CHAPTER 3

CASE STUDIES

We now present seven examples and case studies to illustrate how we can use architectural principles to increase our understanding of software systems. The first case study shows how different architectural solutions to the same problem provide different benefits. The second case study summarizes experience in developing a domain-specific architectural style for a family of industrial products. The third case study contrasts several styles for implementing mobile robotics systems. The fourth case study illustrates how to apply a process-control style to system design. The remaining three case studies present examples of heterogeneous architectures.

3.1 Key Word in Context

In his paper of 1972, Parnas proposed the following problem [Par72]:

> The KWIC [Key Word in Context] index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

Parnas used the problem to contrast different criteria for decomposing a system into modules. He describes two solutions, one based on functional decomposition with shared access to data representations, and a second based on a decomposition that hides design decisions. Since its introduction, the problem has become well-known and is widely used as a teaching device in software engineering.

While KWIC can be implemented as a relatively small system, it is not simply of pedagogical interest. Practical instances of it are widely used by computer scientists. For example, the "permuted" [*sic*] index for the Unix Man pages is essentially such a system.

From the point of view of software architecture, the problem is appealing because we can use it to illustrate the effect of changes on software design. Parnas shows that different problem decompositions vary greatly in their ability to withstand design changes. Among the changes he considers are the following:

1. **Changes in the processing algorithm:** For example, line shifting can be performed on each line as it is read from the input device, on all the lines after they are read, or on demand when the alphabetization requires a new set of shifted lines.
2. **Changes in data representation:** For example, lines, words, and characters can be stored in various ways. Similarly, circular shifts can be stored explicitly or implicitly (as pairs of index and offset).

Garlan, Kaiser, and Notkin also use the KWIC problem to illustrate modularization schemes based on implicit invocation [GKN92]. To do this, they extend Parnas's analysis by also considering the following:

3. **Enhancement to system function:** For example, modify the system to eliminate circular shifts that start with certain noise words (such as *a, an, and,* etc.). Change the system to be interactive, and allow the user to delete lines from the original lists (or from the circularly shifted lists).
4. **Performance:** Both space and time.
5. **Reuse:** To what extent can the components serve as reusable entities?

We now outline four architectural designs for the KWIC system. All four are grounded in published solutions (including implementations). The first two are those considered in Parnas's original article. The third solution uses an implicit invocation style and represents a variant on the solution examined by Garlan, Kaiser, and Notkin. The fourth is a pipeline solution inspired by the Unix index utility.

After presenting each solution and briefly summarizing its strengths and weakness, we contrast the different architectural decompositions in a table organized along the five design dimensions itemized above.

3.1.1 Solution 1: Main Program/Subroutine with Shared Data

The first solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output. These computational components are coordinated as subroutines by a main program that sequences through them in turn. Data is communicated between the components through shared storage ("core storage"). Communication between the computational components and the shared data is an unconstrained read-write protocol. This is possible because the coordinating program guarantees sequential access to the data (see Figure 3.1).

This solution allows data to be represented efficiently, since computations can share the same storage. The solution also has a certain intuitive appeal, since distinct computational aspects are isolated in different modules.

However, as Parnas argues, it has a number of serious drawbacks in terms of its ability to handle changes. In particular, a change in the data storage format will affect almost all of the modules. Similarly, changes in the overall processing algorithm and enhance-

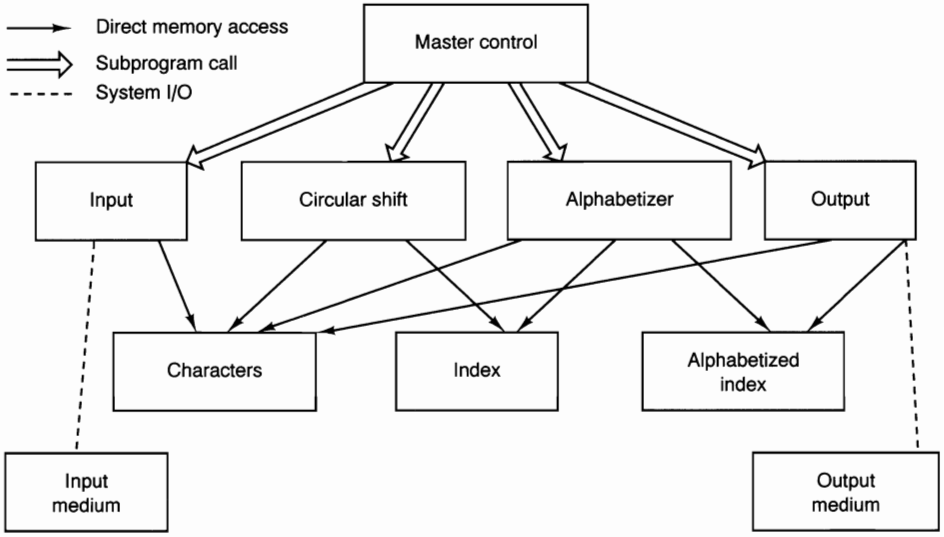

FIGURE 3.1 KWIC: Shared Data Solution

ments to system function are not easily accommodated. Finally, this decomposition is not particularly supportive of reuse.

3.1.2 Solution 2: Abstract Data Types

The second solution decomposes the system into a similar set of five modules. However, in this case data is no longer directly shared by the computational components. Instead, each module provides an interface that permits other components to access data only by invoking procedures in that interface. (See Figure 3.2, which illustrates how each component now has a set of procedures that determine the form of access to it by other components in the system.)

This solution provides the same logical decomposition into processing modules as the first. However, it has a number of advantages over the first solution when design changes are considered. In particular, both algorithms and data representations can be changed in individual modules without affecting others. Moreover, reuse is better supported than in the first solution because modules make fewer assumptions about the others with which they interact.

On the other hand, as discussed by Garlan, Kaiser, and Notkin, the solution is not particularly well suited to certain kinds of functional enhancements. The main problem is that to add new functions to the system, the implementor must either modify the existing modules—compromising their simplicity and integrity—or add new modules that lead to performance penalties. (See [GKN92] for a detailed discussion.)

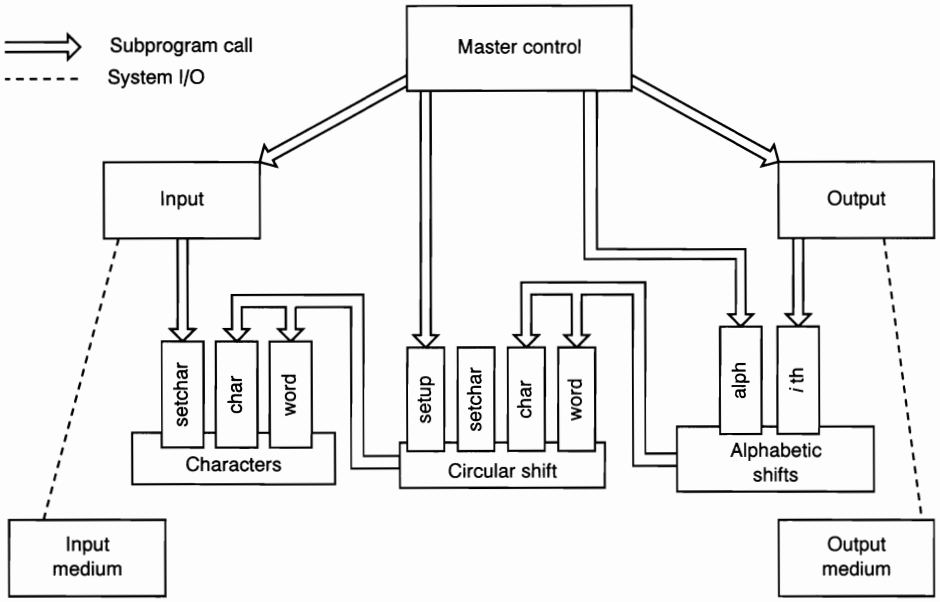

FIGURE 3.2 KWIC: Abstract Data Type Solution

3.1.3 Solution 3: Implicit Invocation

The third solution uses a form of component integration based on shared data, similar to the first solution (see Figure 3.3). However, there are two important differences. First, the interface to the data is more abstract. Rather than exposing the storage formats to the computing modules, this solution accesses data abstractly (for example, as a list or a set). Second, computations are invoked implicitly as data is modified. Thus interaction is based on an "active data" model. For example, the act of adding a new line to the line storage causes an event to be sent to the shift module. This allows it to produce circular shifts (in a separate, abstract shared-data store), which in turn causes the alphabetizer to be implicitly invoked, so that it can alphabetize the lines.

This solution easily supports functional enhancements to the system: additional modules can be attached to the system by registering them to be invoked on data-changing events. Because data is accessed abstractly, the solution also insulates computations from changes in data representation. It also supports reuse, since the implicitly invoked modules only rely on the existence of certain externally triggered events.

However, the solution suffers from the fact that it can be difficult to control the processing order of the implicitly invoked modules. Further, because invocations are data-driven, the most natural implementations of this kind of decomposition tend to use more space than the previously considered decompositions.

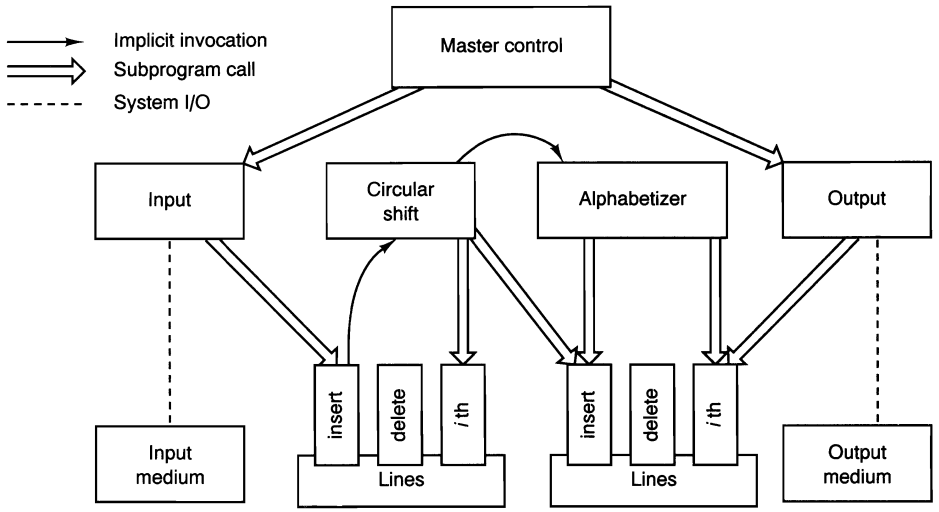

FIGURE 3.3 KWIC: Implicit Invocation Solution

3.1.4 Solution 4: Pipes and Filters

The fourth solution uses a pipeline approach. In this case there are four filters: input, shift, alphabetize, and output. Each filter processes the data and sends it to the next filter. Control is distributed: each filter can run whenever it has data on which to compute. Data sharing between filters is strictly limited to that transmitted on pipes (see Figure 3.4).

