Languages for operating systems description, design and implementation

by PHILIP H. ENSLOW, JR.
Georgia Institute of Technology
Atlanta, Georgia

INTRODUCTION

The transfer of information about operating systems is severely hampered by the lack of suitable means for precise communication. If major progress is to be made in reducing the high cost of developing and maintaining system software, new communication techniques must be fashioned to improve both vertical and horizontal transfer of information about operating systems.

"Vertical information transfer" refers to communication between individuals or groups working at the various levels on the development or use of some specific operating system. Designations of these levels might be specification, description, design, experimentation, implementation, evaluation, use and maintenance. However, these classifications are not all mutually exclusive, nor is this the only taxonomy possible. In contrast, "horizontal information transfer" refers to communication between different projects or between individuals working on an operating system and outsiders (i.e., a new member of the team, another operating system development project, students, etc.). The essential factor characterizing the horizontal flow of information is that the recipient is not already intimately familiar with the total environment of the system under study, and does not want to be forced to learn more about that environment than necessary.

Although some work has been done in support of information transfers along both axes, most of the effort has been focused on vertical flow, with only a small amount of effort devoted to horizontal transfer to a different environment. In only a very few instances has the combined problem been considered. (Examples of the combined approach are References 17, 58 and 59.) Previous work was performed primarily within the framework of problem-solving or systems development and focused on the development of formalized approaches involving a statement of the functional requirements of the system, specifying these requirements in a formal language, similarly defining process interactions, and progressing through various other steps such as the use of a high-level programming language, assembly language, etc. Unfortunately, such work results in the development of different languages for specific purposes without regard to their position in the hierarchy.

From a vertical point of view, the concept of structured programs suggests the possibility of utilizing the same language at different levels of abstraction. This possibility may prove workable when the requirements of only two levels are considered; however, the requirements for efficient horizontal transfer of information may well make such a solution a poor one.

In the vertical direction, the value of a complete operating system language(s) system meeting the characteristics to be described will be primarily economic. The total time, and hence the cost, required to design, implement, test and maintain an operating system will be most directly affected. In the horizontal direction, the resulting effect will be primarily an improved transfer of technology and general knowledge about operating systems. This will of course have an ultimate effect on the economics of operating system development, but the emphasis here is on knowledge transfer. The question has often been raised as to why operating system developments from universities are not utilized better by industry. The availability of a suitable means of accurate, succinct and informative communication would certainly aid this transfer process.

The emphasis of this paper is on the role of the languages as effective information transfer mechanisms, but this orientation is not meant to deny the value of complete software development systems which not only require "good" languages but also the ability to handle the design data bases. (Good examples of discussions of this type of system are References 3 and 50.) The primary thesis of this paper is that the horizontal transfer of information, though certainly as important as the vertical one, has been seriously neglected in most of the work thus far.

LEVELS OF USAGE

The language available at each "level" must provide the ability to "describe" the operating system; however, the purpose of each of these is quite different:

* **Specification**—Functional requirements of the system and the criteria of performance in providing those functions;
* **Design**—Proposed solutions to meet the specifications;
* **Simulation**—A functional model of the proposed system
that can be utilized for experimenting with various designs;

*Implementation*—The detailed solution of the problem in a form that can be directly translated to a machine-executable form;

*Description*—A description of the functionality, implementation and performance of the operating system usable for instruction and training, maintenance, demonstration or study.

The relationships of these activities one to another and to the other activities involved are shown in Figure 1.

It appears that none of these is driven totally by either the vertical or horizontal transfer requirements. For example, instruction about an operating system is performed horizontally for general students; it is also an extremely important, but expensive and time-consuming, factor in the training of new individuals on a specific operating system project. Similarly, design and experimentation may be carried on horizontally for instructional or research projects on a specific aspect of operating systems in general or vertically in the analysis of the performance of a specific system. The one level that does seem to have almost total vertical orientation is check-out and test support; however, even here the horizontal influences and features of the system implementation, such as good programming practices, will certainly be felt.

**Specification**

A specification language should be problem-oriented. It is a statement of the problems to be solved, the functionality of the system, and the criteria to be met in solving these problems. The specification may also include restrictions on system behavior. What must not be included in the specification are any *unnecessary restrictions on how* the problem will be solved. The specification for an operating system should be organized on the basis of functions provided (i.e., a problem orientation) and should be as non-prescriptive as possible. A specification is also a statement of the *policies* to be implemented in the system but not the mechanisms that will be implemented.

