Box structured information systems

by H. D. Mills
R. C. Linger
A. R. Hevner

The box structure methodology for information systems development is based on a usage hierarchy of data abstractions, in which each abstraction is defined in three distinct forms, called its* black box, *its* state machine, *and its* clear box. *Each of these three box structures defines identical external behavior, but with increasing internal visibility, to provide a hierarchical structure which supports the systems development principles of referential transparency, transaction closure, state migration, and common services. This hierarchy of box structures provides, in turn, a basis for orderly management of information systems development by a finite set of analysis and design tasks in a spiral development process. The methodology and its use are described.

Since their inception, information systems have been used in government and business, but research and development in information systems have increased dramatically since the advent of the computer some thirty years ago. As a result, a recognizable discipline of Information Systems is emerging in business and in university curricula. However, Information Systems is still a young field in terms of intellectual growth and development. Even with all the current excitement and progress, there is still a lot to discover. The search for fundamental ideas and deep simplicities takes time.

Structures and data flows. The revolution that changed trial-and-error computer programming into software engineering was triggered by Dijkstra's idea of structured programming.[1] Structured programming cleared a control flow jungle that had grown unchecked for twenty years in dealing with more and more complex software problems. It replaced that control flow jungle with the astonishing assertion that software of any complexity whatsoever could be designed with just three basic control structures—sequence (begin-end), alternation (if-then-else), and iteration (while-do)—which could be nested over and over in a hierarchical structure (the structure of structured programming). The benefits of structured programming to the management of large projects are immediate. The work can be structured and progress measured in a top-down development in a direct way.

Even so, information systems development is much more than software development. The operations of a business involve all kinds of data that are transmitted, stored, and processed in all kinds of ways. The total data processing of a business is defined by the activities of all of its people and computers, as they interact with one other and with customer, vendor, and government personnel and computers outside the business. In a large company, it is a massively parallel operation with many thousands of interactions going on simultaneously. Information systems are called on to automate more and more of the information processing in business—in many cases these systems are required for survival in a competitive environment. And for these systems, a complete description of their data operations and uses leads to a data flow jungle that is even more tangled and arcane than the control flow jungle of software.

We will replace that data flow jungle with just three system structures that can be nested over and over in a hierarchical structure (the structure of box structures). Any information system—automatic, manual, or hybrid—can be described or designed in a hierarchy of these system structures step by step in a provable way. The benefits of box structures to the management of large projects are also immediate. The work can be structured and progress measured in top-down system development in a direct way.

Software counterparts of state machines have been called data abstractions.

State machines and data abstractions. The origins of these system structures are in the hierarchical state machine methodology of software engineering found in References 2, 3, and 4 and taught at the IBM Software Engineering Institute.[5,6] As discussed in the book by Mills, Linger, and Hevner,[7] this methodology was used in the New York Times Information Bank, as reported by Baker,[8,9] with remarkable results in reliability.[9] A very large-scale use of this methodology in the modernization of U.S. Air Force satellite tracking and control systems has been reported by Jordano.[10]

The software counterparts of state machines have also been called data abstractions,[11,12] and more recently, software objects.[13] Their common feature is the presence of a state, represented in stored data, and accessed and altered by procedures that collectively define the state machine transition function. Since these data are accessed and altered by reusing the data abstraction or object, the hierarchy is a usage hierarchy, in the sense found in Parnas,[14] rather than a parts hierarchy. That is, data abstractions appear in the hierarchy at each occasion of use in the design, rather than as a part in the design.

This usage hierarchy of data abstractions cuts a Gordian knot for the effective dual decomposition of data flows and processes in information systems. Data flows are convenient heuristic starting points in information systems analysis, as developed in References 15 through 18, but require a mental discontinuity to move to information systems design. The problem is that data flows describe all that can possibly happen, whereas processes must deal with one data instance at a time and prescribe precisely what will happen at each such instance. Each use of a data abstraction is an instance of data flow through a process, which provides for storage in its state as well. And the collective effects of the usage of the data abstraction throughout a hierarchy are summarized by a data flow through the process. Data abstractions have proved useful in software engineering in several specific languages and systems, as in CLU,[19] VDM,[20,21] HDM,[22] Larch,[23] and object-oriented design.[13]

Box structures and data abstractions. The box structure methodology develops the usage hierarchy of data abstractions in a way especially suited for information systems development, in which the emphasis is jointly on mathematical rigor and management simplicity.[7] For this purpose, we not only need strong system development principles, but must also make these principles obvious in the methodology. We define three distinct forms for any data abstraction, namely, its *black box*, its *state machine*, and its *clear box*. A black box defines a data abstraction entirely in terms of external behavior, in transitions from stimuli to responses. A state machine defines a data abstraction in terms of transitions from a stimulus and internal state to a response and new internal state. A clear box defines a data abstraction in terms of a procedure that accesses the internal state and possibly calls on other black boxes. This recursion of black boxes with clear boxes that call on other black boxes defines a usage hierarchy that supports important principles for system development.

In the next section of this paper we summarize the principal concepts of the box structure methodology and explain their mathematical foundations. In the subsequent section, box structure hierarchies are defined, and the system development principles of referential transparency, state migration, transaction closure, and common services are described. Finally, we discuss the benefits of these structures in managing a spiral system development process.

Box structures

The behavior of any information system (or subsystem) can be rigorously described in three distinct

box structure forms previously mentioned—the black box, state machine, and clear box of the system. We first define each of these structures and then show relationships among them.

Black box behavior. The *black box* gives an external view of a system or subsystem that accepts stimuli, and for each stimulus, S, produces a response, R (which may be null), before accepting the next stimulus. A diagram of a black box is shown in Figure 1. The system of the diagram could be a hand calculator, a personal computer, an accounts receivable system, or even a manual work procedure that accepts stimuli from the environment and produces responses one by one. As the name implies, a black box description of a system omits all details of internal structure and operations and deals solely with the behavior that is visible to its user in terms of stimuli and responses. Any black box response is uniquely determined by its stimulus history.

For example, an interactive workstation is a computer system that accepts keystrokes, one by one, and returns a new screen with each keystroke. Most keystrokes change the screen in small ways, say, by adding or deleting a character, but some keystrokes bring up entirely new screens, say, by an enter key or a menu choice. Each such keystroke is a stimulus for the black box. The user need have no idea of the internal structure—that some screens are created locally, some indirectly by remote computers, etc. The workstation behaves as a black box for the user.

The idea of describing a system as a black box is useful for analyzing the system from the user's point of view. Only system externals are visible; no system state or procedure is described. The mathematical semantics of black box behavior is a function from system stimulus histories to system responses. A black box is specified by its traces.[24,25] In fact, Parnas uses the term *black box* to motivate the study of traces.[24]

State machine behavior. The *state machine* gives an intermediate system view that defines an internal system state, namely an abstraction of the data stored from stimulus to stimulus. It can be established mathematically that every system described by a black box has a state machine description. (Consider each stimulus history to be a state.) A state machine diagram is shown in Figure 2. The state machine part called Machine is a black box that accepts as its stimulus both the external stimulus and the internal state and produces as a response both the external response and a new internal state which replaces the old state. The role of the state machine is to open up the black box description of a system one step by making its state visible. State machine behavior can be described in the transition formula

(Stimulus, Old State) → (Response, New State)

Much of the work in formal specification methods for software applies directly to specification of the state machine system view. These methods, such as those presented in the literature,[11,12,26,27] specify the

