for every build in which it was available.

Activity data is shown in Tables 1 through 4. Tables 1 through 3 show the man-hours spent per subsystem in the activity categories of detailed design, code, unit test, library maintenance, throwaway code development, and management and technical direction by task leaders in the subsystem plus quality control functions performed by subsystem personnel. Data is presented for Builds 2, 3, and 4 of the system. Build 1 of the system is not presented since it occurred at a time prior to the institution of the reporting mechanism. However, total man-hours statistics are available for Build 1. Table 4 presents the total man-hours spent per subsystem for each of the four builds of the system. Tables 5 through 7 show the number of executable lines of code, the number of DD paths, and the number of test cases,

TABLE 1. BUILD 2 MAN-HOURS IN REPORTED ACTIVITY CATEGORIES

CATEGORY	APS SUBSYSTEM	DA SUBSYSTEM	DB SUBSYSTEM	RA SUBSYSTEM	TOTALS
Design	1293	885	462	0	2640
Code	731	432	362	0	1525
Test	1114	834	711	0	2659
Library Maint.	315	111	122	0	548
Throwaway Code	77	121	198	0	396
Other*	564	346	175	0	885
TOTALS	4094	2729	2030	0	8853

TABLE 2. BUILD 3 MAN-HOURS IN REPORTED ACTIVITY CATEGORIES

CATEGORY	APS SUBSYSTEM	DA SUBSYSTEM	DB SUBSYSTEM	RA SUBSYSTEM	TOTALS
Design	1937	775	378	1557	5247
Code	717	381	414	591	2303
Test	1215	985	591	1198	4080
Library Maint.	192	118	24	83	417
Throwaway Code	96	113	65	145	419
Other*	913	165	309	414	1801
TOTALS	5171	2537	1981	3979	14667

TABLE 3. BUILD 4 MAN-HOURS IN REPORTED ACTIVITY CATEGORIES

CATEGORY	APS SUBSYSTEM	DA SUBSYSTEM	DB SUBSYSTEM	RA SUBSYSTEM	TOTALS
Design	416	154	753	1085	2408
Code	170	196	498	397	1261
Test	763	957	201	529	2450
Library Maint.	489	199	109	24	821
Throwaway Code	60	9	7	85	161
Other*	861	172	522	247	1802
TOTALS	2779	1387	2090	2637	8893

TABLE 4. MAN-HOURS EXPENDED BY SUBSYSTEM FOR EACH BUILD

BUILD	APS SUBSYSTEM	DA SUBSYSTEM	DB SUBSYSTEM	RA SUBSYSTEM	SUBTOTAL
1	1480	1843	898	0	5221
2	4094	2729	2030	0	8853
3	5171	2537	1981	3979	14567
4	2779	1387	2090	2637	8893
TOTAL	13524	8996	7399	6615	36034

TABLE 5. EXECUTABLE LINES OF CODE PER SUBSYSTEM PER BUILD

BUILD	APS	DA	DB	RA	TOTAL
1	760	1357	840	0	3257
2	2759	1656	922	0	5337
3	2360	2933	2061	3035	10389
4	1503	939	1153	1364	4959
TOTAL	7482	7185	4676	4399	23742

TABLE 6. DD PATHS PER SUBSYSTEM PER BUILD

BUILD	APS	DA	DB	RA	TOTAL
1	233	530	146	0	909
2	987	403	250	0	1640
3	799	543	503	1068	2915
4	435	154	323	343	1255
TOTAL	2454	1632	1222	1411	6719

TABLE 7. TEST CASES PER SUBSYSTEM PER BUILD

BUILD	APS	DA	DB	RA	TOTAL
1	119	170	76	0	365
2	344	135	173	0	652
3	283	127	134	197	741
4	169	26	139	88	422
TOTAL	915	558	482	385	2340

*Includes management and technical direction by subsystem leader and quality assurance functions performed by subsystem personnel.

P. Belford
CSC
4 of 20

respectively, for each of the four system builds. Table 3 gives the error statistics collected for Builds 2 and 3 of the system and the total number of software errors detected during acceptance testing. Build 1 was completed before error reporting mechanisms were in place, and Build 4 results were not available.

TABLE 3. ERROR STATISTICS FOR THE CENTRAL FLOW CONTROL SYSTEM

BUILD	DESIGN WALKTHROUGH	MODULE* LEVEL TESTING	SYSTEM LEVEL TESTING
2	44	290	18
3	63	223	20
Acceptance Testing	N/A	N/A	21

*APS and DA Subsystems only.

Presentation of Reduced Data

The purpose of this section is to analyze the data presented in the preceding section. This analysis attempts to provide quantitative evaluation of the effectiveness of the software engineering techniques utilized on the CFC Project. Since the project has been completed and accepted by the FAA, this analysis evaluates the final results of the project.

The first analysis performed concerned the relationship of the sizes of software entities compared to the cost of their development. The size of a software entity was measured in terms of both the number of lines of executable source code and the number of decision paths (DD paths) in the design. Cost was always measured in man-hours expended.

Module Level and Build Level were the two software entities evaluated. At the module level, Figure 1 presents a plot of the number of man-hours expended versus the number of lines of executable code for each of a randomly selected set of 50 modules. Any conclusive trend is not at all obvious by analysis of the curve. However, since the true comparison of developmental costs is in terms of the number of lines of code produced per man-hour expended, further evaluation was performed.

The data, therefore, was evaluated in terms of the number of lines of code developed per man-hour (a "relative" measure of cost) as a function of the number of lines of code in the module (Figure 2). If a curve could be formed from the data, the optimal module size would be the highest point on the curve (i.e., the module size for which the maximum number of lines of code would be developed in each man-hour expended).

