AT&T Technical Journal
Vol. 64, No. 1, January 1985
Printed in U.S.A.

System 75:

Software Development Tools

By T. J. PEDERSEN, J. E. RITACCO, and J. A. SANTILLO*

(Manuscript received July 11, 1984)

Efficient development of high-quality software requires a comprehensive set of software development tools. Tools used within the System 75 office communication system project support a hierarchical model of development, and range from compilers, assemblers, and debuggers to high-level tools that drive the software manufacturing process and aid in monitoring software quality and performance. Tools are applied in each of the many steps required to develop, test, integrate, and maintain product releases. Staff roles and procedures for use of these tools encompass a set of development and release-management cycles that begin with the individual developer and extend into the field support organization. The roles and procedures are flexible and easily customized to support various individual and group assignments. Specific tools to be described include the Object Generation System, the Local Administrative Tool Kit; the Oryx/Pecos Test, Inquiry, and Control System; tools for testing processes in isolation; manufacturing and distribution tools; performance-measurement tools; source-control and change-management tools; and tools for program update and system analysis at field sites.

I. INTRODUCTION

Tools play important parts in every aspect of producing System 75 office communication system software, from initial development and

* Authors are employees of AT&T Information Systems Laboratories, an entity of AT&T Information Systems, Inc.

unit test to support of systems installed at field sites. A group within the System 75 development organization is responsible for acquiring and developing special tools for the project. Tool developers and product developers are therefore closely allied and draw upon each other's expertise in planning, implementing, adapting, and applying tools as project needs are perceived. This paper presents the results of that collaboration.

Section II provides background on the product hardware, programming languages, and host computing environment from a tools perspective. Section III gives an abstract description of the software development cycle. Section IV describes the software structure that models the System 75 product. The remaining sections describe the subsystem development, project integration, system test and field support stages, along with specific tools and procedures used at each stage.

II. PROJECT ENVIRONMENT

Certain elements of the System 75 project environment have important influence on development support. These elements include processors and operating systems used in the product, programming languages used for product development, computing resources, and size and composition of the software development community.

2.1 Processors and operating systems

System 75 uses 8086, 8088, and 8051 microprocessors from Intel Corporation. The processors serve in different functional roles[1] and vary in size and complexity. For example, the Switch Processing Element (SPE)* is an 8086 with a large main memory, which runs the Oryx/Pecos operating system.[2] The network control and port circuit "angel" processors are 8051's with smaller memories. High-level software functions are implemented in SPE application processes; lower-level functions are implemented in network control and angel processor firmware.

2.2 Languages and compilation tools

The 8086 and 8088 processors are programmed almost entirely in C language.[3] The compiler is supplemented by a large collection of tools that operate on object files. Of particular importance are enhanced linkers that produce multisection object files to model the product operating system run-time environment.

The 8051 processors are programmed in SMAL51, an enhanced

* Acronyms and abbreviations used in the text are defined at the back of the *Journal.*

assembly language. The SMAL (Structured Macro Assembly Language)[4] syntax resembles C in that its instructions are similar to C language assignment statements or "function calls." Higher-level control constructs (e.g., if-then-else, switch) are also provided. SMAL51 thus provides a compromise between the clarity of expression of a higher-level language and the memory space and run-time performance advantages of an assembly language.

2.3 Computing resources

Figure 1 illustrates the computing resources used in System 75 development. Software development and laboratory model support take place on *host computers* that run the *UNIX*™ operating system. Computing resources are divided along organizational, and therefore functional, lines. A high-speed Local Area Network (LAN) permits rapid communication among the host computers, and a shared file system arrangement allows large collections of files to be delivered and shared efficiently.

A test environment consists of System 75 laboratory models supported by the host computers. A laboratory model is an instrumented System 75 with a sufficient variety of terminal equipment to demonstrate feature operation. High-speed data links are used to transfer programs between host computers and models and to access symbol-table information for software testing.

2.4 Software development community

The software community is organized into groups of developers. The number of developers supporting a given processor ranges from only

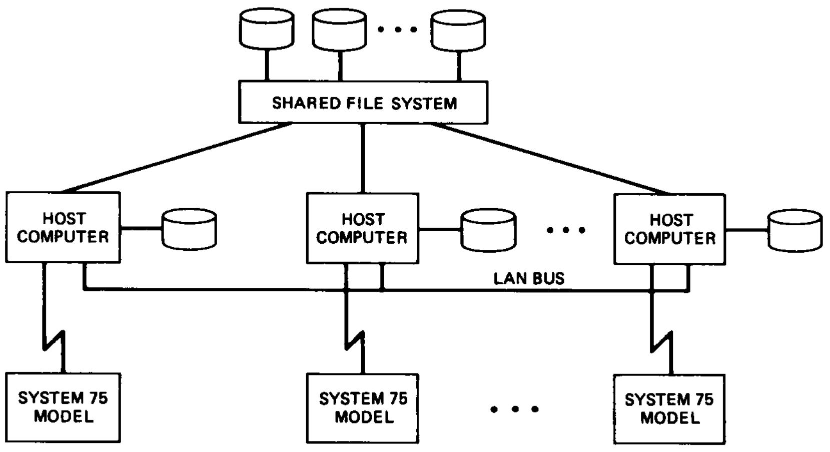

Fig. 1—Computing resources.

