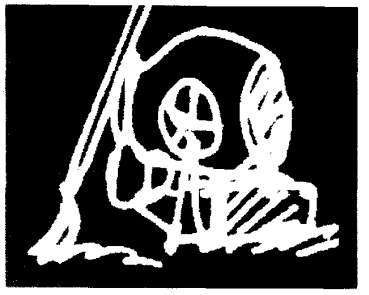

Operational Profiles in Software-Reliability Engineering

JOHN D. MUSA, *AT&T Bell Laboratories*

◆ *An operational profile describes how users employ a system. Created to aid software-reliability engineering, this technique can help you in many other ways. Here's how to develop an operational profile step by step.*

A software-based product's reliability depends on just how a customer will use it.[1] Making a good reliability estimate depends on testing the product as if it were in the field. The operational profile — a quantitative characterization of how a system will be used — is thus essential in software-reliability engineering. In addition, the operational profile shows you how to increase productivity and reliability and speed development by allocating development resources to functions on the basis of use.

Using an operational profile to guide testing ensures that if testing is terminated and the software is shipped because of imperative schedule constraints, the most-used operations will have received the most testing and the reliability level will be the maximum that is practically achievable for the given test time. For guiding regression testing, it efficiently allocates test cases in accordance with use, so the faults most likely to be found of those introduced by changes, are the ones that have the most effect on reliability. Although I and my colleagues at AT&T developed this technique to guide testing, we have found it to be very helpful in many other ways, as the box on the facing page describes.

BENEFITS AND COST

The benefit-to-cost ratio in developing and applying the operational profile is typically 10 or greater.

On AT&T's International Definity project (a PBX switching system), combining the operational profile with other quality-improvement techniques reduced customer-reported problems and maintenance costs by a factor of 10, system-test interval by a factor of two, and product-introduction interval by 30 percent.[2] The

0740-7459/92/0300/0014/$03.00 © IEEE

system experienced no serious service outages in the first two years of deployment; customer satisfaction improved significantly. The marked quality improvement and a strong sales effort resulted in an increase in sales by a factor of 10.

In a similar quality-improvement program, Hewlett-Packard applied software-reliability engineering and the operational profile to reorganize their system-test process for a multiprocessor operating system. With automated test and failure recording and using the operational profile to guide testing, they reduced system-test time and cost by at least 50 percent.

The cost of developing an operational profile varies. Our experience indicates that the effort to construct the operational profile for an "average" project — about 10 developers, 100,000 source lines, and a development interval of 18 months — is about one staff month. Large projects can cost more, but the increase is clearly less than linear with project size.

International Definity invested two to three staff years in extensive customer study that led to an operational profile. Of course, every project requires good knowledge of the customer base, so only a portion of this effort can reasonably be charged to the operational profile. Also, the work can be written off over several releases, with only minor updating needed between them.

DEVELOPMENT PROCEDURE

The operational profile is usually developed by some combination of systems engineers, high-level designers, and test planners, with strong participation from product planning and marketing professionals.

To determine an operational profile, you look at use from a progressively narrowing perspective — from customer down to operation — and, at each step, you quantify how often each of the elements in that step will be used. In this article, I explain underlying concepts by developing, step by step, a simple operational profile for a PBX. I've included more detail and more advanced concepts in the boxes.

As we created and used this procedure, we identified many problems and researched many solutions. I have distilled the "best" current approach, based on our experience with a variety of projects, most of them real-time telecommunications systems. Of course, the procedure I describe here will evolve with use.

Profiles. In the course of developing the operational profile, you will generate several other profiles. A *profile* is simply a set of disjoint (only one can occur at a time) alternatives with the probability that each will occur. If A occurs 60 percent of the time and B 40 percent, for example, the operational profile is A, 0.6 and B, 0.4. Profiles are often shown as plots. In fact, because of the wide range of probabilities, often the logarithm of the probabilities is plotted. For the simple example I use throughout this article, I present profiles as tables.

In many cases, usage information is available or can be estimated, most easily in terms of rates like transactions per hour. But this data is not a true profile until you convert it to probabilities, by dividing by the total transactions per hour. Converting to probabilities is helpful because you can make a quick completeness check by seeing if the probabilities add to 1. On the other hand, the raw data is useful to re-create actual traffic levels in test.

The consideration of net economic gain is the ultimate criterion for deciding how accurate these occurrence probabilities must be. The economic gain is the financial effect of the better decisions that result from more accurate data. In many cases, however, you will use informed engineering judgment rather than a formal economic analysis.

Is use the only factor you should consider in developing operational profiles? What about infrequently executed functions

OTHER USES FOR THE OPERATIONAL PROFILE

Although it was developed to guide testing, the operational profile can also guide managerial and engineering decisions throughout the life cycle, including requirements specification, design, implementation, and testing. Because it ranks features by how often they will be used, it suggests development priorities.

A prioritized *operational development* approach is a very competitive way to sequence new-product introduction: Make the most-used features (operations) available very quickly and provide less-used features in subsequent releases. This approach is different from traditional development approaches, which proceed module by module.

The operational profile improves communication between developer and customer and within the customer organization by making expression of needs more precise. For example, when a developer asks what operations are needed to support maintenance and how often they will be used, it stimulates users to think about, discuss, and study what the maintenance procedures should be.

The operational profile can also be used in performance analysis: If you multiply each operation's occurrence probability by the system's overall run or transaction-execution rate, you obtain the run or transaction rates for each operation. This information is used for performance analysis and performance testing. Among other uses, it can help determine the number of servers a client-server system requires.

Finally, the operational profile is an educational aid. It organizes work in a manner that is closely related to user work processes. It can direct the customer's training efforts toward the most-used operations. For user manuals, the operational profile suggests the order in which material should be presented (most-used first) and the space, time, and care that should be devoted to preparing and presenting it.

The fact that the information collected to develop an operational profile can be used many ways lowers its cost.

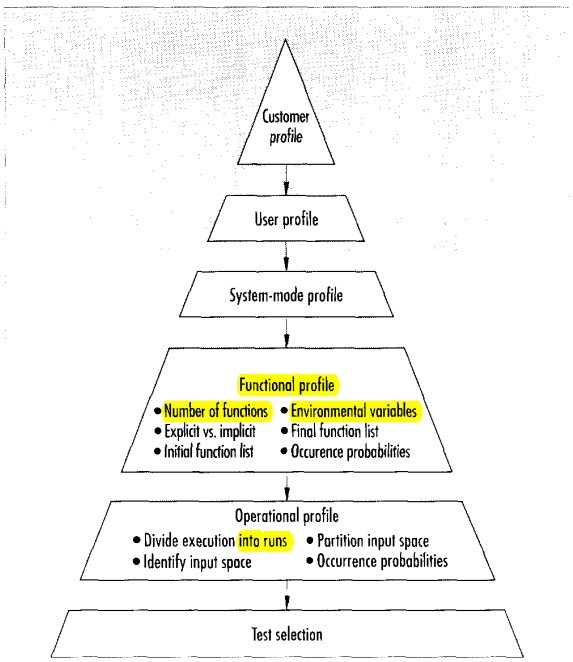

Figure 1. *Developing an operational profile involves progressively breaking system use down into more detail. Not every application will require all five steps. You then apply the operational profile to guide test selection.*

whose failure might lead to disastrous results, such as the function that shuts down an overheating nuclear reactor? This concern can be handled by considering criticality as well as use of operations, which I cover later.

Five steps. As Figure 1 illustrates, developing an operational profile to guide testing involves as many as five steps:

- Find the customer profile.
- Establish the user profile.
- Define the system-mode profile.
- Determine the functional profile.
- Determine the operational profile itself.

In the first four steps, you progressively break system use down into more detail. Customer groups (large retail stores) break down into user groups (sales clerks and information-system specialists). A single user group may invoke several system modes (IS specialists perform both database cleanup and generate reports). In turn, each system mode has several functions (the generate-reports mode has several types of reports). In the fifth step, functions evolve into operations as the system is implemented.

Some steps may not be necessary in a particular application. A customer profile is unnecessary if you have only one customer or if all customers use the system the same way. Sometimes you can skip the functional profile, especially when the requirements are so detailed they specify how users will execute the system to accomplish their tasks. Other times the information you need to develop the operational profile is not available until the design or even a substantial part of the implementation is complete, so you must develop the functional profile if you want guidance in allocating development resources during design and implementation.

Even if you have all the information you need to develop the operational profile before design commences, you may prefer to generate the functional profile first. Because it generally has fewer elements, it is easier to develop and use for guiding pretest development than an operational profile. Should you decide to develop the operational profile directly, you should consider the tasks outlined in the functional-profile step in combination with those in the operational-profile step.

At each step, you must determine the level of detail you need. Whether you distinguish and treat an item differently at a given step will depend on the net economic gain for doing so. For example, in developing a customer profile broken down by industry, it may be cost-effective to distinguish certain important customers with the intention of performing special testing on the product supplied to them. The degree of detail does not have to be uniform across the system. For example, some important system modes may require greater levels of detail.

When attaining an "average" reliability over all a system's applications is acceptable, there may be no need for more than one operational profile. But if it is important to assure a particular reliability for a particular use (even if all reliability objectives are the same), then you must determine multiple operational profiles. In either case, you must separately identify different customers, users, and system modes so that you can weight the contribution each makes to the operational profile.

Sometimes a software product is part of a network of systems. In that case, it may be useful to develop an operational profile for the entire network before developing the operational profile for the product. This "supersystem" profile can be very useful to determine which systems in the network are most important and should receive the most attention.