The data presented in Figure 2 shows that module sizes of between 0 and 40 lines of executable source code never (in fact, without exception in this sample) produce a productivity of more than one line of code produced per man-hour expended. However, for modules of greater

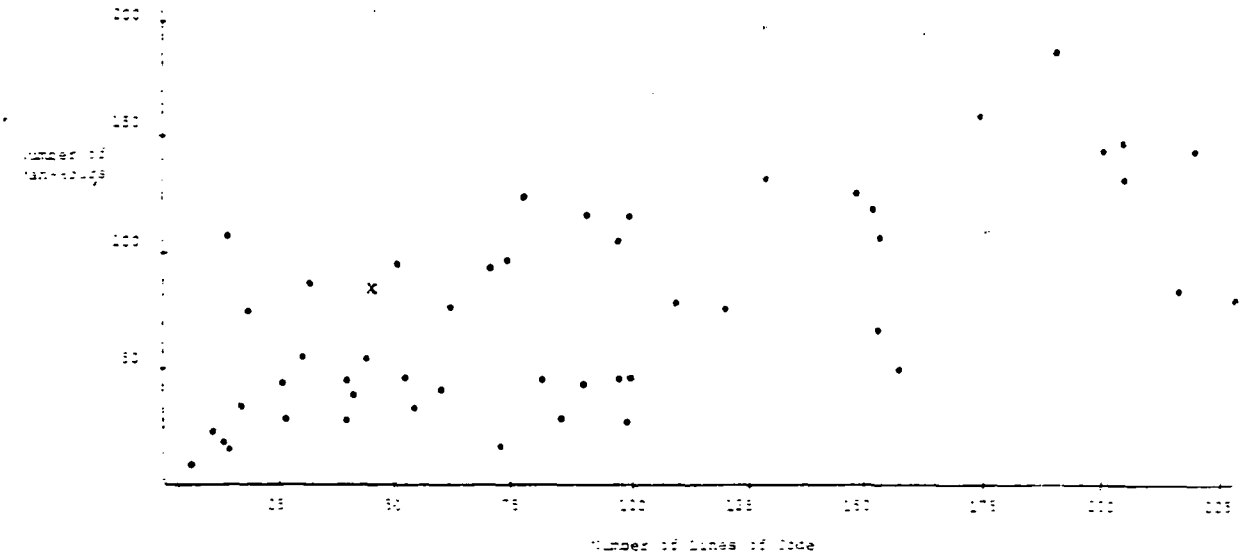


FIGURE 1. CORRELATION OF MAN-HOURS TO MODULE SIZES

P. Belford
CSC
5 of 20

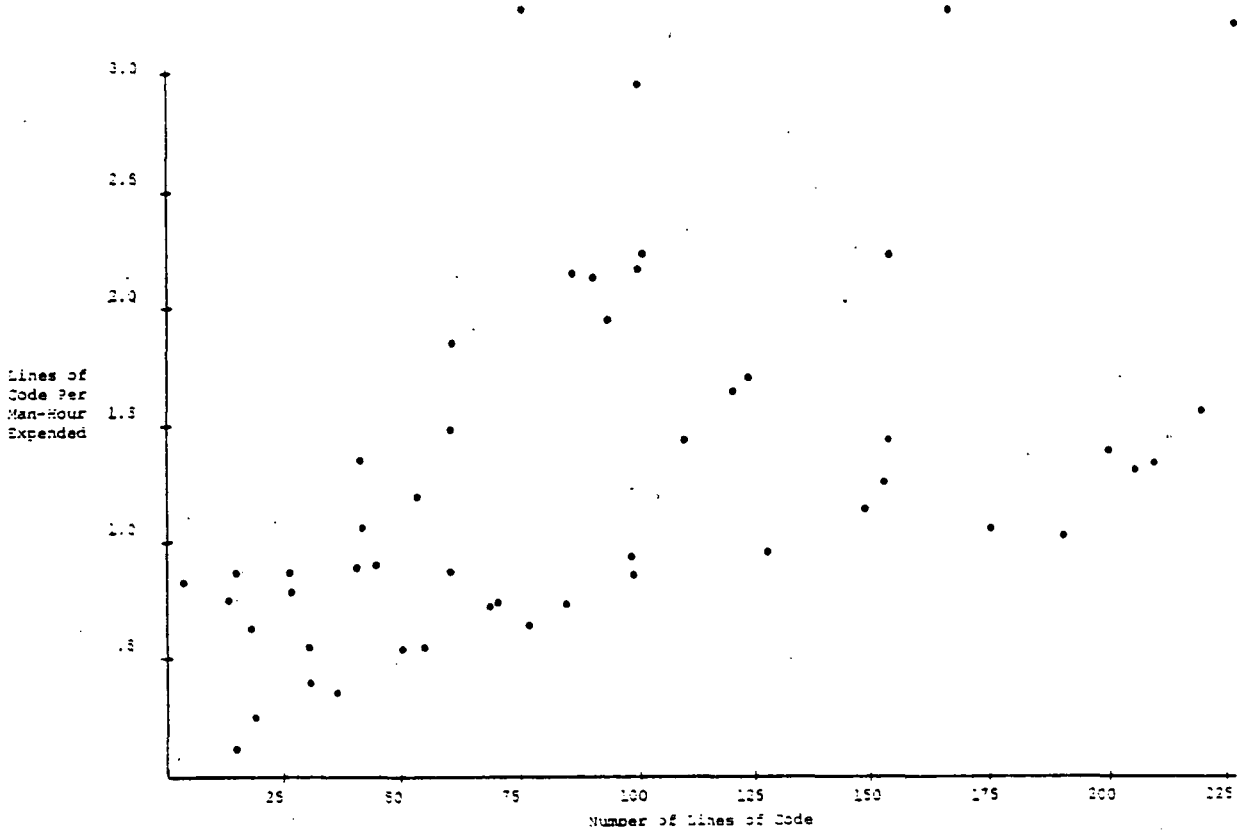


FIGURE 2. CORRELATION OF LINES OF CODE DEVELOPED PER MAN-HOUR COMPARED TO MODULE SIZE

than 40 lines of executable source code (greater than 100 without exception), the productivity is generally greater than one line of code produced per man-hour expended.

