On the construction of interactive systems*

by MARTIN FREEMAN**

Stanford University
Stanford, California

and

WALTER W. JACOBS
The American University
Washington, DC

and

LEON S. LEVY
University of Delaware
Newark, Delaware

INTRODUCTION

Computer-based systems that perform sophisticated tasks in the face of challenging and variable environments are being developed in increasing numbers. Among the examples that can be cited are space exploration modules, highly automated weapon systems, and data-base management systems. The design and development of systems of this class involve many difficulties, and no generally accepted, successful methodology for the task is available.

This paper proposes a contribution to such a methodology in the form of a model that is believed to be generally applicable to the construction of interactive systems. The model is based on the identification of key features common to such systems. These features will be described first. To illustrate the concepts involved, running examples of a large timesharing operating system, and a robot driving an automobile in traffic will be used. Later, applications of the model to the design of microprocessor systems and multi-language processing systems will be discussed.

INTERACTIVE SYSTEMS

Interactive systems are complexes of people, hardware, and software, performing a set of well-defined tasks. These tasks are subject to, or initiated by, conditions met in the environment, and are carried out under general performance objectives. A timesharing operating system has an environment consisting of user-manned terminals and other peripheral devices, and its tasks are the standard types of user requests. A robot chauffeur has an environment consisting of its passengers and the world outside the car, including roads, other cars, traffic signals, parking areas, and so on. Some of its tasks are starting, entering and leaving traffic lanes, parking, and obtaining fuel. Performance objectives for the robot include safety, comfort of passengers, and obedience to traffic regulations while for the timesharing system they include response time, throughput, and reliability.

As these examples show, tasks are the standard components of system activity. Tasks are refined by breaking them down into common subtasks, which are further decomposed until they result in relatively low-level units called actions. For example, the robot chauffeur's activity is expressed in terms of actions such as turning the ignition on or off, accelerating, steering and braking. These actions are the elements of the subtasks of turning a corner, obeying a traffic signal, occupying a parking place, and so on. Actions in the case of the timesharing operating system include compiling a program, searching a directory, or moving a block of data to permanent storage. Actions require resources for their execution. For example, an action for the operating system may be programmed as a machine language subroutine, but a processor and some core memory must be assigned in order that the action can be performed.

An interactive system may also receive assignments of unfamiliar tasks which must be carried out by plans—combinations of standard tasks which are formulated during the operation of the system. A plan will require a decision as each task is completed about the next task to be undertaken. A plan is needed by the robot chauffeur for an auto trip to an unfamiliar destination, while the operating system needs a plan for a large job, involving a number of tasks which must be completed quickly with minimum disruption to other system users.

An interactive system has input from, and output to, its

* The research reported here was supported by the National Science Foundation under grant MCS 76-07682.
** On leave from the American University.
environment. Input may include tasks, as well as data about the conditions in the environment that are involved in executing tasks. For example, users of the timesharing system can input requests, while the robot chauffeur may sense traffic conditions. Output may modify the environment or provide information related to tasks. The operating system may output an error message; the robot chauffeur may turn on the automobile's headlights.

Finally, an interactive system must store and maintain access to a considerable amount of information. Some of this information is general, concerned with the set of tasks and their translation into actions the system can execute. Another part of this information is environmental data—e.g., automobile traffic conditions for the chauffeur and job mix for the operating system. Yet another part is information which must be maintained about the states of the various parts of the system as well as planning information. The former might refer to the status of a task for a particular user of an operating system, while the latter might refer to information to be used to determine how well the chauffeur is performing an unfamiliar task assigned to it.

**METHODOLOGY**

In constructing a complex interactive system, one must find adequate solutions for three major aspects of system organization. First, the execution of system tasks must be controlled. The actual course taken in completing a task depends on the particular conditions met in the environment; and it can be recognized that the control problem is a substantial one. Second, the information needed by the system must be managed so as to be available when needed. And third, system resources (such as memory and processor time for an operating system) must be managed.

A complex interactive system must be organized into logical parts, and this must be done so as to meet two conflicting constraints. For economy, costly resources must be shared by system elements wherever possible, and functions that are the same should not be duplicated. But for effective performance, the parts should be as independent as possible, so as to avoid unnecessary communication and to keep down competition for resources, both of which add to the cost of overhead. A good trade-off between independence and sharing must be found if the design is to achieve performance objectives at reasonable cost.