As far as we now know, the operational profile is independent of design methodology — its determination will not be affected by an object-oriented approach, for example. The one exception might be a case in which functions designed with one methodology map to a considerably different set of operations when designed with another.

CUSTOMER PROFILE

A customer is the person, group, or institution that is acquiring the system. A customer group is a set of customers that will use the system in the same way. Large pharmacies use a switching system very much like other large retailers and thus should be grouped with them, even if they are not in the same market segment. The customer profile is the complete set of customer groups and their associated occurrence probabilities.

You obtain information on potential customers from marketing data from related systems, modified by marketing estimates that take into account the new system's appeal. The business case developed for a proposed product usually includes the expected customer base. It is a valuable source for developing operational profiles, analyzing performance, and reviewing requirements.

If there is a trend in customer composition over the product's

life, you can either use a fixed average-customer profile or a different, updated profile for each new version. If you choose a fixed profile, use the average over the entire product life for the functional profile (since this is used in allocating development resources) and the average over the release life for the operational profile (since this is used primarily in planning and executing tests and may vary with each release).

The best measure of each customer group's probability is the proportion of use it represents. If this isn't available, a simple approximation is to use the proportion of total deliveries you expect to make to that customer group.

As an example, consider a hypothetical PBX that is sold to institutions for internal use and, of course, external connections. Assume there are two customer groups, large retail stores with 60 percent of the use and hospitals with 40 percent. The customer profile is large retail stores, 0.6; hospitals, 0.4.

USER PROFILE

A system's users are not necessarily identical to its customers. A user is a person, group, or institution that employs, not acquires, the system. A user group is a set of users who will employ the system in the same way. By identifying different user groups, you can divide the task of developing the operational profile among different analysts, each an expert on their user group. The user profile is the set of user groups and their occurrence probabilities.

Sometimes the users are customers of your customers; sometimes they are internal to your customer's institution. In any case, different users may employ the system differently. The differences may be the result of job roles — an entry clerk will view an insurance company's claim-processing system differently than an actuary.

You derive the user profile from the customer profile by refinement: looking at each customer group and determining what user groups exist. If you find similar user groups among different customer groups, you should combine them.

As before, the best way to establish the occurrence probability of a user group within a customer group is to use the proportion of the customer group's usage it represents. If this isn't available, use the proportion of total users. If an insurance company group has 900 entry clerks and 20 actuaries out of 1,000 users, the entry-clerk user group has a probability of 0.90 and the actuary user group, 0.02.

If there are multiple customer groups, each user-group probability should be multiplied by its customer-group probability to obtain its overall probability. When user groups are combined across customer groups, their overall user-group probabilities should be added to yield the total user-group probability.

In our example, user groups in each customer group include telecommunications users (people making calls and sending data), attendants (internal operators who answer the main number), a system administrator (who manages the system and adds, deletes, and relocates users), and maintenance personnel (who test the system periodically and diagnose and correct problems). Each of these user groups employs the system differently.

Table 1 shows the proportion of use by each user group in each customer group. Use by attendants in the large retail store is higher because few people call a retail-store department directly. On the other hand, use by the system administrator in a hospital is higher because of the frequency with which patients are admitted and discharged.

Table 1 also shows the overall probability for each user group, computed by multiplying each user group's probability by its customer group's probability and adding all the results. Large retail stores represent 0.6 of customer-group use and attendants represent 0.07 of that, or 0.042 overall. Hospitals represent 0.4 of customer-group use and attendants represent 0.05 of that, or 0.02 overall. The total use by attendants is 0.062 of system use.

TABLE 1
SAMPLE USER PROFILE

	Large retail store probability = 0.6		*Hospital probability = 0.4*		
User group	User-group probability within customer group	Overall user-group probability for customer group	User-group probability within customer group	Overall user-group probability for customer group	Total user-group probability
Telecommunications users	0.900	0.540	0.900	0.360	0.900
Attendants	0.070	0.042	0.050	0.020	0.062
System administrator	0.010	0.006	0.035	0.014	0.020
Maintenance personnel	0.020	0.012	0.015	0.006	0.018

SYSTEM-MODE PROFILE

A system mode is a set of functions or operations that you group for convenience in analyzing execution behavior. A system can switch among modes so that only one is in effect at a time, or it can allow several modes to exist simultaneously, sharing the same resources. A system-mode profile is the set of system modes and their associated occurrence probabilities.

For each system mode, you must determine an operational (and perhaps functional) profile. Thus multiple system modes means multiple operational and perhaps functional profiles. The same function or operation can occur in different system modes.

There are no technical limits on how many system modes you can establish. You must simply balance the effort and cost to determine and test their associated operational profiles against the value of more specialized information and organizational convenience they provide.

Some bases for characterizing system modes, with examples, are

- *User group.* Administration mode; maintenance mode.
- *Significant environmental conditions.* Overload versus normal traffic; initialization (startup or reboot for failure recovery) versus

USING THE FUNCTIONAL PROFILE FOR MODULAR DEVELOPMENT

If functions do not relate to modules on a one-to-one basis, you may want to map a functional profile into a modular usage table. Here, I use "module" in the generic sense of program components and do not imply a size.

Modular usage tables are used when a project organizes work by modules rather than functions. A modular usage table gives the occurrence probabilities of the execution of modules, providing a basis for allocating resources and assigning priorities by modules.

A modular usage table is not a profile in that module usage is not disjoint — the occurrence probabilities do not necessarily add to one. For example, suppose function *A* uses modules 1 and 4; *B*, 1 and 2; *C*, 2 and 3; and *D*, 1, 3, and 4. Assume the functional profile is the one in Table A.

TABLE A
SAMPLE FUNCTIONAL PROFILE

Function	Occurrence probability
A	0.4
B	0.3
C	0.2
D	0.1

The function-module matrix, which indicates (with a "1") which modules are used by each function is shown in Table B.

TABLE B
SAMPLE FUNCTION-MODULE MATRIX

	Modules			
Function	1	2	3	4
A	1	0	0	1
B	1	1	0	0
C	0	1	1	0
D	1	0	1	1

Let q_j be the module occurrence probability and p_i the function occurrence probability. Then

$$q_j = \sum_i a_{ij} p_i$$

where a_{ij} is the binary variable indicating if module j is employed by function i. The modular usage table is shown in Table C.

TABLE C
SAMPLE MODULAR USAGE TABLE

Module	Occurrence probability
1	0.8
2	0.5
4	0.5
3	0.3

The operational profile cannot generally be used for test-case selection at the module level, because operations do not have meaning there.

continuous operation (includes warm start after an interruption); system location; time.

- *Operational architectural structure.* Online retail sales versus after-hours billing modes.
- *Criticality.* Shutdown mode for nuclear power plant in trouble.
- *User experience.* Novice versus expert mode.
- *Hardware component.* Distributed system with hardware components performing different functions.

Defining different system modes is a convenient way of accommodating changes in the operational profile as users become more experienced. In practice, you can capture the variations in experience with two extremes, novice and expert, mixed in different proportions.

Some systems control the operations they will accept on the basis of environmental variables like traffic level and system-capability status, in order to reject noncritical, nonfunctioning operations and dedicate capacity to more critical ones. If a system must function in these conditions, each of these situations should be established as a system mode and tested with the guidance of separate operational profiles.

The sample PBX has five system modes:

- business use,
- personal use,
- attendant use,
- administration, and
- maintenance.

The last three system modes represent user groups and are essentially disjoint in that they do not share functions or operations. The first two modes share most functions or operations and could be combined. However, the functional and operational profiles are expected to be very different. Because statistics for these two uses are readily available, separating them is a matter of considerable convenience.

Assume that 100 percent of use in the large retail stores is business use, but that hospital use is 60-percent business and 40-percent personal. Using Table 1, you compute the overall probability of business use by multiplying the occurrence probabilities of the telecommunications user groups in each customer group (large retail stores, 0.54; hospitals, 0.36) by the proportion of business use (large retail stores, 1.0; hospitals, 0.6) and adding the results.

Using this formula, you derive the system-mode profile for the switching system, shown in Table 2.

TABLE 2
SAMPLE SYSTEM-MODE PROFILE

System mode	Occurrence probability
Business use	0.756
Personal use	0.144
Attendants	0.062
System administration	0.020
Maintenance	0.018

FUNCTIONAL PROFILE

The next step is to break each system mode down into the functions it needs — creating a function list — and determine each function's occurrence probability. A function is a task or part of the overall work to be done by the system, sometimes in a particular environment, as viewed by the system engineer and high-level designer. The functional profile provides a quantitative picture of the relative use of different functions.

In many cases, it is helpful to construct a work-flow profile that shows the overall process you are implementing, including software, hardware, and people, before you determine the functional profile. A work-flow profile shows the context in which the software will operate and suggests functions.

You usually develop a functional profile during the requirements phase, possibly extending into high-level design, as part of the feasibility study and requirements definition. A functional profile is part of the requirements document and should be kept updated when changes occur. It is independent of design methodology.

Because you determine the functional profile before design begins, it can help guide the allocation of resources during design, coding, unit test, and possibly subsystem test. Of course, to allocate resources and set priorities, you must consider other factors as well, such as risk and developer expertise.

Number of functions. A functional profile does not have a set number of functions, but it typically involves between 50 and several hundred. The number generally increases with project size, the number of system modes, the number of major environmental conditions, and function breadth — the extent to which a function accommodates processing variations.