Although the data presented only reflects development costs and not operational and maintenance costs, the general theory of restricting modules to 50 lines of executable source code or less does not appear to be cost effective when considering developmental costs only. If the "single entry, single exit, single function" rule is strictly adhered to, the module size should not be utilized as a standard. In contrast, it appears that modules of 40 lines of code or greater should be encouraged. It is important to note that the CFC modules, regardless of size, followed the "single entry, single exit, single function" concept of module definition.

In order to support these conclusions, a measure of module complexity, number of DD paths in the design, was also compared to developmental costs. The same 50 modules were utilized. This time, the relative cost in number of DD paths generated per man-hour was plotted against the complexity in number of DD paths. Figure 3 shows the results of this analysis. Again, the more complex the module, the lower the relative developmental costs.

These cost analyses seem to point out that within the "single entry, single exit, single function" concept, the larger the module, the lower the per-unit-of-size developmental costs.

The results of this analysis seem to indicate that restrictions limiting module size should not be a driving factor. Single-function modules of 100 or even 200 lines of executable source code should be acceptable.

The relative size of a build was also analyzed. Figure 4 shows the relationship between the size of a build in number of lines of executable source code per subsystem and the number of man-hours expended against that subsystem in a build. This plot shows an obvious correlation of size to cost. With the single exception of the DA Subsystem for Build 3, which was accomplished on third shift (the implications of which will not be discussed here), the cost-to-size relationship is linear. Hence, if the single-function module concept is controlled, the size of a build appears to have no impact on the relative cost of producing that build.

Another analysis was performed to evaluate the relationship between actual development costs and cost estimation techniques. The number of man-hours expended

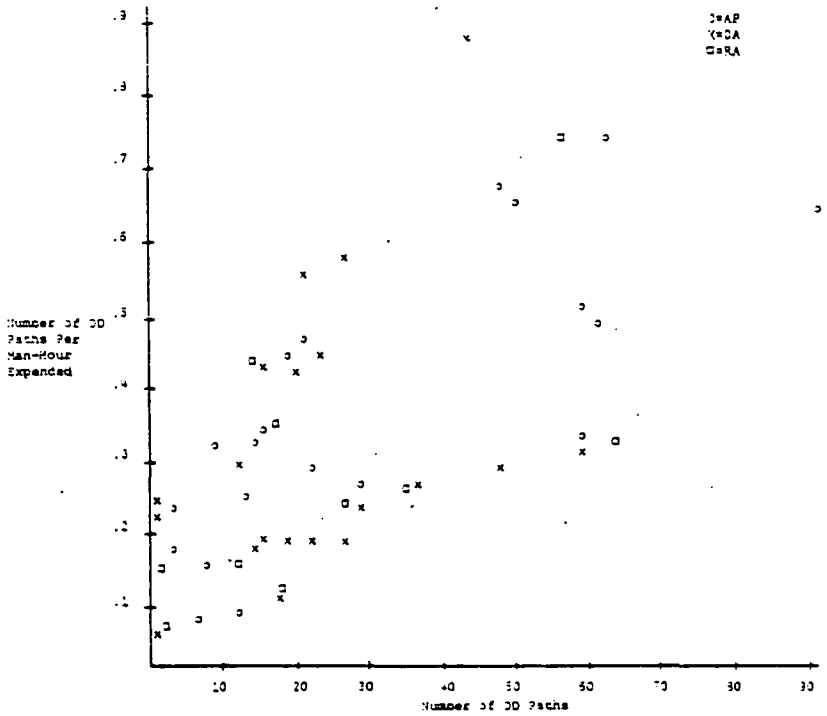


FIGURE 3. CORRELATION OF NUMBER OF DD PATHS PER MAN-HOUR TO MODULE COMPLEXITY

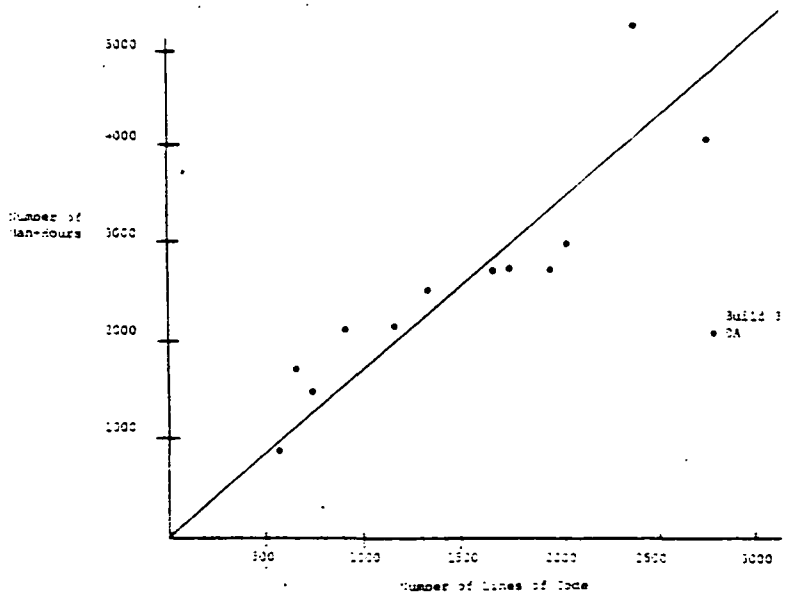


FIGURE 4. CORRELATION OF MAN-HOURS TO BUILD SIZE

per line of executable source code was compared to the number of man-hours expended per DD path in the design and to the number of man-hours expended per test case (see Table 9). This comparison was accomplished using a coefficient of variation (i.e., the ratio of the standard deviation to the mean).