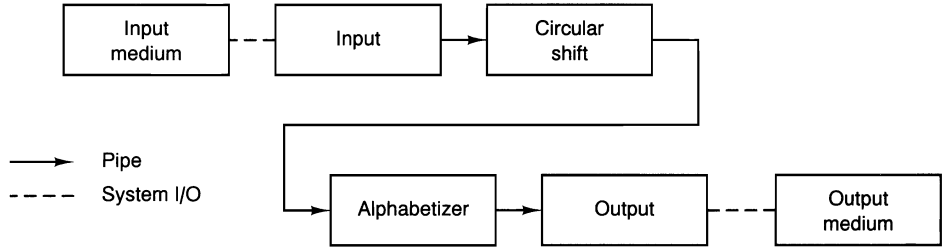

FIGURE 3.4 KWIC: Pipe-and-Filter Solution

This solution has several nice properties. First, it maintains the intuitive flow of processing. Second, it supports reuse, since each filter can function in isolation (provided upstream filters produce data in the form it expects). New functions are easily added to the system by inserting filters at the appropriate points in the processing sequence. Third, it is amenable to modification, since filters are logically independent of other filters.

On the other hand the solution has a number of drawbacks. First, it is virtually impossible to modify the design to support an interactive system. For example, deleting a line would require some persistent shared storage, violating a basic tenet of this approach.

Second, the solution uses space inefficiently, since each filter must copy all of the data to its output ports.

3.1.5 Comparisons

We compare the solutions by tabulating their ability to address the design considerations itemized earlier. For a detailed comparison, we would have to consider a number of factors concerning the intended use of the system: for example, whether it is batch or interactive, update-intensive or query-intensive, and so on.

Figure 3.5 provides an approximation to such an analysis, based on the discussion of architectural styles introduced earlier. As Parnas pointed out, the shared-data solution is particularly weak in its support for changes in the overall processing algorithm, data representations, and reuse. On the other hand it can achieve relatively good performance, by virtue of direct sharing of data. Further, it is relatively easy to add a new processing component (also accessing the shared data). The abstract data type solution allows changes to data representation and supports reuse, without necessarily compromising performance. However, the interactions between components in that solution are wired into the modules themselves, so changing the overall processing algorithm or adding new functions may involve a large number of changes to the existing system.

The implicit invocation solution is particularly good for adding new functionality. However, it suffers from some of the problems of the shared-data approach: poor support for change in data representation and reuse. Moreover, it may introduce extra execution overhead. The pipe-and-filter solution allows new filters to be placed in the stream of text processing. Therefore it supports changes in processing algorithm, changes in function, and reuse. On the other hand, decisions about data representation will be wired into the assumptions about the kind of data that is transmitted along the pipes. Further, depending on the exchange format, there may be additional overhead involved in parsing and unparsing the data onto pipes.

	Shared Data	Abstract Data Type	Implicit Invocation	Pipe and Filter
Change in Algorithm	–	–	+	+
Change in Data Rep	–	+	–	–
Change in Function	+	–	+	+
Performance	+	+	–	–
Reuse	–	+	–	+

FIGURE 3.5 KWIC: Comparison of Solutions

3.2 INSTRUMENTATION SOFTWARE

Our second case study describes the industrial development of a software architecture at Tektronix, Inc. This work was carried out as a collaborative effort between several Tektronix product divisions and the Computer Research Laboratory over a three-year period [DG90, GD90].

The purpose of the project was to develop a reusable system architecture for oscilloscopes. An oscilloscope is an instrumentation system that samples electrical signals and displays pictures (called *traces*) of them on a screen. Oscilloscopes also perform measurements on the signals, and display them on the screen. Although they were once simple analogue devices involving little software, modern oscilloscopes rely primarily on digital technology and have quite complex software. It is not uncommon for a modern oscilloscope to perform dozens of measurements, supply megabytes of internal storage, support an interface to a network of workstations and other instruments, and provide a sophisticated user interface, including a touch-panel screen with menus, built-in help facilities, and color displays.

Like many companies that have had to rely increasingly on software to support their products, Tektronix was faced with a number of problems. First, there was little reuse across different oscilloscope products. Instead, different oscilloscopes were built by different product divisions, each with its own development conventions, software organization, programming language, and development tools. Moreover, even within a single product division, each new oscilloscope typically had to be completely redesigned to accommodate changes in hardware capability and new requirements on the user interface. This problem was compounded by the fact that both hardware and interface requirements were changing increasingly rapidly. Furthermore, there was a perceived need to address "specialized markets," which meant that it would have to be possible to tailor a general-purpose instrument to a specific set of uses, such as patient monitoring or automotive diagnostics.

Second, performance problems were increasing because the software was not rapidly configurable within the instrument. These problems arose because an oscilloscope may be configured in many different modes, depending on the user's task. In old oscilloscopes reconfiguration was handled simply by loading different software to handle the new mode. But the total size of software was increasing, which led to delays between a user's request for a new mode and a reconfigured instrument.

The goal of the project was to develop an architectural framework for oscilloscopes that would address these problems. The result of that work was a domain-specific software architecture that formed the basis of the next generation of Tektronix oscilloscopes. Since then the framework has been extended and adapted to accommodate a broader class of systems, while at the same time being better adapted to the specific needs of instrumentation software. In the remainder of this section, we outline the stages in this architectural development.

3.2.1 AN OBJECT-ORIENTED MODEL

The first attempt at developing a reusable architecture focused on producing an object-oriented model of the software domain, which clarified the data types used in oscilloscopes:

waveforms, signals, measurements, trigger modes, and so on (see Figure 3.6). While this was a useful exercise, it fell far short of producing the hoped-for results. Although many types of data were identified, there was no overall model that explained how the types fit together. This led to confusion about the partitioning of functionality. For example, should measurements be associated with the types of data being measured, or represented externally? Which objects should the user interface interact with?

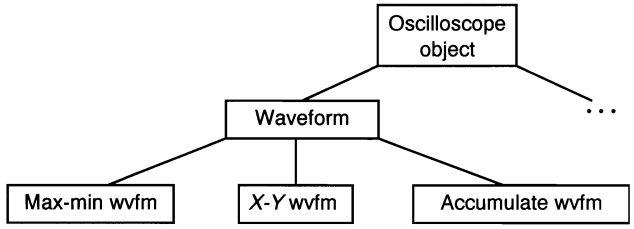

FIGURE 3.6 Oscilloscopes: An Object-Oriented Model

3.2.2 A LAYERED MODEL

The second phase attempted to correct these problems by providing a layered model of an oscilloscope (see Figure 3.7), in which the core layer represented the signal-manipulation functions that filter signals as they enter the oscilloscope. These functions are typically implemented in hardware. The next layer represented waveform acquisition. Within this layer signals are digitized and stored internally for later processing. The third layer consisted of waveform manipulation, including measurement, waveform addition, Fourier transformation, and so on. The fourth layer consisted of display functions, which were responsible for mapping digitized waveforms and measurements to visual representations. The outermost layer was the user interface. This layer was responsible for interacting with the user and for deciding which data should be shown on the screen (see Figure 3.7).

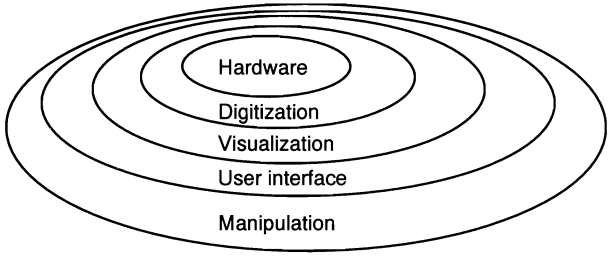

FIGURE 3.7 Oscilloscopes: A Layered Model

The layered model was intuitively appealing since it partitioned the functions of an oscilloscope into well-defined groups. Unfortunately it was the wrong model for the application domain. The main problem was that the boundaries of abstraction enforced by the layers conflicted with the needs for interaction among the various functions. For example, the model suggests that all user interactions with an oscilloscope should be in terms of the

visual representations. In practice, however, real oscilloscope users need to directly affect the functions in all layers, such as setting attenuation in the signal-manipulation layer, choosing acquisition mode and parameters in the acquisition layer, or creating derived waveforms in the waveform-manipulation layer.

3.2.3 A Pipe-and-Filter Model

The third attempt yielded a model in which oscilloscope functions were viewed as incremental transformers of data. Signal transformers are used to condition external signals. Acquisition transformers derive digitized waveforms from these signals. Display transformers convert these waveforms into visual data (see Figure 3.8).

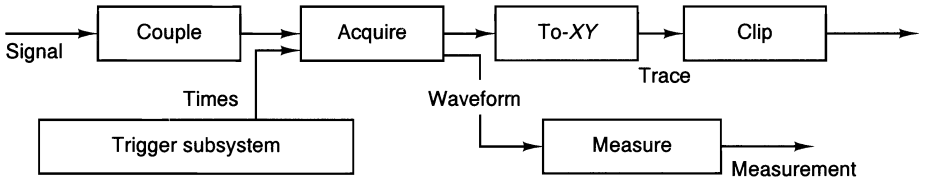

FIGURE 3.8 Oscilloscopes: A Pipe-and-Filter Model

This architectural model was a significant improvement over the layered model because it did not isolate the functions in separate partitions. For example, nothing in this model would prevent signal data from feeding directly into display filters. Further, the model corresponded well with the engineers' view of signal processing as a dataflow problem, and allowed the clean intermingling and substitution of hardware and software components within a system design.

The main problem with the model was that it was not clear how the user should interact with it. If the user were simply at the visual end of the system, this would represent an even worse decomposition than the layered system.

3.2.4 A Modified Pipe-and-Filter Model

The fourth solution accounted for user inputs by associating with each filter a control interface that allowed an external entity to set parameters of operation for the filter. For example, the acquisition filter could have parameters that determined sample rate and waveform duration. These inputs serve as configuration parameters for the oscilloscope. One can think of such filters as having a "control panel" interface that determines what function will be performed across the conventional input/output dataflow interface. Formally, the filters can be modeled as "higher-order" functions, for which the configuration parameters determine what data transformation the filter will perform. (In Section 6.2 we sketch this formal model.) Figure 3.9 illustrates this architecture.

The introduction of a control interface solves a large part of the user interface problem. First, it provides a collection of settings that determine what aspects of the oscilloscope can be modified dynamically by the user. It also explains how the user can change

oscilloscope functions by incremental adjustments to the software. Second, it decouples the signal-processing functions of the oscilloscope from the actual user interface: the signal-processing software makes no assumptions about how the user actually communicates changes to its control parameters. Conversely, the actual user interface can treat the signal-processing functions solely in terms of the control parameters. Designers can therefore change the implementation of the signal-processing software and hardware without affecting the implementation of the user interface (provided the control interface remained unchanged).

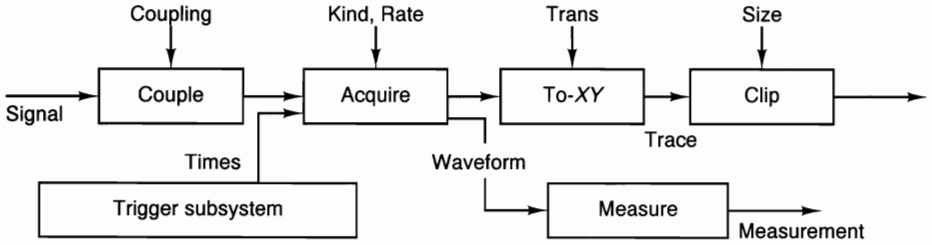

FIGURE 3.9 Oscilloscopes: A Modified Pipe-and-Filter Model

3.2.5 Further Specialization

The adapted pipe-and-filter model was a great improvement, but it, too, had some problems. The most significant problem was that the pipe-and-filter computational model led to poor performance. In particular, because waveforms can occupy a large amount of internal storage, it is simply not practical for each filter to copy waveforms every time they process them. Further, different filters may run at radically different speeds: it is unacceptable to slow one filter down because another filter is still processing its data.