**Design**

A system design prescribes the general mechanisms that will be utilized to implement the policies and functions set forth in the specification. The major difference between a design and an implementation is in the level of abstraction. The design will not include all the details of the mechanisms implementations. There is, in fact, a large degree of machine-independence present in a design.

**Simulation**

One well demonstrated requirement of the total design methodology is "the need to test proposed modifications to a complex system before they are made, regardless of how beneficial these changes might appear. The number of times that the system...did not react as expected in response to adjustments should serve as a warning." Although much work has been done on the simulation of operating systems...
and on special languages to facilitate the construction of simulation models, the fact is that the cost of simulation studies of operating systems remains almost as high as the cost of experimenting with the actual system. The reason for this is that there is an almost total lack of continuity or direct transferability between the design description of the system and its simulation description. Although it is shown in Figure 1 as an adjunct activity, the language support for simulation must be unified with the “main-line” activities.

Implementation

A statement of the implementation of the system must be a description containing sufficient detail that it can be executed after appropriate processing by a language translator. The description of mechanisms in the implementation are machine-dependent except to the degree that there may exist some capability for portability of the programming language utilized for the implementation description.

Description

Whereas the problems associated with specifying, designing and implementing operating systems are well recognized, the difficulties encountered in describing such a system are not so well recognized—nor, as any student of the subject is well aware, have effective techniques been developed to accomplish such a task. What is commonly seen is a description of only one portion of the system, such as the kernel (e.g., References 53 and 72); a specific feature, such as protection (e.g., Reference 7); or the design/implementation policy (e.g., Reference 34). The weakness of our present capabilities to describe operating systems succinctly and completely is evidenced by the few overview papers that have been produced (e.g., References 14, 54, 73). Although each of these papers is informative, none provides a large amount of insight nor the detail necessary for understanding the design and implementation of the complete system. At the other end of the spectrum of descriptions of a single system is the complete coverage of MULTICS.48 It is interesting to note that even in non-system specific discussions of a topic such as security, there is no common basis for discussion and presentation. The presentation techniques encountered in open publication range from totally verbal (e.g., References 15 and 44) to theorem-based discussions (e.g., Reference 22) and a combination of theorem and code (e.g., Reference 52).

DESIRABLE CHARACTERISTICS AND CAPABILITIES

A single unified system

It is important that the “language” support provided be a single unified system, not a large collection of separate design and programming aids each applicable to merely one level. When the various levels are considered, this unification may initially be only a philosophical one; however, the ultimate goal should be physical unification through capabilities to automatically translate the representation at one level into that applicable at the next.

Graphical techniques

Graphical or pictorial techniques have always been extremely useful and powerful tools for explaining the operation of relatively complex systems. In computer systems, such techniques have usually taken the form of flowcharts. Though one experimental study has indicated that flowcharting has little value in increasing programmer productivity,60 it is nonetheless clear that flowcharts do have a positive value in facilitating information transfer about the logic and sequencing in an existing program. On the other hand, the extreme complexity of operating systems and the inability of flowcharts to portray concurrency work together to greatly diminish the value of this technique for documentation and information transfer with respect to operating systems. A modification of flowcharting called flowgraphs has been presented in Reference 32 as a technique for detailed analysis to include timing studies of real-time systems. However, since the level of detail presented in flowgraphs is quite high, its value for the analysis and description of a complete operating system is also questionable.

Language-based

Formal languages have been studied extensively and have a strong theoretical base for their design and the automation of checks for internal consistency and completeness as well as translation. These characteristics, plus the ability to easily manipulate a collection of statements in a formal language system, are strong arguments for having the support for each level be based on a formal language.

It is possible to modify a formal language definition tool such as the Vienna Definition Language70 to support “the formal description of operating system features, like the concepts of parallelism and information sharing.”43 However, based on the example applications of this technique, it does not appear that the results are very easy to understand. The problems here are primarily the language notation and semantics.