TABLE 9. COSTING PARAMETER COMPARISONS

Build/Subsystem	Man-Hours/Lines Code	Man-Hours/DD Path	Man-Hours/Test Case
Build 1 APS	1.35	6.35	12.44
Build 1 CB	1.56	6.15	11.92
Build 1 CA	1.45	5.36	10.53
Build 2 APS	1.48	6.15	11.90
Build 2 CB	1.20	5.12	27.81
Build 2 CA	1.35	5.77	20.21
Build 3 APS	1.19	5.07	18.27
Build 3 CB	1.40	5.73	14.85
Build 3 CA	.86	4.66	13.98
Build 3 PA	1.31	5.72	13.29
Build 4 APS	1.73	6.39	15.44
Build 4 CB	1.81	6.47	15.34
Build 4 CA	2.25	12.25	32.58
Build 4 PA	1.93	7.69	29.97
COV	35.13%	32.13%	75.53%

The statistical correlation was not diverse enough to support differentiation between DD paths in the design and lines of code in terms of total cost estimation techniques. Number of man-hours per test case was shown not to be a viable cost estimation technique due to its high coefficient of variation (COV). Therefore, a separate analysis was performed to determine a better costing parameter that could be utilized at each of the detail design, coding, and testing phases of development. Since the known quantity at the completion of each phase is DD paths in the detail design phase, lines of code in the coding phase, and test cases in the testing phase, this information could be utilized to refine original cost estimates as a project progresses. The data presented on costing provides enough information to support development of initial cost estimation algorithms based on actual development products (e.g., DD paths, lines of code, and test cases).

An analysis was performed on these cost estimations utilizing the data from the APS Subsystem for all four builds. Three cost estimation algorithms were developed, one for each development phase. The basis for these algorithms is the data previously presented in Table 9. The "Lines of Code" algorithm utilizes 1.34 times the number of lines of executable source code to yield the estimated number of man-hours. The "DD-Path" algorithm utilizes 5.34 times the number of DD paths in the design to yield the estimated number of man-hours. The "Test Case" algorithm utilizes 14.76 times the number of test cases to be performed. All three algorithms were then applied to the other subsystems. These results are presented in Table 10.

The low "average percentage deviation" of the DD-path algorithm shows that the number of DD paths provides an accurate prediction of the total man-hours required. This technique provides the added advantage of supporting periodic updating of the estimation as the actual number of DD paths is finalized. If PDL is used, this variable is known early (i.e., at the completion of the design phase).

The effectiveness of the software engineering techniques utilized was also analyzed. The true effectiveness of the software engineering techniques can best be measured by the reliability and maintainability of the product. Although data is not yet available to support definitive reliability and maintainability measures of the CFC System, error rate data was available within CFC and was used to estimate the effectiveness of the software engineering techniques employed on the project.

In order to evaluate the CFC error rate with some defined industry averages, the Rome Air Development Center (RADC) Software Reliability Study[9] was utilized. This report presents the error rate of two JOVIAL projects at system level testing. This error rate worked out to be about 1 error in every 35 lines of code. The CFC error rate, calculated from Tables 5 and 9, shows 1 error in every 28 lines of code, detected at the module level. At the system level testing of CFC, an order of magnitude fewer number of errors (i.e., 1 error for every 382 lines of code) than at the module level were found. During final acceptance level testing, a 3-month user/customer-conducted testing phase, still fewer errors were found. In the 23,742 lines of executable code discussed in this paper, only 21 software errors were detected. This is an error rate of 1 error in every 1,131 lines of code.

The error rate implies two conclusions about the development approach. First, the software engineering techniques utilized were very effective. The CFC error detection rate comparable to that reported in the RADC study was noticed an entire development phase earlier. More errors were found earlier, presumably leaving fewer errors in the final system. This should result in a significant cost savings, since the cost to correct an error increases the longer it remains in the code. Second, the testing approach proved to be quite effective. The quantification of module level testing requirements, specifying that all DD paths in the design must be exercised at the module level, provided significant advantages over the traditional testing approach carried on by the programmers. This concept proved to exercise 98 percent of all the decision paths in the code. Additionally, within one subsystem where these unique module level test specifications were rigidly applied, the acceptance testing error rate was only 1 error detected in every 1,371 lines of executable code; whereas within a subsystem where module level test specifications were loosely applied, the acceptance error rate was 1 error detected in every 733 lines of code. This roughly implies an overall effectiveness increase of over 100 percent

per line of executable source code was compared to the number of man-hours expended per DD path in the design and to the number of man-hours expended per test case (see Table 9). This comparison was accomplished using a coefficient of variation (i.e., the ratio of the standard deviation to the mean).