Figure 1 A black box diagram

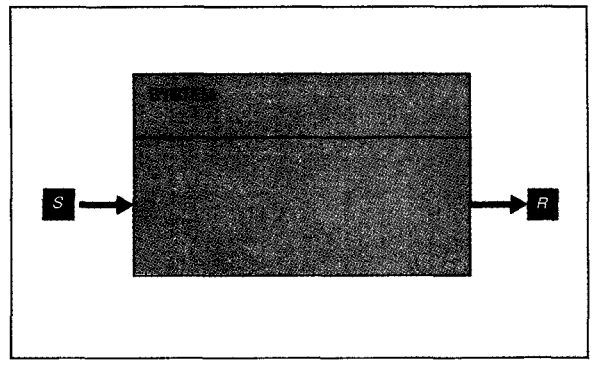


Figure 2 A state machine diagram

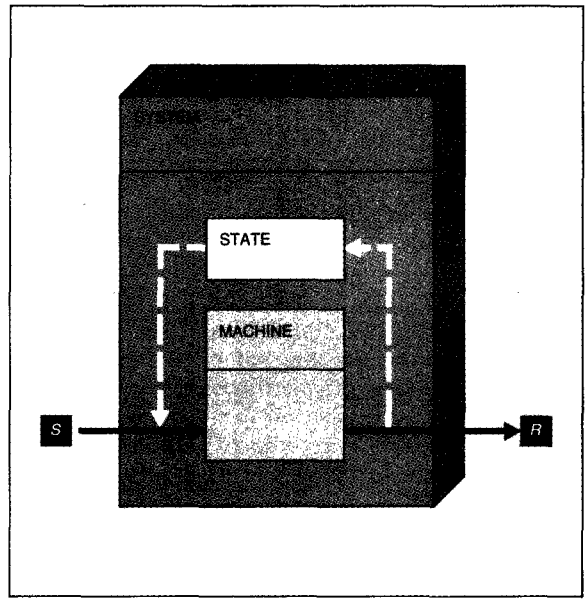
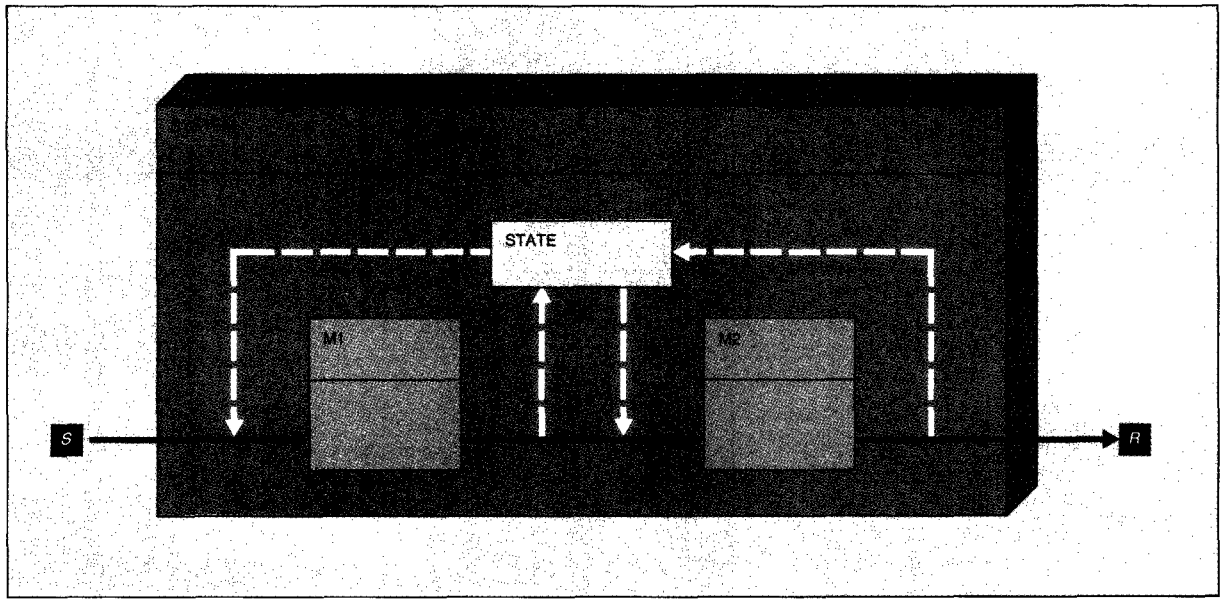


Figure 3 The clear box sequence structure



required properties of programs and abstract data types in axiomatic and algebraic models. The models represent behavior without presenting implementation details.

For information systems, however, we believe that direct descriptions are often sufficient—that indirect axiomatic and algebraic methods of describing data abstractions tend to obscure the essential simplicity of state machines. Also, the conceptual work required to derive axioms or algebras for a complete system state can require deep research itself. (For an example of problems associated with the axiomatization of even a simple data abstraction, see Ferrentino and Mills.[2])

Clear box behavior. The *clear box*, as the name suggests, opens up the state machine description of a system one more step in an internal view that describes the system processing of the stimulus and state. The processing is described in terms of three possible sequential structures, namely sequence, alternation, and iteration, and a concurrent structure. Figure 3 shows a clear box sequence structure with two internal subsystems represented as black boxes; each accepts both a stimulus and a state and produces both a response and a new state. In the sequence structure, the clear box stimulus is the stimulus to black box M1, whose response becomes the stimulus to M2, whose response is the response of the clear box. At this point, a hierarchical, top-down description can be repeated for each of the embedded black boxes at the next lower level of description. Each black box is described by a state machine, then by a clear box containing even smaller black boxes, and so on.

Figures 4, 5, and 6 show, respectively, the alternation, iteration, and concurrent clear box structures. The internal machines, Mi, can be expanded at lower levels of description in a box structure hierarchy. In alternation and iteration clear boxes, the condition C (denoted by a diamond) is a special black box that accesses the stimulus and old state to return responses T or F (True or False). The function of C is to direct the stimulus to the proper black box.

The clear box is an essential step of system description that is lacking in many information systems development methods. It specifies the procedurality that connects the usage of subsystems to be described at the next lower level in the box structure hierarchy. This explicit connection supports the principle of referential transparency, to be discussed in the next section.

Box structure derivation and expansion. The relationships among the black box, state machine, and