YOU USUALLY DEVELOP A FUNCTIONAL PROFILE DURING REQUIREMENTS DEFINITION, AS PART OF THE FEASIBILITY STUDY.

You should define two tasks as different functions if the processing, and hence development, is so different that you are likely to manage their development with different priorities and resource allocations. This is especially true if the functions differ considerably in frequency of use or criticality. Consider some different ways of defining functions for command X with parameters A and B. Parameter A can have values $A1$ or $A2$ and parameter B can have values $B1$, $B2$, or $B3$. Suppose that setting parameter A has much more effect on the difference in code executed than setting parameter B. Two possibilities are likely:

- One function, X, which includes all sets of parameter values.
- Two functions, $XA1\ B$ and $XA2\ B$, which considers only A's different parameter values as separate functions.

You would probably not define six functions, $XA1\ B1$, $XA1\ B2$, $XA1\ B3$, $XA2\ B1$, $XA2\ B2$, and $XA2\ B3$, with each set of parameter values as a separate function, because the value of B doesn't have a major effect on the code executed. In all three cases, there are six ways to execute X — the only difference is how they are grouped.

The command and its parameters are input variables: They exist external to the run and are used by or affect the run. Input variables can include data that influences processing in a run, as well as commands and their parameters. Input variables that differentiate one function or operation from another are called *key input variables*. In the first case, only X is a key input variable; in the second case, A is also a key input variable. In many cases, the values of a key input variable that differentiate functions are actually ranges, which are called levels.

The degree of function differentiation you choose to specify is a balance between the greater flexibility in allocating resources and setting priorities that the more detailed profile gives you and the higher cost of gathering and analyzing more detailed data — and managing on that basis. Differentiation is independent of the task of ensuring that almost all the important input values and environmental variables are covered by the defined function. In practice, because a functional profile is completed early in the design phase, the information available limits how refined the function list can be.

Explicit versus implicit. At this point you must choose between an explicit and implicit operational profile or some combination of the two because that determines if you should develop an explicit or implicit functional profile.

As its name implies, an explicit profile consists of one enumerated set of all variables taken together, with their associated occurrence probabilities. Say you have two key input variables, C and D, each with three values. The set is then $\{(C1,D1), (C1,D2), (C1,D3), (C2,D1), (C2,D2), (C2,D3), (C3,D1), (C3,D2), (C3,D3)\}$. An implicit profile, on the other hand, consists of sets of the key input variables' values, with their associated occurrence probabilities. The sets for C and D are $\{C1, C2, C3\}$ and $\{D1, D2, D3\}$.

An implicit profile can be used only when the key input variables are independent (at least approximately) of each other with regard to the occurrence probabilities of their values. If they interact, you must develop an explicit profile, because such occurrence probabilities must be measured or estimated directly, not by multiplying the occurrence probabilities of values of individual variables. In reality, there is usually some interaction among key input variables.

To illustrate the difference between explicit and implicit functional profiles, suppose key input variable C has levels $C1$, $C2$, and $C3$, with occurrence probabilities 0.6, 0.3, and 0.1, respectively, and key input variable D has levels $D1$, $D2$, and $D3$, with occurrence probabilities 0.7, 0.2, and 0.1, respectively. The occurrence probabilities of the two variables are independent. The implicit profile is given by the individual profiles of these variables. The explicit profile is shown in Table 3.

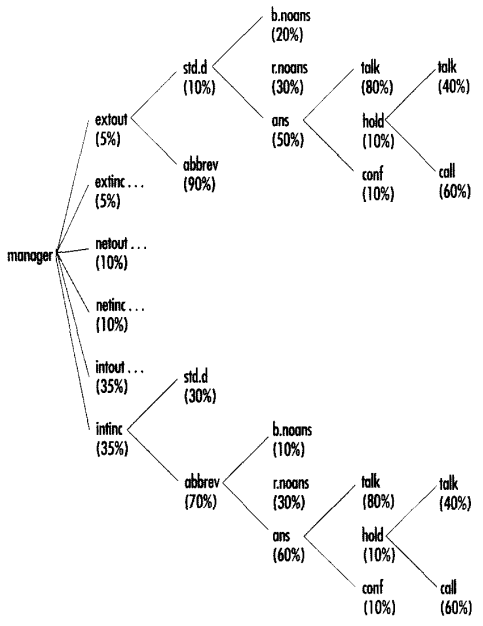

Figure 2. *The developers of International Definity used a call tree, which shows the path of a call through the network and the probability of each branch, to determine an implicit operational profile. Each branch represents a value of a key input variable, and all the branches that diverge from a node represent that key input variable. Instead of selecting test cases from a list of all possible paths, they selected from each set of branch alternates.*

TABLE 3
SAMPLE EXPLICIT FUNCTIONAL PROFILE

Input state	Occurrence probability
C1D1	0.42
C2D1	0.21
C1D2	0.12
C3D1	0.07
C1D3	0.06
C2D2	0.06
C2D3	0.03
C3D2	0.02
C3D3	0.01

The chief advantage of the implicit profile is that it requires you to specify fewer elements — a number equal to the *sum* of the number of levels of the key input variables. Using an explicit profile, the number of elements you must specify can be as high as the *product* of the number of levels of the key input variables.

A disadvantage of the implicit approach is that you can miss executing some seldom-used but critical operations in test, unless you record all combinations of key input variables executed. With an explicit operational profile, the selection of input states is direct and hence easily recorded and recognized.

The implicit approach is suited to transaction-based systems, in which processing depends primarily on transaction attributes that are generally independent and have known occurrence probabilities. A system to generate personalized direct mail, for example, depends on customer attributes (location, income, home ownership) that are essentially independent.

The developers of International Definity[2] used an implicit operational profile. *Call trees* determined the occurrence probabilities of calls in networks. As Figure 2 shows, each call has a path through the network, and each branch in that path has an associated probability. Instead of selecting test cases from a complete list of all possible paths with associated probabilities, the developers selected from each set of branch alternates with their associated branch probabilities. The result is an implicit operational profile based on a vector of key input variables. Each branch represents a value of a key input variable, and all the branches that diverge from a node represent that key input variable.

In the greatly simplified PBX example, an implicit profile is feasible because the input variables are independent, but for illustration I developed an explicit profile.

Sometimes a combination of explicit and implicit operational profiles is appropriate. If some sets of the key input variables interact, those sets can be used to determine explicit operational subprofiles. You can treat each subprofile as a key input variable with an associated occurrence probability and use it, along with key input variables that are independent of it, to determine an implicit operational profile.

Initial function list. The initial function list focuses on features, which are functional capabilities of interest and value to users. System requirements are usually the best source of information on features. If you have trouble identifying the functions, it is often because the requirements are incomplete or fuzzy.

You can use a prototype as a second verifying source, but you must use it with care, because often a prototype implements only some functions. The most recent version of the product can also serve as a valuable check, but of course it lacks the new functions planned for the next version. Sometimes there is a draft user manual, written to enhance communication with users, that you can check, but it will probably emphasize functions activated by commands, not functions activated by events or data.

User input is vital in creating the initial function list. Only those who have practical experience with the existing work process can uncover problems with a proposed process.

In the PBX example, the initial function list has four elements because the system-administration mode has four principal functions: adding a new telephone to the exchange, removing a telephone, relocating a telephone or changing the nature of service provided, and updating the online directory.

Environmental variables. Now you should identify the environmental input variables and their values or value ranges that will require separate development efforts, such as substantial new modules.

Environmental variables describe the conditions that affect

the way the program runs (the control paths it takes and the data it accesses), but do not relate directly to features. Hardware configuration and traffic load are examples of environmental variables. Probably the best approach is to have several experienced designers brainstorm a list of environmental variables that might cause the program to respond in different ways, and then decide which of these would likely have a major effect on the program.

In the PBX example, telephone type is an environmental variable that has a major effect on processing. Although telephone type can have several values, here only analog (*A*) and digital (*D*) telephones have substantially different effects on processing. So there are two levels for the environmental input variable, *A* and *D*.

Final function list. Before you create the final function list, you should examine dependencies among the key environmental and feature variables. If one variable is totally or almost totally dependent on another, you can eliminate it from the final function list. If one variable is partially dependent on another, you must list all the possible combinations of the levels of both variables, along with all the independent variables.

The number of functions in the final function list is the product of the number of functions in the initial list and the number of environmental variable values, minus the combinations of initial functions and environmental variable values that do not occur.

The final function list for the PBX, shown in Table 4, has seven elements: the three initial functions with two environmental variable values, plus the initial function "online-directory updating," which is not affected by telephone type.

TABLE 4
FINAL FUNCTION LIST

Function	Environmental variable
Relocation/change	*A* *D*
Addition	*A* *D*
Removal	*A* *D*
Online-directory updating	

Occurrence probabilities. The best source of data to determine occurrence probabilities is usage measurements taken on the latest release, a similar system, or the manual function that is being automated. Usage measurements are often available in system logs, which are usually machine-readable. Note that these measurements are of operations, not functions, so they must be combined (usually by simple addition) when a function maps to more than one operation.

Occurrence probabilities computed with this data must be adjusted to account for new functions and environments and expected changes due to other factors. Most systems are a mixture of previously released functions, for which you have measurements, plus new functions, for which you must estimate use. Although estimates are less accurate than measures, the total proportion of new functions is usually small, perhaps five to 20 percent, so the functional profile's overall accuracy should be good.