TABLE 9. COSTING PARAMETER COMPARISONS

Build/Subsystem	Man-Hours/Lines Code	Man-Hours/DD Path	Man-Hours/Test Case
Build 1 APS	1.35	5.35	12.40
Build 1 CB	1.66	6.15	11.12
Build 1 DA	1.45	5.38	13.53
Build 2 APS	1.48	4.15	11.90
Build 2 CB	1.20	3.12	17.81
Build 2 DA	1.65	6.77	20.22
Build 3 APS	1.13	4.47	16.17
Build 3 CB	1.40	5.73	14.85
Build 3 DA	.98	4.66	19.38
Build 3 RA	1.31	3.72	13.19
Build 4 APS	1.73	6.39	16.40
Build 4 CB	1.81	6.47	15.36
Build 4 DA	1.35	12.25	22.58
Build 4 RA	1.33	7.69	19.37
COV	15.13%	32.13%	21.53%

The statistical correlation was not diverse enough to support differentiation between DD paths in the design and lines of code in terms of total cost estimation techniques. Number of man-hours per test case was shown not to be a viable cost estimation technique due to its high coefficient of variation (COV). Therefore, a separate analysis was performed to determine a better costing parameter that could be utilized at each of the detail design, coding, and testing phases of development. Since the known quantity at the completion of each phase is DD paths in the detail design phase, lines of code in the coding phase, and test cases in the testing phase, this information could be utilized to refine original cost estimates as a project progresses. The data presented on costing provides enough information to support development of initial cost estimation algorithms based on actual development products (e.g., DD paths, lines of code, and test cases).

An analysis was performed on these cost estimations utilizing the data from the APS Subsystem for all four builds. Three cost estimation algorithms were developed, one for each development phase. The basis for these algorithms is the data previously presented in Table 9. The "Lines of Code" algorithm utilizes 1.34 times the number of lines of executable source code to yield the estimated number of man-hours. The "DD-Path" algorithm utilizes 5.34 times the number of DD paths in the design to yield the estimated number of man-hours. The "Test Case" algorithm utilizes 14.76 times the number of test cases to be performed. All three algorithms were then applied to the other subsystems.. These results are presented in Table 10.

The low "average percentage deviation" of the DD-path algorithm shows that the number of DD paths provides an accurate prediction of the total man-hours required. This technique provides the added advantage of supporting periodic updating of the estimation as the actual number of DD paths is finalized. If PDL is used, this variable is known early (i.e., at the completion of the design phase).

The effectiveness of the software engineering techniques utilized was also analyzed. The true effectiveness of the software engineering techniques can best be measured by the reliability and maintainability of the product. Although data is not yet available to support definitive reliability and maintainability measures of the CFC System, error rate data was available within CFC and was used to estimate the effectiveness of the software engineering techniques employed on the project.

In order to evaluate the CFC error rate with some defined industry averages, the Rome Air Development Center (RADC) Software Reliability Study[9] was utilized. This report presents the error rate of two JOVIAL projects at system level testing. This error rate worked out to be about 1 error in every 35 lines of code. The CFC error rate, calculated from Tables 5 and 8, shows 1 error in every 28 lines of code, detected at the module level. At the system level testing of CFC, an order of magnitude fewer number of errors (i.e., 1 error for every 382 lines of code) than at the module level were found. During final acceptance level testing, a 3-month user/customer-conducted testing phase, still fewer errors were found. In the 23,742 lines of executable code discussed in this paper, only 21 software errors were detected. This is an error rate of 1 error in every 1,131 lines of code.

The error rate implies two conclusions about the development approach. First, the software engineering techniques utilized were very effective. The CFC error detection rate comparable to that reported in the RADC study was noticed an entire development phase earlier. More errors were found earlier, presumably leaving fewer errors in the final system. This should result in a significant cost savings, since the cost to correct an error increases the longer it remains in the code. Second, the testing approach proved to be quite effective. The quantification of module level testing requirements, specifying that all DD paths in the design must be exercised at the module level, provided significant advantages over the traditional testing approach carried on by the programmers. This concept proved to exercise 98 percent of all the decision paths in the code. Additionally, within one subsystem where these unique module level test specifications were rigidly applied, the acceptance testing error rate was only 1 error detected in every 1,371 lines of executable code; whereas within a subsystem where module level test specifications were loosely applied, the acceptance error rate was 1 error detected in every 733 lines of code. This roughly implies an overall effectiveness increase of over 100 percent

TABLE 10. COST ESTIMATION COMPARISONS

		"Lines of Code" Algorithm		"DD Path" Algorithm		"Test Case" Algorithm	
	Actual Man-Hours	Estimated Man-Hours	Deviation (%)	Estimated Man-Hours	Deviation (%)	Estimated Man-Hours	Deviation (%)
Build 1 DB	898	994	10.69	953	6.01	1122	24.94
Build 1 DA	2843	3601	26.66	3095	8.86	3985	40.17
Build 2 DB	2030	1696	16.45	1460	28.08	1077	46.95
Build 2 DA	2729	3047	11.65	2354	13.54	1993	26.97
Build 3 DB	2861	3792	31.62	2938	1.98	2863	.62
Build 3 DA	2537	5397	112.73	3193	25.86	1975	26.09
Build 3 RA	3979	5584	40.37	6237	56.79	4384	10.21
Build 4 DB	2090	2122	1.53	1886	9.76	2052	1.82
Build 4 DA	1887	1176	37.68	899	52.36	384	79.65
Build 4 RA	2637	2510	4.82	2003	24.04	1299	50.74
Average Deviation			29.42		22.61		30.32

attributable to using these module level test specifications.

Summary

In conclusion, the authors feel that a significant baseline has been established in the quantification of software engineering techniques. The success of the CFC project, together with the supporting data that was collected, provides a foundation upon which to build successor systems. The key factors to be considered in setting up these future system programs is to understand the development environment, to collect data during the development effort to support the upgrading of projections, and to support the periodic evaluation of the data to provide insight into the product. This should provide sufficient visibility to keep a project out of trouble, while at the same time supporting evolution of more effective project plans.

References

1. Boehm, B. W., et al., Characteristics of Software Quality, TRW Systems Group, TRW-SS-73-09, December 1973.

2. McCall, J. A., et al., Factors in Software Quality, Final Technical Report (3 vols.), Rome Air Development Center, RADC-TR-77-369, November 1977.