Figure 4 The clear box alternation structure

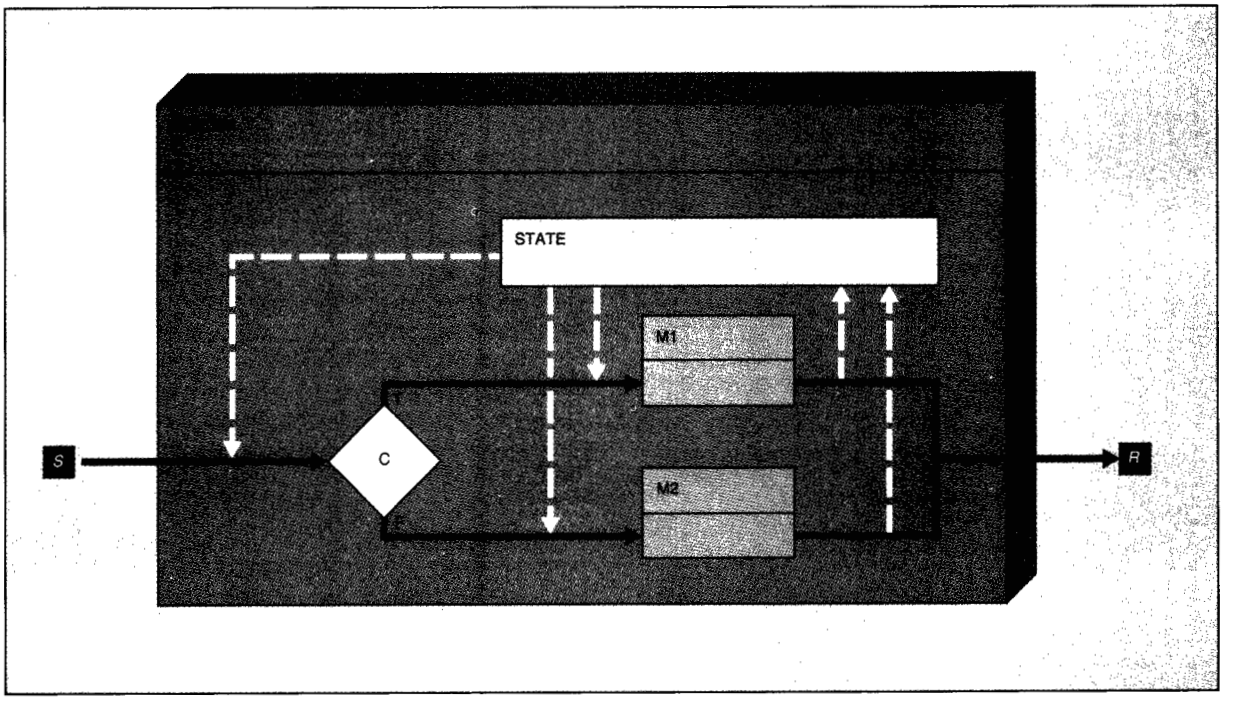


Figure 5 The clear box iteration structure

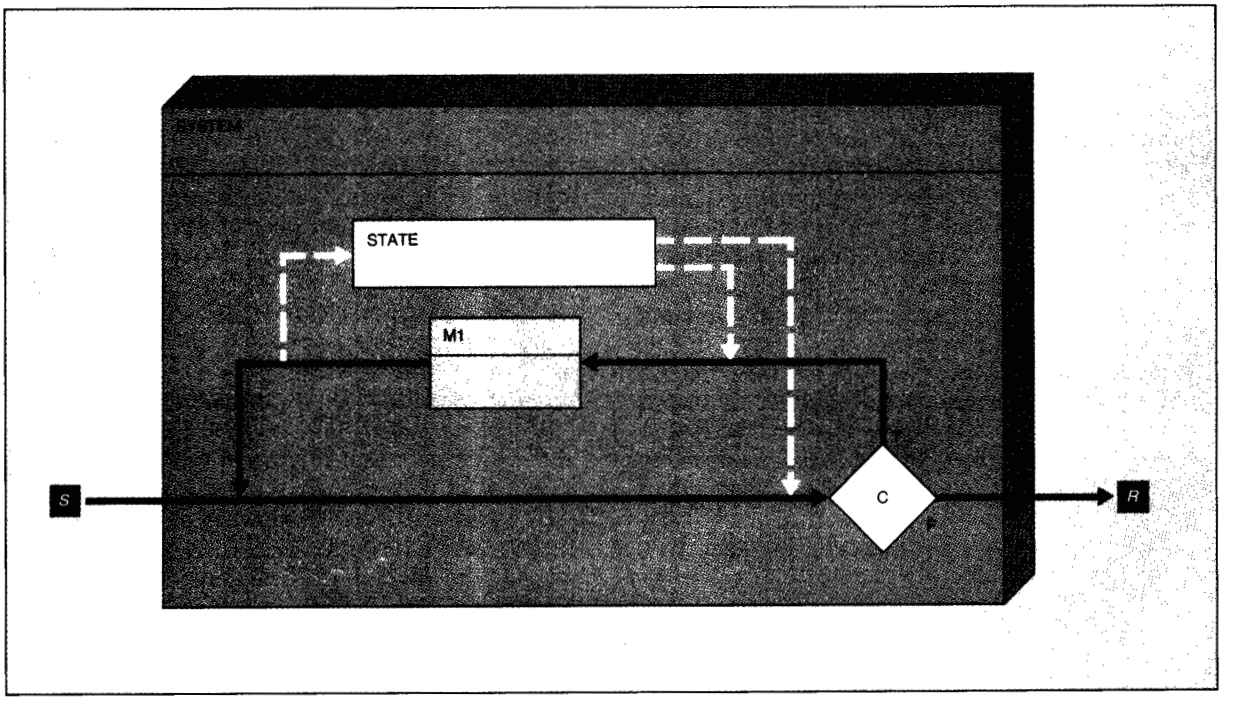
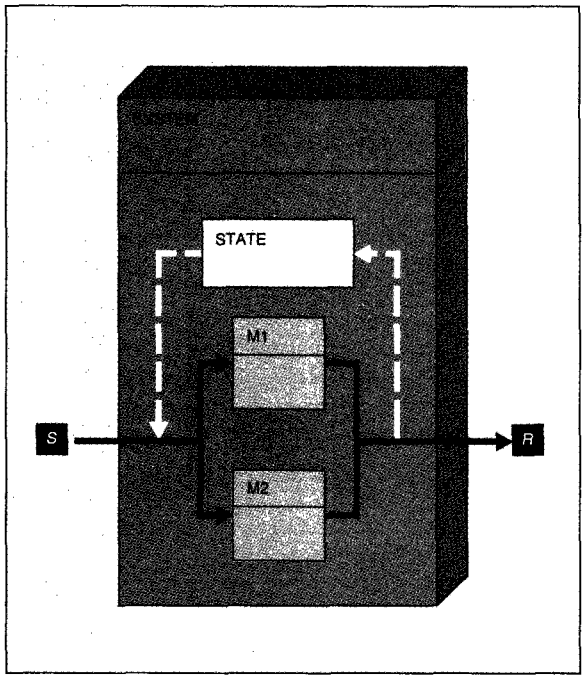


Figure 6 The clear box concurrent structure (shown with two machines)



clear box views of a system or subsystem precisely define the tasks of *derivation* and *expansion*. As shown in Figure 7, it is a derivation task to deduce a black box from a state machine or to deduce a state machine from a clear box, whereas it is an expansion task to induce a state machine from a black box or to induce a clear box from a state machine. That is, a black box derivation from a state machine produces a *state-free* description, and a state machine derivation from a clear box produces a *procedure-free* description. Conversely, a state machine expansion of a black box produces a *state-defined* description, and a clear box expansion of a state machine produces a *procedure-defined* description. The expansion step does not produce a unique product because there are many state machines that behave like a given black box and many clear boxes that behave like a given state machine. The derivation step does produce a unique product because there is only one black box that behaves like a given state machine and only one state machine that behaves like a given clear box.

In summary, black box, state machine, and clear box expansions provide behaviorally equivalent views of an information system or subsystem at increasing levels of internal visibility. This equivalence relationship is depicted in Figure 8.

A box structure illustration. Although the concept of box structures is easy to grasp, its use in actual business systems requires business knowledge. In fact, box structures provide forms in which to describe business knowledge in a standard way. The principal value of a black box is that any business information system or subsystem will behave as a black box whether consciously described as such or not. In turn, any black box can be described as a state machine (actually in many ways), and any state machine can be described as a clear box (also in many ways), possibly using other black boxes. In practice, information systems or subsystems often have their own natural descriptions that can be reformulated as box structures.

As an illustration, a 12-month running average defines a simple, low-level black box that might be used in sales forecasting, for example, for a variety store with 10 000 items. A stimulus of last month's sales of an item produces a response of the past year's average monthly sales of the item. Each month a new sales amount produces a new average of the past 12 months. In the case of new items with less than 12 months of sales history, the response can be the average of sales to date. If i is the age of an item in months, then the number of months to average is $min\ (i,12)$, the minimum of i and 12, no matter how long the sales history is.

Figure 9 shows the Running Average black box where for an item of age i, $S.1 = S$ is last month's sales, $S.2$ is the next previous month's sales, and so on. The symbol ":=" means that the term on the left side (R) is assigned the value of the expression on the right side.

One possible state machine with the same behavior as this black box would store the previous $min\ (i,12)$ monthly sales $S.1, S.2, \cdots$ in state variables $S1, S2, \cdots$, and item age i in state variable I. Then, with each new stimulus S, there is sufficient information to calculate the response and update the state. The state variable I must be initialized, say to 1, and incremented with each stimulus. The state variables $S1, S2, \cdots$ will be initialized as the first 12 months of sales materialize.

Figure 10 shows the corresponding Running Average state machine. The multiple assignments are to be

Figure 7 Box structure derivation and expansion (shown with sequence clear box)

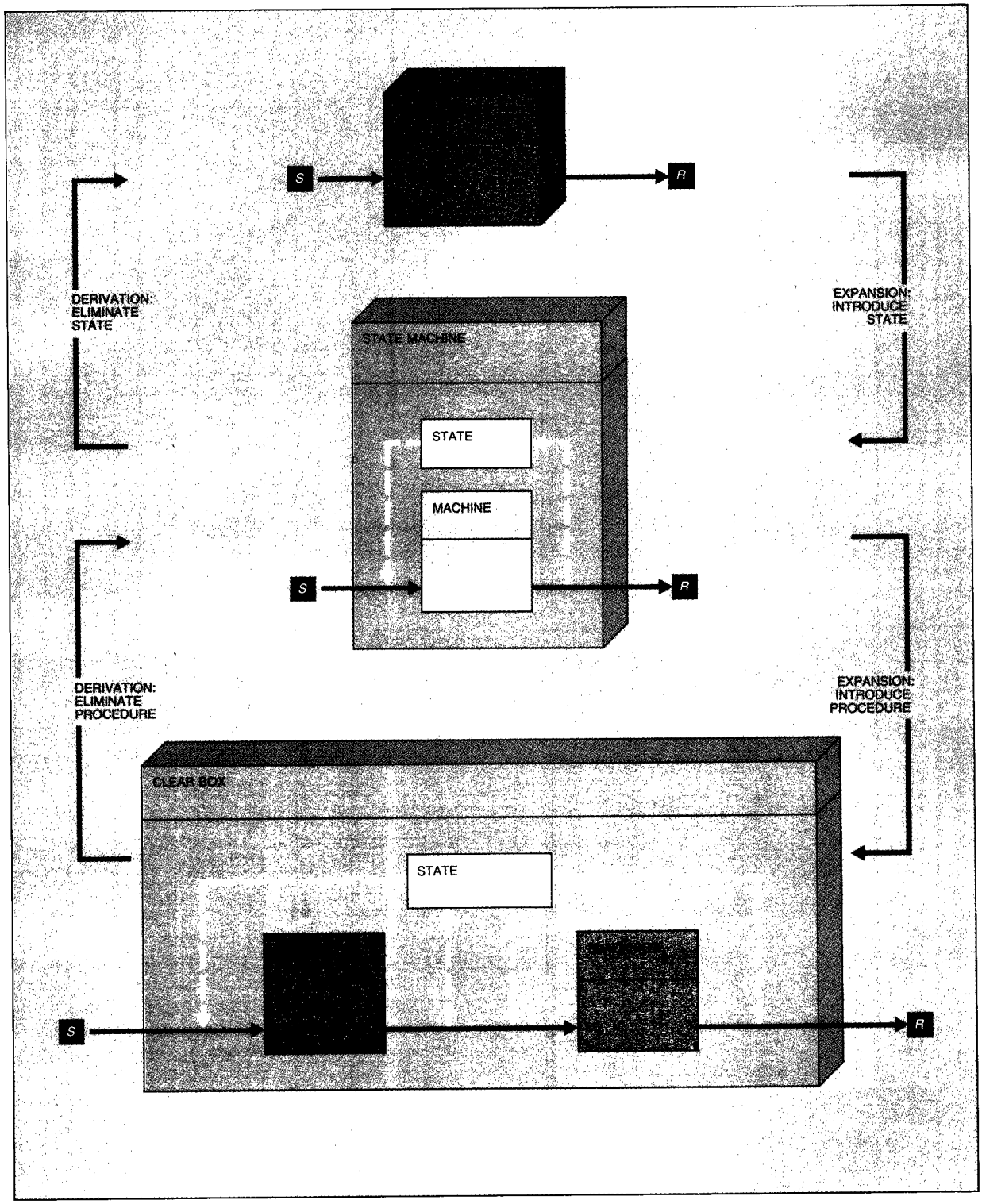


Figure 8 Three behaviorally equivalent views of an information system or subsystem