One approach that has been used in constructing complex systems may be called bottom-up. Development begins with small elements that handle critical details of the system's operation. Larger aggregates are formed using these parts, and the process is continued. The resulting system is structured in layers, each with a particular capability; the top layer handles the tasks in the form that the system receives them. In the case of an operating system; this type of layered structure has been called a hierarchy of virtual machines. This type of hierarchy can also be realized by an approach called top-down which starts with a gross description of the system and calls for successive refinement until the small critical elements of the system are reached. The difficulty with the former approach is that considerations of performance and economy do not enter in the early stages of the development, and the early decisions may well make it impossible later on to achieve performance objectives economically. While this difficulty is removed in the latter approach, it is hard to apply unless one has a good understanding of how the system operates.

Another approach to construction is to dissect the system into modules, which operate independently except for well-defined controls and intercommunication; see e.g., Parnas. The problem of the designer using this approach is to specify modules that require little duplication of information and low-level functions, and that can share resources to an appropriate degree. Useful guides to the choice of such modules do not seem to be available.

The proposed approach to modularization is intended to provide a guide or model of general applicability. Called structural decomposition, it derives from work on artificial intelligence. It leads to breaking up the system into modules that either directly execute some action, or else support its execution. Support may take the form of control, allocation of resources, or management of information.

An important feature of the model is its separation of the short-range, intermediate-range, and long-range aspects of control. Short-range control is involved in executing and supporting individual actions. Intermediate control relates to the determination of the next action to be called when the outcome of the current action occurs. Long-range control arises in planning how to carry out assignments of unfamiliar tasks in terms of known tasks, in adapting the system to changes in the environment, or in otherwise modifying the system. Separation of control in this way reduces the average frequency of communication without restricting the kinds of communication that can take place, and can therefore lead to lower overhead.

Handling input and output in interactive systems often presents technical difficulties. Thus, the model places communication with the environment in a separate module.

**STRUCTURAL DECOMPOSITION**

An interactive system is constructed out of numerous components, with well-defined interfaces. Some parts are hardware, others are software; they are likely to be human components also. Hardware is usually specific to the particular type of system, but the other aspects of the modules that make up the system are being discussed. For the purposes of this exposition, it is irrelevant whether they are realized in software, firmware, or people.

The proposed approach to the design of such systems is to break them down into elements in a particular way. Structural decomposition implies that each module corresponds directly to some aspect of executing the task as a series of actions. The decomposition proceeds in stages, each stage refining the parts identified in the preceding one, until the system has been expressed as off-the-shelf elements.

The first two stages of this decomposition are general enough to serve as a model for a wide class of interactive systems. The description of these stages will emphasize the role played by each part in carrying out tasks, and also the communications between parts.
At each stage, a specification for each part, the communications it receives and the communications it issues is determined. This specification guides the further refinement of the part, considered as a finite-state machine.

The first stage

The initial decomposition of the system is into parts that deal respectively with the long-range, mid-range and short-range aspects of activity. These parts are known as the Executive, Supervisory, and Operating components (see Figure 1).

The Executive is responsible for changes to the system, including both structural modifications and adjustments to adapt to variation in the environment. (This component usually is realized at least partly by a human administrator, but since changes to the system as initially put together are inevitable, the design should make explicit provision for them.) The Executive plans the expression of new tasks, if any, in terms of simpler tasks already known to the system. It also handles system start-up.

The Supervisor's job is to translate tasks into sequences of actions. It does this by initiating the next action on the basis of the outcome of the action just completed. The Supervisor also maintains knowledge of the current status of the system's tasks.

The Operator handles input from and output to the environment. Also, it carries out the actions called for by the Supervisor, making available resources and supporting functions as required. It keeps information pertaining to the environment and to the status of resources.
Communications between these components must take place (see Figure 2). The Operator transmits to the Supervisor any tasks that it recognizes among the inputs from the environment. It also reports the outcome of each completed action, including any status information needed for specifying subsequent actions.

The Supervisor calls on the Operator for each action as appropriate, and provides the parameters that particularize the action to the current situation. Also, it provides to the Executive status summaries needed for the decisions the Executive makes.

The Executive supplies the Operator with start-up information, adjustments in operating rules, and structural modifications, as these become appropriate. If necessary, it passes to the Supervisor the task knowledge for handling a new task.