a few people for a typical 8051 processor to several groups for the SPE. The SPE groups are organized to correspond to a functional decomposition of the software. Development takes place simultaneously on all components of the system in accordance with a set of product specifications and development plans and a schedule.[5] Milestones in the schedule are planned for collecting, validating, and distributing software among the development groups at key intermediate stages.

Subsequent discussion will center on software development for the SPE. Developers of software for other processors apply a subset of the tools and methods to be discussed.

III. THE DEVELOPMENT CYCLE CONCEPT[6]

The overall process of software production and maintenance encompasses development, integration, system test, and release of a complete product. At a more microscopic level, this process can be viewed conceptually to take place in a set of cyclic activities. Repeated sequences of "develop, integrate, release" steps take place within each cycle. Semantics of the terms Develop, Integrate, and Release differ somewhat among the different types of cycles, and the term "development" is commonly applied to practically any activity connected with software production. However, the following definitions apply generally to the discussion in this section:

- Develop—Create or modify code
- Integrate—Combine and synchronize work of several developers
- Release—Deliver the results to others.

Note that transitions between the steps usually imply satisfaction of acceptance criteria. The development cycle concept is particularly useful because it expresses interfaces among individuals and organizations that deal with the software product.

A somewhat idealized model of System 75 software production is shown in Fig. 2. Stages in the overall process are shown at the left side, and microscopic D-I-R cycles are shown within each stage. A brief explanation follows; later sections explain activities at each stage in more detail. Independent groups of developers build and unit test (D) software components and then deliver (R) the components to project integration. Project integrators combine delivered components from all groups (I), make changes to correct integration flaws (D), and deliver the product back to developers (R). Each development group "buys back" the integrated software into its own environment (I) to test against in producing the next release. Some releases are forwarded from project integration to system test, where the D-I-R cycle models correction of troubles found during final testing (D, I) and delivery to controlled introduction field sites (R).

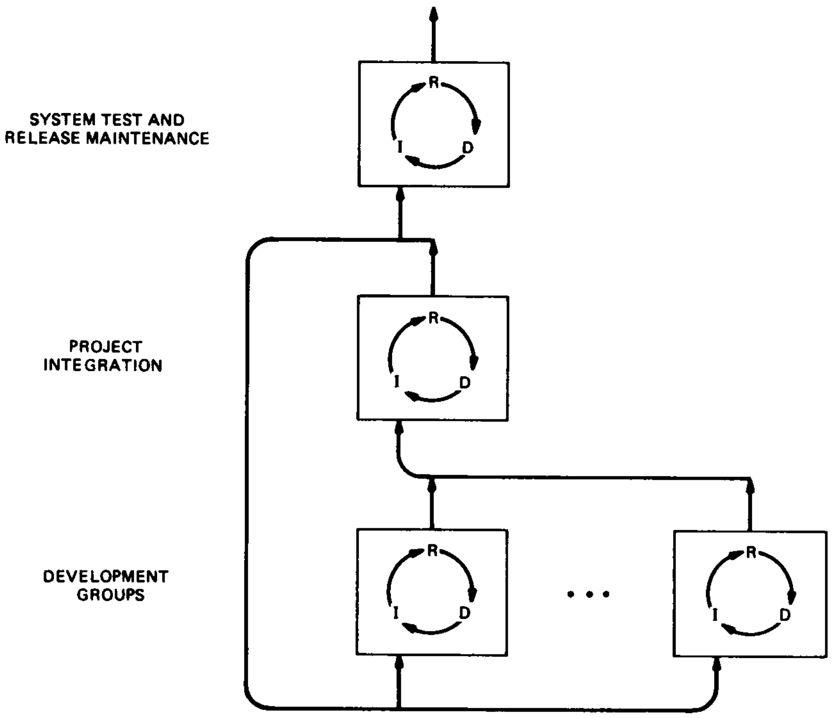

Fig. 2—Development cycle concept.