To handle these problems the model was further specialized. Instead of using a single kind of pipe, we introduced several "colors" of pipes. Some of these allowed data to be processed without copying. Others permitted slow filters to ignore incoming data if they were in the middle of processing other data. These additional pipes increased the stylistic vocabulary and allowed the pipe/filter computations to be tailored more specifically to the performance needs of the product.

3.2.6 Summary

This case study illustrates some of the issues involved in developing an architectural style for a real application domain. It underscores the fact that different architectural styles have different effects on the solution to a set of problems. Moreover, it illustrates that architectural designs for industrial software must typically be adapted from pure forms to specialized styles that meet the needs of the specific domain. In this case, the final result depended greatly on the properties of pipe-and-filter architectures, but adapted that generic style so that it could also satisfy the performance needs of the product family.

3.3 Mobile Robotics

By Marco Schumacher

A mobile robotics system is one that controls a manned or partially manned vehicle, such as a car, a submarine, or a space vehicle. Such systems are finding many new uses in areas such as space exploration, hazardous waste disposal, and underwater exploration.

Building the software to control mobile robots is a challenging problem. These systems must deal with external sensors and actuators, and they must respond in real time at rates commensurate with the activities of the system in its environment. In particular, the software functions of a mobile robot typically include acquiring input provided by its sensors, controlling the motion of its wheels and other moveable parts, and planning its future path. Several factors complicate the tasks: obstacles may block the robot's path; the sensor input may be imperfect; the robot may run out of power; mechanical limitations may restrict the accuracy with which it moves; the robot may manipulate hazardous materials; and unpredictable events may demand a rapid response.

Over the years, many architectural designs have been proposed for robotic systems. The richness of the field permits interesting comparisons of the emphases chosen by different researchers. In this section we consider four representative architectural designs.

3.3.1 Design Considerations

To help create a framework for comparison of the architectural approaches, we enumerate below some of the basic requirements (Req1 through Req4) for a mobile robot's architecture.

- **Req1:** The architecture must accommodate *deliberative and reactive behavior.* The robot must coordinate the actions it undertakes to achieve its designated objective (e.g., collect a rock sample) with the reactions forced on it by the environment (e.g., avoid an obstacle).
- **Req2:** The architecture must allow for *uncertainty.* The circumstances of a robot's operation are never fully predictable. The architecture must provide a framework in which the robot can act even when faced with incomplete or unreliable information (e.g., contradictory sensor readings).
- **Req3:** The architecture must *account for dangers* inherent in the robot's operation and its environment. By considering fault tolerance, safety, and performance, the architecture must help maintain the integrity of the robot, its operators, and its environment. Problems like reduced power supply, dangerous vapors, or unexpectedly opening doors should not lead to disaster.
- **Req4:** The architecture must give the designer *flexibility.* Application development for mobile robots frequently requires experimentation and reconfiguration. Moreover, changes in tasks may require regular modification.

The degree to which these requirements apply in a given situation depends both on the complexity of the work the robot is programmed to perform and the predictability of its environment. For instance, fault tolerance is paramount when the robot is operating on another planet as part of a space mission; it is still important, but less crucial, when the robot can be brought to a nearby maintenance facility.

We now examine four architectural designs for mobile robots: Lozano's control loops [LP90], Elfes's layered organization [Elf87], Simmons's task-control architecture [Sim92], and Shafer's application of blackboards [SST86]. We will use the four requirements listed above to guide the evaluation of these alternatives.

3.3.2 Solution 1: Control Loop

Unlike mobile robots, most industrial robots only need to support minimal handling of unpredictable events: the tasks are fully predefined (e.g., welding certain automobile parts together), and the robot has no responsibility with respect to its environment. (Rather, the environment is responsible for not interfering with the robot.) The open-loop paradigm applies naturally to this situation: the robot initiates an action or series of actions without bothering to check on their consequences [LP90].

Upgrading this paradigm to mobile robots involves adding feedback, thus producing a closed-loop architecture. The controller initiates robot actions and monitors their consequences, adjusting the future plans according to the monitored information. Figure 3.10 illustrates the control-loop paradigm. Let us consider how this model handles the four requirements listed above.

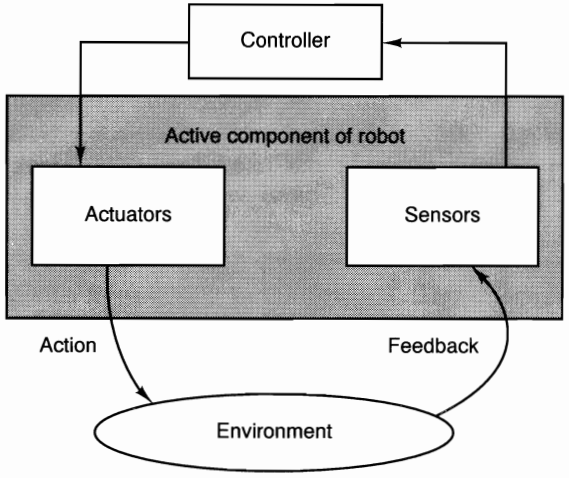

FIGURE 3.10 A Control-Loop Solution for Mobile Robots

- **Req1:** An advantage of the closed-loop paradigm is its simplicity: it captures the basic interaction between the robot and the outside. However, its simplicity is also a drawback in the more unpredictable environments. One expert [LP90] notes that the feedback loop assumes that changes in the environment are continuous and require continuous reactions (e.g., the control of pressure through the gradual opening and closing of a valve). Robots, though, are mostly confronted with disparate, discrete events that require them to switch between very different behavior modes

(e.g., between controlling manipulator motions and adjusting the base position, to avert loss of equilibrium). The model does not suggest how such mode changes should be handled.

For complex tasks, the control loop gives no leverage for decomposing the software into cooperating components. If the steps of sensing, planning, and acting must be refined, other paradigms must provide the details that the control-loop model lacks.

- **Req2:** For the resolution of uncertainty, the control-loop paradigm is biased towards one method: reducing the unknowns through iteration. A trial-and-error process with action and reaction eliminates possibilities at each turn. If more subtle steps are needed, the architecture offers no framework for integrating these with the basic loop or for delegating them to separate entities.
- **Req3:** Fault tolerance and safety are supported by the closed-loop paradigm in the sense that its simplicity makes duplication easy and reduces the chance of errors creeping into the system.
- **Req4:** The major components of a robot architecture (supervisor, sensors, motors) are separated from each other and can be replaced independently. More refined tuning must take place inside the modules, at a level of detail that the architecture does not show.

In summary, the closed-loop paradigm seems most appropriate for simple robotic systems that must handle only a small number of external events and whose tasks do not require complex decomposition.

3.3.3 Solution 2: Layered Architecture

Figure 3.11 shows Alberto Elfes's definition of the idealized layered architecture [Elf87], which influenced the design of the Dolphin sonar and navigation system, implemented on the Terregator and Neptune mobile robots [CBCD93, PDB84].

- At level 1, the lowest level, reside the robot control routines (motors, joints, etc.).
- Levels 2 and 3 deal with input from the real world. They perform sensor interpretation (the analysis of the data from one sensor) and sensor integration (the combined analysis of different sensor inputs).
- Level 4 is concerned with maintaining the robot's model of the world.
- Level 5 manages the navigation of the robot.
- Levels 6 and 7 schedule and plan the robot's actions. Dealing with problems and replanning is also part of the level-7 responsibilities.
- The top level provides the user interface and overall supervisory functions.
- **Req1:** Elfes's model sidesteps some of the problems encountered with the control loop by defining more components to which the required tasks can be delegated. Being specialized to autonomous robots, it indicates the concerns that must be addressed (e.g., sensor integration). Furthermore, it defines abstraction levels (e.g., robot control vs. navigation) to guide the design.

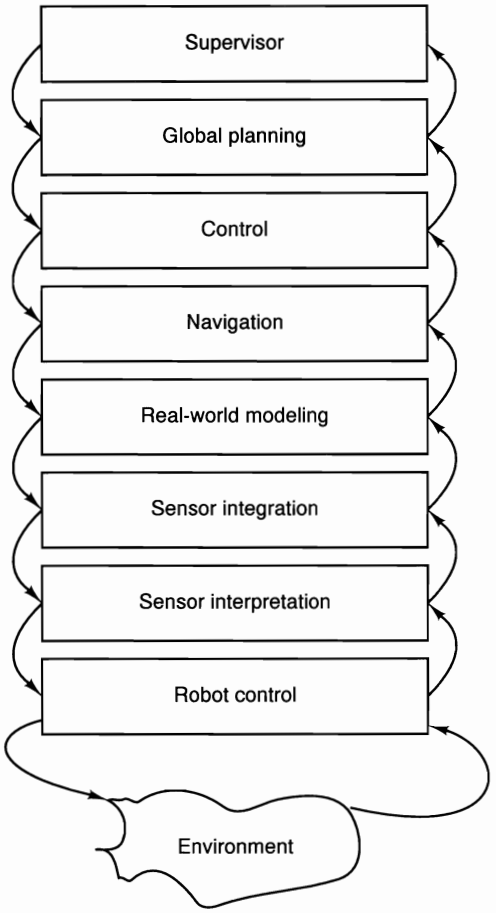

FIGURE 3.11 A Layered Solution for Mobile Robots

While it nicely organizes the components needed to coordinate the robot's operation, the layered architecture does not fit the actual data and control-flow patterns. The layers suggest that services and requests are passed between adjacent components. In reality, as Elfes readily acknowledges, information exchange is less straightforward. For instance, data requiring fast reaction may have to be sent directly from the sensors to the problem-handling agent at level 7, and the corresponding commands may have to skip levels to reach the motors in time.

Another problem in the model is that it does not separate two abstraction hierarchies that actually exist in the architecture:

- The data hierarchy, with raw sensor input (level 1), interpreted and integrated results (2 and 3), and finally the world model (4).
- The control hierarchy, with motor control (level 1), navigation (5), scheduling (6), planning (7), and user-level control (8).

(Some other layered architectures, such as NASREM, do a better job in this respect. We will mention some of these later.)

- **Req2:** The existence of abstraction layers addresses the need for managing uncertainty: what is uncertain at the lowest level may become clear with the added knowledge available in the higher layers. For instance, the context embodied in the world model can provide the clues to disambiguate conflicting sensor data.
- **Req3:** Fault tolerance and passive safety (when you strive not to do something) are also served by the abstraction mechanism. Data and commands are analyzed from different perspectives. It is possible to incorporate many checks and balances into the system. As already mentioned, performance and active safety (when you have to do something rather than avoid doing something) may require that the communication pattern be short-circuited.
- **Req4:** The interlayer dependencies are an obstacle to easy replacement and addition of components. Further, complex relationships between the layers can become more difficult to decipher with each change.