In the rare event that a system is completely new and the functions have never been executed before, even by a similar system or manually, the functional profile could be very inaccurate. However, it is still the best picture of customer use you have and so is valuable.

The process of predicting use alone, perhaps as part of a market study, is extremely important because the interaction with the customer that it requires highlights the functions' relative value. It may be that some functions should be dropped and others emphasized, resulting in a more competitive product. Reducing the number of little-used functions increases reliability, speeds delivery, and lowers cost.

In the sample PBX, there are 80 telephone additions, 70 removals, and 800 relocations or changes per month. Online-directory updating represents five percent of the total use in the system-administration mode.

In Table 2, the occurrence probability for the system-administration mode is listed as 0.02. Thus the overall occurrence probability for each of these functions, without consideration of environmental factors, is the product of their occurrence probability and the system-adminstration mode's occurrence probability in the overall system. Table 5 shows the resulting initial functional profile segment.

TABLE 5
SAMPLE INITIAL FUNCTIONAL PROFILE SEGMENT

Function	System-administration-mode occurrence probability	Overall occurrence probability
Relocation/change	0.80	0.0160
Addition	0.08	0.0016
Removal	0.07	0.0014
Online-directory updating	0.05	0.0010

To take into account environmental factors, assume that 80 percent of the telephones are analog and 20 percent are digital. The environmental profile is shown in Table 6.

TABLE 6
SAMPLE ENVIRONMENTAL PROFILE

Telephone type	Occurrence probability
Analog (*A*)	0.8
Digital (*D*)	0.2

Also assume that the occurrence probabilities of the first three functions and telephone type are independent. To determine the

ADVANCED CONCEPTS: RUNS

A run should probably be at least as long as the work accomplished between interventions by the external environment that influences the program. In an aircraft-control system, for example, a run might be the time it takes to receive sensor inputs and compute and transmit orders, at which point sensor inputs for the next cycle occur. In a switching system, a run might be a telephone call. In interactive systems, it might be each command input by a user. In transaction-based systems, it might be each transaction.

A run should not be so long that an excessive amount of information on inputs and outputs is necessary to characterize its interaction with its environment. Furthermore, it should not be so long that the chance of similarities among runs is very small. A complete flight trajectory, which might last several hours, is too long.

If the total number of runs that can occur during the life of the system is less than several thousand, the runs are almost certainly too long.

In many cases, system engineers implicitly establish the desired run length and build it into the structure of the requirements or even the system. In that case, you need only recognize the natural division that already exists. In a control program that dispatches tasks, the tasks or task sequences may logically form runs.

In a distributed system, a run may be limited to an execution sequence that occurs on one machine. However, if you consider all failures on a distributed system together, not by separate computer elements,[1] you can define runs that encompass execution on different machines.

Functions may not only be divided into runs, they may be regrouped into operations according to runs. In a bill-processing system, you might divide execution into two phases, the first for merging information about transactions from different sources and posting it to accounts and the second for processing accounts and generating bills. Thus the task of processing a particular account consists of several individual transaction runs plus an account-processing run.

The usefulness of relating runs to functions depends on whether the functions are fairly reliably isolated, with respect to sharing machine resources, so that interactions are minimized. If this is not the case, because there are no isolation features like memory protection or reinitialization or there is frequent queueing for resources in high-traffic conditions, then it may be better to define a run as a period of time that encompasses the interactions. Unfortunately, a time-based definition can require a very large number of input variables to define the input state, particularly as the time period increases.

If the system is reinitialized from time to time, it is a good idea to define runs so that they do not cross reinitialization boundaries, since interactions can't occur across those boundaries. This often means that when sequences of events, commands, or actions occur in such a way that data set by one of them is used by a succeeding one and interaction is unavoidable, it may be best to define the entire sequence as a run, preferably delimited by reinitializations.

For example, your first choice in considering a flight-control system might be to define each cycle as a run. However, it's clear that one cycle sets much of the data used by the next. A better choice in this case is a run defined as a flight maneuver (a climb), because the amount of interaction between flight maneuvers is relatively small. A complete flight trajectory (from New York to Boston) would be too long.

REFERENCE

1. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.

final functional profile, you multiply the values of the environmental profile by the values of the initial functional profile to obtain the final functional profile of the system-administration mode in Table 7.

TABLE 7
SAMPLE FINAL FUNCTIONAL PROFILE SEGMENT

Function	Environment (telephone type)	Overall occurrence probability
Relocation/change	*A*	0.0128
	D	0.0032
Addition	*A*	0.00128
	D	0.00032
Removal	*A*	0.00112
	D	0.00028
Online-directory updating		0.00100

If development is organized by modules more than by functions, you may wish to create a modular-usage table, described in the box on p. 18.

OPERATIONAL PROFILE

The functional profile is a user-oriented profile of functions, not the operations that actually implement them. But it is operations, not functions, that you test. An operation represents a task being accomplished by the system, sometimes in a particular environment, as viewed by the people who will run the system (also as viewed by testers, who try to put themselves in this position). To allocate testing effort, select tests, and determine the order in which tests should be run, the operational profile must be available when you start test planning.

Functions evolve into operations as the operational architecture of the system is developed. The *operational architecture* is the way the user will employ operations to accomplish functions. There is often some but rarely complete correlation between the operational architecture and the system architecture, which determines how modules and subsystems combine into the system.

A function may evolve into one or more operations, or a set of functions may be restructured into a different set (and different number) of operations. Thus the mapping from functions to operations is not necessarily straightforward. For example, an administrative function in a switching system might be to relocate a telephone. This single function may be implemented by two operations, removal and installation, because these tasks may be assigned to different work groups. Generally there are more operations than functions, and operations tend to be more refined. An operation is usually more differentiated than a function, in that it represents a particular task with certain specific input-variable values or value ranges.

The first three steps in determining the operational profile are to divide the execution into runs, identify the input space, and partition the input space into operations.

Divide execution into runs. Operations are associated with runs. A run is a segment of a program's execution time, usually based on the accomplishment of a user-oriented task, such as a command or transaction, always in a particular environment. A run is a somewhat arbitrary concept that exists only so that you can conveniently divide the execution and characterize the patterns of a system's use. Usually a run is a quantity of work or a set of tasks initiated by some intervention or input. It is usually terminated by the next intervention or run, unless it terminates because of failure.

In many cases, it helps to define a run as an end-to-end user activity. A program does not necessarily stop or pause between runs. In most cases, the terms "initiation" and "termination" refer merely to distinct changes in control flow.

Often runs will relate to functions, but not always, because functions may be broken into multiple runs when they are implemented. In the PBX example, for instance, the relocation/change function could be implemented as a removal run followed by an addition run.

Runs are logical, not machine entities. A run does not have to be continuous in time; it can proceed in segments. For example, a run on a multiprogrammed system may exist as a set of execution-time slices.

Identical runs form a run type. Airline-reservation transactions that are exact duplicates are runs of the same type. However, reservations made for different people, even on the same flight, are different run types. So an operation like "remove a telephone" involves many run types, each of which processes the removal of a specific telephone under specific environmental conditions (a unique number, location, type, feature set, administrative traffic load, and so on). If the telephone was removed too soon, replaced, and then removed at the correct time, the system would experience two runs of the same run type for that specific telephone.

The box on the facing page describes more advanced run concepts.

Each run type has an associated *input state*, a set of input-variable values that exist external to the run and are used by or affect it. In this definition, "used by" implies a deliberate design decision, while "affects" implies an inadvertent influence. There need not be a physical input process — the input variable may be waiting in memory or in a file to be accessed.

The input state is not the same thing as the machine state, which is the much larger set of all variable values accessible to the computer.

The input state uniquely determines the instructions a run will execute and the value of their operands; it establishes the path of control a run takes through a program. It also uniquely establishes the values of all intermediate variables and all output variables the program computes. Output variables are the data items external to the program that are set by the program. Thus the input state uniquely determines the output state.

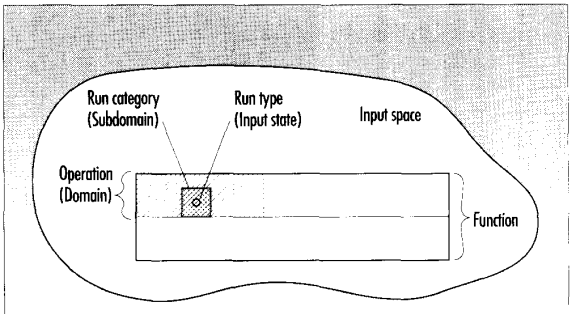

Figure 3. *A function may comprise several operations. In turn, operations may comprise many run types. Grouping run types (represented by points in the input space) into operations partitions the input space into domains. A domain can be partitioned into subdomains, which define run categories. To use the operational profile to drive testing, you first pick the domain that characterizes the operation, then the subdomain that characterizes the run category (if defined), and then the input state that characterizes the run type.*

The box on p. 24 describes more advanced input-state concepts.

Identify input space. A program's input space, illustrated in Figure 3, is the set of input states that can occur during its operation. This set is not infinite, but it is astronomically large for any program of practical use. Many programs are expected to take some rational action for environmental variations like data-entry errors, data degradation, or heavy traffic. These considerations expand the *required* input space that a program must respond to and be tested for. It may not be the same as the *design* input space that the program has been developed to work with. The areas of the required input space that do not fall in the design input space will contain input states with a higher likelihood of failure.