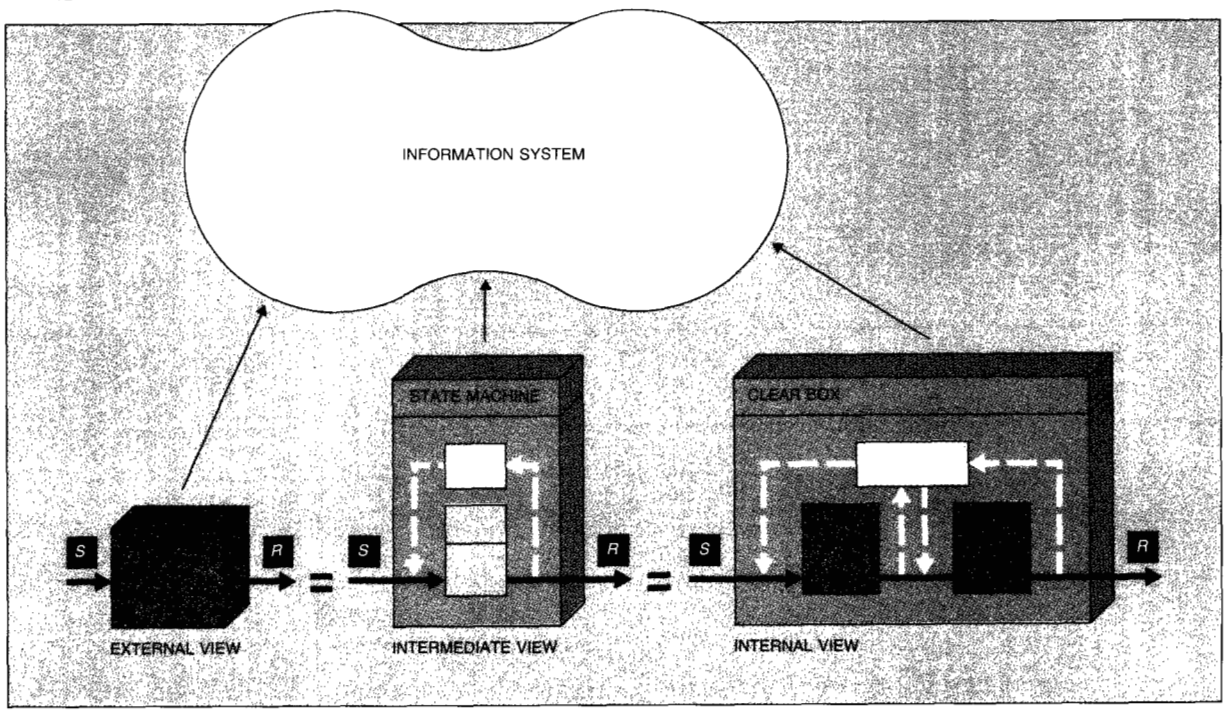
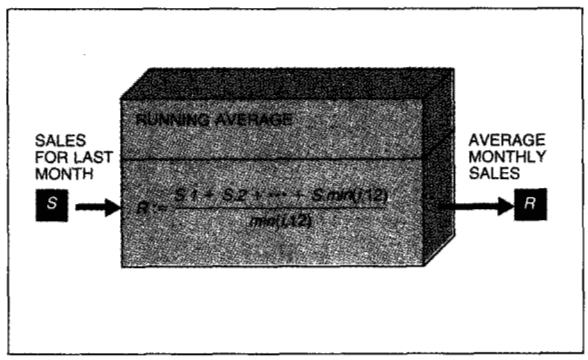


Figure 9 Running average black box



understood as concurrent. That is, all expressions on the right sides use the data available at the beginning of the transition, not data computed in assignments above them.

Note a distinction between $S.1$, $S.2$, $\cdots$, which are monthly sales, and $S1$, $S2$, $\cdots$, which are state variables. The values are the same (at the end of each transition), but unless $S.1$, $S.2$, $\cdots$ are recorded in $S1$, $S2$, $\cdots$, they will be lost to the state machine because it does not access stimulus history, as does the black box. The assignments made to $S2$, $S3$, $\cdots$ before $S1$, $S2$, $\cdots$ are initialized reference undefined values, but do no harm because they are not used in R.

A clear box will describe how the response and new state are computed in a sequential or concurrent structure of other black box uses. One possible design is to first update the sales data, then compute the running average from the new state data and increment the age of the item, as shown in Figure 11. In this case, no further black box expansion will be needed because both black boxes, Update Sales and Find Average, require no more than their last stimuli to compute their response [they can be defined as mathematical functions from stimuli (not stimulus histories) to responses]. Of course, the stimuli on which they operate include the state variables of Running Average.

Note that many other state machine and clear box designs could have been chosen to implement the

Figure 10 Running average state machine

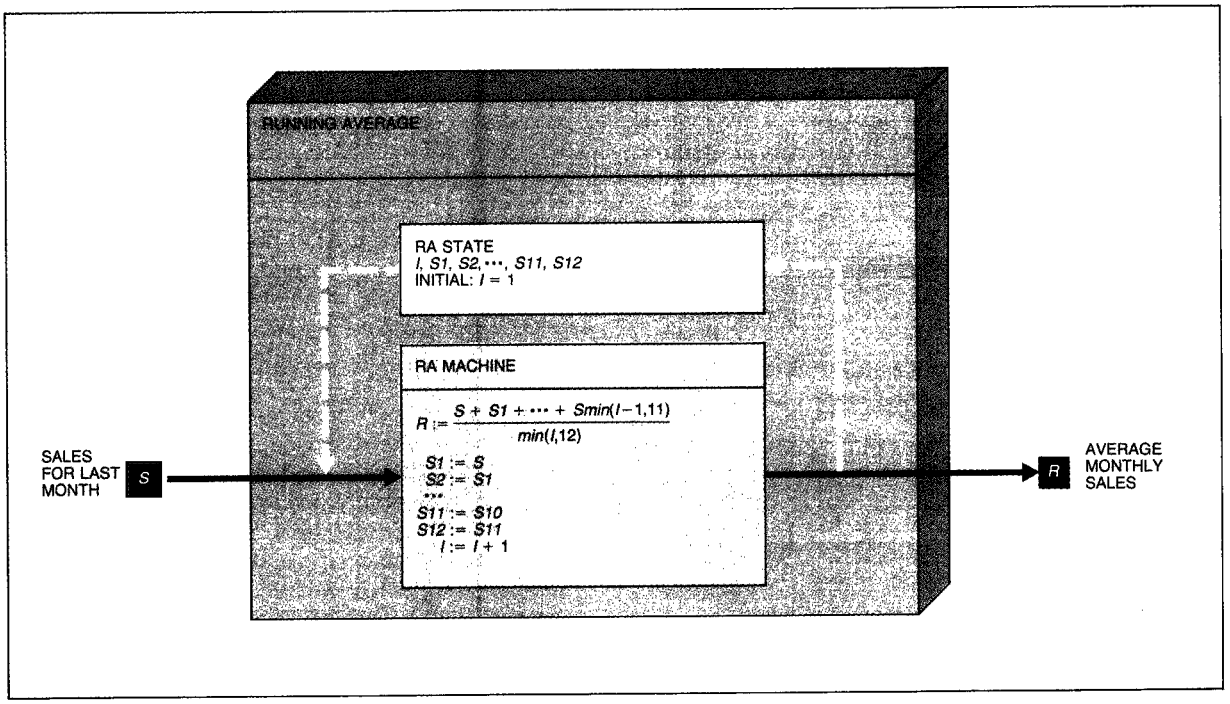
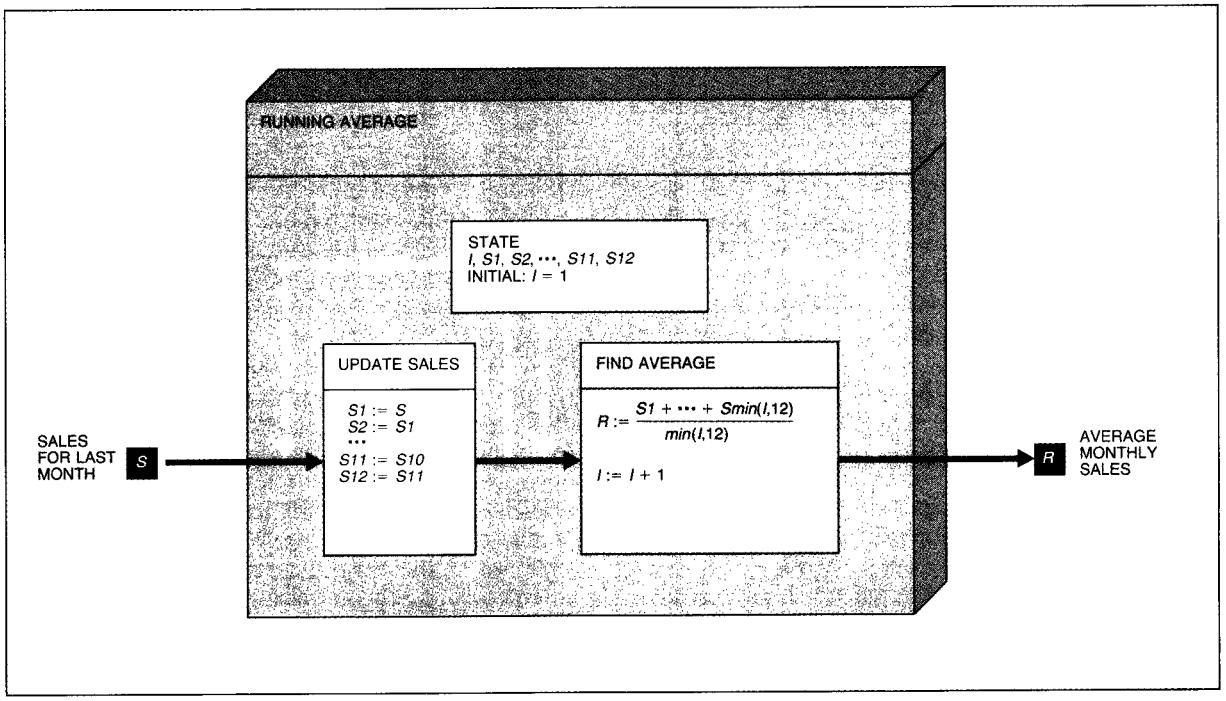