Ability to express the same abstraction or control structure at all levels

The capability of the language system to perform this task is central to the entire argument for its development. However, basic research is still needed to provide full “abstraction expression” capabilities at even one level. Work is presently being performed on languages that may provide the facilities necessary to fully describe (define) the types of structures used by Dijkstra in the T.H.E. operating system.
but it does not appear that current projects will adequately support the two dimensions of information transfer presented here.

Separation of policies and mechanisms

It is on this point that there might be divergence of goals in considering both horizontal and vertical information flows. From a vertical point of view, an objective might be that “We would like to develop a language mechanism which allows us to capture an abstraction and its implementation in the program text.” However, from the point of view of horizontal flow of information, if the abstraction of the policy and its implementation mechanism are inextricably combined, there is probably too much information being passed, since the boundary between the levels of design and implementation has been removed. There is nothing sacred about these boundaries, nor has it been proved that they are essential; however, the desirability of this characteristic should be examined. At the descriptive level there is no question that one would wish to be able to describe various policies without considering their implementations.

There are several readily-apparent reasons for the separation recommended above. The study of operating systems can be divided into two general classifications; (1) The investigation of the logic and operation of algorithms that control specific functions of the operating system; and (2) The consideration of the various implementations (mechanisms) possible for the logic underlying these policies. In addition to clearly identifying just what it is that is being examined, this fundamental separation directly supports two other objectives of any design and implementation system—a systematic approach to performance analysis (Which is the cause and which is the effect, the policy or the mechanism?) and the identification and cataloging of reusable policy modules even though their implementations might be unique (i.e., target system dependent).

Separation of policy and mechanism should greatly assist in the development of automatic checks of the program developed. The methodology employed for such checks will be different at each level, but all methodologies will have the goals of ensuring consistency and facilitating verification. A full consideration of this requirement is present in the work of several workers at the implementation level. However, all of the problems have not been solved at even that single level.

Understandability and intelligibility

William Wulf has commented that “Whatever it is that makes a program understandable must be a property of the program text.” The problem is one of attaining and retaining understandability for both vertical and horizontal information transfer. A descent of one vertical level might be characterized as the removal of one level of abstraction whereas a horizontal transfer implies a complete understanding of the abstraction at that level. Entering into the discussion here is the old argument about the use of comments vs. a verbose language (e.g., COBOL). It appears at this time that the total “understanding” required for either direction of transfer should be provided by the program statements themselves; and, since the language itself must support horizontal transfers, the demands made on the information content of the language exceed those made in the past. There is a grave danger that, in attempting to increase the information content of the basic language statements so as to improve understanding, the result will be the development of an extremely complex or artificial notation system that greatly diminishes intelligibility.

Promote (enforce) good programming practices

The promotion—or better, the enforcement—of good programming practices is one of the most desirable characteristics of the complete system from the point of view of actual usage in a vertical manner. Much has been said about the desirability of omitting several types of constructs from the language; unfortunately, much less has been said about what should be included to provide the capability to do “good systems design and implementation;” and research on the enforcement of procedures that will result in quality work is almost non-existent. It appears that the problem is divisible into two general subjects—the organization of the control structures used in the program and the physical organization of the program.

SURVEY OF PREVIOUS WORK

In the past, the primary motivation for language work in this area has been the need to develop tools that will facilitate the evaluation of various features included in an operating system, the selection of parameter values, etc. This is obvious from the number of citations given below in the section on simulation. The area with the next higher level of activity has probably been implementation languages, although only a small selection of references is cited here. There has been some work on design languages, but descriptive systems have received very little attention. The work on languages or programming systems that addresses more than one level of usage has been very small.

Hardware languages

It is not advisable to attempt too close an analogy between software systems design languages and those developed for hardware; however, a few comments on the latter appear to be in order. One is inclined to think of the problem of “common” notation as being much simpler for hardware, but such a situation is not the case. A 1967 review of documentation in use at the lowest (logic gate) design level showed nine techniques then in use. The problem was
actually even worse than this diversity of systems would seem to indicate. The motivations for the use of higher-level languages or machine manipulatable notations for hardware are almost the same as they are for software—vertical and horizontal information transfer plus automatic translation and expansion to take care of the simple repetitive tasks, error and consistency checks, and interfaces with automated systems that carry the design process through the next logical step.

