Performance Measures for Control Computers

C. M. KRISHNA AND KANG G. SHIN, SENIOR MEMBER, IEEE

Abstract—**As real-time systems become more complex, the problem of controlling them safely becomes more difficult. When computers are in control of time-critical systems such as aircraft, nuclear reactors, and life-support systems, they must meet stringent performance requirements. However, these requirements cannot be properly stated without appropriate performance measures. We review and analyze such measures in this paper.**

I. Introduction

A. Motivation

THERE has been an increasing trend toward automating the control of industrial and other processes. There are two reasons for this. First, computerized or otherwise automated control tends in many cases to be less costly than manual control. Second, there are instances where the processes are too complex to be controlled manually.

One good example of processes which are becoming too complex to be manually controlled is aircraft. The highly fuel-efficient aircraft of the future are likely to be far less stable aerodynamically than the aircraft that we fly in today. This calls for improved methods to maintain stability; in particular, for the capacity to respond very quickly to any onset of instability. The automatic maintenance of aerodynamic stability—something done today by relatively simple means—will become a more complex problem and require the use of computers.

The difference between control computers of the future and the machines used in apparently similar roles today is to be found in the consequences of their failure. In most cases today, the failure of a control computer does not lead to catastrophe: human overseers can at least shut the process down safely. By contrast, the failure of a computer guiding an intrinsically unstable aircraft can be expected to result in an air crash, and one can think of analogous disasters in highly complex industrial processes.

The criticality of such systems makes accurate performance prediction imperative. If one is to characterize performance accurately, one needs good performance measures. And that is the justification for the material of this paper.

B. Scope and Organization

We intend this paper to be used as a bibliography as well as guidance to performance analysts and intelligent laymen of this relatively new and growing field. While this paper concentrates on performance measures and not on modeling or measurement, the techniques of modeling and measurement play a major role in determining the practical relevance of a measure.

In the rest of the Introduction, we first consider the challenges and opportunities offered to computers in the control context. Then follows a brief subsection in which we convey the fact that performance measures are a language or medium of communication, and that by the very act of choosing a set of measures, an analyst imposes a scale of values on his analysis. Thus, choosing a performance measure should not be taken lightly.

In Section II, we study the properties that good performance measures must satisfy. In Section III, we discuss ways by which the demands of the application can be expressed in a form fit for analysis. Section IV contains a list of performance measures, and the paper concludes with Section V.

C. Definitions

We have found it convenient to collect some useful terms and definitions in this section.

A *distributed computer* is a collection of processing elements embedded in an *interconnection network,* which may or may not have a global memory equally shared by all processors.

The *state* of anything is an encoding of all that is relevant about its condition. For example, the current state of a distributed computer will include the number of processors and links currently operational. The current state of an aircraft might include its velocity and heading as well as an encoding of the condition of the aircraft hardware. The operative word in the definition of *state* is "relevant." The *state-model* of a system—which is a collection of all states possible for the system to be in—will depend for its complexity on how much detail is necessary to capture in the model. For a further discussion, see, for example, [2].

Automatic control means the control of mechanical, electrical, or other systems without human intervention. For a good introduction, see [30].

The *controlled system* or *controlled process* is, as the term implies, whatever is being controlled. Similarly, the *control computer* is the digital computer entrusted with the automatic control of the controlled system or process.

A *performance measure* is a language, framework, or index, through which to express what is relevant about the capability of systems. Here, as in *state,* the operative word is "relevant."

The objective of control is to operate the controlled system safely and optimally. Optimality is calculated with respect to some performance index such as fuel, energy, time, etc. [1], [27], [37]. The quality of the control provided is measured by the performance index.

A system is said to be *gracefully degradable* if there exists one or more states in which the system is neither at full capability nor in total collapse.

Consider a stochastic process $X = \{X(t, \omega) | t \in T, \omega \in \Omega\}$, where T represents time and Ω the event space [12]. Specific realizations of X are called *sample paths.* For example, when the random variable $X(t, \omega_i)$ for a realization $\omega_i \in \Omega$, and $t \in T = [0, \infty)$ represents the number of functional processors at time t, the sample path would be the actual "movement" undertaken by the system through the states of complete functionality (all processors functional) all the way to total collapse (no processor functional). Since there is an infinite number of ways in which this progression can occur (for example, there is an infinite number of epochs at which the first failure will occur), the stochastic process in this example can follow any of an infinity of sample paths.

Manuscript received January 21, 1986; revised October 20, 1986. Paper recommended by Past Associate Editor, W. F. Powers. This work was supported in part by NASA under Grant NAG-1-296.