Figure 11 Running average clear box



Running Average black box. For example, after the value of I exceeds 11, the state data could be stored as monthly sales values divided by 12. The running average would then be found by adding all the state data.

A Running Average black box is a simple sales forecaster. However, if sales are seasonal or have definite trends, a more suitable black box may be required. Such a forecaster will differ in details, but can still be described in a black box/state machine/clear box structure.

Box structure verifications. In the foregoing example, we began with an informal description of a black box ("12-month running average"), then formalized it into an assignment from stimulus histories to responses,

$$R := \frac{S.1 + S.2 + \cdots + S.min\,(i,12)}{min\,(i,12)},$$

accounting for new items with less than 12 months of sales history. Next, we expanded this black box into one of many possible state machines, as in Figure 10, then expanded the state machine into one of many possible clear boxes, as in Figure 11. These two expansions were simple and direct, because the black box itself is quite simple. Even so, these designs are possibly faulty, and in more complex cases the probability of faulty designs increases, even with the greatest of care.

Fortunately, there is a direct and rigorous way to check these designs: Independently derive the state machine of the final clear box expansion and compare it with the intended state machine designed above. If the intended state machine is recovered by derivation, the expansion into the clear box has been verified. Next, we can independently derive the black box from the verified state machine and compare it with the intended black box formulated initially.

We call this rederivation and comparison process a *box structure verification.* It works on the same principle used in division to check that the division has been done correctly, that is, a multiplication of quotient and divisor added to the remainder to independently derive the dividend.

Box structure verification defines an objective, rigorous process for self-checking and peer inspections. Even though people are fallible, this fallibility can be reduced dramatically by such inspections based on an objective, rigorous foundation.

Box structure verification defines an objective, rigorous process for self-checking and peer inspections.

In this example, beginning with the clear box of Figure 11, the first task is to eliminate the procedurality—in this case the sequence of Update Sales and Find Average—to obtain the black box machine of the derived state machine. In Find Average, the expression for R references $S1, \cdots, Smin\,(I,12)$, which were updated in Update Sales, where $S1$ was assigned S, $S2$ assigned $S1, \cdots$, and $S12$ assigned $S11$. Therefore, in terms of the stimulus and original state at the beginning of the transition, the assignment to R is

$$R := \frac{S + S1 + \cdots + Smin\,(I - 1,11)}{min\,(I,12)}$$

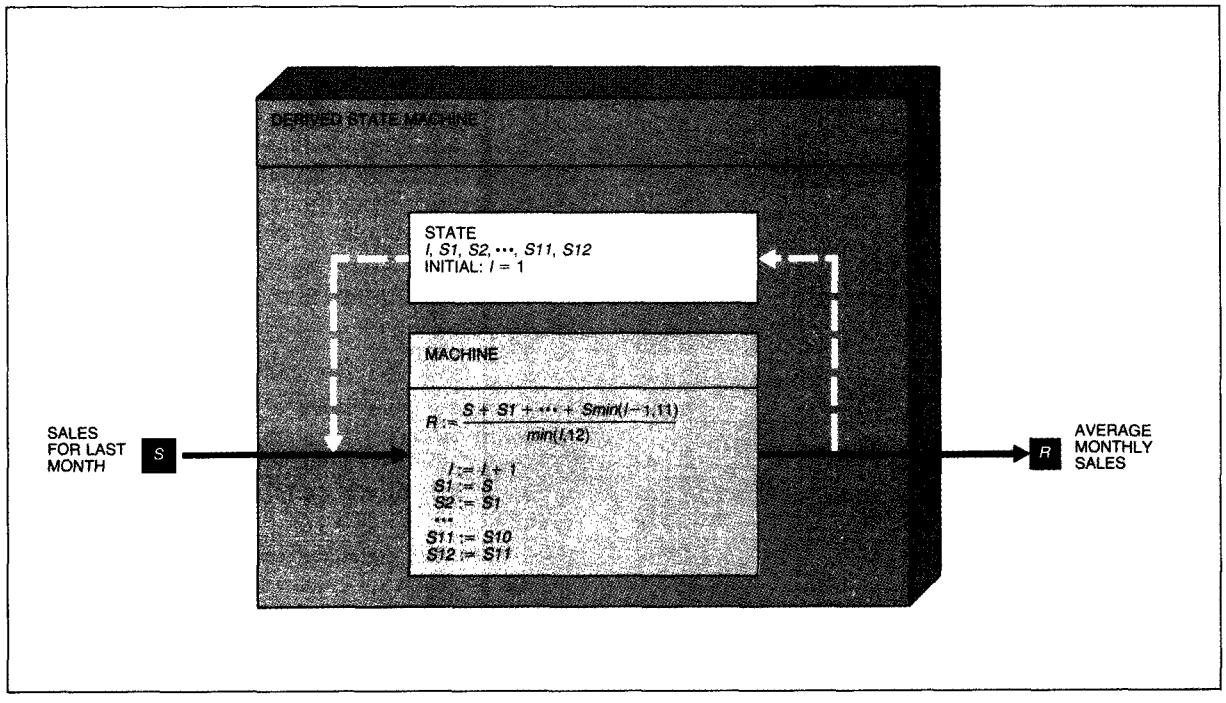
Also, in Find Average, the expression for I references only I which is not changed in Update Sales, so this assignment remains as before,

$$I := I + 1$$

Since Update Sales is the first black box used, its assignments are from the stimulus and original state, so those assignments remain the same. The result of collecting all these assignments into a single black box machine results in the derived state machine shown in Figure 12.

Now we can compare the derived state machine of Figure 12 with the Running Average state machine as shown in Figure 10. They are not identical, line by line, but they differ only in the placement of the line $I := I + 1$ in the multiple assignments. But since these multiple assignments are concurrent, the order of placement of $I := I + 1$ has no effect on the responses of these two machines. So they are identical in effect in returning a response and updating the state of the state machines. With this derivation and comparison, the Running Average clear box of Figure 11 has been verified to be a correct (and com-

Figure 12 Derived state machine



plete) expansion of the Running Average state machine of Figure 10.

Now that the Running Average clear box has been verified, we can turn our attention to the verification of the Running Average state machine, by the independent derivation of its black box, to be compared with the original Running Average black box of Figure 9.

Each value in the state of the Running Average state machine is the cumulative result of its initial value and all subsequent transitions to date. Our objective is to determine the value assigned to R, which is

$$R := \frac{S + S1 + \cdots + Smin(I - 1,11)}{min(I,12)}$$

not in terms of stimulus and state data, but in terms of stimulus history data instead.

First, at age i of the item, we observe that $I = i$ at the beginning of the transition, because at age 1, $I = 1$ by initialization, and I is incremented by 1 at each transition. Therefore, at age i, by direct substitution of i for I in the assignment above,

$$R := \frac{S + S1 + \cdots + Smin(i - 1,11)}{min(i,12)}$$

Furthermore, at the beginning of the transition at age i, the state variables $S1, S2, \cdots, Smin(i - 1,11)$ will contain the sales values $S.2, S.3, \cdots, S.min(i,12)$ for the following reason.

At age 1 of the item, all values of $S1, S2, \cdots, S12$ are uninitialized, but $S1 = S.1$ after the transition at age 1. At age 2, $S2$ is assigned the value of $S1$, which is the value of $S.1$ at age 1, but is renamed $S.2$ at age 2, so $S2 = S.2$, and $S1$ is assigned $S.1$. Continuing, at age i, $S1, S2, \cdots, Smin(i - 1,11)$ are assigned $S.1, S.2, \cdots, S.min(i - 1,11)$ after the transition. But at the beginning of the next transition, these sales values will have all aged one month. So, in fact, at the beginning of the transition at age i, the state variables $S1, S2, \cdots, Smin(i - 1,11)$ will contain the sales values $S.2, S.3, \cdots, S.min(i,12)$.