An example of a notational system that was originally designed to support horizontal information transfer is ISP. Of particular interest is the later work on ISP that focused on expanding its use along the vertical dimension with complete, automatic translation support down through simulation and implementation. ISP corresponds to a lower-level of language hierarchy discussed above; however higher-level languages have not been neglected. Some early work was on the development of an Algol-type language, and today there is much activity within a group known as the Conference on Hardware Design Language (CHDL). Of particular note is the sentiment that several members of the CHDL group have expressed on the importance of the use of the output of this group as a teaching or descriptive language.

**Descriptive languages**

The first language developed specifically for operating system description was a notation introduced by Leo Cohen in his preliminary text, *Theory of the Operating System*. Cohen was one of the first individuals presenting introductory courses on operating systems and his notation reflects an appreciation of the importance of understanding the concurrent operation of the operating system with the user program and concurrency within the operating system itself as well as recognition of the fact that it is a "transaction" that is being processed. The technique, which involved identifying the "Current Operating Transaction" and the "Available Transaction" as the operating system crossed logical boundaries was described fully and used extensively in Cohen's book. Although the COT-AT technique was quite limited in its capabilities, it did provide the basis for a rather concise description of an existing operating system, OS/360 MVT, in 18 small flowcharts and a conceptual design for a proposed operating system. The early work with the COT-AT notation system also greatly influenced the design of the first operating system simulation language, S3, to be discussed.

Another notational technique applied to the descriptions of both software and hardware is the contour model. This technique, which can illustrate the nesting of control and concurrent execution environments, was used extensively in a complete treatise on the Burroughs 5700 and 6700 systems. This technique may not have been available for use by the same author in an earlier text on the Multics system which also has a block-type structure; however, there also appears to be some question of its applicability to this situation. The contour model also formed part of the basis of the picture-system.

**Experimentation and simulation languages**

Simulation has long been recognized as a very useful tool in the study and evaluation of operating systems. As can be seen from the references to be cited, there has been much work done in this area—far more than that done at any other level.

Almost any general-purpose discrete event simulation language, e.g. GPSS and SIMSCRIPT, or even a general-purpose language such as FORTRAN could be used to model an operating system, and many such models have been constructed. However, most of these models cover only a small portion of the complete system such as the scheduler/dispatcher subsystem, the memory allocator/manager, etc. GPSS has been popular for this purpose. Although a model programmed in a general-purpose language can provide an accurate description of the logic of the target system, the source code is not easy to read, and the depiction of many of the operations found in operating systems requires some rather involved coding that obscures the basic logic. Some work of note in this area has been the use of SIMULA as the simulation language. A specific example is a complete model of the CDC 6600/SCOPE. SIMULA has several features, particularly the CLASS concept, which made it extremely attractive for use in this application and its applicability has been considered by several individuals at the Norwegian Computer Center.

The earliest known example of a programming system developed expressly for simulation studies of computer systems is the Computer System Simulation (CSS) which was utilized by IBM in three ways: "First, in the development of a programming system . . . so that proposed changes can be evaluated . . . second, in establishing a system configuration for a given workload . . . third, after a system is operational . . . to predict the effect of expected or proposed changes to the actual system." The CSS model consisted of (1) a description of the system, (2) description of the operating system, and (3) description of the workload. The CSS language provided a means to describe the model of the operating system so that each block in the flowchart was represented by a single statement. What was lacking was a clear picture of the concurrency present. (See Reference 39 for an application of CSS.)

The earliest known example of a simulation language developed expressly for operating systems is S3, the Systems and Software Simulator. This system was developed under contract to the U.S. Army which was exploring alternatives to benchmarking as a system evaluation procedure for procurement purposes. (Although the review of the S3 System indicated that it would provide sufficient accuracy for this purpose, its use was not adopted because of the high cost of maintaining a data base containing up-to-date descriptions of all of the various hardware and software systems that might be proposed by suppliers. Benchmarking was retained as the most cost-effective technique that would satisfy both the vendors and the Army.) The S3 simulation language provided full capabilities to describe both the hardware and the software involved to include both user programs and the operating system. The simulator provided a "true discrete
event simulation” utilizing a complete future events chain. The language provided statements to accurately model the logic of the software as well as statements to maintain the proper timing with respect to the hardware and the system actually being modeled. The power and usefulness of the language is best attested to by the fact that a completely validated and time-adjusted simulation of the GE 635 computer and the GECOS II operating system required only 211 S3 statements. (The instructional value of such a language as a descriptive tool was first appreciated by this author during his review and evaluation of the S3 contract.12 S3 was used internally by Cohen Associates for verifying the performance of the operating system for an airborne computer which had extremely stringent timing requirements, e.g., proper recovery and handling of six different interrupts that could occur during one microsecond. The work on S3 obviously influenced Cohen in the development of the commercially available simulation system SAM.13