In summary, the abstraction levels defined by the layered architecture provide a framework for organizing the components that succeeds because it is precise about the roles of the different layers. The major drawback of the model is that it breaks down when it is taken to the greater level of detail demanded by an actual implementation. The communication patterns in a robot are not likely to follow the orderly scheme implied by the architecture.

3.3.4 Solution 3: Implicit Invocation

The third solution is based on a form of implicit invocation, as embodied in the Task-Control Architecture (TCA) [Sim92]. Figure 3.12 sketches the architecture. It has been used to control numerous mobile robots, such as the Ambler robot [Sim90].

The TCA architecture is based on hierarchies of tasks, or *task trees*. Figure 3.13 shows a sample task tree, in which parent tasks initiate child tasks. The software designer can define temporal dependencies between pairs of tasks; for example, a common temporal constraint is that *A* must complete before *B* starts. These features permit the specification of selective concurrency. The implementation of TCA includes many operations for dynamic reconfiguration of task trees at run time in response to changing robot state and environmental conditions.

TCA uses implicit invocation to coordinate interactions between tasks. Specifically, tasks communicate by multicasting messages via a message server, which redirects those messages to tasks that are registered to handle them.

In addition to the communication of messages, TCA's implicit invocation mechanisms support three functions:

1. **Exceptions:** Certain conditions cause the execution of an associated exception handler. Exceptions override the currently executing task in the subtree that causes the exception. They quickly change the processing mode of the robot and are thus better suited for managing spontaneous events (such as a dangerous change in terrain) than the feedback loop or the long communication paths of the pure layered architecture. Exception handlers have at their disposal all the operations for manipulating the task trees: for example, they can abort or retry tasks.
2. **Wiretapping:** Messages can be intercepted by tasks superimposed on an existing task tree. For instance, a safety-check component can use this feature to validate outgoing motion commands.
3. **Monitors:** Monitors read information and execute some action if the data fulfills a certain criterion. An example from the TCA manual is the battery check: if the battery level falls below a given level, the actions necessary for recharging it are invoked. This feature offers a convenient way of dealing with fault-tolerance issues by setting aside agents to supervise the system.

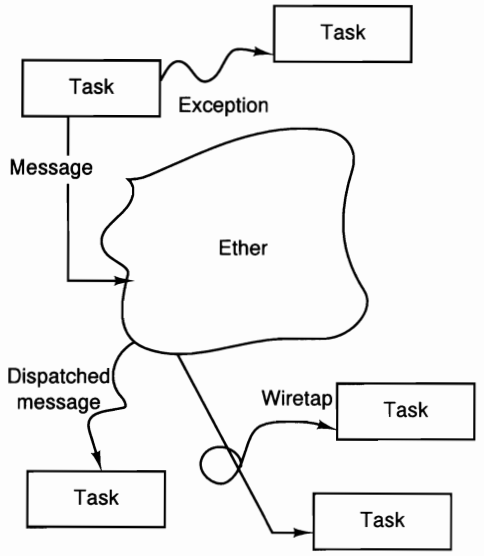

FIGURE 3.12 An Implicit Invocation Architecture for Mobile Robots

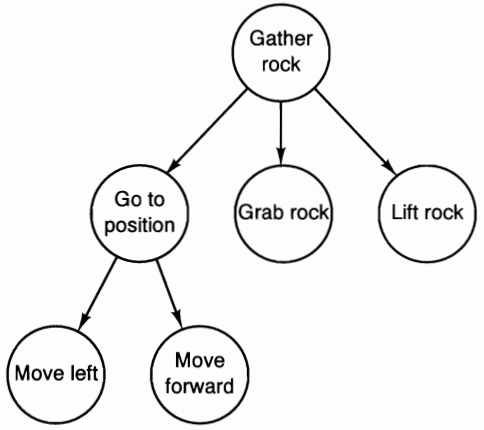

FIGURE 3.13 A Task Tree for Mobile Robots

We turn now to the requirements.

- **Req1:** Task trees on the one hand, and exceptions, wiretapping, and monitors on the other, permit a clear-cut separation of action (the behavior embodied in the task trees) and reaction (the behavior dictated by extraneous events and circumstances).

  TCA also distinguishes itself from the previous paradigms by explicitly incorporating concurrent agents in its model. In TCA it is evident that multiple actions can proceed at the same time, more or less independently. The other two models do not explicitly address concurrency.

  In practice, the amount of concurrency of a TCA system is limited by the capabilities of the central server. In general, its reliance on a central control point may be a weak point of TCA.
- **Req2:** How TCA addresses uncertainty is less clear. If imponderables exist, a tentative task tree can be built, to be modified by the exception handlers if its fundamental assumptions turn out to be erroneous.
- **Req3:** As illustrated by the examples above, the TCA exception, wiretapping, and monitoring features take into account the needs for performance, safety, and fault tolerance.

  Fault tolerance by redundancy is achieved when multiple handlers register for the same signal; if one of them becomes unavailable, TCA can still provide the service by routing the request to another. Performance also benefits, since multiple occurrences of the same request can be handled concurrently by multiple handlers.
- **Req4:** The use of implicit invocation makes incremental development and replacement of components straightforward. It is often sufficient to register new handlers, exceptions, wiretaps, or monitors with the central server; no existing components are affected.

In summary, TCA offers a comprehensive set of features for coordinating the tasks of a robot while respecting the requirements for quality and ease of development. The richness of the scheme makes it most appropriate for more complex robot projects.

3.3.5 Solution 4: Blackboard Architecture

Figure 3.14 describes a blackboard architecture for mobile robots. This paradigm was used in the NAVLAB project, as part of the CODGER system [SST86]. The *whiteboard* architecture, as it is termed in [SST86], works with abstractions reminiscent of those encountered in the layered architecture. The components of CODGER are the following:

- The "captain": the overall supervisor.
- The "map navigator": the high-level path planner.
- The "lookout": a module that monitors the environment for landmarks.
- The "pilot": the low-level path planner and motor controller.
- The perception subsystem: the modules that accept the raw input from multiple sensors and integrate it into a coherent interpretation.

Let us consider the requirements.

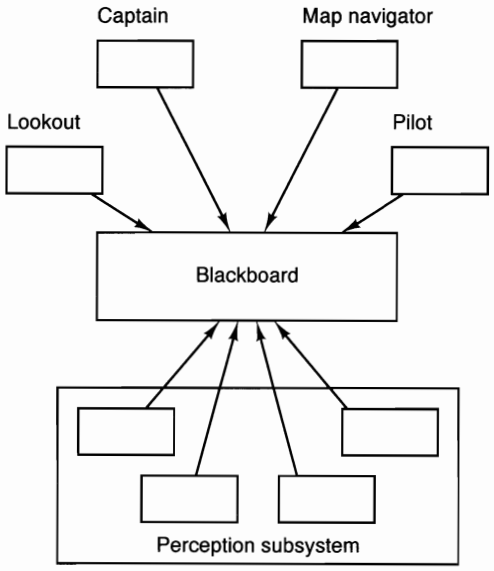

FIGURE 3.14 A Blackboard Solution for Mobile Robots

- **Req1:** The components (including the modules inside the perception subsystem) communicate via the shared repository of the blackboard system. Modules indicate their interest in certain types of information. The database returns them such data either immediately or when some other module inserts it into the database. For instance, the lookout may watch for certain geographic features; the database informs it when the perception subsystem stores images matching the description.

  One difficulty with the CODGER architecture is that control flow has to be coerced to fit the database mechanism, even under circumstances where direct interaction between components would be more natural.
- **Req2:** The blackboard is also the means for resolving conflicts or uncertainties in the robot's world view. For instance, the lookout's landmark detections provide a reality check for the distance estimation by dead-reckoning; both sets of data are stored in the database. The modules responsible for resolving the uncertainty register with the database to obtain the necessary data. (A good example of this activity is sensor fusion, performed by the perception subsystem to reconcile the input from its diverse sensors.)
- **Req3:** Communication via the database is similar to the communication via TCA's central message server. The exception mechanism, wiretapping, and monitoring—guarantors of reaction speed, safety, and reliability—can be implemented in CODGER by defining separate modules that watch the database for the signs of unexpected occurrences or the beginnings of troublesome situations.
- **Req4:** As with TCA, the blackboard architecture supports concurrency and decouples senders from receivers, thus facilitating maintenance.

In summary, the blackboard architecture is capable of modeling the cooperation of tasks, both for coordination and resolving uncertainty in a flexible manner, thanks to an implicit invocation mechanism based on the contents of the database. These features are only slightly less powerful than TCA's equivalent capabilities.

3.3.6 Comparisons

In this section we have examined four architectures for mobile robotics to illustrate how architectural designs can be used to evaluate the satisfaction of a set of requirements. The four architectures differ substantially in their control regimes, their communications, and their specificity of components, which affects the degree to which they satisfy the requirements, as shown in Figure 3.15.

	Control Loop	Layers	Implicit invocation	Blackboard
Task Coordination	+ –	–	+ +	+
Dealing with uncertainty	–	+ –	+ –	+
Fault intolerance	+ –	+ –	+ +	+
Safety	+ –	+ –	+ +	+
Performance	+ –	+ –	+ +	+
Flexibility	+ –	–	+	+

FIGURE 3.15 Table of Comparisons

Naturally, these architectures are not the only possibilities; dozens of other architectures exist. Many of these are hybrids. For example, the The NASA/NBS Standard Reference Model for Telerobots (NASREM) [LFW90] is an architecture that combines control loops with layers, while Hayes-Roth's architecture combines layers with data flow [HRPL+95].

3.4 Cruise Control

In this section we illustrate how to apply the control-loop paradigm to a simple problem that has traditionally been cast in object-oriented terms. As we will show, the use of control-loop architectures can contribute significantly to clarifying the important architectural dimensions of the problem.

Booch and others have used the cruise-control problem to explore the differences between object-oriented and functional (traditional procedural) programming [AG93, Boo86,War84]. As described by Booch, this problem is:

> A cruise-control system that exists to maintain the speed of a car, even over varying terrain. Figure 3.16 shows the block diagram of the hardware for such a system.
>
> There are several inputs:

System on/off	If on, denotes that the cruise-control system should maintain the car speed.
Engine on/off	If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
Pulses from wheel	A pulse is sent for every revolution of the wheel.
Accelerator	Indication of how far the accelerator has been pressed.
Brake	On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
Increase/Decrease Speed	Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
Resume Speed	Resume the last maintained speed; only applicable if the cruise-control system is on.
Clock	Timing pulse every millisecond.

> There is one output from the system:

Throttle	Digital value for the engine throttle setting.

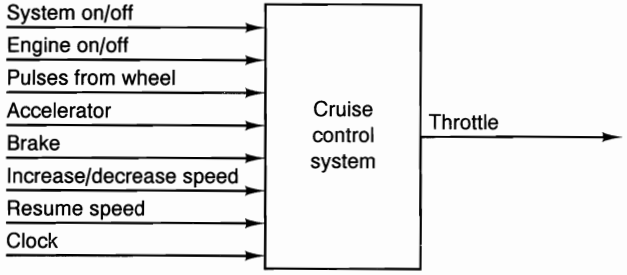

FIGURE 3.16 Booch Block Diagram for Cruise Control

The problem does not clearly state the rules for deriving the output from the set of inputs. Booch elaborates the description in the form of a dataflow diagram, but some questions remain unanswered. In the design below, missing details are supplied to match the apparent behavior of the cruise control on the authors' cars. Moreover, the inputs pro-

vide two kinds of information: whether the cruise control is active, and if so, what speed it should maintain.