IV. SOFTWARE STRUCTURE AND TOOLS

A software structure, and tools that build and administer components within the structure, are introduced in this section. An important property of the structure and tools is that they promote, but do not rigidly enforce, uniformity. The project environment, the cyclic model of software development, and the nature of the product itself all influenced the design of these tools. Acceptance of the tools by the software development community was fostered by involving product developers early in the design. Application of the tools at successive stages of software production will be discussed in later sections.

4.1 Object generation system

Because build procedures can become as complex as the product itself, the same build tool should be used at every stage of software production. The primary software build tool used in generating System 75 software is OGS (Object Generation System), an interface to the `make`[7] program supplied with the *UNIX* operating system. The basic components of OGS are conventions for naming files and directories, a collection of makefiles, and a set of build commands.

OGS assumes that source and object code is organized in a uniform directory structure as shown in Fig. 3. The structure has three major levels:

- Book—a directory that contains source and object codes for either a process or a library
- Subsystem—a collection of related books
- Project—a collection of subsystems.

Subdirectories within a book, or subbooks, are also supported. Each directory level and source file type has a unique suffix. Example directory and file suffixes are

`.pj` for project
`.ss` for subsystem
`.p` for process book
`.b` for library book
`.d` for process subbook
`.db` for library subbook
`.O` for object directory
`.c` for C source file
`.h` for header file
`.o` for object file
`.a` for library archive file.

OGS allows code within a single directory structure to be built for

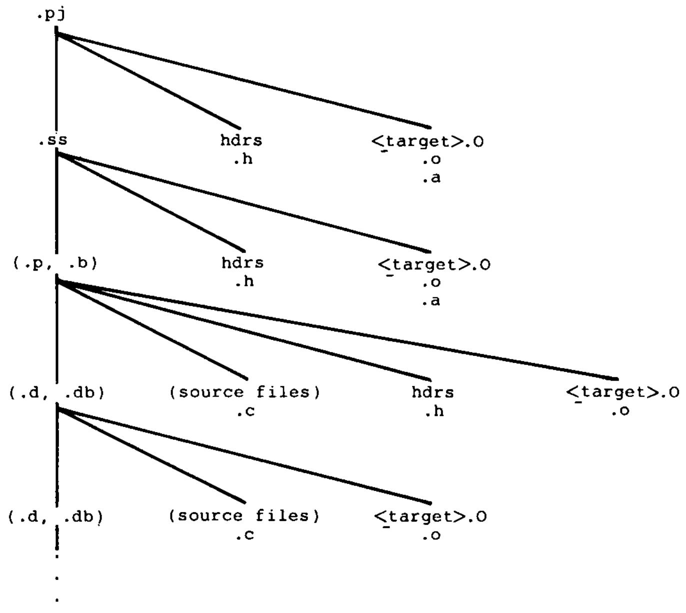

Fig. 3—Generic OGS directory structure.

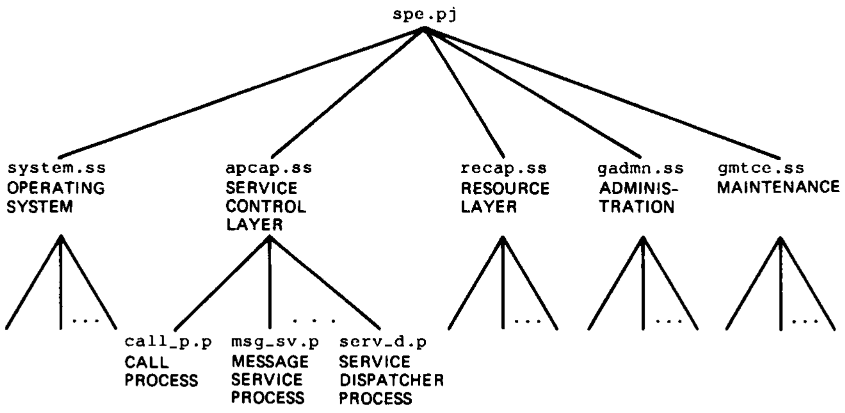

Fig. 4—System 75 OGS directory structure.

several target processors. Object files are placed in target-specific object directories. In addition, the structure incorporates header and library scoping conventions. The scope of a header file is determined by its placement in a book, subsystem, or project "hdrs" directory. Similarly, the scope of a library is determined by the object directory in which the archive is placed.

OGS takes advantage of the uniform directory structure by employing *generic makefiles*. OGS uses one standard makefile per directory level. This has two major advantages. First, all directories at the same level, e.g., all processes, are normally built in the same way. When necessary, build directives can be customized to handle exceptions. Second, developers need not be concerned with constructing makefiles. Once the generic makefile for each level is installed in a standard place, the developer just invokes simple build commands. For example, `ogs mk` (OGS make), executed at the appropriate level, will build a process, a subsystem, or the entire project.