3. Central Flow Control Computer Program Specifications, Final Report (5 vols.), Federal Aviation Administration, FAA-RD-76-157, September 1976.

4. CODASYL Data Base Task Group Report, Conference on Data Systems Languages, April 1971.

5. Central Flow Control Quality Assurance Plan, Final Report, Computer Sciences Corporation, CSC/SD-78/6060, April 1978.

6. Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Vol. 15, No. 3, 1976.

7. NAS Operational Support System, IBM 9020 JOVIAL Language Manual, NASP-9288-02, May 1975.

8. Gannon, C., et al., JAVS - JOVIAL Automated Verification System, JAVS Technical Report (3 vols.), General Research Corporation, CR-1-722/1, June 1978.

9. Software Reliability Study, Rome Air Development Center, RADC-TR-74-250, October 1974.

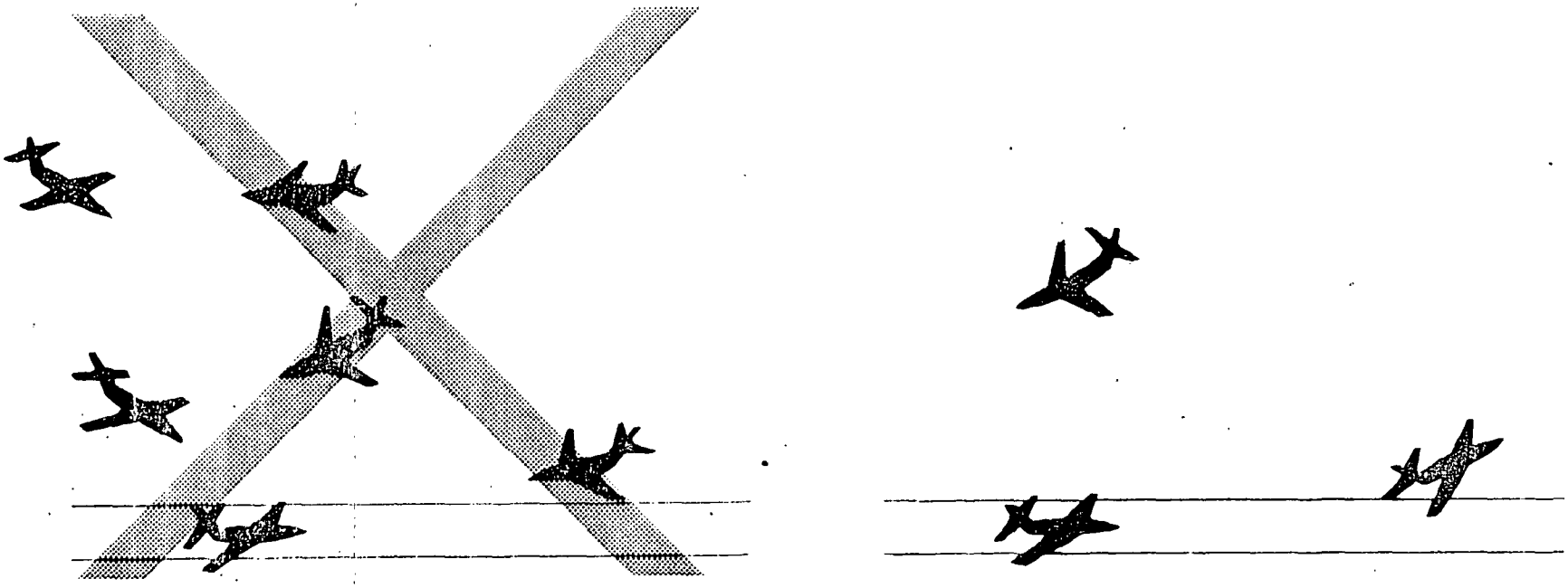
CENTRAL FLOW CONTROL

T. L. HANNAN, FEDERAL AVIATION ADMINISTRATION

R. A. BERG, COMPUTER SCIENCES CORPORATION