OSSL was another language developed primarily for generalized simulation studies of computer systems.10 The OSSL model had three components: “(a) The hardware characteristics and system configuration, (b) the operational philosophy of the system [the manner in which user jobs flow through the system], and (c) the environment in which the system is to function [the services that the system is called upon to perform].”10 Simulation languages and systems developed expressly for operating system study do provide models with a rich content of system description. Their major weakness in application has been the cost in processor time of executing the model. An example is the GECOS II/635 model referred to above which produced validated results within five percent of actual system runs but required 10-15 times as much time even when executing on a UNIVAC 1108. The high time requirement for executing the simulator is not totally the result of using a “high-level” simulation language. Even for models written in assembly language “the simulated-time/real-time ratio is much greater than one.”14

Analytic models can provide the same degree of accuracy with execution costs much less than actual runs. Although the state-of-the-art in analytic models has progressed to very high levels of both completeness and detail (a comprehensive survey is presented by Reference 65), the description of the model, in whatever language is utilized, provides little, if any, intuitive understanding or insight into the nature of the system being modeled.64

A compromise between the two forms of modelling or simulation has resulted in the development of hybrid techniques in which both discrete event and analytic procedures are utilized.16,31,61 “In computer system models, it is convenient to partition the resources of the system into two mutually exclusive sets, denoted long-term resources and short-term resources respectively, creating a two-phase model. In a hybrid model, discrete-event simulation is used to model the first phase, which is the arrival of tasks and the allocation of long-term resources. This second phase—use of short-term resources by active tasks—is then modeled by any technique which produces an expected residency time (active time) for each active task.”61 The intuitive understanding obtainable from the “language” statement of a hybrid model does exceed that present in pure analytic models, but the logic of the system being simulated is often submerged in programming details of the model.

**Instructional laboratory project languages**

It is not clear exactly where to place those languages developed expressly to support operating system laboratory courses. To support the general teaching objectives, they certainly should be highly descriptive in nature, but it appears that this aspect of the language has often been sacrificed for the sake of expediency in getting the laboratory exercise completed in a limited amount of time. However, this latter comment certainly does not apply to all of the work in this area. Several of these instruction-oriented systems provide very usable simulation capabilities, e.g., OASIS66 and ITS,36,37 while others provide a language intended for the actual implementation of a complete operating system or portions of a system, e.g., Concurrent PASCAL,23 and Concurrent SP/k.53

Certainly, a primary objective of any instruction language is the transfer of information about concepts as well as implementation techniques and examples. Also, the influence of an operating system simulation language on system descriptions is quite beneficial,45 however, the instructional environment must be able to accommodate student time limitations, which results in shifts in emphasis on the various capabilities of the language as contrasted to what they might be in a “production” environment.

**Design languages**

There are two examples of design languages. These are definitely multi-level, and either could possibly be made to span the entire spectrum of language levels being discussed here. Both of these projects date from 1972—one as academic research that was utilized for one small job and the other as a totally operational support system. The first phase in the academic project was a study of “A Programming Language for Concurrent Processing.”24 The underlying model of this system is based on the contour model discussed earlier.27 Of more interest to this paper is the follow-up work which led to the development of the Picture-System model and the PS notation which supports model development.24,55 Picture-system models “are useful in defining, communicating, and simulating computer system designs, especially in the early design stages.”24 The PS system generates the models “from a description of a computer system as a structure of finite-state components. The PS system also does an analysis of the state-transition graph of the subject computer system which detects design problems such as deadlocks, looping, and races.”24

The other project,18-21 known as Higher Order Software (HOS), is “a formal methodology for reliable systems specification and development.”21 A specific goal of the HOS system is to tightly control those areas causing most of the
design errors, such as interface problems, and to automate that control.