The OGS structure used in developing System 75 software[2,8–10] is given in Fig. 4. Since the directory structure models the software architecture, a developer can easily identify the major architectural components of any subsystem by looking at the directory levels. The scoping conventions for shared headers and libraries facilitate methods for managing changes that may affect several processes. Further, a standard directory structure promotes a generic set of software administrative tools because, like OGS, these tools can infer their operation from the directory level upon which they are executed.

4.2 Local administrative tool kit

LATK (Local Administrative Tool Kit) is a collection of tools that are applied in various activities in the development and integration of

System 75 software. LATK tools operate on OGS directory structures to construct developer work areas, reserve file-edit privileges, submit completed work to an official area, ensure that all subsystems are developed with the same project-level files, verify that OGS structure conventions are followed strictly, and place files under change control.

The functions of the LATK tools in subsystem development, project integration, and system test and field support will be addressed in Sections V, VII, and VIII, respectively. Certain developers are assigned roles that carry responsibilities for coordinating day-to-day work and carrying out routine administrative operations. These roles will be explained in the context of the activities.

V. SUBSYSTEM DEVELOPMENT

System 75 software is logically grouped into the five sybsystems shown earlier in Fig. 4. A subsystem is developed entirely on one host computer. Subsystem isolation is important in a project where a large quantity of new software is being developed. In this environment, the developers of one subsystem are not subject to daily changes in the software of other subsystems. A project integration group periodically brings together and tests the software of all subsystems. Key technical and administrative responsibilities are given to the *subsystem coordinator* and *subsystem administrator* appointed for each subsystem. Project integrators, together with the subsystem coordinators, approve changes to files having project scope, schedule and conduct project integrations, etc.

5.1 Areas

An area is an OGS structure that has been populated with a particular subset of files. *Official subsystem areas* and work areas (or *work spaces*) exist on each host computer. The official subsystem area contains a complete image of the current source and object files for one subsystem and object files for the other subsystems in the project. Changes to this area are controlled by the subsystem coordinator. A work space duplicates a portion of the official subsystem area and contains files that an individual developer is modifying.

5.2 Procedure

A developer sets up a work space for a process book by invoking an LATK tool. The fact that several developers may be working on a given process creates a conflict between the desire to work with the latest copy of each other's code and the need for a stable environment. The work space setup tool copies all of the object code for the process from the official subsystem area, thereby isolating the developer from changes by others. An LATK tool can be invoked to refresh these object files from the official subsystem area at any time.

The developer uses an LATK tool to restrict edit permission on source files and obtain copies of them in the work space for modification. The OGS build tool compiles these source files and loads them with other object files for the process in the work space. The final load module in the work space can consist of any combination of processes from the work space, the official subsystem area, or even other developers' work spaces.

The developer tests the code in the work space and then uses an LATK tool to submit the work space to the subsystem administrator. The subsystem administrator uses an LATK tool to perform checks on the work space (e.g., file-edit permission checks, directory structure checks, time/date checks on source and object files) and to merge the changed files into the official subsystem area.

VI. UNIT TESTING

Developers must unit test their software before submitting it to an official area. Tools to support testing and debugging are tailored to the product software design. System 75 software consists of the Oryx/Pecos operating system and a collection of application processes that communicate via interprocess messages. Unit test tools for System 75 accommodate testing at both the operating system and application level. At the application level, tools facilitate testing the interactions between processes, as well as testing within a single process.

Three primary test tools are used in System 75 development. They are a low-level monitor for operating system and hardware testing, a high-level symbolic debugger for applications testing, and a process test environment that allows isolated testing of processes in the system.

The monitor and symbolic debugger are part of a laboratory debugging environment pictured in Fig. 5.

6.1 Monitor

The *monitor* is an 8086 debug/test tool that resides in read-only memory and runs as a stand-alone package. The monitor contains mechanisms for transferring 8086 executable files from and to a host computer via either a 9600-b/s asynchronous link or a 50-kb/s synchronous link. It provides a basic software debugging environment with capabilities such as the ability to set and display memory and I/O locations, registers, and memory management descriptors; software instruction execution breakpoints; and single stepping of assembly language execution. As a low-level tool, it supports a simple command language and has no symbolic capability.