The problem statement says the output is a value for the engine throttle setting. In classical process control, the corresponding signal would be a change in the throttle setting; this avoids calibration and wear problems with the sensors and engine. A more conventional cruise-control requirement would thus specify control of the current speed of the vehicle. However, current speed is not explicit in the problem statement, though it does appear implicitly as "maintained speed" in the descriptions of some of the inputs. If the requirement addresses current speed, throttle setting remains an appropriate output from the control algorithm. To avoid unnecessary changes in the problem we assume accurately calibrated digital control and achieve the effect of incremental signals by retaining the previous throttle value in the controller.

The problem statement also specifies a millisecond clock. In Booch's object-oriented solution, the clock is used only in combination with the wheel pulses to determine the current speed. Presumably the process that computes the speed will count the number of clock pulses between wheel pulses. The problem is overspecified in this respect: a slower clock or one that delivered current time on demand with sufficient precision would also work and would require less computing. Further, a single system clock is not required by the problem, though it might be convenient for other reasons.

These considerations lead to a restatement of the problem: *Whenever the system is active, determine the desired speed, and control the engine throttle setting to maintain that speed.*

3.4.1 Object View of Cruise Control

Booch organizes an object-oriented decomposition of the system around objects that exist in the task description. The elements of the decomposition correspond to important quantities and physical entities in the system, as shown in Figure 3.17, where the blobs represent objects, and the directed lines represent dependencies among objects. Although the target speed did not appear explicitly in the problem statement, it does appear in Figure 3.17 as "Desired Speed."

3.4.2 Process-Control View of Cruise Control

Section 2.8.1 suggests that a control-loop architecture is appropriate when the software is embedded in a physical system that involves continuing behavior, especially when the system is subject to external perturbations. These conditions hold in the case of cruise control: the system is supposed to maintain constant speed in an automobile despite variations in terrain, vehicle load, air resistance, fuel quality, and so on. To develop a control-loop architecture for this system, we begin by identifying the essential system elements as described in Section 2.8.1.

Computational elements

- **Process definition:** Since the cruise-control software is driving a mechanical device (the engine), the details are not relevant. For our purposes, the process receives a throttle setting and turns the car's wheels. There may in fact be more computers involved, for example, in controlling the fuel-injection system. From the standpoint of the cruise-control subsystem, however, the process takes a throttle setting as input and controls the speed of the vehicle.
- **Control algorithm:** This algorithm models the current speed from the wheel pulses, compares it to the desired speed, and changes the throttle setting. The clock input is needed to determine current speed from the intervals between wheel pulses. Since the problem requires an exact throttle setting rather than a change, the current throttle setting must be maintained by the control algorithm. The policy decision about how much to change the throttle setting for a given discrepancy between current speed and desired speed is localized in the control algorithm.

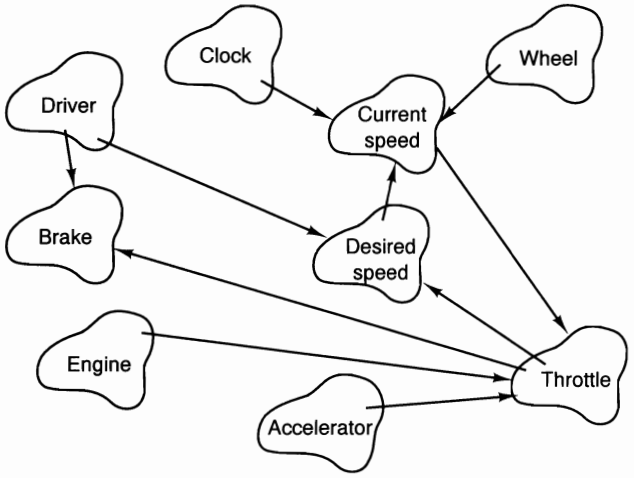

FIGURE 3.17 Booch's Object-Oriented Design for Cruise Control

Data elements

- **Controlled variable:** For the cruise control, this is the current speed of the vehicle.
- **Manipulated variable:** For the cruise control, this is the throttle setting.
- **Set point:** The desired speed is set and modified by the accelerator input and the increase/decrease speed input, respectively. Several other inputs determine whether the cruise control is currently controlling the car: system on/off, engine on/off, brake, and resume. These inputs interact: resume restores automatic control, but only if the entire system is on. These inputs are provided by the human driver (the operator, in process terms).
- **Sensor for controlled variable:** For cruise control, the current state is the current speed, which is modeled on data from a sensor that delivers wheel pulses, using the clock. (See the discussion below about the accuracy of this model.)

In the restated control task, note that only the current speed output, the wheel pulses input, and the throttle manipulated variable are used outside the set point and active/

inactive determination. This leads immediately to two subproblems: the interface with the driver, concerned with "Whenever the system is active determine the desired speed," and the control loop, concerned with "Control the engine throttle setting to maintain that speed."

The latter is the actual control problem; we'll examine it first. Figure 3.18 shows a suitable architecture for the control system. The first task is to model the current speed from the wheel pulses. The designer should validate this model carefully. The model could fail if the wheels spin, which could affect control in two ways. If the wheel pulses are being taken from a drive wheel and the wheel is spinning, the cruise control would keep the wheel spinning (at constant speed) even if the vehicle stops moving. Even worse, if the wheel pulses are being taken from a nondrive wheel and the drive wheels are spinning, the controller will act as if the current speed is too slow and continually increase the throttle setting. The designer should also consider whether the controller has full control authority over the process. In the case of cruise control, the only manipulated variable is the throttle; the brake is not available. As a result, if the automobile is coasting faster than the desired speed, the controller is powerless to slow it down.

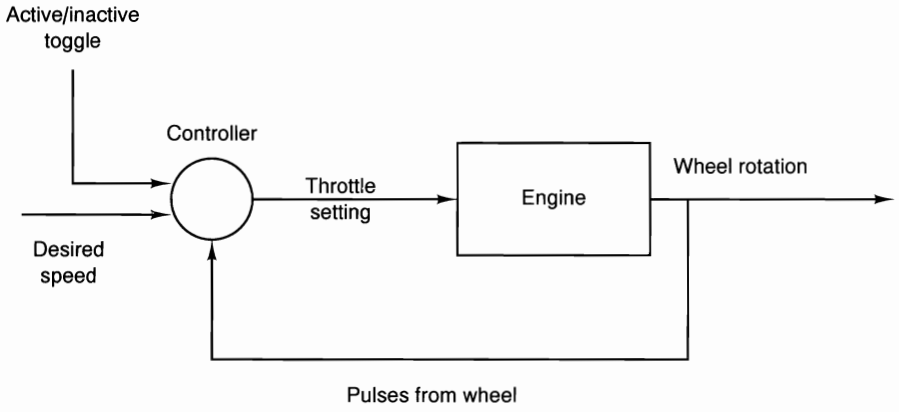

FIGURE 3.18 Control Architecture for Cruise Control

The controller also receives two inputs from the set-point computation: the active/inactive toggle, which indicates whether the controller is in charge of the throttle, and the desired speed, which only needs to be valid when the vehicle is under automatic control. All this information should be either state or continuously updated data, so all lines in the diagram represent dataflow. The controller is implemented as a continuously evaluating function that matches the dataflow character of the inputs and outputs. Several implementations are possible, including variations on simple on/off control, proportional control, and more sophisticated disciplines. Each of these has a parameter that controls how quickly and tightly the control tracks the set point; analysis of these characteristics is discussed in Section 3.4.3. As noted above, the engine is of little interest here; it might very well be implemented as an object or as a collection of objects.

The set-point calculation divides naturally into two parts: (1) determining whether or not the automatic system is active—in control of the throttle—and (2) determining the desired speed for use by the controller in automatic mode.

Some of the inputs in the original problem definition capture state (system on/off, engine on/off, accelerator, brake), and others capture events (wheel pulses, increase/decrease speed, resume, clock). We will treat accelerator as state, specifically as a continuously updated value. However, determining whether the automatic cruise control is actively controlling the car is cleaner if everything it depends on is of the same kind. We will therefore use transitions between states for system on/off, engine on/off, and brake. For simplicity we assume brake application is atomic, so that other events are blocked when the brake is on. A more detailed analysis of the system states would relax this assumption [AG93].

The active/inactive toggle is triggered by a variety of events, so a state transition design, shown in Figure 3.19, is natural. The system is completely off whenever the engine is off. Otherwise there are three inactive states and one active state. In the first inactive state no set point has been established. In the other two, the previous set point must be remembered. When the driver accelerates to a speed greater than the set point, the manual accelerator controls the throttle through a direct linkage (note that this is the only use of the accelerator position in this design, and it relies on relative effect rather than absolute position); when the driver uses the brake, the control system is inactivated until the resume signal is sent. The active/inactive toggle input of the control system is set to active precisely when this state machine is in state Active.

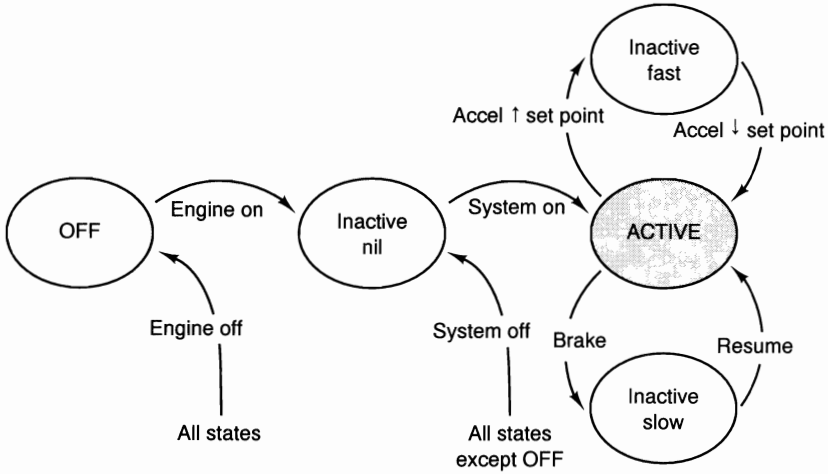

FIGURE 3.19 State Machine for Activation

Determining the desired speed is simpler, since it does not require state other than the current value of desired speed (the set point). Whenever the system is off, the set point is undefined. Whenever the system on signal is given (including when the system is already on), the set point is set to the current speed as modeled by wheel pulses. The driver also has a control that increases or decreases the set point by a set amount. This, too, can be invoked at any time (assume that arithmetic on undefined values yields undefined values). Figure 3.20 summarizes the events involved in determining the set point. Note that this process requires access to the clock in order to estimate the current speed from the pulses from the wheel.

Event	Effect on desired speed
Engine off, system off	Set to "undefined."
System on	Set to current speed as estimated from wheel pulses.
Increase speed	Increment desired speed by constant.
Decrease speed	Decrement desired speed by constant.

FIGURE 3.20 Event Table for Determining Set Point

We can now (Figure 3.21) combine the control architecture, the state machine for activation, and the event table for determining the set point into an entire system. Although the control unit and set point determination do not need to use the same clock, we have them do so to minimize changes to the original problem statement. Then, since current speed is used in two components, it would be reasonable for the next elaboration of the design to encapsulate that model in a reusable object; this would encapsulate the clock.