C. M. Krishna is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003.

K. G. Shin is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8714451.

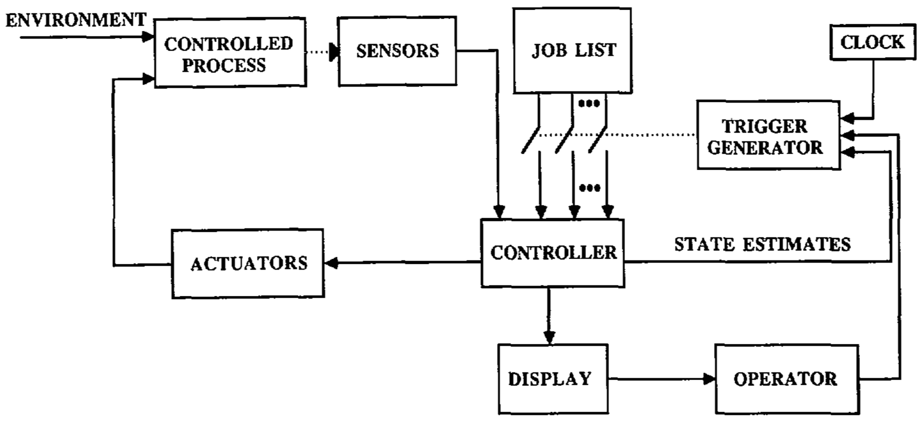


Fig. 1. A typical real-time control system.

D. Real-Time Systems [38]

Fig. 1 shows the block diagram of a typical real-time control system. The inputs to the control computer are from sensors that provide data about the controlled process and from the environment. This is typically fed to the control computer at regular intervals. Data rates are usually low: generally fewer than 20 words per second for each sensor. The job list, which is the set of tasks or jobs to be run on the control computer represents the fact that all the control software is predetermined and partitioned into individual jobs.

Central to the operation of the system is the *trigger generator* that initiates execution of one or more of the critical tasks. In most systems, this is physically part of the control computer itself, but we separate them here for purposes of clarity. Triggers can be classed into three categories.

1) Time-Generated Trigger: These are generated at regular intervals, and lead to the corresponding control computer job(s) being initiated at regular intervals. In control-theoretic terms, these are *open-loop* triggers.

2) State-Generated Trigger: These are closed-loop triggers, generated whenever the system is in a particular set of states. A simple example is a thermostat that switches on or off according to the ambient temperature. For practicality, it might be necessary to space the triggers by more than a specified minimum duration. If time is to be regarded as an implicit state variable, the time-generated trigger is a special case of the state-generated trigger. One can also have combinations of the two.

3) Operator-Generated Trigger: The operator can generally override the automatic systems, generating or canceling triggers at will.

The output of the control computer is fed to the actuators or the display panel(s). Since the actuators are mechanical devices and the displays are meant as a human interface, the data rates here are usually low. Indeed, a control computer generally exhibits a fundamental dichotomy from many points of view. First, the I/O is carried out at rather low rates (the only exceptions to this that we know of are control systems that depend on real-time image-processing: such applications have extremely high input data rates), and the computations have to be carried out at high rates owing to real-time constraints on control. Second, the complexity of the data processing carried out at the sensors and the actuators is much less than that carried out in the main data-processing area. Third, the sensors, actuators, and the associated equipment are entirely dedicated to the performance of a particular set of tasks, while the hardware in the region where the complex data processing takes place is usually not dedicated.

It is therefore possible to logically partition real-time computer systems into *central* and *peripheral* areas. The peripheral area consists of the sensors, actuators, displays, and the associated processing elements used for the preprocessing and formatting of data that are to be put into the central area, and the "unpacking" of data that are put out by the central area to the actuators or displays. The central area consists of the processors and associated hardware where all the higher level computation takes place. Designing the peripheral area is relatively straightforward; the most difficult design problems that arise in these systems usually concern the central area. Fig. 2 emphasizes these points.

A control system executes "missions." These are periods of operation between successive periods of maintenance. In the case of aircraft, a mission is usually a single flight. The operating interval can sometimes be divided into consecutive sections that can be distinguished from each other. The sections are called *phases*. For example, Meyer *et al.* [33] define the following four distinct phases in the mission lifetime of a civilian aircraft:

1) takeoff/cruise until VHF omnirange (VOR/distance measuring equipment (DME) out of range,
2) cruise until VOR/DME in range again,
3) cruise until landing is to be initiated,
4) landing.

The current phase of the control computer partially determines its job load, job mix, job priorities, and so on.

A real-time system typically has to function under more constraints than does its general-purpose counterpart. First, there are deadlines for tasks to meet. Timing is crucial to successful job completion. Second, some physical constraints are more stringent than for the general-purpose computer. Examples are weight and power consumption.

The applications software has the following properties.

1) The effects of tasks on one another are well understood.
2) Clear lines of authority are recognized.
3) Clear lines of information flow are recognized.
4) The products of the tasks are well defined.
5) The system of tasks is nearly decomposable.

These also happen to be the five conditions for the efficient design of a distributed system listed by Fox [16]. Because of this, and due to their potentially high reliability, distributed systems are particularly suited for use as control computers.

Also, the problems that arise when one attempts to run general-purpose software on a distributed system do not occur with these special-purpose machines. The chief problem in the former case is that to run anything on a distributed system, one needs to disentangle the original distributed program into nicely interacting sequential streams: a task which is not always possible to do reliably, and almost impossible to do efficiently. In contrast, the software for a control computer is not so much a single partitionable package as a set of cleanly interacting subroutines. Macro-instruction languages show much promise in this context [21].