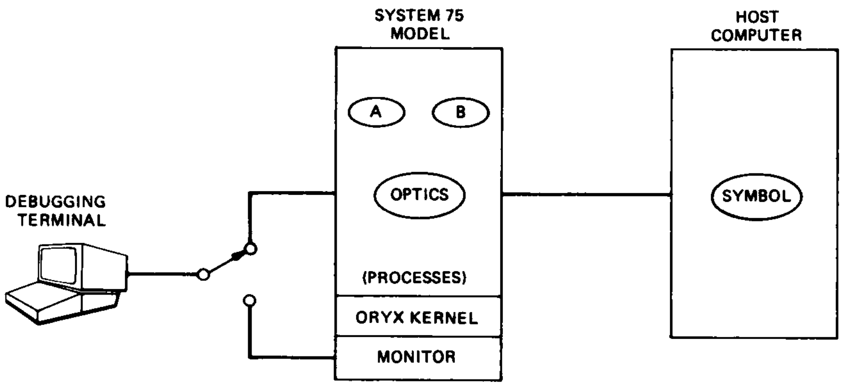

Fig. 5—Debugging environment.

6.2 OPTICS

The Oryx/Pecos Test, Inquiry, and Control System (OPTICS) is a debug/test tool that is part of the Oryx/Pecos operating system. OPTICS has a user-friendly command interface that supports line editing, command completion, and a help facility. It contains a rich set of classical debugging features, such as symbolic referencing, software breakpoints, kernel call tracing, and C stack backtrace. However, the key aspect of OPTICS is that it allows the user to control and monitor Oryx/Pecos processes. Thus, OPTICS can be used not only to debug/test software within a process, but also to debug/test the interactions among a collection of cooperating processes. OPTICS features oriented toward multiple-process debugging include creating, halting, and killing of processes; process status display; and display of a process' path records, path descriptors, and queued messages.

As shown in Fig. 5, OPTICS uses a symbol process on the host computer for symbol-table lookup. The debugging terminal is under control of the OPTICS process when the Oryx/Pecos system is running, and under control of the monitor otherwise.

6.3 Process test environment

The process test environment is a method for testing the internal logic of a process under test in isolation from the rest of the processes in System 75. A sample process test environment is given in Fig. 6. A unit test process and a stub process are available for each real process in the system. The labeled arrows in Fig. 6 represent Oryx/Pecos paths over which interprocess messages are transmitted. In both the unit test and stub processes, program intelligence is replaced by an interactive interface by which a developer can enter data from a control terminal and have it formatted into interprocess messages. A unit test

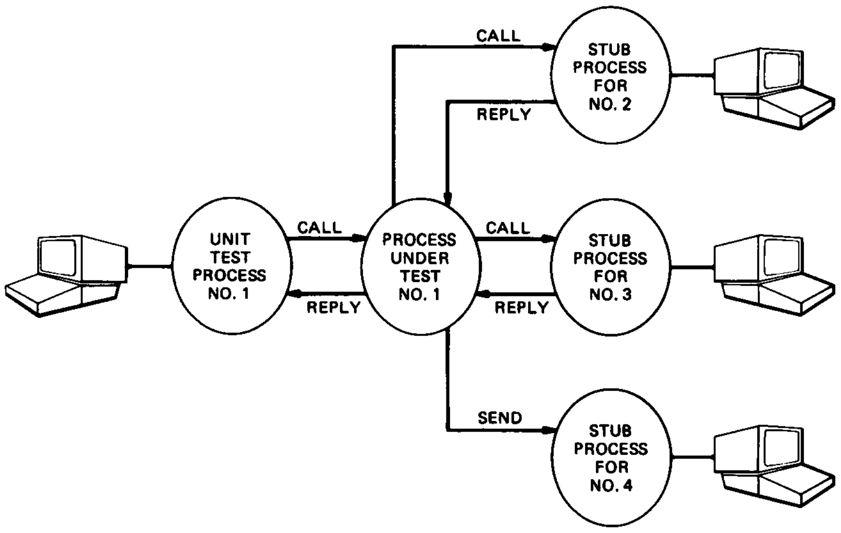

Fig. 6—Process test environment.

process issues the data entered by a developer to the process under test as a message. It can issue all the message types that the process under test can receive and thus can be thought of as a unit test driver. A stub process can be used as a substitute for any process to which the process under test sends messages. It prints at the terminal the contents of a received message, and prompts the developer to enter data to be formatted into a reply message. The functions of the several terminals shown in Fig. 6 are performed by a single physical terminal.

VII. PROJECT INTEGRATION

The System 75 project integration group performs three principal activities. These are combining, manufacturing, and testing software delivered periodically by the development groups (referred to as a *major cycle*); controlling changes to files having project scope (referred to as a *minor cycle*); and other functions such as source code quality checking, performance measurement, and collection of product statistics.

7.1 Major cycle

The major cycle procedure consists of collection from all development groups of formally submitted software, along with specification of the state of development. The main purpose is to ensure that the entire product is consistent, meaning that it can be built with a single set of tools and project-level files and that interfaces between independently developed components work correctly. All computable files are rebuilt independently on a separate host computer, and the result is tested and distributed to each development group. The type and degree of testing depend upon the stage of development. In general terms, the aim is to ensure that the quality of the integration delivery is at least sufficient to serve as a base for development of the next stage.