Finally, we observe that $S = S.1$, the last sales value, so we can complete the substitution of sales values for state values in the calculations for R in the assignment above, to get the derived black box shown in Figure 13.

Figure 13 Derived black box

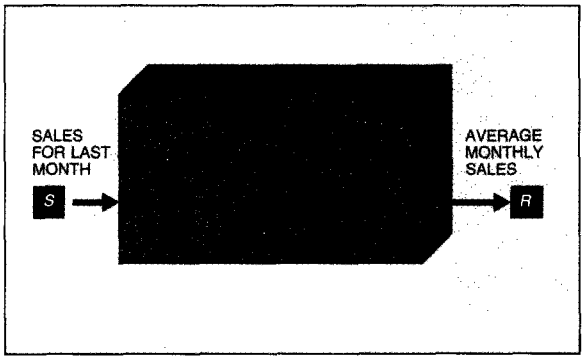
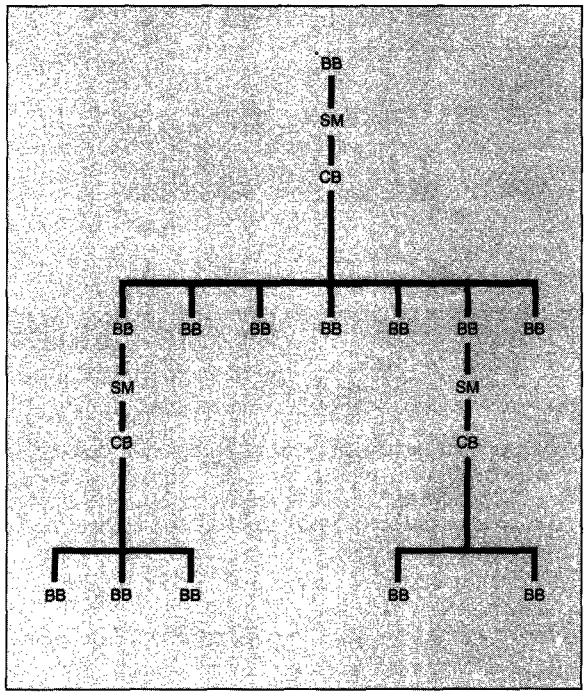


Figure 14 A box structure hierarchy



The derived black box of Figure 13 is identical to the Running Average black box of Figure 9. With this derivation and comparison, the Running Average state machine of Figure 10 has been shown to be a correct (and complete) expansion of the Running Average black box of Figure 9.

The joint result of these two verifications is the verification that the Running Average clear box of Figure 11 is a correct (and complete) expansion of the Running Average black box of Figure 9.

Information systems development with box structures

The box structure concepts presented in the previous section can be expanded into a complete methodology for information systems development. The first step is to describe an information system as a multilevel usage hierarchy wherein each node is a box structure expansion of an independent system part. We then demonstrate how fundamental principles of system development can be applied in the box structure hierarchy.

Box structure hierarchies. A box structure hierarchy, as shown in Figure 14, provides an effective means of control for managing and developing complex information systems. By identifying black box subsystems in higher levels of the system, state data and processing are decentralized into lower-level box structures. Each subsystem becomes a well-defined, independent module in the overall system. Although the progression from black box to state machine to clear box at any point in the hierarchy may appear to be a triplication of effort, this is not the case. Each subsystem can be initially described in its most natural form, with the other forms determined as necessary for analysis and design.

The concept of hierarchies is crucial in system and program development. Top-down programming is based on the principle of stepwise refinement of program modules in a hierarchy. Similarly, usage hierarchies of system modules allow a top-down discipline of system specification and implementation.

The box structure hierarchy, in particular, provides for the systematic application of four essential principles of system development. These principles, called referential transparency, transaction closure, state migration, and common services, are discussed next.

Referential transparency. The box structure hierarchy provides a formal method for defining system modules while preserving referential transparency between levels. Referential transparency is a guiding principle for forming system hierarchies.

Principle of Referential Transparency—In the delegation of any system part for design and implemen-

tation, all requirements should be specified explicitly and independently, so that no further communication or coordination is logically required to complete the system part.

The principle of referential transparency provides a crisp discipline for management delegation and assignment of responsibility. The lack of referential transparency can lead to management nightmares where nothing works and no one is to blame.

The kinds of system parts required to make the principle possible are data abstractions. The specification of such a part must be defined at the stimulus/response level (i.e., black box) for each access to the part, and account for the effect of any previous access to the part. Popularized system development methods that use plausible ideas such as HIPO charts,[28] structure charts,[15] or data flow diagrams[16] can still lose vital information and thus make referential transparency impossible. It takes the right kinds of system parts to defer details without losing them.

The clear box view of a system provides the key abstraction that ensures referential transparency in a box structure hierarchy. The procedurality of the clear box makes precise the control flow and data flow into and out of all embedded black boxes. At the next level of the system hierarchy, each black box can be designed and implemented independently of its surroundings in a system, so accountability is achieved in the delegation. Flexibility is achieved in the delegation because a black box can be redesigned with different state machines and clear boxes as required. As long as the new black box behavior is identical to that of the original, the rest of the system will operate exactly as before. Such black box replacement may be required or desirable for purposes of better performance, changing hardware, or even changing from manual to automatic operations.

When designers and implementers are required to discuss and coordinate details of separate parts after their assignment of responsibilities, gamesmanship becomes an important part of a day's work, in addition to system development. It's only sensible, with ill-defined responsibilities, to cover one's bets and tracks with activities and documents designed as much to protect as to illuminate.

Even with the best of intentions, extensive communication and coordination with respect to design and implementation details opens up many more opportunities for misunderstandings and errors. Such errors are always written off as human fallibilities (nobody is perfect), but errors of unnecessary communication and coordination should be charged to the methodologies that require them, not to the people forced to do the unnecessary communication and coordination.

Transaction closure defines a systematic, iterative specification process.

Transaction closure. *Principle of Transaction Closure*—The transactions (transitions) of a system or system part should be sufficient for the acquisition and preservation of all its state data, and its state data should be sufficient for the completion of all its transactions. In particular, system integrity as well as user function should be considered in achieving transaction closure.

The principle of transaction closure can forestall many surprises and afterthoughts in specifying and designing systems. A common mistake for amateur (and not so amateur) analysts and designers is concentrating so much on primary user transactions that the secondary transactions to make primary user transactions available and reliable become awkward or impossible. For example, if system security or recovery requirements are not identified up front, an ideal user system (imagined in a perfect world of hardware and people) may end up with data structures that make security or recovery difficult or impossible. Therefore, transactions provided for security and recovery need to be defined as early as user transactions, and as carefully.

The principle of transaction closure defines a systematic, iterative specification process, in ensuring that a sufficient set of transactions is identified to acquire and preserve a sufficient set of state data. The iteration begins with the transactions for the primary users, and the state data needed for those transac-

tions, then considers the transactions required for the acquisition and preservation of those state data, then identifies the state data needed for those transactions, and so on. Eventually, no more transactions will be required in an iteration, and transaction closure will have been achieved.

The concept of system integrity plays a special role in transaction closure. Transaction closure assuming perfect hardware and people is not enough; many transactions can only be defined once specific hardware and people are identified for system use. For example, an information system using an operating system with automatic checkpoint and restart facilities will not need checkpoint and restart transactions, but one without them will. The problem of system security provides a classic example. Many operating systems and database systems in wide use today cannot be retrofitted for high-level multilevel security because they were conceived and specified before such security requirements were identified.

In simplest terms, information systems integrity is the property of the system fulfilling its function while handling all of the system issues inherent in its implementation. For example, systems are expected to be correct, secure, reliable, and capable of handling their applications. These requirements may not be explicitly stated by managers, users, or operators, but it is clear that the designed system must have provisions for such properties. Questions of system integrity are largely independent of the function of the system, but are dependent on its means of implementation, manual or automatic. Manual implementations must deal with the fallibilities of people, beginning with their very absence or presence (so backup personnel may be required), that include limited ability and speed in doing arithmetic, limited memory capability for detailed facts, lapses in performance from fatigue or boredom, and so on. Automatic implementation must deal with the fallibilities of computer hardware and software, beginning with their total lack of common sense, that include limited processing and storage capabilities (much larger than for people, but still limited), hardware and software errors, security weaknesses, and so on.

The process of transaction closure is essential in the development of a top-level black box for any system. A useful beginning of this search for a top-level black box begins with the most obvious users of the system but seldom ends there. These most obvious users often interact with the system daily, even minute by minute, in entering and accessing data (for example, a clerk in an airline reservations system). Usually, however, the data they use are provided in part by other users who enter and access data less frequently, such as those entering flight availability information. And other users even more distant from the obvious users enter and access data even less frequently (for example, users who add route schedule information). All the while, an entirely different group, the operators of the system, is entering and accessing system control data that affect the users in terms of more or less access to the system because of limited capacity or availability.

The top-level black box must accommodate the transactions of all these users and operators, not just the most obvious ones. A cross-check can be made between the top-level black box and its top-level state machine. Every item of data in the top-level state must have been loaded with the original system or acquired by previous black box transactions. Are there any items not so loaded or acquired? It is easy, in concentrating on one set of transactions, to assume the existence of data to carry them out. A comprehensive scrutiny of these needed data items can discover such unwarranted assumptions early.