The constraints on real-time systems, as well as the properties

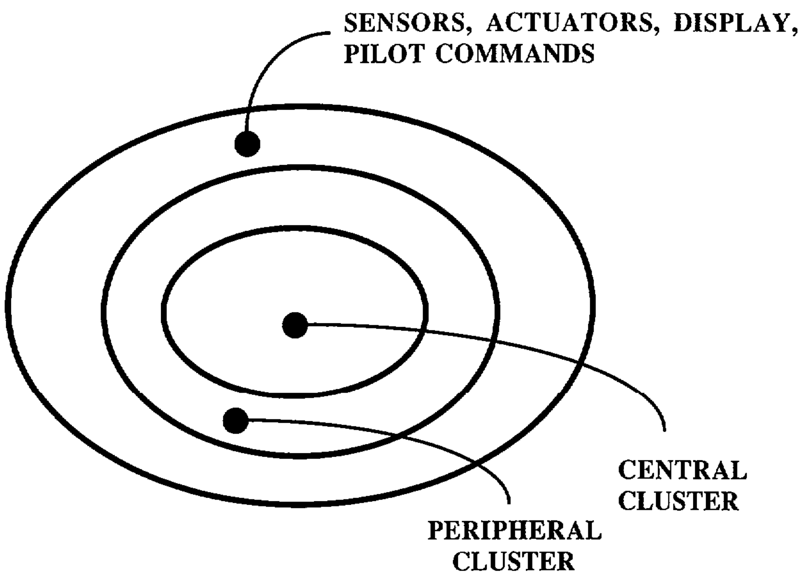


Fig. 2. Schematic decomposition of a real-time control computer.

of the applications software, have a great influence on the system architecture and the executive software.

The system architecture should be fault-tolerant since the overall computer system has to be much more reliable than any of its components. Massive replication of hardware is commonplace, as are high interconnection-link bandwidths. Recovery from failure must be quick, and both the hardware and the system software should be designed with this in mind.

The executive software must reflect the constraints on time and resources. The executive is responsible for the control of queues at shared resources, for the scheduling of events, for the handling of interrupts, and the allocation of memory. While all these tasks are common to general-purpose systems, the existence of hard deadlines makes the efficient and reliable execution of such activities imperative. The designer of a real-time system does not have the luxury of being able to assume that occasional serious degradation of performance is acceptable, if unfortunate.

An additional important task of the executive is fault-handling and recovery. This includes reconfiguration where that is possible, and the rescheduling of tasks upon processor failure. Here again, the constraints on time make this a difficult problem.

We can now see that real-time control computers differ from more conventional computers in a number of ways.

First, as we have pointed out above, the consequences of a control computer's failing are likely to be far more serious than for a general-purpose computer, so that a good analysis of its capabilities is needed.

Second, there is far more information available *a priori*. In the case of a control computer, the job load consists of the same jobs being repeatedly executed. Since one is dealing here with *critical* systems, such as nuclear reactors or aircraft, much modeling, simulation, and measurement needs to be done before any detailed specifications can be drawn up. The job mix is known: although the actual programs to be run may not always be available to the analyst, one can expect to have a good deal of information on the nature of the jobs. For instance, a certain job may be Kalman filtering [26], implemented according to some given algorithm. By contrast, the analyst of a general-purpose computer used, for example, in a university computation center must make do with benchmarks [7], [15], synthetic programs [8], or kernels [39] knowing full well that the validity of the results obtained depends on the actual jobs run on the computer, on the nature of which, nobody, except the users, has much low-level control.[1] For a good discussion of the drawbacks in benchmarks, synthetic programs, and kernels, see [32]. Also, since the operating environment of critical systems is stochastically modeled to a fair degree of accuracy, and the operating environment determines the job load to a large extent, the job load can be expected to be known much more accurately for real-time systems than for their general-purpose counterparts.

[1] Hatt cites instances where benchmarks were in error by as much as 600 percent in predicting performance [7].

The two differences above make it both possible and more necessary to obtain a good performance characterization of real-time control computers.

E. Some Observations about Performance Measures

No single measure yet invented has the power to describe computer characteristics completely. Every measure is, in effect, a partial view of the system since it is either sensitive to only a few dimensions of performance, or so general in scope as to have very poor resolution. For this reason, the choice of a performance measure implies the imposition of a scale of values on the various attributes of a computer. The choice of mean response time, for example, implies that occasional abnormalities in response time are tolerable.

The incorrect choice of performance measures can therefore lead to an unsatisfactory characterization of performance. The performance analyst must first consider the appropriateness of his measures before proceeding to determine and use them. Unfortunately, while it is possible to quantify performance measures, it is impossible—at present anyway—to quantify their appropriateness. Perhaps as a result of their training, most analysts are extremely uncomfortable about admitting to anything about their trade that cannot be quantified. This is surely one reason there is so little in the literature on the appropriateness of performance measures.

II. Properties of Performance Measures

A performance measure must do each of the following if it is to be comprehensive:

1) express the *benefit* gained from a system, and
2) express the *cost* expended to receive this benefit.