In defining the input space, the most important thing is to develop a practically complete list of input variables. A "practically complete" list identifies all input variables except those that take on one value with very high probability. You are ignoring and thus won't be testing the alternate values. This is acceptable because they occur so rarely that they have little effect on reliability even if they fail. The degree to which you can do this decreases for systems requiring higher reliability.

If you miss input variables that appreciably affect the program's operation, you cannot clearly identify run types and hence faulty input states. This makes it impossible to reproduce either failures or successful behavior unambiguously. For example, some variables may not be considered as inputs because it is not obvious that they influence a run until they interact with other runs. Rather than wait for such interactions, you should make a conscious effort to identify "hidden" input variables.

Because the identification process will never be perfect, you

ADVANCED CONCEPTS: INPUT STATES

An input variable can be a numerical quantity, logical variable, character string, abstract data type, or an array or structure of these items.

An input variable can have only one value for each of its elements with respect to a run. Therefore, you must consider each use or setting of a quantity that is varying as a separate variable. Each activation of an interrupt of a given type is a separate input variable.

An input variable is not a physical or symbolic memory location. The same physical or symbolic location can be "time shared" with different input variables.

Externally initiated interrupts, such as interrupts generated by the system clock, by operator actions, and by other components of the system outside the program, are input variables. Intermediate data items computed by the program during a run and not existing external to the program are not input variables. Hence, interrupts generated directly by a run or interrupts that are determined from other input variables (for example, overflow and underflow) are not input variables.

All runs will terminate sooner or later as a consequence of the input state selected. Some terminations may be premature in the sense that no useful function is completed. If a run is terminated early by operator action, the termination action represents an interrupt that signifies a different input state from a run with a normal termination.

should employ other strategies, such as reducing the input space and using indirect input variables, described later, to ensure that you handle hidden input variables properly. The amount of effort you put into this should be based on reliability requirements, the cost of the extra effort, and any information you have on the probability that these interactions will occur.

The input-state profile is the set of input states and their associated occurrence probabilities. The input-state profile unambiguously establishes failure behavior. However, it is a concept only — the input-state profile is virtually never measured in practice. But you must understand this concept to grasp the idea of perfect testing. Then you can understand how to select appropriate engineering approximations that cost-effectively approach this ideal to a reasonable degree.

The required input space may be difficult and time-consuming to delineate, especially if input states have nonzero occurrence probabilities only for certain values of input variables. Instead, define a *specified* input space by simply listing the set of input variables involved. You assume that each input variable can take on any possible value (a finite number, based on the range and granularity of the machine representation) and ignore the test resources wasted because some input states will have zero-occurrence probabilities. The specified input space will more than cover the required input space and will be much easier to define and select tests for.

A failure is a departure of the output state from program requirements. The implication is that the requirements are the operational behavior that the user and (at least eventually) the customer expects. Thus, in theory, you should be able to determine the occurrence or nonoccurrence of a failure by testing each input state (run type). I say "in theory" because the number of input states involved generally makes this determination impractical to pursue.

Partition input space. In practice, you must limit profiles to several hundred (several thousand, at most) elements, because the cost of developing them increases approximately in proportion to the number of elements. By grouping run types into operations, you can partition the input space and so reduce the number of elements. The portion of the input space that corresponds to an operation is called a domain.

The run types you group should share the same input variables so that you can set up a common and efficient test procedure for the domain. You don't necessarily group all the run types that share the same input variables. If one or more input variables have values or value ranges with easily determined and substantially different occurrence probabilities, you will probably define corresponding multiple operations so that you can differentiate the amount of testing applied. In other words, there is a trade-off between the cost of a larger operational profile and the benefit of the more efficient testing it makes possible.

Consider an airline reservation for a single-leg flight. The airline-reservation system's run types or input variables will differ in passenger name, flight number, originating city, terminating city, and so on, but all belong to the single-leg operation. A two-leg reservation, on the other hand, is a different operation with a different set of key input variables, which include the second flight number and the connecting city.

In the PBX example, "command type" is a key input variable; "location" is not. Different locations represent different run types or points within the domain of the operation associated with a command. In transaction-based systems, "transaction type" will almost always be a key input variable.

Partitioning does not imply that you select only one test per operation; it simply provides the framework for sampling nonuniformly across the input space. If you select operations randomly in accordance with the operational profile and then select input states randomly within the domain, you will have selected nonuniform random tests that match the operational profile. When an important input variable is known to be nonuniformly distributed with respect to its values, it may be desirable to pick ranges of values and use these to define operations, using the foregoing criteria.

The set of operations should include all operations of high criticality, even if they have low use. The effect of not including operations of low criticality and low use will be negligible unless reliability requirements are very high. To increase the likelihood

that all high-criticality operations are included, you should focus on tasks whose unsatisfactory completion would have a severe effect and carefully consider all the environmental conditions in which they may be executed. Postmortems of serious failures in previous or related systems often suggest some of these situations.

It may be desirable to partition at a second level by dividing the domain of an operation into subdomains that represent run categories. The division should be in terms of the input variable's levels, or value ranges. A run category is a group of run types that exhibit (as nearly as possible) homogeneous failure behavior. "Homogeneous" in this context means that all input states within the subdomain have the same failure behavior so that one run type can represent an entire run category or large set of run types. To define run categories that approach homogeneity, look for run types that at least execute the same code path. A modular usage table may help you to do this.

Try to find levels or value ranges of homogeneity for each input variable. The usefulness of this division into run categories depends on the increased testing efficiency you gain with relation to the added cost of partitioning. The alternative is to select run types randomly from the entire domain of the operation. In theory, only one run type is required per run category, greatly increasing testing efficiency. In practice, you may want to use more than one test to reduce risk, because you can approach homogeneity but very rarely achieve it. In either case, it is best to select the run types randomly from the run category.

Run categories should have approximately equal occurrence probabilities, because testers will be selecting uniformly from among them (or executing all of them), and it is too complex and time-consuming to account for executing them different numbers of times.

The requirements that all run categories should be approximately homogeneous and have equal occurrence probabilities can conflict. When they do, emphasize equal occurrence probability. Because you can achieve only near-homogeneity, the possible testing waste from dividing a homogeneous region into multiple test cases is not important. Dividing run categories to equalize occurrence probabilities (avoid grouping them) often reduces the risk of nonhomogeneity because smaller run categories tend to be more homogeneous.

Because the operational profile used in test may not match the true operational profile in the field, it is called the test operational profile. In this case, of course, an appropriate transformation will be required to convert failure intensity experienced under this profile to what would occur in the field.

Occurrence probabilities. There are two ways to determine occurrence probabilities for operations:

- record the input states in the field, group them into operations, and count them; or
- rely on estimates derived by refining the functional profile.

The first is more accurate, but obviously can be done only if a previous release exists. When adding new operations to an existing system, you must supplement use records with estimates.

Recording. It may take some effort to develop recording software, but you may be able to develop a generic recording routine that requires only an interface to each application. The recording software must instrument the system so that it extracts sufficient data about input variables to identify the operations being executed. Then it is simply a matter of counting the execution of each operation. An operational profile can be recorded in either explicit or implicit form.

Data for certain systems will require some adjustment. If a system initiates operations on a priority basis from a queue, when it is lightly loaded it executes low-priority operations (such as audits and maintenance) more frequently. Consequently, measurements of these operations' occurrence probabilities will overemphasize their role. Similarly, measurements made when the system is heavily loaded will underemphasize their importance or even indicate zero probability.

The recording process usually adds some overhead to the application. As long as this overhead is not excessive, it may be feasible to collect data from the entire community. However, if the overhead is large, you will probably have to employ a user survey instead of recording. In sampling users, the same guidelines and statistical theory used for polling and market surveys apply: a sample of 30 or even fewer users may suffice to generate an operational profile with acceptable accuracy.

TAKE CARE TO ENSURE THAT YOU HAVEN'T IGNORED EASIER WAYS TO RECORD USAGE, ESPECIALLY IF YOU ARE FACED WITH BUILDING SPECIAL TOOLS.

Take care to ensure that your measurement effort is cost-effective and that you haven't ignored easier alternatives, especially if you are faced with building special recording and measurement tools. For example, you may already have data on the frequency of events the system responds to. Or you might use a simple operational profile with a moderate number of elements and moderate accuracy for the first version of a software product, refining it for later versions only if you discover in the field that the reliability predictions from test are in error.

If you will be using the usage data for testing only, it is acceptable to directly drive testing with the complete input states recorded in the field. You need not determine the operational profile, partition the input space, or create test cases. This saves time and effort, but these savings will be reduced by the extent to which you add new operations. For the new operations, you must estimate occurrence probabilities and develop test cases.

Recording just the use of operations is much easier than recording the complete input state because it involves much less data. Of course, you must then partition the input space and generate test cases.

Estimation. For new systems or new operations added to existing systems, the operational profile is derived from estimated occurrence probabilities, with the aid of data recorded on the use of operations already deployed.

You start with the set of functions and refine them into operations by considering how they map to runs and how they have been detailed as they have been developed. A convenient way to capture this detail is to list events that initiate runs. If the events are commands, the system is command-driven. If there is a command list, examine the commands and their parameters. If a value of a parameter causes significantly different processing to occur, you may want to define a new operation. If N kinds of processing depend on the value of the parameter, then the operation derived from the command should be replaced by N operations associated with the N parameter values. If the events are data items, the system is data-driven. Look for variables that cause significant differences in processing.