The second stage

Each of the components of the first stage is itself decomposed into parts. Among those parts are certain active elements that perform special functions. Such elements are known as processors, and indeed they may be realized by microprocessors; on the other hand, their functions may be supplied by programs that share general processing units. At the level of this discussion, it does not matter which way the element will be implemented. Also, it is possible that a processor is busy when a new call for its function arises. In such a case, the processor is assumed to have its own queue.

The “knowledge” or “information” used by the components has already been mentioned. The data structures that embody the knowledge are identified elements of the components in the second stage.

The Supervisor consists of a task processor, an action sequencer, and two data structures—the task knowledge and the task status. When a task is received from the Executive or the Operator, or when the latter reports the outcome of an action, the task processor updates the task status and determines the next action to be called. The task knowledge is used in this process. If the new action must wait for attention along with others already called, the action sequencer decides its place in the line. The task processor passes the results of tasks to the Executive.

The Operator contains an input/output processor, a set of action processors, support units, and environmental data. The input/output processor handles inputs from the environment, in the light of task status information received from the Supervisor, and recognizes tasks, as well as task-related data to be added to the store of environmental data. The processor also issues output to the environment as directed by the Supervisor. The action processors execute actions, using environmental data and task status information; outcomes are reported back to the Supervisor. The support units make their resources available to the processors, manage the environmental data, and do low-level scheduling when necessary.

The Executive has a planning unit and decision information. The planning unit can express a novel job in terms of tasks that the Supervisor is able to manage, and it directs the sequence of such tasks, calling each new one in light of the result obtained in the current one. In the process it uses its decision information, which includes measures of performance objectives and summaries of past experience. A task result may also indicate that rules in use by the Operator need modification, or even that some Operator element should be replaced, and the Executive will then issue the necessary change.

The diagram in Figure 3 summarizes the structural decomposition arrived at in the second stage. Solid dots represent active modules, and open circles represent information elements. Active modules may themselves be constructed by applying structural decompositions to them.

Actions

As mentioned earlier, the concept of action is central to the approach being discussed. An action is an atomic unit of activity that occurs in many tasks. It is executed when it has been supplied with the resources and information it requires, and it continues until it reaches some appropriate termination point. (This is in contrast to the notion of process, used in many software systems. A process can be interrupted at arbitrary points, and resumed later. The interruption incurs an overhead cost in storing the process state and restoring it when it resumes.)

In effect, actions are the operations in terms of which tasks are programmed. The choice of the set of actions for a system is a fundamental part of the effort of constructing the system. An action should take a reasonable number of the basic clock cycles of the system. This is because each action initiation involves the Supervisor, and also requires that resources and information be supplied; the longer the duration of the action, the smaller is the ratio of system overhead to task execution. On the other hand, no action should be allowed to hold on to its resources so long that other requirements of the system are affected adversely, and this condition will set a limit to the duration of any action.

Actions should be general, so that they can respond to the variations in the environment. The Supervisor, when it calls for an action, supplies the parameters that specialize the action to the current situation.

The example in Table 1, representing a very small part of the task knowledge needed by a robot chauffeur, serves to
show how a subtask might be expressed in terms of actions; and how the actions are particularized by parameters. The subtask calls for the robot to enter some specified type of location—parking lot, service station, driveway and so forth—arrived at in a previous subtask; after entering, it is to park in an appropriate spot.

The action seek requires that some place in the immediate environment be found that matches the type description given in the task knowledge. Therefore, the action is executed by the I/O module rather than some other processor. If the time allocated for the action is insufficient for it to find the place, the action seek may be continued. The subtask is terminated when the car has been parked, or when completion has become impossible. In either case, a new subtask will be determined by the Supervisor.

**EXAMPLES**

The application of the interactive system model to the design of a robot chauffeur has been outlined above in general terms; for further details, see Reference 7. The model has also been applied to the development of a timesharing operating system,8 and a brief description of this application will be given.

**An operating system**

The environment of the operating system, as mentioned earlier, are users at terminals, and other peripheral devices. The tasks are job requests issued by users. The environmental data are user program and data files—both temporary and permanent. Examples of actions, in addition to those given earlier, are running as much of a user program as can be completed within an allowed time quantum, and inputting and outputting lines of text at terminals.

System functions are handled by the support units of the Operating component (called the Interface in the reference cited). These functions include managing core memory; allocating I/O channels, buffers, and similar resources to actions that require them; and scheduling processors to execute actions.