Preferably, both cost and benefit should be expressed in the same or convertible units, so that the net benefit can be computed.

Benefit relates to the rewards that accrue from the system when it is functional. Benefit may be a vector—which happens when we wish to relate each system state (e.g., number of components surviving) to the reward that accrues. How to estimate the benefit, or even how to quantify it, is usually a moot point.

The *cost* has three components associated with it. They are as follows.

1) Costs that arise when the computer does not function even at the lowest level of acceptability.

2) Life-cycle cost—capital, installation, repair, and running costs. This includes design and development costs, suitably prorated over the number of units manufactured.

3) Design and development costs.

(1) If the control computer does not function even at the lowest level of acceptability, then catastrophe can follow. One can put a price on this by, for example, estimating the legal damages that must be paid as a result.

(2) Life-cycle costing is extremely difficult to do. The large amount of effort that has been spent on developing methods to do life-cycle costing have as yet produced no convincing results, although some approaches have been proposed. There is a large literature on the subject: see, for example, [3], [9], [41].

(3) Techniques for the accurate estimation of design and development costs are nowhere in sight.

Given all of this, it is not surprising that the design, or even the choice, of appropriate performance measures is nontrivial. Indeed, no performance measure that accurately expresses either the benefits or the costs in terms of the attributes (i.e., characteristics) of a computer has yet been found.

Lowering our sights a little, we can list requirements that are more easily attained. Performance measures must:

R1: represent an efficient encoding of relevant information,

R2: provide an objective basis for the ranking of candidate controllers for a given application,

R3: be objective optimization criteria for design,

R4: represent verifiable facts.

R1: One of the problems of dealing with complex systems is the volume of information that is available about them and their interaction with their environment. Determining relevance of individual pieces of data is impossible unless the data are viewed within a certain context or framework. Such a framework suppresses the irrelevant and highlights the relevant.

To be an efficient encoding for what is relevant about a system, the measures must be congruent to the application. The application is as important to "performance" as the computer itself: while it is the computer that is being assessed, it is the application that dictates the scale of values used to assess it.

If the performance measure is congruent to the application, that is to say, if it is a language natural to the application, then specifications can be written concisely and without contortion. This not only permits one to write specifications economically: it is important in the attempt to write—and check for—*correct* specifications. The simpler a set of specifications, the more likely it is in general to be correct and internally consistent.

R2: Performance measures must, by definition, quantify the goodness of computer systems in a given application or class of applications. It follows from this that they should permit the ranking of computers for the same application. It should be emphasized that the ranking must always depend on the application for the reasons given above.

R3: The more complex a system, the more difficult it is to optimize or tune its structure. There are numerous side-effects of even simple actions—of changing the number of buses, for example. So, intuition applied to more and more complex computers becomes less and less dependable as an optimization technique. Multiprocessors are among the most complex computers known today. They provide, due to their complexity, a wealth of configurations of varying quality; this complexity can be used to advantage or ignored with danger.

Multiprocessors that adapt or *reconfigure* themselves—by changing their structure, for example—to their current environment (current job mix, expected time-to-go in a mission-oriented system, etc.) to enhance productivity are likely to become feasible soon. All the impressive sophistication of such a reconfigurable system will come to naught if good, application-sensitive, optimization criteria are unavailable.

R4: A performance measure that is impossible to derive is of no use to anyone. To be acceptable, a performance measure should hold out some prospect of being estimated reasonably accurately. What constitutes "reasonably accurate" depends, naturally enough, on the purpose for which the performance characterization is being carried out. Sometimes, when the requirements are too stringent—extremely low failure probability, for example—to be validated to the required level of accuracy, it is difficult to decide which, if any, is to blame: the performance measure itself, or the mathematical tools used to determine it.

III. Expressing Application Requirements

A computer must be judged in the context of its application: without such a context, all performance evaluation is meaningless. The computer designer cannot be expected to know very much about the application, so the application requirements must be abstracted through a suitable language. If this is done correctly, the designer needs to know nothing about the controlled process: all the information he needs is available through the formally expressed application requirements. Ideally, this should take the form of a scalar function of measurable attributes of the computer system. We look in this section at three methods of doing this: the methods of *linear combination* [23], of *performability* [33], and of *cost functions* [38].

A. Linear Combination

One straightforward scalar function is a linear combination of measurable attributes of a computer. Thus, if $a_1, a_2, \cdots, a_N$ were the weights (indicators or relative or absolute importance to the controlled process of the corresponding attributes) allocated to attributes $x_1, x_2, \cdots, x_N$, respectively, of the computer, its overall performance in that context is $\Sigma_{i=1}^{N} a_i x_i$.

This, of course, raises the issue of obtaining the weights $\{a_i\}$. One way of doing this is the following. While the weights themselves may be hard to obtain objectively, there is less subjectivity involved in ranking the attributes according to importance. Better still, one can define attributes as being made up of subattributes, and rank the subattributes in the context of the application. When this is done, number the ranked items starting with the one of lowest priority. The weight of each attribute is given by the mean (or some other function) of the number associated with each of its subattributes. A list of recommended attributes is provided by Gonzalez and Jordan [23]. Alternatively, one might consider the subattributes themselves as attributes.