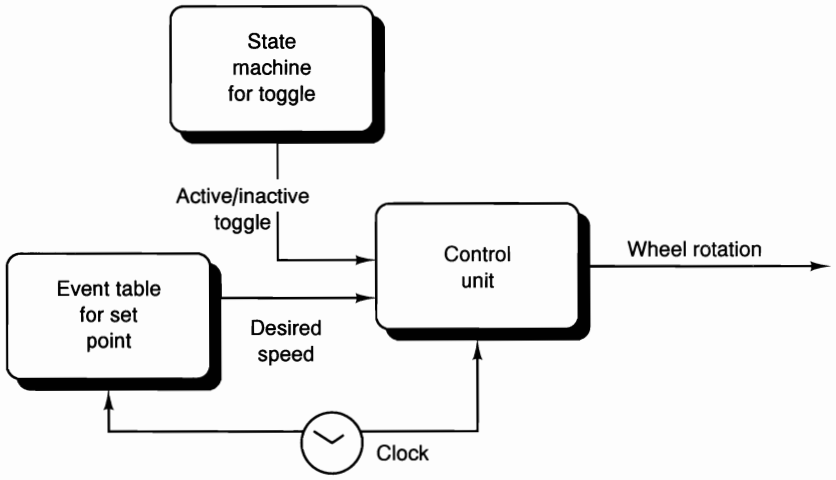

FIGURE 3.21 Complete Cruise Control System

All of the objects in Booch's design (Figure 3.17) have clear roles in the resulting system. It is entirely reasonable to anticipate a design strategy that uses the control-loop architecture for the system as a whole, and uses a number of other architectures, including objects and state machines, to elaborate the elements of the control-loop architecture. The shift from an object-oriented view to a control view of the cruise-control architecture raised a number of design questions that had previously been slighted. The separation of process from control concerns led to explicit choice of the control discipline. The limitations of the control model also became clear, including possible inaccuracies in the current speed model and incomplete control at high speed. The dataflow character of the model

showed irregularities in the way the input was specified, for example, mixture of state and event inputs and the inappropriateness of absolute position of the accelerator.

We now compare the two approaches in more detail.

3.4.3 Analysis and Discussion

Correspondence between Architecture and Problem

The selection of an architecture commits the designer to a particular view of a problem. Like any abstraction, this view emphasizes some aspects of the problem and suppresses others. Booch [Boo86] characterizes the views inherent in object-oriented and functional architectures as follows:

> Simply stated, object-oriented development is an approach to software design in which the decomposition of a system is based upon the concept of an object. An object is an entity whose behavior is characterized by the actions that it suffers and that it requires of other objects. Object-oriented development is fundamentally different from traditional functional methods, for which the primary criteria for decomposition is that each module in the system represents a major step in the overall process.

The issue, of course, is deciding which abstractions are most useful for any particular problem. We have argued that the control view is particularly appropriate for a certain class of problems. In this case, the control view clarifies the design in several ways:

- The control view leads us to respecify the output as the actual speed of the vehicle.
- The separation of control from process makes the model of actual speed explicit and hence more likely to be validated; similarly it raises the question of control authority.
- The explicit element for the control algorithm also sets up a design decision about the kind of control to be exercised (see Section 3.4.3).
- By establishing special relations among components, the control paradigm discriminates among different kinds of inputs, and makes the feedback loop more obvious.
- The control paradigm clearly separates manual operation from automatic operation.
- Determination of the set point is easier to verify when it's separated from control; for example, Booch's design does not appear to reset the desired speed to undefined when the engine is turned off.

Methodological Implications

Object-oriented architectures are supported by associated methodologies. What can we say about methodologies for control-loop organizations and when they are useful? First, a methodology should help the designer decide when the architecture is appropriate. Second, a methodology should help the designer identify the elements of the design and their interactions. This corresponds to instructions for "finding the objects" in object-oriented methodologies. Third, a methodology should help the designer identify critical design decisions. In the case of control, these include potential safety problems.

Åström and Wittenmark give a collection of examples of common solutions for process control problems [AW90]. Each identifies a typical control situation and gives advice

for suitable strategies. They provide a top-down methodology that also serves as the control paradigm for software:

- Choose the control principle.
- Choose the control variables.
- Choose the measured variables.
- Create subsystems.

A methodology should also provide for system modifications. Booch proposes two for the object-oriented design; both would be simple changes in the control-loop design:

- Add a digital speedometer: The wheel pulses are directly available as a control signal that can be picked up by any other component and used independently of the control paradigm. In addition, Section 3.4.1 suggested the creation of an object for current speed.
- Use separate microcomputers for current/desired speed and throttle: The most likely assignment of function to multiple processors would put the control on one and engine-related operations on another. This corresponds directly to the design.

Performance: System Response to Control

Process control provides powerful tools for selecting and analyzing the response characteristics of systems. For example, the cruise controller can set the throttle in several ways [P+84]:

- **On/Off Control:** The simplest and most common mode of control simply turns the process off and on. This is more appropriate for thermostats than throttles, but it could be considered. In order to prevent the power from fluttering rapidly on and off, on/off control usually provides some form of hysteresis (actual speed must deviate from the set point by some amount before control is exercised, or power setting can't be switched more often than a preset limit).
- **Proportional Control:** The output of a proportional controller is a fixed multiple of the measured error. The gain of a cruise controller is the amount by which the speed deviation is multiplied to determine the change in throttle setting. This is a parameter of control. Depending on the properties of the engine, this can lead to a steady-state value not quite equal to the set point or to oscillation of the speed about the set point.
- **Proportional plus Reset Control:** The controller has two parts; the first is proportional to the error, and the second causes the controller output to change as long as an error is present. This has the effect of forcing the error to zero. Adding a further correction based on the derivative of the error speeds up the response but is probably overkill for the cruise-control application.

For each of these alternatives, mathematical models of the system responses are well understood.

Correctness

When software controls a physical system, correctness and safety are critically important. We have seen how the control paradigm's methodology leads the designer to consider the accuracy of design assumptions that have significant safety implications. For cruise control, the possibility of runaway feedback is a significant safety concern.

3.4.4 Summary

Much of the power of a design methodology arises from how well it focuses attention on significant decisions at appropriate times. Methodologies generally do this by decomposing the problem in such a way that development of the software structure proceeds hand in hand with the analysis for significant decisions. This localizes decisions and limits the ripples caused by changes. Our exploration of cruise control has shown that the significant high-level decisions are better elicited by a methodology based on process control than on the more common object-oriented methodology. In particular, the control paradigm separates the operation of the main process from compensation for external disturbances. This separation of concerns yields appropriate abstractions and reveals design issues that might otherwise be neglected.

Cruise control exemplifies a class of design problems in which a real-time process is controlled by embedded software. Conceptually, such processes update the control status continuously. Thinking about these designs explicitly as process-control problems leads the designer to a software organization that separates process concerns from control concerns and requires explicit attention to the appropriateness and correctness of the control strategy. This leads to early consideration of performance and correctness questions that might not otherwise arise.

3.5 Three Vignettes in Mixed Style

3.5.1 A Layered Design with Different Styles for the Layers

The PROVOX™ system by Fisher Controls offers distributed process control for chemical production processes [Fis89]. The process-control capabilities of the system range from simple control loops that control pressure, flow, or levels to complex strategies involving interrelated control loops. Provisions are made for integration with plant management and information systems in support of computer-integrated manufacturing. The system architecture integrates process control with plant management and information systems in a five-level layered hierarchy, shown in Figure 3.22. The right side of the figure shows the software view, and the left side shows the hardware view. Each level corresponds to a different process-management function with its own decision-support requirements.

- **Level 1:** Process measurement and control—direct adjustment of final control elements.
- **Level 2:** Process supervision—operations console for monitoring and controlling Level 1.

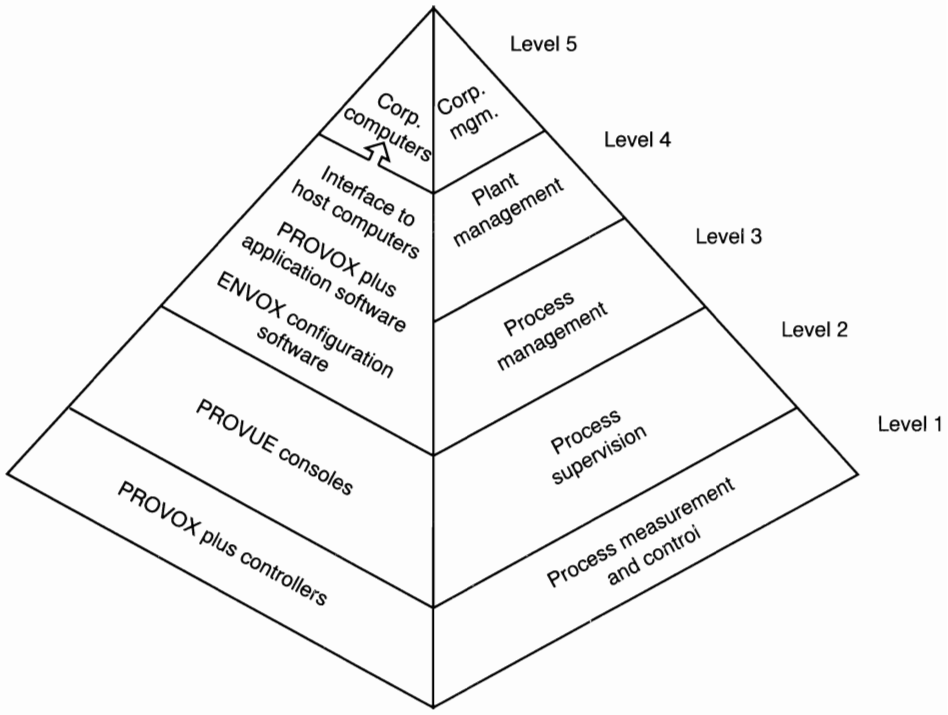

FIGURE 3.22 PROVOX—Hierarchical Top Level

- **Level 3:** Process management—computer-based plant automation, including management reports, optimization strategies, and guidance to the operations console.
- **Levels 4 and 5:** Plant and corporate management—higher-level functions such as cost accounting, inventory control, and order processing/scheduling.

Different computation and response times are required at different levels of this hierarchy. (This is similar to the concerns addressed by the layered architecture for mobile robotics discussed in Section 3.3.3.) Accordingly, different computational models are used. Levels 1 to 3 are object-oriented; Levels 4 and 5 are largely based on conventional data-processing repository models. In this section we examine only the object-oriented model of Level 2 and the repositories of Levels 4 and 5.

For the control and monitoring functions of Level 2, PROVOX uses a set of *points,* or loci of process control. Figure 3.23 shows the canonical form of a point definition; seven specialized forms support the most common kinds of control. Points are, in essence, object-oriented design elements that encapsulate information about control points of the process. The points are individually configured to achieve the desired control strategy. Data associated with a point includes the following: operating parameters, including current process value, setpoint (target value), valve output, and mode (automatic or manual); tuning parameters, such as gain, reset, derivative, and alarm trip-points; and configuration parameters, including tag (name) and I/O channels.