Next, develop a list of environmental variables that have significant influence. In general, a given environmental variable will affect processing only for certain events.

It often helps to create an interaction matrix of key input variables plotted against other key input variables. This matrix reveals combinations of key input variables that do not occur or that interact. The remaining areas of the matrix represent regions where you can assume key input variables are independent and estimate the occurrence probability as the product of individual key input variable probabilities.

Next, you must consider each operation and estimate its occurrence probability. This is usually best done by an experienced systems engineer who has a thorough understanding of the businesses and the needs of the expected users, and how they will likely take advantage of the new functions. It is vital that experienced users review these estimates. Often, new functions implement procedures that had been performed manually or by other systems, so there may be some data available to improve the accuracy of the estimates.

You can view a system modification as adding a new operational profile to an old one. The operational profile for the old system can be measured; for the new operations, estimated. The two parts are joined by weighting each one's occurrence probabilities by the fraction that that part's total occurrence probabilities represents of the entire system.

When a system's operational profile depends on the frequencies of events that occur in an associated system, it may help to simulate the second system to establish event frequencies. For example, the operational profile of a surveillance system depends on the frequencies of certain conditions arising in the systems being monitored.

Examples. Let's examine two common types of systems, command-driven and data-driven.

Command-driven system. The PBX is a command-driven system.

In implementing the system-administration mode, this command set was developed:

```
reloc    <old location ><new location>
add      -s <service grade> <location>
remove   <location>
dirup
```

Relocation is being handled by removing the old location and adding a new one.

As you consider the parameters, you note that location does not affect the nature of the processing. However, the type of user does because the features provided are substantially different for staff, secretaries, and managers. So you refine the add command into three operations:

```
reloc    <old location> <new location>
add      -s staff <location>
add      -s secretary  <location>
add      -s manager <location>
remove   <location>
dirup
```

All these commands, except dirup, account for 0.019 of the occurrence probability; dirup accounts for 0.001. Suppose that the expected 80 additions of service per month break down into 70 staff, five secretaries, and five managers. There will be 780 relocations and 20 changes of service each month, the latter representing promotions to manager. There will be 70 removals proper and 20 removals created as the result of "change" functions, yielding a total of 90 removals. The part of the operational profile for the system-administration mode is shown in Table 8.

TABLE 8
OPERATIONAL-PROFILE SEGMENT BASED ON FEATURES

Command	Transactions per month	Occurrence probability
reloc	780	0.0153
remove	90	0.0017
add -s staff	70	0.0014
dirup		0.0010
add -s manager	25	0.0005
add -s secretary	5	0.0001

You proceed in this fashion until you have accounted for all the ways the system can be employed. Now you must consider the possible expansion of the operation list to account for environmental variables that could change the processing (and thus result in different failure behavior). There will usually be environmental variables that affect the processing sufficiently to require testing based on some of their values that you must now consider, even though they were not sufficiently major to be considered in the development of the functional profile. For simplicity, assume that the environmental and feature variables are independent of each other. In practice, you must be alert to the possibility that environmental variables may interact with feature variables in determining occurrence probabilities. For example, certain features may be executed at constant occurrence *rates* but as traffic increases, their occurrence *probabilities* decrease.

In our sample system, the environmental variable is telephone

type: The system must handle both analog and digital telephones. The operational profile in Table 8 will thus expand under this environmental variable into 11 operations (online-directory update is not affected by telephone type). Let's exclude directory update and consider the part for analog telephones *A*, which will have occurrence probabilities that are 80 percent of those for all configurations.

Assume that system load is such an environmental variable. If system-administration functions are performed when the system is in an overload condition because of heavy communication traffic, processing may be affected (administrative requests might be queued, for example). Assume that this occurs 0.1 percent of the time.

To generate the segment of the operational profile we are considering, first multiply all values in Table 8 by 0.8 to give the occurrence probabilities for analog telephones. Then multiply by 0.999 to obtain the occurrence probabilities for normal load, or by 0.001 to obtain the occurrence probabilities for overload. Table 9 is the new operational profile segment.

TABLE 9
OPERATIONAL PROFILE SEGMENT BASED ON FEATURES AND ENVIRONMENT

Command	Environment	Occurrence probability ($\times 10^{-6}$)
reloc	Normal load	12,228.00
remove	Normal load	1,359.00
add -s staff	Normal load	1,119.00
add -s manager	Normal load	400.00
add -s secretary	Normal load	79.90
reloc	Overload	12.24
remove	Overload	1.36
add -s staff	Overload	1.12
add -s manager	Overload	0.40
add -s secretary	Overload	0.08

Some operations in Table 9 occur very infrequently. You should seriously question if it is really necessary to test all of them. Consider eliminating the "add - s secretary under overload conditions."

Data-driven system. Financial and billing systems are commonly data-driven.

Suppose a telephone billing system was designed as two subsystems. The first receives call transactions and sorts them by billing period and account number, grouping all the items for one account and the current billing period. The second processes the charge entries for each account for the current billing period and generates bills.

The reliability you want to evaluate is the probability of generating a correct bill. This involves determining the reliability of each subsystem over the time required to process the bill or the entries associated with the bill, and then multiplying the reliabilities. You must determine an operational profile for each subsystem.

Because this design was not anticipated when the functional profile was developed, the relationship between the functional profile and the two operational profiles is complex. For example, typical functions may have been bill processing, bill correction, and the identification of delinquent customers. The bill-processing function relates to operations in both subsystems, but the other two functions relate only to the second subsystem.

The first subsystem, the sort subsystem, will have relatively few operations and a simple operational profile. The operation for processing correct charge items has an occurrence probability greater than 0.99; other operations handle missing data, data with recognizable errors, and so on. You should be able to estimate occurrence probabilities from past data on the frequency and type of errors.

The second subsystem, the account-processing subsystem, has an operational profile that relates to account attributes. Its operations are classified by service (residential or business), use of a discount calling plan (none, national, or international), and payment status (paid or delinquent), resulting in 12 operations.

Assume that the service classification is 80-percent residential and 20-percent business. A national discount calling plan is used by 20 percent of subscribers; international, five percent. Only one percent of accounts are delinquent. Table 10 shows the set of operations and their associated probabilities.

TABLE 10
OPERATIONAL PROFILE FOR BILLING SYSTEM

Operation	Occurrence probability
Residential, no calling plan, paid	0.5940
Residential, national calling plan, paid	0.1580
Business, no calling plan, paid	0.1485
Business, national calling plan, paid	0.0396
Residential, international calling plan, paid	0.0396
Business, international calling plan, paid	0.0099
Residential, no calling plan, delinquent	0.0060
Residential, national calling plan, delinquent	0.0016
Business, no calling plan, delinquent	0.0015
Business, national calling plan, delinquent	0.0004
Residential, international calling plan, delinquent	0.0004
Business, international calling plan, delinquent	0.0001

If transaction use is described in terms of transaction rates, you obtain the occurrence probabilities by dividing the individual transaction rates by the total transaction rate and multiplying this by the probability of transaction occurrence (with respect to other operations). If all the operations are transactions, the last step is not necessary.

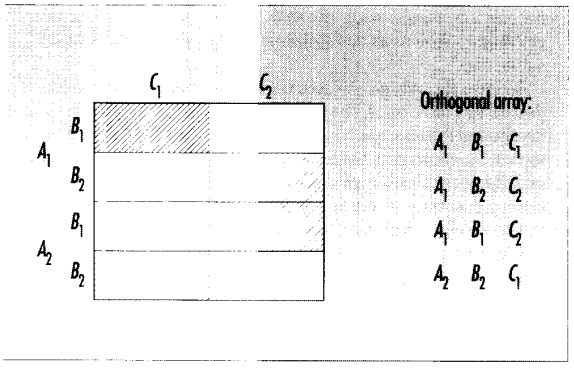

Figure 4. In this example, input variables A, B, and C each have two levels, 1 and 2. An orthogonal array assumes that failures are influenced only by the variables themselves (A, B, and C) and their pairwise interactions (AB, AC, BC), which means you need test only half the run categories to activate all possible failure behaviors.

TEST SELECTION

The operational profile is used to select operations in test in accordance with their occurrence probabilities. You then select run categories (if defined) and finally specific run types. You can see this in Figure 3, where you first pick the domain that characterizes the operation in input space, then the subdomain that characterizes the run category, and then the point that represents the input state that characterizes the run type.

Testing driven by an operational profile is very efficient because it identifies failures (and hence the faults causing them), on average, in order of how often they occur. This approach rapidly increases reliability — reduces failure intensity — per unit of execution time because the failures that occur most frequently are caused by the faulty operations used most frequently. Users will also detect failures in order of their frequency, if they have not already been found in test.

Selection should be with replacement for operations and run types: After test, replace the element in the population, allowing reselection. Because the number of operations is relatively small, you are likely to repeat one. But the run types will almost certainly be different and their failure behavior will often also differ. The number of run types is so large that the probability of wasting test resources by repeating many run types is infinitesimal. You should select run categories with replacement, except when you use experimental design techniques, as described later.

Selection could be performed without replacement, in which an element can be chosen only once. This is unwise for operations because they can be associated with multiple faults. There is a high risk that different run types within an operation may show different behavior.

In practice, it is difficult to enumerate all environmental variables in the input-state definition. Thus, tests are usually selected from the incomplete design input space, not the required input space. Because many environmental variables change over time, repeating the same operation at different times (selecting with replacement) may in effect be yielding samples from an approximate required input space.