This approach seems open to several objections, the most serious of which is that the attributes are usually *mutually dependent;* this affects any sensitivity analysis that uses this technique.

B. Performability

The basic idea behind performability is that the payoff from the application is the only relevant measure of the computer's effectiveness. In other words, computer A is better than computer B *in the context of a specific application* if and only if A results in a greater payoff than does B.

The payoff is defined to be the user's view of how well the application system (in which the computer is embedded) performs. There may be more than one class of user, and therefore more than one scale of values. In such an event, the perceived worth of the computer will depend on the class of user. The user's world-view, therefore, imposes a set A of *accomplishment levels.* The set A may be finite or infinite, discrete or continuous. The *performability* of the controlled process (or any other application system, for that matter) is then defined as the vector function function $p_S(A) = [p_S(a_1), \cdots, p_S(a_N)]$, where

$$p_s(a) = \text{Prob } \{\text{Application system performs at accomplishment level } a \in A\}.$$

All that remains to do is to connect the behavior of the computer to the consequent behavior of the application system. This is done by creating a Markov model of the computer system and linking each state trajectory in that model with an accomplishment level.

Detailed case-studies involving the use of performability can be found in [17], [34]. As an example, [34] lists the following accomplishment set to define the performance levels for a computer controlling an aircraft.

a_0: No economic penalties, no operational penalties, no change in mission profile, and no fatalities.

a_1: Economic penalties, no operational penalties, no change in mission profile, and no fatalities.

a_2: Operational penalties, no change in mission profile, and no fatalities.

a_3: Change in mission profile and no fatalities.

a_4: Fatalities.

The performability of the computer system would then be the probability that the computer's performance made the aircraft perform at each of these accomplishment levels.

C. Cost Functions

The cost function approach [28] focuses attention on the computer response time for the various computational tasks. *Response time* is the interval between the moment that a task is triggered and the moment the result is put out at an actuator or display.

Computer response time is a good parameter to focus on because a control computer is in the feedback loop of the controlled system (see Fig. 1), so that response time constitutes feedback delay. As the feedback delay increases, so the controlled system becomes less stable. Beyond a certain value of response time, instability sets in, and disaster could follow. Even if the system is not driven to instability, its quality of performance degrades with an increase in computer response time.

Also, the response time of a computer strongly depends on the computer architecture, the operating system, the failure handling mechanism, and the software, so that it is possible, in theory, to use a function of response time as an optimization criterion to tune computer structure, etc.

The central idea is as follows. Let us suppose that some performance index is available for the controlled system. This is not as limiting an assumption as might at first appear because controlled systems have performance indexes such as energy, time, fuel, etc., that are physically relevant and are widely accepted as being objective expressions of performance [27], [37]. The cost associated with a response time of ξ for computer task i is defined by

$$g_i(x, \xi) = \Omega_i(x, \xi) - \Omega_i(x, 0)$$

where $\Omega_i(x, \eta)$ is the contribution of task i to the performance index if the response time for that task is η and the controlled system is in state x. That is, $g_i(x, \eta)$ is the incremental portion in the performance index due to the control computer delay ξ. It is assumed that if ξ is so large as to cause catastrophic failure of the controlled process, the corresponding cost is infinite.

If it is not possible to decouple the contributions of individual tasks to the performance index of the controlled system, then joint cost functions may be analogously defined. These are conceptually similar to joint probability distributions.

Measures based on the cost functions can now be defined. For instance, the *probability of dynamic failure*, p_{dyn}, is the probability that the computer causes the controlled process to fail catastrophically. The *mean cost* is the expected value of the cost accrued over a mission, given that the controlled process does not fail. For an example of obtaining cost functions that apply when an aircraft is about to land, see [38].

Performability and cost functions have a forgotten precursor. In 1960, Drenick [14] proposed a set of measures of "generalized reliability": the *mission success ratio* (the fraction of missions that end successfully), and the *mission survival probability.*

Somewhat related to performability and cost functions measures is *dependability* [10], [31]. Dependability is there defined as the "trustworthiness and continuity of computer system service such that reliance can justifiably be placed on this service." This measure is not very useful because it begs the question as to what constitutes "reliance" on service. In [31], having defined the measure, the author goes on to consider reliability and availability as instances of dependability.

D. Discussion

This section has dealt with methods whereby requirements R1, R2, and R3 may be satisfied. The method of linear combinations and that of performability are general: they can, in theory, be applied to general-purpose systems, such as computer centers, as well as to control computers. By contrast, the method of cost functions limits itself to control computers. Indeed, this latter method might be viewed as a special case of performability, with the mean cost and the probability of dynamic failure, for instance, defining the set of accomplishment levels for the system.