LATK tools support the major cycle by automating routine operations associated with submission and delivery of a large volume of files. An LATK structure verifier tool is used to locate files that do not conform to OGS conventions, and a structure printing tool can be used to get a formatted display of any part of the directory and file structure. Other LATK tools ensure that all submitted software was developed with correct project-level files and report discrepancies. The verification tools are available to both subsystem administrators and project integrators. Serious problems discovered during integration testing are fixed by developers prior to distribution. LATK work space procedures are the same as those followed for subsystem development.

7.2 Minor cycle

Certain software components apply to the entire project and are expressed in files that have project scope. For example, structures of messages used by applications to communicate with the operating system are declared in project-level header files. While it is important to define and freeze files that have project scope as early as possible, it is also important not to delay or hamper development until all such files can be defined completely. Therefore, the minor integration cycle procedure was instituted as a formal means for timely collection and distribution of these files. All minor cycle deliverables are subject to approval by project integration before the distribution takes place. The files are delivered by the development groups and distributed by project integration via shared file systems. The last minor cycle before a major cycle is especially important in that it represents a freeze of all project-level files to be used in the subsequent major cycle.

LATK tools support the minor cycle by assisting subsystem administrators in accepting distributions, validating changes, and ensuring that all components of a distribution are accepted.

7.3 Other functions

As mentioned earlier, project integration conducts some specialized forms of testing. Unlike unit and system tests, these are not functional tests. Instead, they are used to monitor adherence to established criteria for source code quality and performance. The tests are conducted by integrators rather than developers or system testers because of the need to ensure uniform compliance at frequent intervals. Project integration also serves as a central point for gathering statistics on the product and development process, such as source code and memory usage statistics. Two principal types of specialized testing are source code quality checking and hardware-assisted performance measurement.

7.3.1 Source code quality measurement

A set of `lintogs` tools, based on the `lint`[11] tool supplied with the *UNIX* operating system, are used by project integration to measure source code quality. `Lint` examines C source files and points out syntactic, stylistic, and semantic characteristics that may cause bugs, waste, or portability problems. `Lintogs` has additional capabilities to analyze C source files in an OGS structure and to report "local" characteristics of each file, and "global" characteristics of the collection of files. Local characteristics include such things as conformance to strict type rules and detection of unreachable statements. Global characteristics include inconsistent use of function arguments and return values. In principle, `lintogs` could be applied by every developer. However, the large collection of files makes it more practical to apply the tools centrally and report the results back to developers for examination and possible correction.

7.3.2 Performance measurement

System 75 software is required to meet stringent real-time performance standards. Performance measurements to validate designs are carried out by a hardware/software system known as the *spigot* system.

A diagram of the spigot system is shown in Fig. 7. In brief, it consists of hardware that provides high-resolution timing of a sequence of events reported to it by software under performance test. The software reports an event by executing a "spigot call," which sends information to the hardware. Typical events might represent the beginning and end of handling of a message by a process. An event comprises type and data fields in a spigot data packet. The hardware adds a time-stamp field and transmits the packet to a data collection processor, where it is buffered on a disk. The event file is transmitted from the data collection processor to a host computer for analysis at the conclusion of a test run. For example, statistics on each of a sequence of steps required to process an external stimulus might be gathered under varying system loads.

VIII. SYSTEM TEST AND FIELD SUPPORT

Deliveries from project integration to system test occur at major project milestones. Each such delivery represents completion of soft-

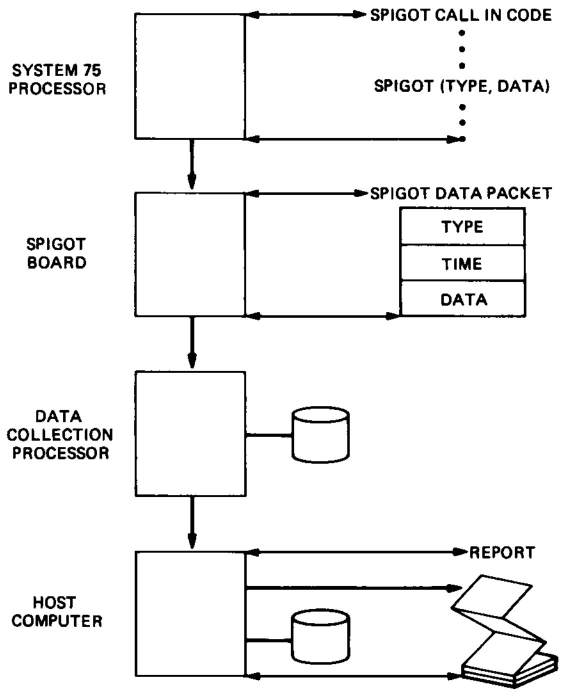

Fig. 7—Spigot system.

ware that implements a well-defined set of product features at a high level of quality. (The principal automated system testing tool, GAMUT, is discussed in Ref. 12.) The first topic of this section is a test coverage system that aids in judging the effectiveness of system testing. The remaining topics deal with tools and procedures to manage and support field releases.