The Supervisor at any time is controlling the carrying out of each user’s current request. When it calls an action, the call is placed on an action queue, according to its priority, by an action sequencer. The action call waits until it has received the processor, core memory, and other resources needed for its execution. The action program then carries out the action, reports the outcome to the Supervisor, and releases resources that are no longer needed. The Executive (called the Policy component in this application) can modify scheduling and other system rules to adjust to workload changes. The system administrator, who is treated as part of the Executive, can plan the handling of unusual jobs.

To illustrate the task knowledge for such a system, consider the subtask Open (file, access) which can be used to open a file. The following actions comprise the subtask:

- **transfer** (file part, buffer)—transfers the next portion of the indicated file into a buffer allocated by the system.
- **search** (buffer, file descriptor, access)—searches the buffer area to find a file descriptor. If present it determines whether the user has the indicated access right.
- **output** (message)—prints the indicated message on user’s terminal.

A representation of a simplified version of the subtask in terms of the above actions is described in Table II; the initial action is transfer.

<table>
<thead>
<tr>
<th>Current Action</th>
<th>Parameters</th>
<th>Outcome</th>
<th>Next Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>transfer</td>
<td>next portion of directory, buffer</td>
<td>transferred</td>
<td>search (file, access, buffer)</td>
</tr>
<tr>
<td>search</td>
<td>file, access, buffer</td>
<td>found and access okay</td>
<td>output (&quot;OPENED&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>found and access not okay</td>
<td>output (&quot;ILLEGAL ACCESS&quot;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not found</td>
<td>transfer (next portion of directory, buffer)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOF</td>
<td>output (&quot;FILE NOT PRESENT&quot;)</td>
</tr>
<tr>
<td>output</td>
<td>message</td>
<td>outputted</td>
<td>—</td>
</tr>
</tbody>
</table>
timesharing operating system
message
I/O subsystem
CRT Terminal

Figure 4—The I/O subsystem

A microprocessor system

As another example of the use of the model, a microprocessor system will be discussed (a more general microprocessor system is described in Reference 9). It is believed that the proposed model with its attendant concept of actions allows a methodology for the integration of hardware and software in a natural way. One example of this is in the realization of actions—an action may be dedicated to one microprocessor (or specialized piece of hardware), or several similar actions may be assigned to one microprocessor.

Consider the I/O terminal subsystem in Figure 4 that has been factored from a timesharing operating system in order to meet certain performance objectives (e.g., efficiency). Lines of input from CRT terminals are considered as messages to be transmitted between the operating system and the I/O subsystem. The subsystem itself can be decomposed applying the model. The Supervisor for such a system would take signals from the terminal, considering them as tasks, and would translate them into actions which would be executed by the Operator component. An Executive component could conceivably be used to regulate I/O traffic.

Many designs and implementations exist for such a subsystem along with their attendant opportunities for parallelism. A simplified hardware/software realization of the I/O subsystem using microprocessors is given in Figure 5. It has been assumed that the number of terminals is so large that it is infeasible to have the I/O subsystem share a processor with other parts of the system. Here there are only two components of the model—the Supervisor and the Operator. The Supervisor has a dedicated microprocessor since it must handle requests from terminals as well as outcomes of action executions. Each action has a dedicated microprocessor since many terminals may be requesting service at the same time. Thus at any given time each action microprocessor can be handling different parts of tasks issued from several terminals and the operating system. Other elements of the configuration include a decoder/router for routing action calls from the Supervisor to their respective action queues, an I/O bus and a random access memory.

The I/O bus requires two message formats—the I/O Supervisor format (see Figure 6) and the random access memory format. There is one sub-format for tasks and one sub-format for outcomes. Tasks are issued by the timesharing operating system or by a CRT terminal, while outcomes are the results of action executions.

During regular operation, terminals signal the system by placing an Input task on the I/O bus in Supervisor format. The task is recognized and placed on the Supervisor queue. When the task gets to the head of the queue, the Supervisor issues the first action call in the action sequence that comprises the task (see Table III).

In this case, the first action is transfer which transfers a message (synchronously) from the specified terminal to an internal buffer. The next action check determines whether the message is a command or a data item. If it is a legal command, it is sent to the operating system for execution, if not transfer outputs an error message to the appropriate terminal. If a data item is encountered, then add makes it the next line of the user's workspace.