As remarked earlier, the purpose of expressing application requirements in a precise, formal manner is to create a clean interface between the application and the control computer. Take, for instance, the cost functions. Once they have been defined for a particular application, they can be used to evaluate computers without very much reference to the controlled process. They can be used, together with information about the control software (e.g., the run times), to optimally schedule jobs, derive methods to shed computational load when failures have reduced computational capacity, and to obtain strategies for recovery in the event that portions of the computer fail [29].

Such formal approaches are useful for another reason. It provides a template that reduces the probability of overlooking something important. The amount of detail that is required to express application requirements formally imposes a salutary discipline upon the analyst.

IV. Attributes of Computers

Attributes marked with an asterisk (*) were originally defined for single-processor computers. They, therefore, assume that the computer is either "up" or "down." There is exactly one "up" state in such cases.

A. Reliability Attributes

1)Interval Reliability, $R(\alpha, T)$ [4]:* This is the probability that the computer continues to operate over an interval $[\alpha, \alpha + T]$, under the assumption that it is operational at α.

2) Strategic Reliability, $SR(T)$ [45]:* The strategic reliability of a computer is $SR(T) = \lim_{\alpha\to\infty} R(\alpha, T)$. This can be regarded as a "steady-state" reliability. It is a function of the frequency with which preventive maintenance is carried out. Truelove shows how to choose the maintenance frequency to optimize strategic reliability [45]. This is the resolution of the following tradeoff. If preventive maintenance is carried out too often, the probability of the system's failing is small, but then the maintenance itself takes so much time that the strategic reliability is reduced. On the other hand, if maintenance is carried out only rarely, the probability of the system's failing becomes appreciable. The optimal frequency of preventive maintenance is a function of the time taken to carry it out and of the failure characteristics of the computer.

3) Job-Related Reliability, $R_{job}(t, J)$ [35]: $R_{job}(t, J)$ is the probability that the computer has, at time t, enough hardware resources to complete job J satisfactorily. A similar measure is called *computational reliability* by Beaudry [6].

4) Capacity Reliability [19]: This is the probability that the system (presumably expressed in a Markov model) remains out of a certain set of states throughout the "interval of interest" (e.g., a mission). Huslende's term for a very similar measure is *performance-reliability* [25]. Related measures include the *expected capacity survival time,* which is the expected time for the capacity reliability to drop to a specific value, and the *expected capacity reduction time,* which is the expected time for the system to move from one set of states to another. Somewhat similar to these is another measure introduced by Beaudry: the *mean computation before failure* [6], whose meaning is obvious. Yet another similar measure, *pseudoreliability* is proposed in [42]: this is given by $\Sigma_{i=1}^{n} p_i\gamma_i$, where p_i is the probability of the computer being in state i and γ_i the "relative performance" of the system at that state.

5) Mean Time Between Failure [36]: The average time between two successive failures of the item under study. One can also define the *mission time between critical failures,* which is self-explanatory.

B. Availability Attributes

1) Pointwise Availability, $A_p(t)$ [24]:* This is the probability that the system will be operating "within tolerance limits" at time t.

2) Interval Availability, $A_i(a, b)$ [24]:* This is the expected fraction of the interval $[a, b]$ that the system will be operating "within tolerance limits." The *limiting interval availability* is defined in [5] as $\lim_{t\to\infty} A_i(0, t)$.

3) Performance Availability [11]: Identical to the *pseudoreliability* of Tillman, Lie, and Hwang [42] (see above), except that γ_i is now the availability of the system at state i.

C. Maintenance Attributes

1) Mean Time between Maintenance [36]: The average time between successive maintenance actions. This can be specialized to measure the mean time between successive maintenance actions of a specific character or type.

2) Mean Maintenance Time [36]: The average length of a maintenance job. This can be expressed either in the time taken to complete maintenance, or the total man-hours expended in doing so.

3) Maintenance Ratio [36]: The ratio of maintenance man-hours (or other measure of maintenance effort) to the lifetime of the system being maintained.

D. Other Attributes

1) Throughput: The average number of instructions that the system is capable of processing per unit time. The instructions are drawn from a standard instruction mix.

2) Response Time: The time that elapses between a job commencement and termination.

E. Discussion

The attributes of this section must satisfy requirement R4. In other words, if the specifications call (to take one well-known case) for a failure probability of less than 10^{-9}, there must be some way of verifying that a system meets—or does not meet—such a standard.

All the attributes in this section can be experimentally determined, in principle. However, when the requirements are as stringent as they are for critical systems, experiments would take too long. (It is an amusing and instructive exercise in elementary statistics to calculate the duration of an experiment that must validate an extremely stringent performance/reliability specification. Frequently, one obtains durations longer than the estimated age of the universe—of the order of 10^{10} years.)

One, therefore, has to fall back on decomposition. While a system may not be tested in its entirety, it can be broken down into subsystems which lend themselves quite readily to testing. A model can then be created to obtain performance values for the whole system as a function of the experimental results for its subsystems.

Perhaps the best known model is CARE-III, developed under contract to NASA [40], [44]. Other models include the complementary analytic-simulative technique (CAST) [13], and the hybrid automated reliability predictor [20]. A good survey of such models appears in [20] (see also [43]), and so we do not discuss any of them here.