8.1 Software test coverage

System test procedures are designed to exercise a load module as thoroughly as possible to verify correct operation of features. Test coverage is one objective measure of how thoroughly a program under test has been exercised.[13,14] Although more elaborate coverage arrangements exist, a low-cost system that simply records which instructions have been executed is a valuable aid in improving tests.

The coverage arrangement used in testing System 75 comprises hardware and software components. Coverage analyzer hardware monitors the SPE physical address bus. There is one bit of coverage analyzer memory for each byte of SPE memory. During a test run, coverage memory bits corresponding to SPE memory locations that have been executed are set to '1'. Commands to dump the coverage memory at the conclusion of the test run and to analyze coverage dumps are provided on the host computers. The analysis commands produce a hierarchy of reports that relate the coverage information to the load module under test. A summary report for the entire load module gives percentages of functions and source lines executed for each process. A summary report for a selected process lists which functions and which source lines in each function have been executed. Marked source code and object file disassembly listings can also be produced. A merging program is used to combine coverage data from several test runs and discard obsolete data from parts of the load module that have changed. Of special importance to the system test application is the fact that modification of the load module under test or the conditions under which it runs is not necessary.

8.2 Change control

Development and integration activities occur at a rapid pace, and files are subject to very frequent revision. There has been no perceived need to maintain a history of software changes at these stages of development. However, a system test or field release is a supported product for which strict control over changes is necessary. The two families of tools used for this purpose are the MR (Modification Request) system and the MESA (Management Environment for Software Administration) system.

An MR is a request to modify a product to fix a problem or add a new capability.[5] The MR system is a database for tracking such requests. One use of the MR system is to record problems discovered in system test or field releases. Entries in an MR form contain the originator's analysis of a problem, the severity, the product component, and the release affected. Other entries describe the eventual resolution of the MR. Reports on open MRs are analyzed to judge the importance of fixing each problem and to schedule further investigation and fixes.

MESA is an interface to the Source Code Control System (SCCS),[15] supplied with the *UNIX* operating system. MESA has added capabilities for control of a structured collection of files. That is, in addition to the standard SCCS function of recording changes to individual files, MESA also records information about the version number of every file in a release and its place in a directory structure. Thus any whole release, as well as any version of any file, can be recovered from the "pool" of SCCS files. Tools are provided to handle three types of MESA operations: initial introduction of a complete release of software into MESA, changes to files made in developer work spaces, and definition and reconstruction of releases.

8.3 Release management

By the time a software package has been delivered to the field, several copies of the package exist in different release-management stages. A field copy of the software represents the package installed at customers' sites. A working copy allows the developers to make minor enhancements and fix problems. Two intermediate copies permit system testing and soaking on an in-house system. The software graduates from one of these environments to the next as certain quality levels are achieved. Each of the environments is under MESA control and constitutes a separate view of the same MESA source pool.

In the course of system testing and field experience, MRs are filed against the product. In the normal case, changes are made to the working copy, and SCCS *deltas* are generated. The changes will be delivered to the field when the working copy has become the field copy. When quicker turnaround is desired, a change may be applied to one of the other copies, and a *branch delta* created. To resolve an MR, a developer uses the same LATK, OGS, and unit test tools discussed earlier to create a work space, check out files, build, test, and submit changes. An administrator invokes MESA options in LATK tools to merge files from the submitted work space into the MESA pool.

8.4 Field support

Field support tools are needed for two purposes. One purpose is to administer field software deliveries. The second purpose is to analyze system behavior at controlled introduction field sites[16] when necessary.

Software is delivered to a field site either as a complete reissue of the system tape or as an electronically transmitted update. Tools are provided to generate tapes on a host computer that supports a System 75 tape drive. The tape generation tools are applied by field support personnel during initial trial of a new release and are also used for factory production of tapes. Small updates to a software release can be transmitted electronically. An update is accomplished by a three-step procedure. First, a tool compares an "old" and "new" software release and generates an update file composed of commands to shift and replace software in the System 75 memory. Next, the update file is transmitted to a field system running the old release and is recorded on its tape. Last, commands within the update file are interpreted by System 75 firmware to transform the software into the new release.