BEL-0-79

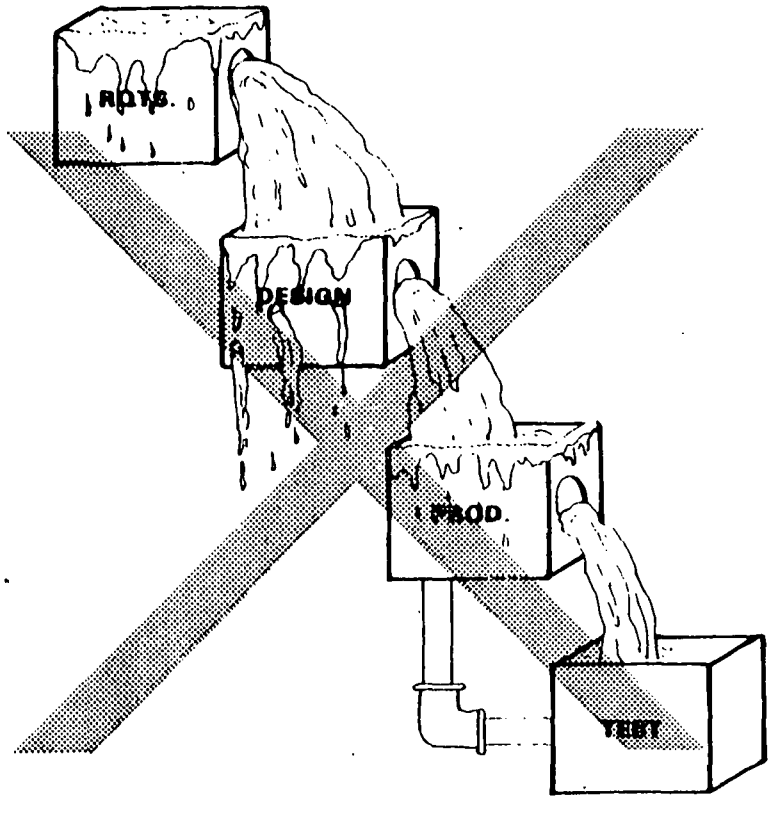
CENTRAL FLOW CONTROL FUNCTIONAL PURPOSE

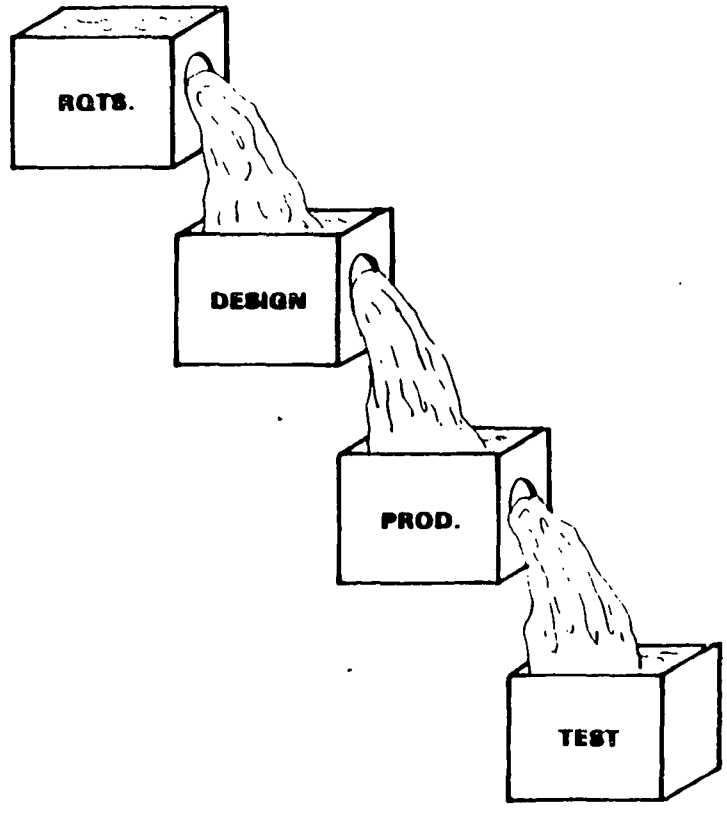


- LONG RANGE TRAFFIC OVERLOAD PREDICTION
- REDUCTION IN AIR TRAFFIC DELAYS
- FUEL SAVINGS

BEL-8-78

CENTRAL FLOW CONTROL
SOFTWARE ENGINEERING PURPOSE





- PLANNED APPROACH
- STRUCTURED TESTING
- COMPREHENSIVE DATA COLLECTION

SEL-8-78

P. Belford
CSC
13 of 20

DATA ANALYSIS AREAS AND RESULTS

- MODULE SIZE ANALYSIS
- BUILD SIZE ANALYSIS
- COSTING METHODOLOGY RESULTS
- TESTING METHODOLOGY RESULTS

BEL-8-78

DATA COLLECTED

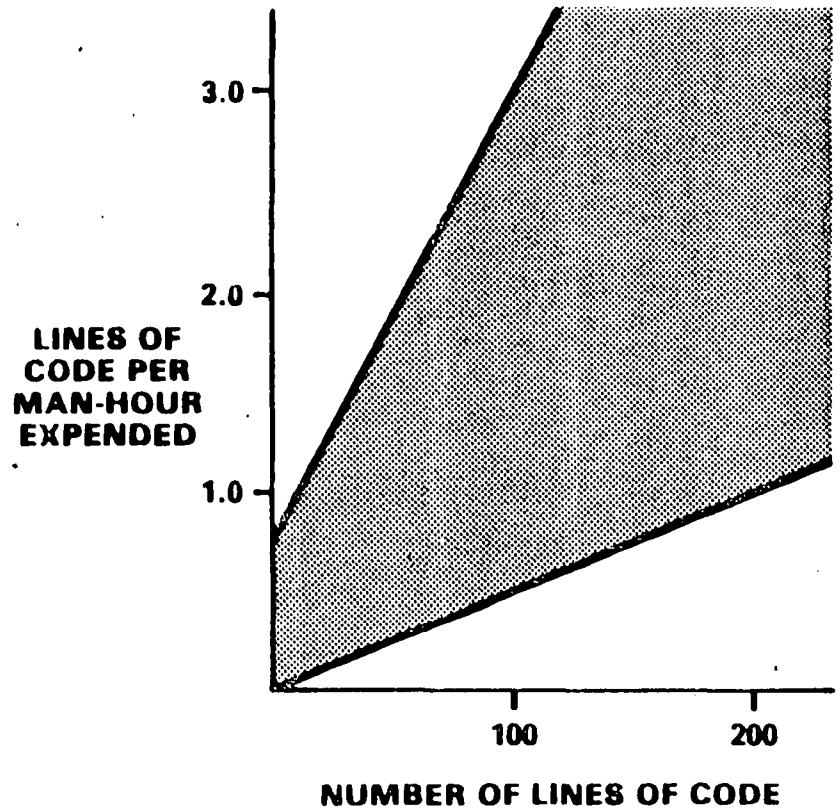
- **ACTIVITY DATA**
  - **PERSONNEL: TIME ACCOUNTING GIVEN AT THE SUBSYSTEM LEVEL**
- **MODULE STRUCTURE DATA**
  - **PHYSICAL STRUCTURE: MODULE SOURCE CODE**
  - **LOGICAL STRUCTURE: MODULE PDL AND TEST SPECIFICATIONS**
- **SOFTWARE ERROR DATA**
  - **MODULE: PROGRAMMER ERROR LOG**
  - **SYSTEM: TEST TEAM TROUBLE REPORTS**