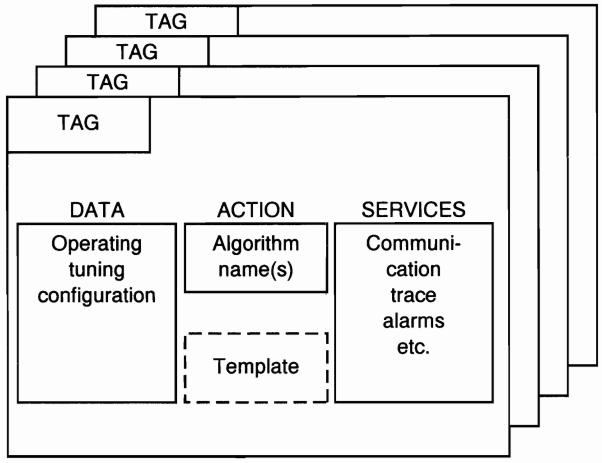

FIGURE 3.23 PROVOX—Object-Oriented Elaboration

In addition, the point's data can include a template for a control strategy. Like any good object, a point also includes procedural definitions such as control algorithms, communication connections, reporting services, and trace facilities. A collection of points implements the desired process-control strategy through the communication services and through the actual dynamics of the process (e.g., if one point increases flow into a tank, the current value of a point that senses tank level will reflect this change). Although the communication through process state deviates from the usual procedural or message-based control of objects, points are conceptually very like objects in their encapsulation of essential state and action information.

Reports from points appear as input transactions to data-collection and analysis processes at higher design levels. The organization of the points into control processes can be defined by the designer to match the process-control strategy. These processes can be further aggregated into Plant Process Areas (points related to a set of equipment, such as a cooling tower) and thence into Plant Management Areas (segments of a plant that would be controlled by single operators).

PROVOX makes provisions for integration with plant management and business systems at Levels 4 and 5. Selection of those systems is often independent of process control design; PROVOX does not itself provide MIS systems directly but does provide for integrating a conventional host computer with conventional database management. The data-collection facilities of Level 3, the reporting facilities of Level 2, and the network that supports distributed implementation suffice to provide process information as transactions to these databases. Such databases are commonly designed as repositories, with transaction-processing functions supporting a central data store—quite a different style from the object-oriented design of Level 2.

The use of hierarchical layers at the top design level of a system is fairly common. This permits strong separation of different classes of functions and clean interfaces between the layers. However, within each layer the interactions among components are often too intricate to permit strict layering.

3.5.2 An Interpreter Using Different Idioms for the Components

Rule-based systems provide a means of codifying the problem-solving skills of human experts. These experts tend to capture problem-solving techniques as sets of situation-action rules whose execution or activation is sequenced in response to the conditions of the computation rather than by a predetermined scheme. Because these rules are not directly executable by available computers, systems for interpreting such rules must be provided. Hayes-Roth has surveyed the architecture and operation of rule-based systems [HR85].

The basic features of a rule-based system, shown in Hayes-Roth's rendering as Figure 3.24, are essentially the features of a table-driven interpreter, as outlined earlier.

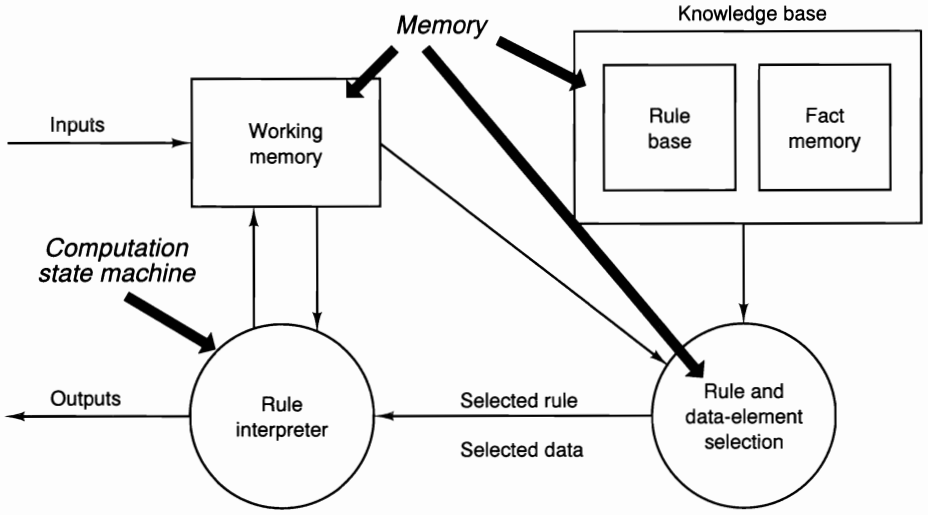

FIGURE 3.24 Basic Rule-Based System

- The *pseudocode* to be executed, in this case the knowledge base
- The *interpretation engine*, in this case the rule interpreter, the heart of the inference engine
- The *control state of the interpretation engine*, in this case the rule and data-element selector
- The *current state of the program* running on the virtual machine, in this case the working memory

Rule-based systems make heavy use of pattern matching and context (currently relevant rules). Adding special mechanisms for these facilities to the design yields the more complicated view shown in Figure 3.25. With this added complexity, the original simple interpreter vanishes in a sea of new interactions and dataflows. Although the interfaces among the original modules remain, they are not distinguished from the newly added interfaces.

However, we can rediscover the interpreter model by identifying the components of Figure 3.25 with their design antecedents in Figure 3.24, as in Figure 3.26. Viewed in this

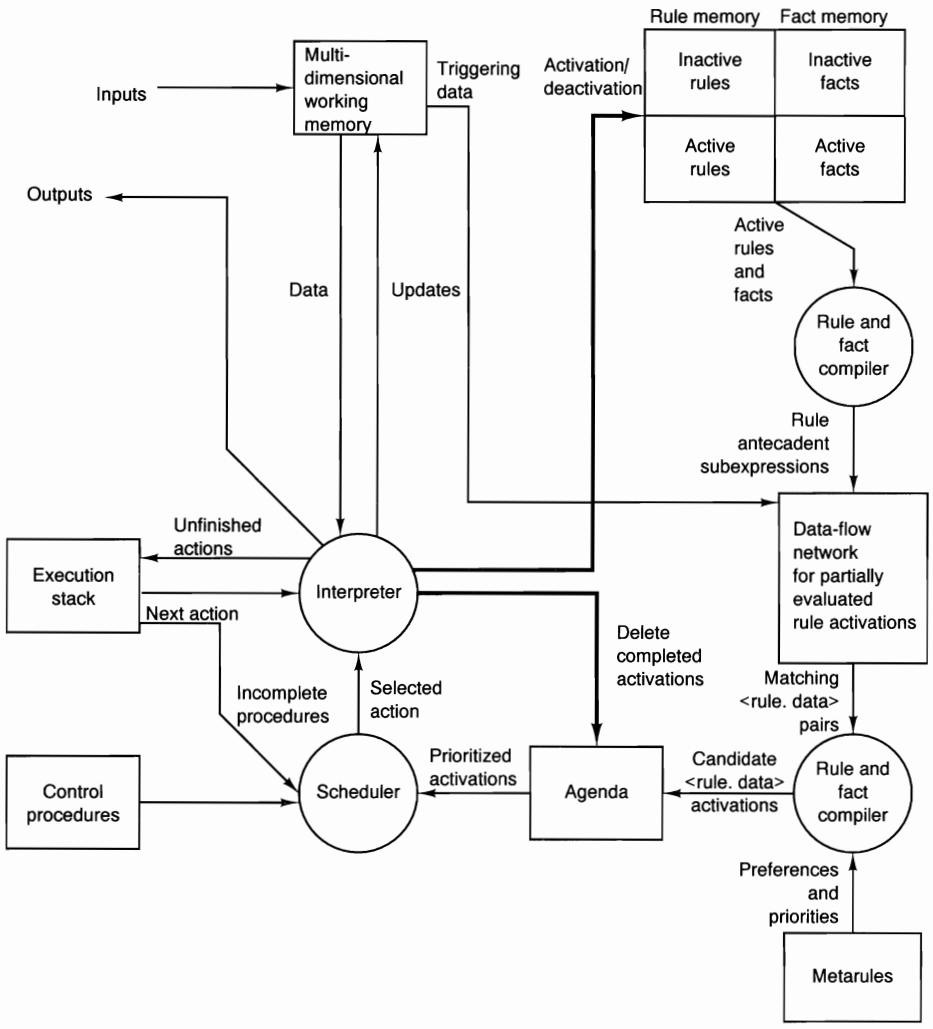

FIGURE 3.25 Sophisticated Rule-Based System

way, the elaboration of the design becomes much easier to explain and understand. For example, we can see the following:

- The knowledge base remains a relatively simple memory structure, merely gaining substructure to distinguish active from inactive contents.
- The rule interpreter is expanded with the interpreter idiom (that is, the interpretation engine of the rule-based system is itself implemented as a table-driven interpreter), with control procedures playing the role of the pseudocode to be executed and the execution stack the role of the current program state.

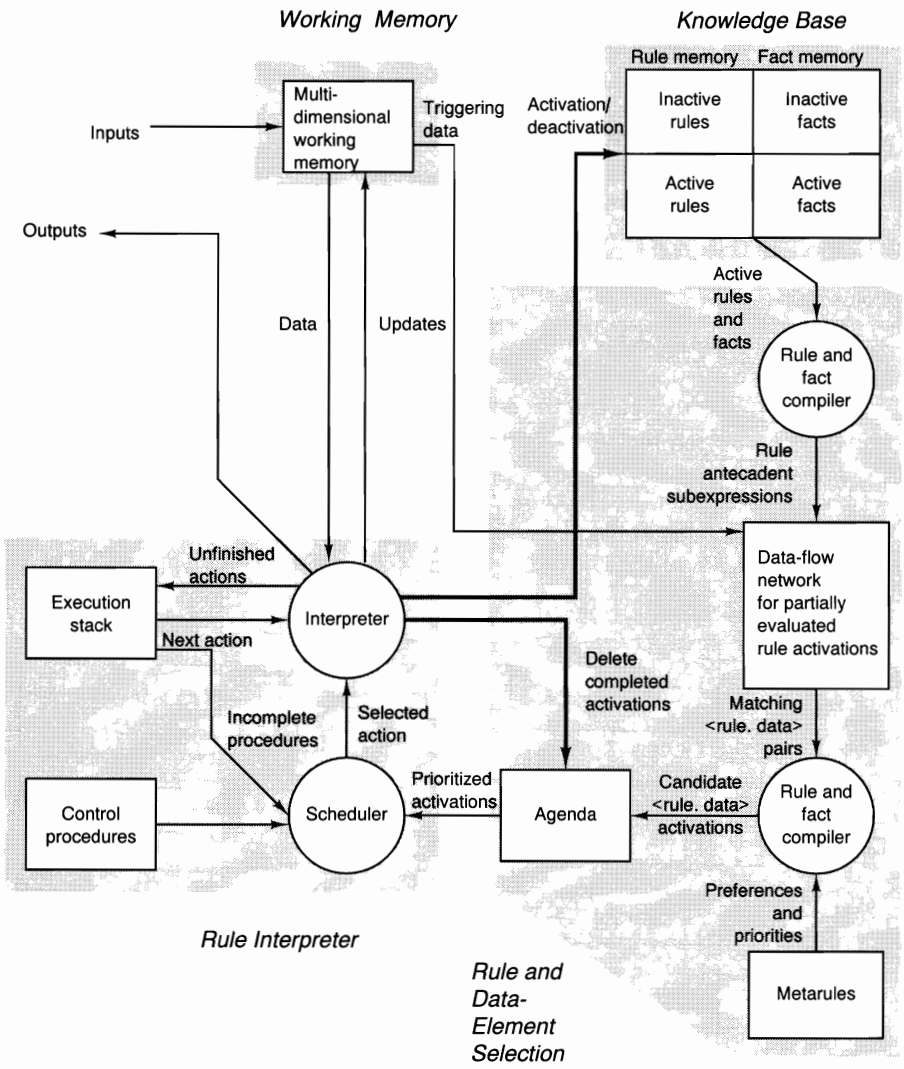

FIGURE 3.26 Simplified Rule-Based System

- "Rule and data-element selection" is implemented primarily as a pipeline that progressively transforms active rules and facts to prioritized activations; in this pipeline the third filter ("nominators") also uses a fixed database of metarules.
- Working memory is not further elaborated.