V. Conclusion

In this paper, we have surveyed the state of the art in performance measures. Much remains to be done, especially in predicting the reliability of critical systems.

Performance measures represent the link between the real world of control computers and controlled processes and the abstract world of the analyst. It is usually possible—albeit with a great deal of effort—to satisfy oneself that an abstract mathematical model of a system is correct, i.e., it has no *internal* contradictions that would invalidate it. It is *never* possible to prove that a performance measure is correct. All one can have is faith, backed up by considerable experience. All the formal techniques for expressing the needs of the application must be regarded as no more than aids to design and evaluation. They are no substitute for human care, intuition, and experience.

Acknowledgment

The authors would like to thank H. Hecht for providing them the source of [36].

References

[1] M. Athans and P. L. Falb, *Optimal Control: An Introduction to the Theory and its Applications.* New York: McGraw-Hill, 1966.

[2] D. M. Auslander, Y. Takahashi, and M. Rabins, *Introducing Systems and Control.* New York: McGraw-Hill, 1974.

[3] R. K. Barasia and T. D. Kiang, "Development of a life-cycle management cost model," in *Proc. IEEE Reliability and Maintainability Symp.,* 1978, pp. 254-259.

[4] R. E. Barlow and L. C. Hunter, "Reliability of a one-unit system," *Oper. Research,* vol. 9, no. 2, pp. 200-208, 1961.

[5] R. E. Barlow and F. Proschan, *Mathematical Theory of Reliability.* New York: Wiley, 1975.

[6] M. D. Beaudry, "Performance-related reliability measures for computing systems," *IEEE Trans. Comput.,* vol. C-29, no. 6, pp. 501-509, 1978.

[7] N. Benwell, Ed., *Benchmarking: Computer Evaluation and Measurement.* Washington, DC: Hemisphere, 1975.

[8] W. Bucholz, "A synthetic job for measuring system performance," *IBM Syst. J.,* vol. 8, no. 4, pp. 309-318, 1969.

[9] "Canadian Forces Procedure CFP113 life management system—Guidance manual," Dept. National Defense, Canada, 1979.

[10] W. C. Carter, "A time for reflection," in *Proc. 8th Int. Symp. Fault-Tolerant Comput.,* 1978, p. 41.

[11] T. C. K. Chou and J. A. Abraham, "Performance/availability model of shared resource multiprocessors," *IEEE Trans. Reliability,* vol. R-29, no. 1, pp. 70-76, 1980.

[12] E. Çinlar, *Introduction to Stochastic Processes.* Englewood Cliffs, NJ: Prentice-Hall, 1975.

[13] R. Conn, P. Merryman, and K. Whitelaw, "CAST—A complementary analytic-simulative technique for modeling complex, fault-tolerant computing systems," in *Proc. AIAA Comput. Aerospace Conf.,* 1977.

[14] R. F. Drenick, "Mathematical aspects of the reliability problem," *J. SIAM,* vol. 8, no. 4, pp. 680-690, 1960.

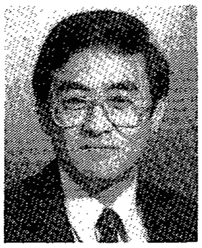
[15] M. E. Drummond, "A perspective on systems performance evaluation," *IBM Syst. J.,* vol. 8, no. 4, pp. 252-263, 1969.

[16] M. S. Fox, "An organizational view of distributed systems," *IEEE Trans. Syst., Man, Cybern.,* vol. 11, no. 1, pp. 70-80, 1981.