State migration. *Principle of State Migration*—System data should be decentralized to the smallest system parts that do not require duplicating data updates. If, for geographic or security reasons, system data should be decentralized to smaller system parts, the system should be designed to ensure correctly duplicated data updates.

The principle of state migration eliminates the need for instant decisions (often faulty) about how data should be structured and how the data should be stored in a system. Instead, it permits the definition of system data at a conceptual level, and permits the concrete form and location of the data to be worked out interactively with the system design and decomposition into system parts. As better design ideas emerge, system data can be relocated effectively to accommodate such ideas, all the while maintaining correct function as required in the system transactions.

When system data need to be decentralized to smaller system parts than allowed by the principle of state migration, the smallest system defined by this part must be redesigned to accommodate correct duplicate updating. In this case, it is a different system and should be recognized as such from the outset. The problem of incorrect updating of dupli-

cated data is a well-known burden of faulty system designs.

System data in a box structure hierarchy are distributed into the states of their component box structures. State migration through the box structure hierarchy is a powerful tool in managing system development. It permits the placement of state data at the most effective level for its use. Downward migration may be possible when black boxes are identified in a clear box; state data used solely within the state machine expansion of one black box can be migrated to that state machine at the next lower level of the hierarchy. The isolation of state data at proper levels in the system hierarchy provides important criteria for the design of database and file systems. Upward migration is possible when duplicate state data are updated in identical ways in several places in the hierarchy. These data can be migrated up to the common parent state machine for consistent update at one location.

Common services. *Principle of Common Services*—System parts with multiple uses should be considered for definition as common services. A corollary principle is to create as many opportunities as possible for reusability within and between system parts.

Operating systems, data management and database systems, network and terminal control systems are all illustrations of common services between systems. It is axiomatic in today's technology to seek as much reuse of common services as possible to multiply productivity and increase reliability. These common services must satisfy the principle of referential transparency in their use, so their specifications are as important as their implementations. On a smaller scale, effective system design seeks and creates commonality of services and identifies system parts for widespread multiple uses within a system.

Common service box structures are ubiquitous in information systems.

When several black boxes of a clear box expansion access or alter a common state part, it is generally inadvisable to migrate the state part to those lower levels. But it may be advisable to define a new box structure hierarchy to provide access to or to alter this common state part for these several black boxes. Such a new box structure must be invoked in the clear box expansions of these black boxes. This new box structure thereby provides a common service to these several black boxes. Such a common service structure in effect encapsulates a state part, by providing the only means for accessing or altering it in the box structure hierarchy.

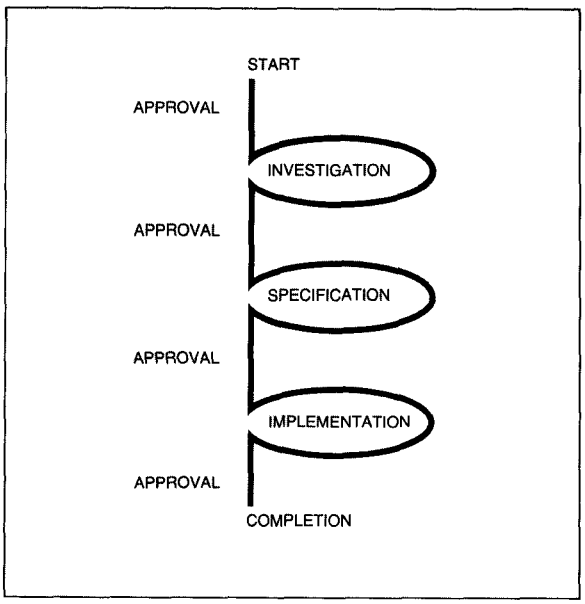
State encapsulation requires a new box structure whose state will contain the common state part and whose transactions will provide common access to that state part for multiple users. In essence, state encapsulation permits state migration to be carried out in another form, with the provision that the only possible access to the migrated state is by invoking transactions of the new box structure that encapsulates it.

Common service box structures are ubiquitous in information systems. For example, any database system behaves as a common service box structure to the people and programs that use it. As a simple illustration, consider a clear box expansion of a master file update state machine. Such a clear box would contain a number of black boxes which operate on the master file, for example, to open, close, read, and write the file, as well as black boxes to access transaction files, directory and authorization information, etc. The master file of the clear box state cannot be migrated to these lower-level black boxes without duplication. However, the master file can be encapsulated, without duplication, in a new box structure that provides the required transactions to open, close, read, and write the file. These transactions can then be invoked from the original box structure hierarchy as required. The new box structure can be designed to ensure the integrity of the master file and all access directed to it. In fact, when the master file is migrated to this common service, it is protected from faulty access by the box structure in an effective way.

The spiral development process

In information systems development, the box structure methodology defines a set of limited, time-phased *activities* to decompose and manage the work required. A formal *development plan* defines and

Figure 15 A system development spiral



schedules the specific activities required to address a specific problem. The development plan represents long-range planning for information system development; the activity plans represent short-range planning. As each activity is completed, the entire development plan is updated to account for the current situation.

Although the activities of a development plan are always specific to a particular system development problem, they can be categorized into three general classes: *investigation*, *specification*, and *implementation*. An investigation is a fact-finding, exploratory study, usually to assess the feasibility of an information system. For example, such a study may define the black box behavior of a projected information system. A specification is more focused to define a specific information system and its benefits to the business. For example, a specification activity may result in definition of state data and high-level clear boxes of a projected information system. An implementation converts a specification into an operational system. For example, implementation may elaborate the black boxes of high-level clear boxes into box structure hierarchies of their own, eventually arriving at human and computer procedures in user guides and software, respectively.

The system development spiral. Many current methods of information systems development reflect appearances rather than principles. One of the obvious appearances in information systems is the system development life cycle. It is certainly apparent that information systems go through various stages of conception, specification, design, implementation, operation, maintenance, modification, and so on. But although these terms are suggestive, real information systems do not pass through these stages in any simple or straightforward way.

In contrast to a fixed life cycle, the box-structured system development process is defined by a set of time-phased activities that are initiated and managed dynamically on the basis of the outcome of previous activities in the development. This progression of activities is conveniently represented in a flexible *system development spiral* that reflects the actual progress of a development effort in terms of box structure analysis and design tasks.

The time-phased set of activities in a spiral can be strictly sequential or may have concurrent parts. If a development is sequential, it can be pictured, in prospect or retrospect, as shown in Figure 15. In this example, the activity sequence is a straightforward progression of

- Investigation
- Specification
- Implementation

with a management approval to enter each activity and to end the entire development. Such a progression for developing a system is ideal, but is not necessarily possible or even desirable.

It may not be possible because the business problem is too complex and needs several investigation activities to arrive at a solution. It may not be possible because the system development problem is too complex and needs several specification/implementation activities in an incremental development. It may not be desirable because the business problem is too acute and a less-than-best implementation is called for as soon as possible. It may not be desirable because the happy outcome of the first investigation activity is the discovery of an existing implementation to meet the business need.

If a development is concurrent, it can be pictured in a network of spirals, as in the example of Figure 16. In this network, activity dependencies are shown by

the approval lines ("A" lines here). For example, Investigation 1 enables both Specification 1 and Investigation 2, whereas both Implementation 1 and Specification 2 must be completed before Implementation 2 can be started. The specific network pictured might, for example, represent the concurrent development of a database system (Implementation 1) and an application system (Implementation 2) that uses it.

Managing spiral development. The system development process generates limited, time-phased activities of investigation, specification, and implementation that must be managed. Formal stages of *planning*, *performance*, and *evaluation* in each activity define an orderly process for this management. The box structure methodology provides a great deal of commonality across these activities for the analysis and design work that is required. The management problems are also very similar. As the names imply, the most challenging stages for management are planning and evaluation, whereas the performance stage is the most challenging for technical professionals.

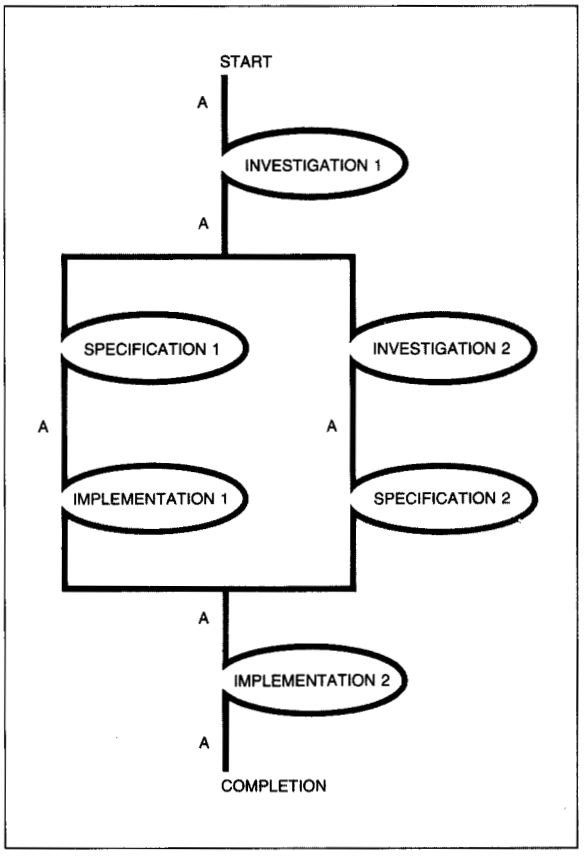
Planning. There are three basic results from the planning stage of any activity:

1. *Activity objective.* A statement of what the activity is to produce.
2. *Activity statement of work.* A statement of how the activity will achieve its objective.
3. *Activity schedule.* An assignment of work items in the Statement of Work to professionals together with agreed-on completion dates.