Selecting operations. With an explicit operational profile, you select operations directly in accordance with their occurrence probabilities. With an implicit operational profile, you select operations by choosing the level of each key input variable in accordance with its occurrence probability, which implicitly selects the operation at the conjunction of these values.

If different profiles (system modes) occur at different times in the field, you should conduct separate tests. However, if they occur simultaneously, testing should be concurrent, because system modes running simultaneously can interact. The execution time allocated to each system mode should be proportional to its occurrence probability. Concurrent testing in effect combines multiple operational profiles into a single one.

If different versions of the software product are supplied to different customers, they may differ primarily in system-mode profiles. If interaction among system modes is nonexistent or small, you can test each system mode independently. Failure intensities for the different customers can be obtained by weighting the system-mode failure intensities by the occurrence probabilities. The result is substantial savings in test time.

You may also want to test several operational profiles that represent the variation in use that can occur among different system installations to determine the resulting variation in reliability.

If possible, you should select operations randomly, to prevent some unrealized bias from entering into the testing process. Bias usually relates to time. Data corruption increases with the length of time between reinitializations, for example, so if one operation is always executed early and another always late, your tests may miss significant failure behavior.

Random selection is feasible for operations with key input variables that are not difficult to change. However, some key input variables can be very difficult and expensive to change, such as one that represents a hardware configuration. In this case, you must select some key input variables deterministically, because changing variables during system test must be scheduled.

It is wise to randomly select as many key input variables as possible. Carefully consider the bias that might result from those you select deterministically and try to counter it. For example, reinitialize the system at random times to avoid data-corruption bias.

Some operations can occur only in sequence with other operations. A switching system can set up a conference call only after it has set up an ordinary call. Operation sequences should be established, counted, and selected as superoperations. In this case, you would have the superoperations of an ordinary call and an ordinary call converted to a conference call. The operation "conversion to a conference call" by itself would not be selected.

Selecting within operations. Consider partitioning the operation into run categories, as previously described. If there is limited interaction among the input variables with respect to failure behavior, you may be able to use statistical experimental design techniques to reduce the number of run categories that must be selected. Because the goal is to reduce the number of selections, you should make them without replacement. One common experimental design is the orthogonal array, which assumes that failures are influenced only by the variables themselves (*A*, *B*, and *C*) and their pairwise interactions (*AB*, *AC*, *BC*). If failures were influenced by the three-way interaction *ABC*, you would not have an orthogonal array. In Figure 4, the absence of higher than second-degree interactions makes it possible to select only half the run categories to activate all possible failure behaviors, substantially improving testing efficiency. If higher order interactions are occurring, you must sample all subdomains (in the terminology of experimental design, conduct a full factorial experiment).

Criteria for determining in practice how to select input variables and levels so that you can best apply orthogonal array and related techniques is the subject of research.

TO HANDLE CRITICALITY, YOU CLASSIFY OPERATIONS BY MAGNITUDE OF EFFECT AND GENERATE AN OPERATIONAL PROFILE FOR EACH CATEGORY.

Selecting for regression test. Regression testing can be a substantial portion of the overall test effort. Regression tests are run after changes have been made, when faults spawned (introduced while removing other faults) by changes have just been introduced. Because changes are frequently grouped and introduced periodically, regression testing is also usually periodic. A week is a common interval, although intervals can be as short as a day or as long as a month.

Some testers say regression testing should focus on operations that contain the changed code. This view makes sense if you are sure the possible effects of the changes are isolated to those operations or if system reliability requirements are low so that cross-effects to other operations do not matter.

However, in most cases you cannot rely on isolation, and potential cross-effects can cause unacceptable deterioration in system reliability. So all operations should be considered when planning a regression test. However, a change generally results in a smaller probability of failure than a new program, so it isn't really necessary to retest every operation after every change.

It is inefficient to cover operations of unequal occurrence frequency with equal regression testing; hence operations should be selected in accordance with the operational profile.

SPECIAL ISSUES

As we used the operational profile on several projects, we encountered some special situations for which solutions had to be researched. I address the most important of these cases here.

Criticality. Operations can be classified by criticality: value added (increased revenue or reduced cost) or the severity of the effect when they fail. Effects include risk to human life, cost, or reduction in capability. Cost consists of direct and indirect (damage to reputation) revenue loss and the cost of failure workaround, resolution, and recovery. In some cases, an operation can fail in different ways, with different severities. We use the average of the severities, weighted by relative probability, as the operation's criticality.

In considering financial effect, old operations can be more critical than new operations because their failure disrupts existing capabilities that users rely on. At least part of this effect may be captured by higher occurrence probabilities for these operations. On the other hand, new operations may be critical to the success of a new product, yet may not have a very high estimated occurrence probability. You may have to assign them a high criticality to reflect their importance.

The best way to handle criticality is to classify operations by the category of criticality and then generate operational profiles for each category. If the most critical operations have low occurrence probabilities, you can test with a constant multiple of the occurrence probabilities, divide the resultant estimated failure intensity by the multiple, and obtain a true failure intensity. It is common to use four categories, each separated from the next most critical by one order of magnitude of effect. You must define a failure-intensity objective for each criticality category and all must be met to ensure satisfactory operation.

The margin for error in reliability estimates is almost always smaller for the most critical category. So the effects of environmental input variables, like traffic fluctuations and entry errors, will often be material for the critical category; testing for operations in that category must cover them.

Another way to handle criticality is use an operational profile weighted by criticality to drive test planning and execution. Let p_k' represent the weighted occurrence probability for operation k. Now

$$p_k' = \frac{c_k}{\sum_l c_l p_l} p_k$$

where c_k is the criticality of operation k and p_k is its occurrence probability. But this approach is undesirable, because you do not obtain a true reliability estimate.

Controlling the number of operations. There are three general approaches to reducing the number of operations:

- Reduce the size of the input space.
- Increase the size of the domain associated with each operation (the operation's breadth).

♦ Ignore the subset of least-frequently occurring operations that have a total occurrence probability appreciably less than the failure-intensity objective.

The first approach involves changing the system requirements. The other two approaches involve changing only the way you model system use.

Reducing the input space. One way to reduce the task of determining an operational profile is to reduce the size of the input space. This has the added benefit of reducing the testing effort.

You reduce the input space primarily by selecting an appropriate software and hardware architecture and by using careful design techniques. If you can reduce the input space early in the project, you may also reduce design and implementation costs. This argues for analyzing the input space at the front end of development.

Practically speaking, the input space can be reduced by reducing the number of input variables or the number of values for each input variable. In general, the number of input variables is more likely to influence program control flow and failure behavior than the number of values, so you should give this more attention. Some ways to reduce the number of input variables are:

♦ Reduce the number of operations.

♦ Reduce the number of possible hardware configurations.

♦ Restrict the environment the program must operate in.

♦ Reduce the number of types of faults (hardware, human, software) the system must tolerate.

♦ Reduce unnecessary interaction between successive runs.

ONE WAY TO REDUCE THE TASK OF BUILDING AN OPERATIONAL PROFILE IS TO REDUCE THE SIZE OF THE INPUT SPACE.

All these approaches have costs. The first four, which change the system's features, may involve customer objections, less flexibility, less robustness, or reduced reliability, respectively. The disadvantage associated with reducing operations may be more apparent than real. It may be possible to build systems with the same functionality but fewer operations by applying the reduced-operation software concept,[3] analogous to reduced-instruction-set computing. With this approach, you do not implement operations that occur rarely. Instead, they are accomplished by executing sequences of other operations or combining other operations with manual intervention. To decide which operations should not be implemented, look at the economic trade-off between reduced development cost and potentially higher operating costs.

The fifth option, reducing unnecessary (not required for functional reasons) interaction between successive runs, is highly desirable. It requires only a change in test plans, not negotiation with customers. Reducing interaction not only simplifies test planning, but substantially reduces the risk of failure from unforeseen causes. You must recognize, however, that the extent to which you can do this may be limited. And there is usually some cost in greater execution time.

Some ways to reduce unnecessary interaction are

♦ Design the control program to limit the input variables that application programs can access at any one time (information hiding).

♦ Reinitialize variables between runs. Because it may be difficult to determine with high confidence which input variables are influencing runs, it may be simplest to reinitialize all of them between runs. If the resulting overhead is excessive, reinitialize periodically, which reduces overhead but allows more interaction.

♦ Use synchronous (time-triggered) instead of asynchronous (event-triggered) design. Synchronous design lets you better control the input variables that are in play at any time. However, it may add overhead, and it requires extra measurement and planning to prevent functions from being aborted when deadlines are missed.

Reducing interactions has a higher risk than the other approaches to reducing input space. It is more complex and hence more error-prone, so "sneak" interactions may remain. Also, we know less about how to best exploit reduced interactions to reduce testing.