[17] D. G. Furchtgott, "Performability models and solutions," Ph.D. dissertation, The University of Michigan, Ann Arbor, MI, 1984.
[18] J. W. Gault, K. S. Trivedi, and J. B. Clary, Eds., "Validation methods research for fault-tolerant computing systems—Working group meeting II," NASA, Rep. CP-2130, 1980.
[19] F. A. Gay and M. L. Ketelson, "Performance evaluation for gracefully degrading systems," in *Proc. 7th Int. Symp. Fault-Tolerant Comput.*, 1979, pp. 51-58.
[20] R. M. Geist and K. S. Trivedi, "Ultrahigh reliability prediction for fault-tolerant computer systems," *IEEE Trans. Comput.*, vol. C-32, no. 12, pp. 1118-1127, Dec. 1983.
[21] J. Gertler and J. Sedlack, "Software for process control—A survey," *Automatica*, vol. 2, no. 6, pp. 613-625, 1975.
[22] B. V. Gnedenko, Yu. K. Belyayev, and A. D. Solovyev, *Mathematical Methods of Reliability Theory.* New York: Academic, 1969.
[23] M. J. Gonzalez and B. W. Jordan, "Evaluation of distributed processor systems," *IEEE Trans. Comput.*, vol. C-29, no. 12, pp. 1087-1094, 1980.
[24] J. E. Hosford, "Measures of dependability," *Oper. Research*, vol. 8, no. 1, pp. 53-64, 1960.
[25] R. Huslende, "A combined evaluation of performance and reliability for degradable systems," in *Proc. ACM SIGMETRICS Conf.*, 1981, pp. 157-164.
[26] R. E. Kalman, "A new approach to linear filtering and prediction problems," *ASME Trans.*, vol. 82D, 1960.
[27] D. E. Kirk, *Optimal Control Theory.* New York: Academic, 1970.
[28] C. M. Krishna and K. G. Shin, "Performance measures for multiprocessor controllers," *Performance '83.* A. K. Agrawala and S. K. Tripathi, Eds. Amsterdam, The Netherlands: North-Holland, 1983, pp. 229-250.
[29] ——, "On scheduling tasks with a quick recovery from failure," *IEEE Trans. Comput.*, vol. C-35, no. 5, 1986, pp. 448-457.
[30] B. C. Kuo, *Automatic Control Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1984.
[31] J. C. Laprie, "Dependability evaluation of software systems in operation," *IEEE Trans. Software Eng.*, vol. SE-10, no. 6, Nov. 1984.
[32] H. C. Lucas, "Performance evaluation and monitoring," *Comput. Surveys*, vol. 3, no. 3, pp. 79-91, 1971.
[33] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Trans. Comput.*, vol. C-29, no. 8, pp. 720-731, 1980.
[34] J. F. Meyer, D. G. Furchtgott, and L. T. Wu, "Performability evaluation of the SIFT computer," *IEEE Trans. Comput.*, vol. C-29, no. 6, pp. 501-509, 1980.
[35] H. Mine and K. Hatayama, "Performance-related reliability measures for computing systems," in *Proc. 7th Int. Symp. Fault-Tolerant Comput.*, 1979, pp. 59-62.
[36] "Definitions of terms for reliability and maintainability," U.S. Government Printing Office, MIL-STD-721C, 1981.
[37] A. P. Sage, *Optimum Systems Control.* Englewood Cliffs, NJ: Prentice-Hall, 1970.
[38] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaluating real-time computer controllers and their application," *IEEE Trans. Automat. Contr.*, vol. AC-30, no. 4, pp. 357-366, 1985.
[39] N. Statland, "Methods of evaluating computer systems performance," *Comput. Automat.*, vol. 13, no. 2, pp. 18-23, 1964.
[40] J. J. Stiffler and L. A. Bryant, "CARE-III phase report—Mathematical description," NASA Rep. 3566, 1982.
[41] "Life-cycle cost analyzer (LCCA) general information guide," Tech. Analytical Sciences Corp., Eng. Memo. E.M.-2184-1, June 1981.
[42] F. A. Tillman, C. H. Lie, and C. L. Hwang, "Analysis of pseudoreliability of a combat tank and its optimal design," *IEEE Trans. Reliability*, vol. R-25, no. 3, pp. 239-242, 1976.
[43] K. S. Trivedi, J. W. Gault, and J. B. Clary, "The validation of system reliability in life-critical systems," in *Proc. Pathways to Syst. Integrity Symp.*, National Bureau of Standards, 1980.
[44] K. S. Trivedi and R. M. Geist, "A tutorial on the CARE-III approach to reliability modelling," NASA Rep. 3488, 1981.
[45] A. J. Truelove, "Strategic reliability and preventive maintenance," *Oper. Research*, vol. 9, no. 1, pp. 22-29, 1961.

C. M. Krishna received the B. Tech. degree from the Indian Institute of Technology, Delhi, India, in 1979, the M.S. degree from Rensselaer Polytechnic Institute, Troy, NY, in 1980, and the Ph.D. degree from the University of Michigan, Ann Arbor, in 1984, all in electrical engineering.

Since September 1984, he has been on the faculty of the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst. He was a Visiting Scientist at the IBM T. J. Watson Research Center during the summer of 1986. His research interests include reliability modeling, queueing and scheduling theory, and distributed architectures and operating systems.



Kang G. Shin (S'75-M'78-SM'83) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and both the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

From 1970 to 1972 he served in the Korean Army as an ROTC officer and from 1972 to 1974 he was on the Research Staff of the Korea Institute of Science and Technology, Seoul, Korea, working on the design of VHF/UHF communication systems. From 1978 to 1982 he was an Assistant Professor at Rensselaer Polytechnic Institute, Troy, NY. He was also a Visiting Scientist at the U.S. Air Force Flight Dynamics Laboratory in Summer 1979 and at Bell Laboratories, Holmdel, NJ, in Summer 1980. Since September 1982, he has been with the Department of Electrical Engineering and Computer Science at The University of Michigan, Ann Arbor, where he is currently a Professor. He has been very active and authored/coauthored over 100 technical papers in the areas of distributed fault-tolerant real-time computing, computer architecture, and robotics and automation. As an initial phase of validation of architectures and analytic results, he and his students are currently building a 19 node hexagonal mesh at the Real-Time Computing Laboratory, The University of Michigan.

Professor Shin is a member of ACM, Sigma Xi, and Phi Kappa Phi. He was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium and served as the Guest Editor of the Special Issue on Real-Time Systems which is scheduled to appear in the August 1987 issue of the IEEE TRANSACTIONS ON COMPUTERS.