With such a plan, each member of the entire development team understands the objectives, Statement of Work, and the individual responsibilities for making good on the work objectives and schedule. Such a plan not only requires the agreement of the professionals, but also requires their direct participation in the planning process. But the planning process must be led by managers to address the proper questions and problems for the activity in the overall development plan.

Performance. If plans are well made, performance is focused and predictable. The management job in performance is to assess and track progress against the Statement of Work and schedules. Management must identify unexpected problems and help professionals decide how to meet them, and must identify unexpected windfalls in solutions that can free up people and resources. It is here that good understandings and agreements on assignments and schedules pay off.

Figure 16 A system development spiral with concurrent activities



Evaluation. Evaluation is both a closing out of one activity and a basis for selecting and commencing one or more following activities. The objectives and results of performance can be compared and related to the business and its situation. Even if objectives are not met, the lessons learned may be useful. If the objectives are met, so much the better, and the expected next activities can be initiated. In particular, the evaluation stage is the point where the development plan for future activities can be assessed and modified.

These activities and stages can be organized in tabular form, as shown in Table 1, which indicates typical tasks in systems development. A detailed discussion of these tasks is found in Mills et al.[7]

Table 1 Stages in activities: Typical tasks

Activities	Stages		
	Planning	Performance	Evaluation
Investigation	Activity objective	Business process and objectives	Feasibility assessment
	Statement of work	Requirements analysis	Review and acceptance
	Scheduling	System prototype	Development plan update
Specification	Activity objective	Systems analysis and design	Design verification
	Statement of work	Operations analysis and design	Review and acceptance
	Scheduling		Development plan update
Implementation	Activity objective	Resource acquisition	System testing
	Statement of work	Systems integration	Review and acceptance
	Scheduling	Operations education	Development plan update

Concluding remarks

The box structure methodology provides a rigorous approach for information systems analysis and design. The black box, state machine, and clear box present three different, yet complementary, views of an information system and any of its subsystems. The methodology provides formal techniques for relating these structures and constructing box structure hierarchies.

The correctness of box structure designs can be verified in stepwise fashion from clear boxes by systematically deriving their actual state machine and black box behaviors, and comparing them to their intended behaviors.

Box structures permit application of specific principles of information systems development that help ensure complete and well-structured designs. Referential transparency permits precise delegation of black box expansions once their clear box connections have been designed. Transaction closure ensures complete system behavior for users and complete state definitions for developers. State migration avoids data flow jungles in systems by decentralizing data storage and access into box structure subsystems. Common service design permits migration of widely used data into new box structure hierarchies that provide all required data access.

Box structures permit a flexible management process of spiral development, in contrast to a fixed life cycle. Spiral development is characterized by steps of investigation, specification, and implementation of box structures that can be dynamically sequenced and managed to best capitalize on the current progress and remaining resources of a development effort.

Cited references

1. O. Dahl, E. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press Inc., New York (1972).
2. A. B. Ferrentino and H. D. Mills, "State machines and their semantics in software engineering," *Proceedings of COMPSAC 1977*, Chicago (November 1977), pp. 242–251.
3. R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Company, Inc., Reading, MA (1979).
4. H. D. Mills, D. O'Neill, R. C. Linger, M. Dyer, and R. E. Quinnan, "The management of software engineering," *IBM Systems Journal* **19,** No. 4, 414–477 (1980).
5. M. B. Carpenter and H. K. Hallman, "Quality emphasis at IBM's Software Engineering Institute," *IBM Systems Journal* **24,** No. 2, 121–133 (1985).
6. M. Schaul, "Designing using software engineering principles: Overview of an educational program," *Proceedings 8th International Conference on Software Engineering*, London (1985), pp. 201–208.
7. H. D. Mills, R. C. Linger, and A. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, Inc., New York (1986).
8. F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal* **11,** No. 1, 56–73 (1972).
9. F. T. Baker, "System quality through structured programming," *AFIPS Conference Proceedings Fall Joint Computer Conference* **41,** 339–343 (1972).
10. A. J. Jordano, "DSM software architecture and development," *IBM Technical Directions* **10,** No. 3, 17–28 (1984).
11. D. Parnas, "A technique for software module specification with examples," *Communications of the ACM* **15,** No. 5, 330–336 (May 1972).
12. J. Guttag and J. Horning, "The algebraic specification of abstract data types," *Acta Informatica* **10** (1978).
13. G. Booch, "Object-oriented development," *IEEE Transactions on Software Engineering* **SE-12,** No. 2, 211–221 (February 1986).
14. D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Transactions on Software Engineering* **SE-5,** No. 3, 128–138 (March 1979).
15. E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Programs and System Design*, 2nd Edition, Yourdon Press, New York (1978).

16. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York (1979).
17. W. P. Stevens, *Using Structured Design*, John Wiley & Sons, Inc., New York (1981).
18. W. P. Stevens, "How data flow can improve application development productivity," *IBM Systems Journal* **21**, No. 2, 162–178 (1982).
19. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *Communications of the ACM* **20**, No. 8, 564–576 (August 1977).
20. D. Bjørner and C. Jones, "The Vienna Development Method: The Meta-Language," *Springer-Verlag Lecture Notes in Computer Science* **61**, Springer-Verlag, New York (1978).
21. D. Bjørner, "On the use of formal methods in software development," *Proceedings 9th International Conference on Software Engineering* (1987), pp. 17–29.
22. K. Levitt, P. Neumann, and L. Robinson, "The SRI Hierarchical Development Methodology and its application to the development of secure software," *Proceedings of Software Engineering Applications*, Capri (1980).
23. J. Guttag, J. Horning, and J. Wing, *Larch in Five Easy Pieces*, Technical Report, Digital Equipment Corporation Systems Research Center, Maynard, MA (1985).
24. D. L. Parnas and W. Bartussek, *Using Traces to Write Abstract Specifications for Software Modules*, UNC Report TR 77-012, University of North Carolina, Chapel Hill, NC 27514 (1977).
25. C. A. R. Hoare, "Some properties of predicate transformers," *Journal of the ACM* **25**, No. 3, 461–480 (July 1978).
26. B. Liskov and S. Zilles, "Specification techniques for data abstraction," *IEEE Transactions on Software Engineering* **SE-1**, No. 3, 114–126 (March 1975).
27. M. Shaw, "Abstraction techniques in modern programming languages," *IEEE Software* **1**, No. 4 (October 1984).
28. H. Katzen, *Systems Design and Documentation: An Introduction to the HIPO Method*, Van Nostrand Reinhold, New York (1976).

General references

R. Burstall and J. Goguen, "An informal introduction to specifications using CLEAR," in Boyer and Moore, editors, *The Correctness Problem in Computer Science*, Academic Press Inc., New York (1981).

L. Robinson and O. Roubine, *SPECIAL—A Specification and Assertion Language*, Technical Report CSL-46, Stanford Research Institute, Stanford, CA (1977).

Harlan D. Mills *Information Systems Institute, 2770 Indian River Boulevard, Vero Beach, Florida 32960.* Dr. Mills is Director of the Information Systems Institute, and a Visiting Professor at the University of Florida. He was formerly an IBM Fellow, a member of the IBM Corporate Technical Committee, and Director of Software Engineering and Technology in the IBM Federal Systems Division. Dr. Mills received his Ph.D. in mathematics from Iowa State University in 1952 and has served on faculties at Iowa State, Princeton, New York, and Johns Hopkins Universities and at the University of Maryland. He has also served as a Regent of the DPMA Education Foundation and as a Governor of the IEEE Computer Society.

Richard C. Linger *IBM Federal Systems Division, 6600 Rockledge Drive, Bethesda, Maryland 20817.* Mr. Linger is Senior Programming Manager of Software Engineering Studies. He is the author of numerous research papers on software engineering technology, and coauthor of *Structured Programming: Theory and Practice* (Addison-Wesley, 1979) and *Principles of Information Systems Analysis and Design* (Academic Press, 1986). Mr. Linger received a B.S. degree in electrical engineering from Duke University. He is a member of the ACM and the IEEE Computer Society.

Alan R. Hevner *Information Systems Department, College of Business and Management, University of Maryland, College Park, Maryland 20742.* Dr. Hevner is an Associate Professor and Chairperson of the Information Systems Department. He has published numerous research papers in the areas of distributed database systems, database design, and information systems analysis and design. Dr. Hevner received his Ph.D. in computer science from Purdue University in 1979. He is a member of the ACM and the IEEE Computer Society.

Reprint Order No. G321-5304.