Increasing operation breadth. The second approach to reducing the number of operations, increasing an operation's breadth, involves increasing the difference in occurrence probabilities among run types required to establish separate operations. Operations that do not meet the higher differentiation standard are merged, provided they share the same input variables. If the number of randomly selected tests in the larger domain equals the sum of the number of tests of the smaller domains, the risk of missing a failure is not substantially changed, providing the failure probability of a run in each of the smaller domains is approximately the same.[4]

Ignoring infrequent operations. Suppose you exclude the set of operations with the lowest occurrence probability from test. Let the sum of their occurrence probabilities equal p_E. Assume the failure intensity at the start of test is λ_0; at the end, λ_F. Assuming that faults initially are distributed uniformly with respect to operations, then operations contribute to the failure intensity in proportion to their occurrence probability. The excluded operations will contribute $p_E\lambda_0$ to the failure intensity at the start of test. This number will be the same or less at the end, assuming that no faults are spawned that could cause any excluded operation to fail. Because this contribution will not be measured, λ_F will be low by at most this amount. Let ε be the maximum acceptable error in measuring failure intensity. Then, setting ε equal to $p_E\lambda_0$, you obtain the allowable value of

$$p_E = \frac{\varepsilon}{\lambda_0}$$

Suppose the failure-intensity objective is 10 failures per 1,000 CPU hours. It might be reasonable to set ε equal to one failure per 1,000 CPU hours. If $\lambda_0 = 10^5$ failures per 1,000 CPU hours, then $p_E = 10^{-5}$. This means that once the total occurrence probabilities of the operations you are testing reaches $1 - 10^{-5}$, you can ignore the rest. The higher the failure-intensity objective (the lower the reliability), the more operations you can exclude. Obviously, as criticality increases, the degree to which functions can be excluded diminishes.

Indirect input variables. Sometimes the relationship among observable task and environmental variables and input states is not clearly discernible, at least not without an expensive effort. In this case, you can establish and employ indirect input variables to control test selection. An indirect input variable is believed to affect processing, but is not used by the program directly.

Consider traffic load. It is neither practical nor enlightening to determine which input-variable values are caused by heavy traffic and directly affect processing. It is better to actually generate a heavy traffic load and observe the results. The program accesses no "traffic level" variable, but you can consider the traffic level generated as an indirect input variable. You can select levels of these indirect input variables randomly, in accordance with estimated occurrence probabilities. But indirect input variables are not mere input variables; they must actually control the program's environment.

Another approach is to define different system modes for heavy and normal traffic, but you must be careful not to define too many system modes, because each requires an operational profile.

Indirect input variables are particularly useful for handling the effects of data corruption. Data corruption is the accumulated degradation in data with execution time that results from anomalies in intermediate variables that do not represent failures. Some of this data is in reality input variables for other operations, but the interaction is often not known. In this case, you can define an indirect input variable called "soak time" in terms of hours of execution and plan to test several different values of this variable. You may implicitly select the values by performing a "soak test," in which you continuously increase soak time up to a limit, with operations randomly chosen in accordance with the operational profile in this interval. In either case, you should include a soak time slightly less than the reinitialization interval you expect to use in the field.

Correlated occurrence probabilities. Although many operations are independent or essentially independent, others can affect the selection of the next operation. There are two ways to handle this:

- Define superoperations.
- Select operations separately in accordance with the operational profile.

To define superoperations, you identify those operations for which interactions are significant and determine all their possible sequences. Treating these sets of operations as superoperations, you determine their occurrence probabilities and replace that part of the operational profile by this superoperational profile.

Although this is the most precise way to approximate reality, it is also the most expensive, because it increases the number of elements in the operational profile. But the cost may be acceptably low if the sequences that can occur are limited and highly structured, because their operations are highly dependent on each other.

If you use separate selection with replacement, sequences will be selected naturally. This approach is painless, because it takes no additional effort. The disadvantage is that it may yield unlikely sequences that waste testing effort and distort reliability estimates if there are substantial correlations between the operations. Consider operations *A*, *B*, and *C* with the operational profiles 0.4, 0.3, and 0.3. Suppose that 95 percent of the time operation *A* is followed by operation *B* and that it is rarely followed by operation *C*. Random selection in accordance with the operational profile will produce the unlikely sequence *AC* as frequently as the likely sequence *AB*. If the number of unlikely sequences is small, you could use separate selection of operations and remove unlikely sequences as exceptions. But if the number of unlikely sequences is large, removal might be unwieldy.

EACH NEW RELEASE WILL NECESSITATE MODIFYING THE OPERATIONAL PROFILE, SO YOU WILL NEED A REGULAR MEASUREMENT PROGRAM.

Updating the operational profile. An operational profile can change during the life of a product, especially when new features are regularly made available through new releases. Each new release will necessitate modifying the operational profile. Because it is best to base the modified operational profile on measured data, a regular operational-profile measurement program is recommended.

In the long run, the simplest way to do this is to build the measurement capability into the system. This involves counting and recording the number of runs of each run type. You can combine this measurement system with a failure-detection and -recording function or other performance-measurement system. The most economic and reliable way to collect data is through periodic reporting over telecommunication channels to a central location. If this is not feasible, you can collect data on a removable medium that is mailed periodically to a processing center.

Some designers may be concerned about performance degradation caused by the recording function. For the amount of data needed and the length of the runs involved, performance degradation is unlikely, but recording excessive amounts of data could cause a problem. If performance does become a problem, you can sample a set of sites or a set of time periods, as long as you are careful to sample randomly. Built-in recording may not happen

until suppliers and customers learn through experience to appreciate its value. In this case, the best approach is to take measurements at a randomly selected sample of sites. Root-cause failure analysis may also provide data, because it sometimes leads to uncovering system uses that were not known and hence not tested.

Generally, operational profiles should be updated when there are major releases that represent substantial changes in capabilities and expected use. As a system passes through different versions or releases, the functional profile usually needs updating less frequently than the operational profile, because it is used mainly to prioritize tasks and allocate resources and thus can be less refined.

If new functions are added, their occurrence probabilities must be estimated. Suppose that the new functions' usage totals p_N. If the new functions do not affect the occurrence probabilities of the old functions except by adding to the overall set of functions, the old functions' probabilities are adjusted by multiplying by $(1 - p_N)$. However, if some old functions are replaced or otherwise affected, the probabilities are adjusted individually.

Distributed systems. You can apply operational-profile techniques to distributed or networked systems if they are engineered, tested, and managed as a whole. This implies that "operation" refers to a task that involves part or all of the total system, not just one component. There is nothing that restricts the concept of an "operation" to a program that executes on a single machine.

All the concepts relating to input space — run, run type, run category, operation, and function — and failures are logical, in the sense that they can span a set of software, hardware, and human components. For example, a run can consist of a series of segments, each executed as a process by a server, with the servers being implemented on the same or different machines.

The functional profile can be used to guide resource allocation and set priorities in development with respect to the entire system. The operational profile can guide testing of the entire system as a unit.

Delineating run types, run categories, and operations for distributed and networked systems can be more complex because the set of environmental variables can be appreciably larger. Although going from a centralized to a distributed system does not increase the number of task variables, it often increases the number of environmental variables, such as traffic load and soak time, because you may have to specify them with respect to individual machines.

You can counter the proliferation of environmental variables by carefully designing both the system and its operating procedures. For example, you can design a system so that all machines have similar traffic loads (as a percentage of capacity). And you can equalize soak time by synchronizing reinitializations.

You can, of course, also apply the operational profile to any subsystem as long as that subsystem has operations that relate directly to users.

This systematic approach organizes and makes more efficient the process of determining the operational profile and using the information it provides to guide software development. Many projects are using it, and in one large software-development organization, it is being fully integrated into the development process.

At AT&T, software-reliability engineering, which includes the operational profile, was approved as a "best current practice" in 1991. To qualify as a best current practice, a technique must have substantial project application with a documented, favorable benefit-to-cost ratio, support by world-class technology, and mechanisms for technology transfer (courses, reference material, jump starts and consulting, tools) in place. It must pass probing reviews by two committees of senior software managers.[5]

Although savings will vary, on an average project, we estimate that test costs can be reduced by 56 percent, or 11.5 percent of total project cost. These estimates are supported by the results achieved on AT&T's International Definity project and Hewlett-Packard's multiprocessor-operating-system project. You can expect to save even more if you use the operational profile to guide other development phases as well. ♦

John D. Musa is supervisor of software-reliability engineering at AT&TBell Laboratories, Murray Hill, N.J. He has been involved in software-reliability engineering since 1973,developing two models and making other contributions to theory. He has published more than 60 papers and is the principal author of *Software Reliability: Measurement, Prediction, and Application* (McGraw-Hill, 1987).

Musa received an MS in electrical engineering from Dartmouth College. He is a fellow of the IEEE.

Address questions about this article to Musa at AT&T Bell Laboratories, Room 2C535, 600 Mountain Ave., Murray Hill, NJ 07974; Internet j.d.musa@att.com.

ACKNOWLEDGMENTS

I thank Ed Shaughnessy for suggesting the concept of the customer profile, Bill Everett for recommending the operational profile be viewed as evolving during the life cycle; and Frank Ackerman, Beto Avritzer, Mary Donnelly, Willa Ehrlich, George Estes, Bill Everett, Mario Garzia, Bliss Jensen, Bruce Juhlin, Himanshu Pant, John Stampfel, Elaine Weyuker, and Geoff Wilson for reviewing this article and providing many helpful suggestions.

REFERENCES

1. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
2. S.R. Abramson et al., "Customer Satisfaction-Based Product Development," *Proc. Int'l Switching Symp., Vol. 2*, Inst. Electronics, Information, Communications Engineers, Yokohama, Japan, 1992, pp. 65-69.
3. J.D. Musa, "Reduced Operation Software," *ACM Software Eng. Notes*, Vol. 16, No. 3, p. 78.
4. D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Soft. Eng.*, Dec. 1990, pp. 1402-1411.
5. W.W. Everett and J.D. Musa,, "A Software-Reliability Engineering Practice," *Computer*, Standards, Mar. 1993, 2 pp.