It is now possible within the framework of HOE to develop automatic tools to aid verification as well as design. For example, the interfaces of an HOS system can be exhaustively tested by an automated analyzer without program execution. This is especially significant since interface testing in a large system is known to be a very costly procedure. (73 percent of all problems found during the APOLLO integration effort were interface problems; and verification accounts for 50 percent of the total software development effort.)

Higher Order Software is software expressed in its own meta-language and conforming to a formalized set of laws. The basic components of HOS methodology are: (1) the application of the formal set of laws to the design of a given problem; (2) a meta-language adhering to these laws; (3) the automatic analysis of design interfaces by the design analyzer and the structuring executive analyzer; (4) the architectural virtual layers produced from analyzer output in the form of software, firmware or hardware, and (5) the hardware that is transparent to the user. (Our work, to date, has concentrated on the first three areas of HOS methodology.) In addition, support tools based on axiomatic consistency can enhance a given development process in such areas as: performance analysis, simulation, design automation, definition of subsystem requirements, automatic documentation and automatic management techniques.

There have been several design approaches involving path descriptions. A large amount of it centers around the basic concepts of Petri nets. It is well known that the original work by Petri in this area remained unnoticed and unused for some time after its initial publication. The long-term value of the technique appears to be as difficult to predict as it was to recognize its immediate applicability. There is something inherently attractive about the use of graphical representation to depict parallelism, but this author has difficulty in visualizing clearly how such techniques can be integrated into a unified "vertical" information-transfer system. Activity continues in this area, and it appears quite reasonable to anticipate more progress along these lines.

Implementation languages

Activity in this area has been very high, and work on the topic has been identified by the designation "machine-oriented high-level languages," "systems programming languages" as well as "operating system implementation languages." IFIP has established a technical group on Machine-Oriented Higher-Level Languages and a working conference has been held. This working conference led to the establishment of a permanent working group, WG 2.4, under the auspices of the IFIP Technical Committee on Programming, TC 2. It is not possible to cover all of the work done in this area; however, a typical example is Reference 38. Markstein developed an operating system programming language, PSETL, which is based on SETL, a set-theoretic programming language. The extensions required for PSETL are capabilities "to allow the description of algorithms involving interrupts, parallelism, and to some extent, machine dependent features." Although it is stated that PSETL is "intended for operating system description," it is much more of an implementation language. The extensive examples given in the report illustrate that the program is not very "descriptive" nor very understandable without the liberal use of comments (perhaps as much as 50 percent).

A general discussion of the desired characteristics of implementation languages including specific comments on Concurrent PASCAL and MODULA is presented in Reference 28. Another derivative of PASCAL is CCNPASCAL.

Multi-level language systems

As was stated previously, there has been only a limited amount of work done in this area. A useful survey of languages for specification and design was presented in Reference 56. Language classifications that were identified and briefly discussed in that survey are state-based languages such as TOPD and DREAM, event-based languages such as path expression and flow expression systems, and relational language systems such as ISDOS and REVs. All of these systems do provide some formal linkage between the statement of the specification and the design.

An early example of a development system that extends into even the simulation level is DES. A Design and Evaluation System. "A system which integrates performance evaluation with design and implementation" is described in Reference 17. "[T]his system is based on a simple, high level language which is used to describe the evolving system at all stages of its development. The source language description is used as direct input to performance analysis and simulation routines."

A description of a complete program development methodology focused specifically on "the design, implementation, and proof of large systems, based upon a hierarchical decomposition of the system" is given in Reference 57.

SUMMARY

A major factor contributing to the high cost of the development and use of an operating system is the lack of effective means for transferring information vertically within one operating system project and horizontally between projects. Most of the work in the past has addressed only vertical information flow. It is important that the importance of the description of operating systems and the value of horizontal transfers also be recognized and addressed by future work.

ACKNOWLEDGMENT

The ideas expressed in this paper were developed during discussions with several individuals, among whom were E. Dijkstra, C. A. R. Hoare, Gerhard Seegmueller, and W. Wulf. As I look back on these discussions, I think I detect
a definite dichotomy of points of view—I was interested in a balanced solution considering both horizontal and vertical information transfer, whereas these individuals were focusing primarily on vertical transfer between the lower levels (e.g., Reference 63). In attempting to sort out and properly place the various concepts we discussed, I am sure that I have made errors in interpreting their comments. The responsibility for that is totally mine, as is a sincere appreciation for their help.

REFERENCES