DATA PRESENTED

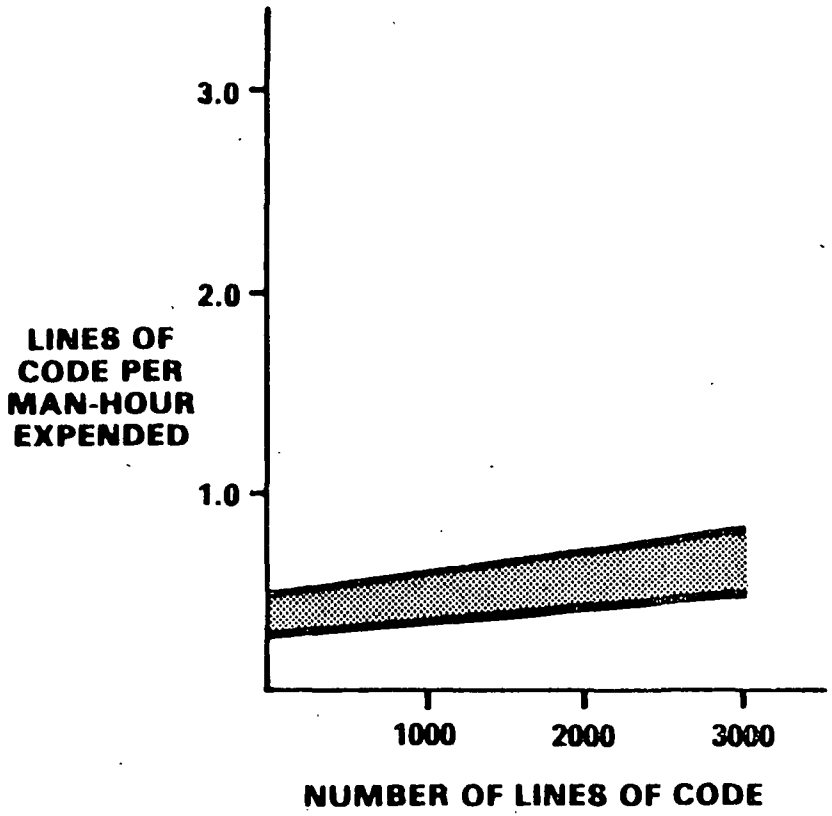
- DATA PRESENTED ON THE MAJOR SUBSYSTEMS OF CFC THAT WERE CODED IN JOVIAL
- 38034 MAN-HOURS OF DIRECT LABOR AND 23742 LINES OF EXECUTABLE CODE REPRESENTED OVERALL
- OVERALL DATA PRESENTED:

  MAN-HOURS, EXECUTABLE LINES OF CODE, DD-PATHS, TEST CASES, ERRORS

OVER LIMITING SIZE
NOT COST EFFECTIVE



MODULE SIZE



BUILD SIZE

COSTING METHODOLOGY

- MAN-HOURS PER LINE OF CODE/DD PATH/TEST CASE WERE CALCULATED FOR EACH SUBSYSTEM FOR EACH BUILD

- RESULTS:

  COV (LINE OF CODE) = 35%
  COV (DD PATH) = 32%
  COV (TEST CASE) = 76% (ELIMINATED AS USEFUL ESTIMATOR)

- APS SUBSYSTEM USE AS ESTIMATOR FOR OTHER SUBSYSTEMS AND RESULTS COMPARED WITH ACTUALS

- RESULTS:

  AVERAGE DEVIATION (LINES OF CODE) = 24%
  (DD PATHS) = 22%

- CONCLUSION:

  DD PATHS CAN BE USEFUL IN REFINING INITIAL COST ESTIMATES

TESTING METHODOLOGY

- DISCIPLINED APPROACH
- TEST SPECS GENERATED FROM PDL
- TESTS EXERCISE ALL DD PATHS IN DESIGN
- VERIFIED AT DESIGN WALKTHROUGHS

RESULTS

- 1 ERROR PER 28 LINES OF CODE AT MODULE LEVEL
- 1 ERROR 382 LINES OF CODE AT SYSTEM LEVEL
- 1 ERROR PER 1131 LINES OF CODE AT ACCEPTANCE LEVEL

NOTE: SUBSYSTEM RIGIDLY FOLLOWING PDL/TESTING STANDARDS
1 ERROR PER 1871 LINES OF CODE DETECTION AND CORRECTION TIME AVERAGED 8 PERSON HOURS.

SUBSYSTEM LOOSELY FOLLOWING DL/TESTING STANDARDS
1 ERROR PER 733 LINES OF CODE DETECTION AND CORRECTION TIME AVERAGED 40 PERSON HOURS.

BEL-9-79

SUMMARY

- BUILD SIZE HAS LITTLE COST IMPACT
- OVER RESTRICTING MODULE SIZE IS NOT COST EFFECTIVE IN THE INITIAL DEVELOPMENT
- RIGID ADHERENCE TO THE METHODOLOGY CAN REDUCE COST
- THE METHODOLOGY DID NOT SIGNIFICANTLY REDUCE THE NUMBER OF ERRORS BUT DID ALLOW EARLY DETECTION OF MOST ERRORS
- THE NUMBER OF DD PATHS IS DIRECTLY RELATED TO LINES OF EXECUTABLE CODE AND CAN BE USED TO REFINE COST ESTIMATES AFTER THE DESIGN STAGE

BEL-9-79