The interfaces among the rediscovered components are unchanged from the simple model except for the two bold lines over which the interpreter controls activations.

This example illustrates two points. First, in a sophisticated rule-based system the elements of the simple rule-based system are elaborated in response to execution characteristics of the particular class of languages being interpreted. If the design is presented in this way, the original concept is retained to guide understanding and later maintenance. Second, as the design is elaborated, different components of the simple model can be elaborated with different idioms.

Note that the rule-based model is itself a design structure: it calls for a set of rules whose control relations are determined during execution by the state of the computation. A rule-based system provides a virtual machine—a rule executor—to support this model.

3.5.3 A BLACKBOARD GLOBALLY RECAST AS AN INTERPRETER

The blackboard model of problem solving is a highly structured special case of opportunistic problem solving. In this model, the solution space is organized into several application-dependent hierarchies, and the domain knowledge is partitioned into independent modules of knowledge that operate on knowledge within and between levels [Nii86]. Figure 2.5 showed the basic architecture of a blackboard system and outlined its three major parts: knowledge sources, the blackboard data structure, and control.

The first major blackboard system was the HEARSAY-II speech-recognition system. Nii's schematic of the HEARSAY-II architecture appears in Figure 3.27. The blackboard

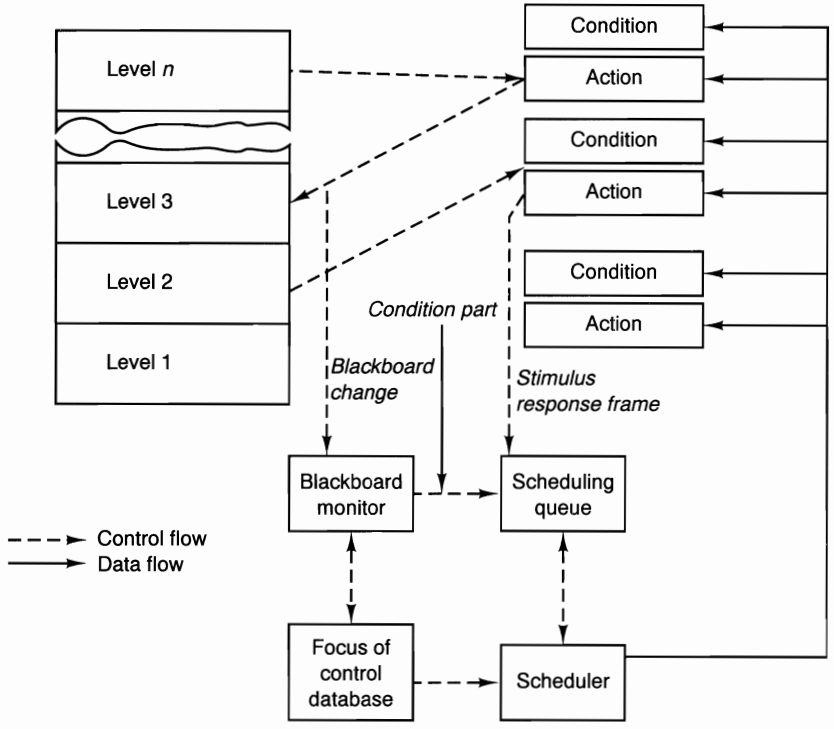

FIGURE 3.27 Hearsay-II

structure is a six- to eight-level hierarchy in which each level abstracts information on its adjacent lower level, and blackboard elements represent hypotheses about the interpretation of an utterance. Knowledge sources correspond to such tasks as segmenting the raw signal, identifying phonemes, generating word candidates, hypothesizing syntactic segments, and proposing semantic interpretations. Each knowledge source is organized as a condition part that specifies when it is applicable and an action part that processes relevant blackboard elements and generates new ones. The control component is realized as a blackboard monitor and a scheduler; the scheduler monitors the blackboard and calculates priorities for applying the knowledge sources to various elements on the blackboard.

HEARSAY-II was implemented between 1971 and 1976 on DEC PDP-10s. These machines were not directly capable of condition-triggered control, so it should not be surprising to find that an implementation provides the mechanisms of a virtual machine that realizes the implicit invocation semantics required by the blackboard model.

Figure 3.27 elaborates the individual components of Figure 2.5 and also adds components for the previously implicit control component. In the process, the figure becomes rather complex, because it is now illustrating two concepts: the blackboard model and realization of that model by a virtual machine. The blackboard model can be recovered as in Figure 3.28 by suppressing the control mechanism and regrouping the conditions and

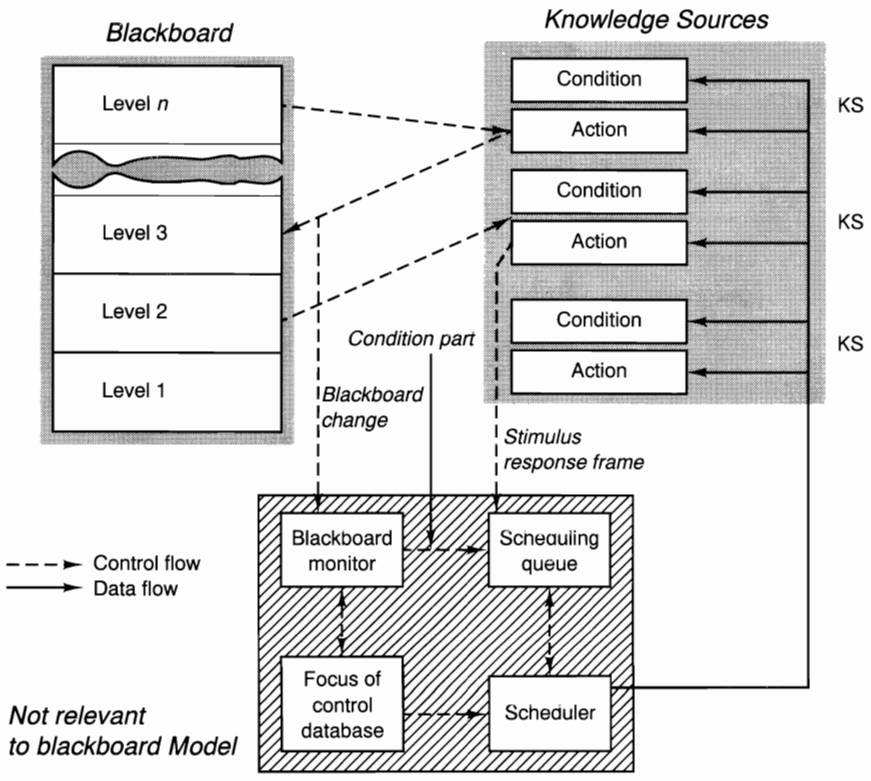

FIGURE 3.28 Blackboard View of Hearsay-II

actions into knowledge sources. The virtual machine can be seen to be realized by an interpreter by using the assignment of function in Figure 3.29. Here the blackboard corresponds cleanly to the current state of the recognition task. The collection of knowledge sources roughly supplies the pseudocode of the interpreter; however, the actions also contribute to the interpretation engine. The interpretation engine includes several components that appear explicitly in Figure 3.27: the blackboard monitor, the focus-of-control database, and the scheduler, as well as the actions of the knowledge sources. The scheduling queue corresponds roughly to the control state. To the extent that execution of conditions determines priorities, the conditions contribute to rule selection as well as forming pseudocode.

Here we see a system initially designed with one model (blackboard, a special form of repository), and then realized through a different model (interpreter). The realization does not involve a component-by-component expansion as in the previous two examples; the view as an interpreter involves a different aggregation of components than the view as blackboard.

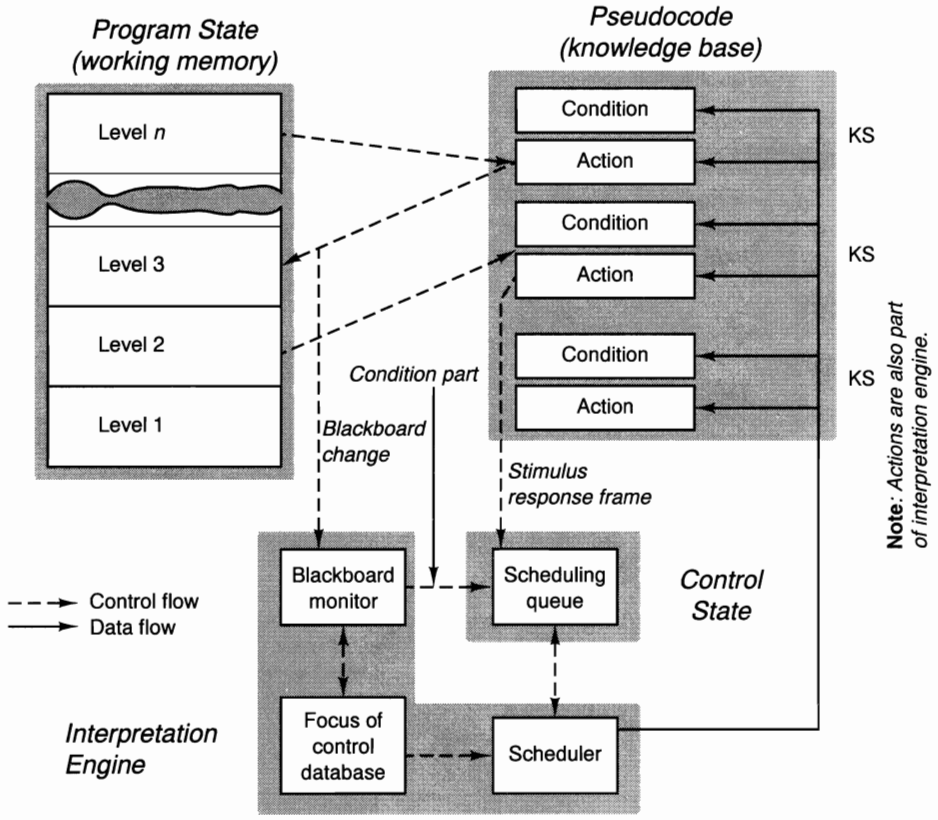

FIGURE 3.29 Interpreter View of Hearsay-II