Maintenance features within System 75 software are designed to detect and guide repair of system troubles.[10] However, other tools can be applied when further analysis is needed. In particular, the OPTICS tool discussed earlier can be made available at a controlled introduction site when necessary. Other field support tools provide for off-line analysis of a dump of the processor memory and of translation[9] data stored on the system tape. These tools are especially valuable for studying behavior that occurs only in a particular system configuration.

IX. CONCLUSIONS

The rich set of software tools described here, coupled with extensive computing resources, and a large-scale laboratory models program are all essential to the development of System 75 software. The tools support initial development and unit testing, automate the activities of software manufacturing, and assist in measuring software quality and performance. Methods and tools were defined jointly by the tool builders and members of the software development community. The tools continue to be enhanced when necessary to reflect evolving project needs.

X. ACKNOWLEDGMENTS

This paper presents the work of many individuals at the Denver and Holmdel locations of AT&T Information Systems Laboratories.

REFERENCES

1. L. A. Baxter et al., "System 75: Communications and Control Architecture," AT&T Tech. J., this issue.
2. K. T. Fong, J. A. Melber, and G. R. Sager, "System 75: The Oryx/Pecos Operating System," AT&T Tech. J., this issue.
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
4. C. Popper, "SMAL—A Structured Macro-Assembly Language for a Microprocessor," *Digest of Papers, COMPCON Fall 74*, 1974, pp. 147–51.
5. T. S. Kennedy, D. A. Pezzutti, and T. L. Wang, "System 75: Project Development Environment," AT&T Tech. J., this issue.
6. D. M. Emerson and G. R. Sager, unpublished work.
7. S. I. Feldman, "Make—A Program for Maintaining Computer Programs," Software—Practice and Experience, *9*, No. 4 (April 1979), pp. 255–65.
8. W. Densmore et al., "System 75: Switch Services Software," AT&T Tech. J., this issue.
9. H. K. Woodland, G. A. Reisner, and A. S. Melamed, "System 75: System Management," AT&T Tech. J., this issue.
10. K. S. Lu, J. D. Price, and T. L. Smith, "System 75: Maintenance Architecture," AT&T Tech. J., this issue.
11. S. C. Johnson, "Lint, a C Program Checker," Computing Science Technical Report #65, Bell Laboratories, Murray Hill, N.J., January 1977.
12. C. J. Lake, J. J. Shanley, and S. M. Silverstein, "System 75: GAMUT: A Message Utility System for Automatic Testing," AT&T Tech. J., this issue.
13. E. F. Miller, Jr., "Program Testing: Art Meets Theory," Computer, *10*, No. 7 (July 1977), pp. 42–51.
14. D. V. Buyansky and J. W. Schatz, "No. 1A ESS Laboratory Support System—Erasable Flag Facility," Proc. 6th Int. Conf. Software Eng., 1982, pp. 279–86.
15. M. J. Rochkind, "The Source Code Control System," IEEE Trans. Software Eng., *SE-1* (December 1975), pp. 365–70.
16. M. A. McFarland and J. A. Miller, "System 75: Introduction Activities and Results," AT&T Tech. J., this issue.

AUTHORS

Thomas J. Pedersen, B.S. (Electrical Engineering), 1960, Iowa State University; M.E.E., 1962, New York University; Bell Laboratories, 1960–1982; AT&T Information Systems Laboratories, 1983—. Mr. Pedersen has worked on a variety of communications research projects and recently on the *Horizon*® communications system and System 75 projects. He is currently working on software tools to support business communication system development. Member, IEEE.

Joseph E. Ritacco, B.S. (Mathematics), 1965, Polytechnic Institute of Brooklyn; M.S. (Mathematics), 1966, University of Michigan; Bell Laboratories, 1965–1977, 1979–1982; American Bell International Inc., 1977–1979; AT&T Information Systems Laboratories, 1983—. Most of Mr. Ritacco's career has been as a system programmer in a computer center environment. His responsibilities included IBM operating system support, performance analysis and tool building, remote job entry, and networking development and operations management. He supervised a group responsible for software tool development for the System 75 project. He is currently supervising a group responsible for developing a software system that performs remote maintenance and administration of AT&T products. Member, ACM.

Jamie A. Santillo, S.B. (Mathematics), 1975, The Massachusetts Institute of Technology; M.S. (Information and Computer Science), 1976, Georgia Institute of Technology; IBM, Poughkeepsie, 1976–1978; Bell Laboratories, 1978–1982; AT&T Information Systems Laboratories, 1983—. Prior to working on System 75, Ms. Santillo did reliability and performance analysis of mainframe computers, and developed operating system and database software. With the System 75 project, she designed development environment and integration tools, and then supervised the development of maintenance software and enhanced switch services software. Ms. Santillo is currently supervising a group responsible for project management, integration, and system test of personal computer/work station software.