The Supervisor determines the next action for a task on the basis of the outcome of the present action for this task. At any point in time an action microprocessor can be processing an action call obtained from the head of its action queue. When finished, the outcome is placed on the I/O bus

<table>
<thead>
<tr>
<th>Current Action</th>
<th>Parameters</th>
<th>Outcome</th>
<th>Next Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>transfer</td>
<td>terminal, action</td>
<td>transferred</td>
<td>check (buffer)</td>
</tr>
<tr>
<td>check</td>
<td>buffer</td>
<td>command</td>
<td>send (buffer)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>illegal</td>
<td>transfer ('ILLEGAL COMMAND', terminal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data</td>
<td>add (buffer, workspace)</td>
</tr>
<tr>
<td>send</td>
<td>buffer</td>
<td>sent</td>
<td>transfer ('WORKSPACE OVERFLOW', terminal)</td>
</tr>
<tr>
<td>transfer</td>
<td>buffer, workspace</td>
<td>added</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>overflow</td>
<td></td>
</tr>
</tbody>
</table>
and a new action call is obtained from the associated action queue. The system recognizing that the information on the bus is an outcome, puts it on the Supervisor queue. Eventually the Supervisor removes this outcome from its queue, and examines the associated task data structure to determine which action call to issue next.

<table>
<thead>
<tr>
<th>0</th>
<th>task code</th>
<th>terminal no.</th>
</tr>
</thead>
</table>

| 1 | outcome code | terminal no. |

The action transfer is a bit different from the others. It essentially initiates an I/O operation; when the operation is complete, the associated terminal places the outcome on the I/O bus. The Supervisor of the operating system can forward a task to the I/O subsystem in an analogous way by placing a call on the I/O bus. The outcome can be returned to the Supervisor of the operating system through the send action.

**A multi-language processor**

As a final example, the application of the model to the design of a processor which executes a user's program written in an arbitrary combination of programming languages will be discussed. The model provides a natural framework...
in which to realize such a processor. As a prelude, the utility of a multi-language system will be elaborated on.

Being able to write a program in a programming language suitable to expressing the solution of a problem in a straightforward manner is very desirable. String processing algorithms are easily written in SNOBOL, algebraic and vector computations are easily represented in APL, and input/output operations are most effectively specified in PL/I. Since a universal programming language which can do all of these things well is impractical at this time, it is desirable to be able to use statements from a variety of languages to express a given algorithm.

The usefulness of this approach is particularly evident in the following program segment which contains PL/I and APL statements:

\[
\text{IF } x \neq 0 \text{ THEN } Y + Z_1 \times W; \\
\text{ELSE } Y + Z_2 \times W; \\
\text{PUT DATA (Y);} \]

When \(Z_1, Z_2 \text{ and } W\) are conformable matrices, even this simple two line segment is difficult to program in either PL/I or APL. It is difficult in PL/I because (1) vector operations are not available, (2) PL/I subroutines for APL operations require a different syntax, and (3) PL/I subroutines do not return arrays as values. It is difficult in APL because the equivalent of PUT DATA naming each element would require the user to generate the names of each element.

The proposed model offers a means to realize such an approach in a natural way. Consider an interactive time-sharing system with a subsystem able to handle interpretively user programs written in arbitrary combinations of programming languages. Here the operating system is used to create programs, store and retrieve files, provide I/O capabilities, etc. The subsystem is invoked as an action interpreted applied to a user program.

The subsystem consists of a Supervisor and an Operator. The Supervisor calls upon actions executed by the Operator which (1) provide a generalized fetch-execute cycle for several language processors, and (2) directly execute [10] statements in these languages. Here, then, operations such as add and compare are no longer the basic units of computation; statements in specific languages are the basic units of computation. Thus, given an assignment statement

\[
X = A*B + C/D
\]

in a program file which is tagged with some language specific information (e.g., language type), an action sequence controlled by the Supervisor gets the statement from the program file, identifies the indicated language, computes the expression and performs the assignment.

Variables and data structures must be specified in such a way that the meaning of each statement is unambiguous, even if it may be interpreted in more than one language. Each action can have more than one outcome. If an action associated with a statement in a particular language executes properly, it may, for example, assign a value to a variable or indicate the next statement to be executed. Otherwise, it may fail to parse or encounter execution difficulty; either of these will cause a return to the operating system.

CONCLUSION

This paper has presented a model and methodology for the construction of interactive systems. Its usefulness has been shown in its application to realizing timesharing operating systems, robot-chauffeured automobiles, microprocessor systems and multi-language processor systems. It is felt that a structural approach of this nature is necessary in managing the complexity of the decomposition of large interactive systems.

REFERENCES