Chapter 7

Functional Architecture Development

7.1 INTRODUCTION

Time-tested engineering of systems has shown that the design process for a system has to consider more than the physical side of the system; the functions or activities that the system has to perform are a critical element for the design process to be successful on a consistent basis. This is not to say that the designs of functions and physical resources for the system proceed independently; they cannot. However, for success these two design elements must be equal partners in the design process, providing checks on each other and complementing each other's progress. The functional architecture of a system contains a hierarchical model of the functions performed by the system, the system's components, and the system's configuration items (CIs); the flow of informational and physical items from outside the system through the transformational processes of the system's functions and on to the waiting external systems being serviced by the system; a data model of the system's items; and a tracing of input/output requirements to both the system's functions and items. Note that functional architecture is called a logical architecture by many people.

There are a number of key terms that need to be defined as part of this chapter. Early in the chapter distinctions are drawn between modes, states, and functions for a system. There is considerable difference in meaning in the literature on systems and software related to the terms of mode, state, and function; to be clear in our discussions these terms have to be defined specifically for use in this book. A system *mode* is a distinct operational

The Engineering Design of Systems: Models and Methods, Second Edition. By Dennis M. Buede

capability of the system; this capability may use either the full or a partial set of the system's functions. An example is the initialization mode versus the full operational mode for your personal computer. A state is a modeling description of the status of the system at a moment in time, as defined by the values on a set of state variables. A function is an activity or task that the system performs to transform some inputs into outputs. Late in the chapter distinctions are drawn between failure, error, and fault. Failure is a deviation between the system's behavior and the system's requirements. An error is a problem with the state of the system that may lead to a failure. A fault is a defect in the system that can cause an error.

After the initial definition of key terms for describing a functional architecture, Section 7.3 defines the method for developing a functional architecture using an IDEF0 (Integrated Definition for Function Modeling) model. This model of the development process for a functional architecture is explained, followed by a discussion of using a decomposition process versus a composition process.

Section 7.4 discusses approaches, examples, and issues for defining a system's functions; this discussion is very important because the modeling of a system's functions is not a common skill that is found in engineers. Section 7.4.1 describes several approaches for developing functional decompositions. Section 7.4.2 addresses an important theme of this entire book; namely, there is always more than one system involved in the engineering of a system. Examples of functional decompositions for several phases of the life cycle of a system are presented. Third and perhaps most importantly, the concepts of feedback and control in a system's functions are introduced in Section 7.4.3. A common hypothesis of many systems engineers is that most systems fail because of inadequate design of the feedback and control functionality into the system. Finally Section 7.4.4 provides a discussion of evaluation topics that are useful for critiquing a functional architecture; critical examination of any model is important for engineers, and Section 7.4.4 provides some metrics for doing so.

Section 7.5 defines the data collection activities associated with developing the functional model of a system. This section provides some guidance on the types of data to collect, on the need to try alternate modeling ideas, and then on the evaluation of these model alternatives in terms of the need to capture the system's capabilities and communicate these capabilities to both the stakeholders and the discipline engineers.

Then in Section 7.6, the introduction of fault tolerance functionality in terms of the functional architecture is described. Adding fault tolerance functionality is very important to the success of most systems and is critical to the success of some systems, for example, air traffic control and life support. Error detection, damage confinement, error recovery, and fault isolation and reporting are the types of functions discussed here.

Finally tracing input/output requirements to functions and items in the functional architecture is described in Section 7.7. This last activity is critical to the process of developing specifications for each component that comprises the

system in such a way that the component specifications are directly related and traceable to the System's Requirements Document (SRD).

The methods described in this chapter relate to the development of the functional architecture. The method relating to defining the elements of the functional architecture is described in detail and presented as an IDEF0 model. In addition, the chapter provides a data collection process for defining the functional architecture based upon the fundamental approaches behind the structured analysis and design technique that led to IDEF0.

The primary modeling technique relied upon in this chapter is IDEF0, as presented in Chapter 3. In addition, feedback and control models are introduced for evaluating the state of the system and improving the system's performance.

The exit criterion for the development of the functional architecture is the coherent matching of the input/output requirements with the functions and items in the functional architecture. Every input/output requirement should be traced to at least one function and one item in the functional architecture. In addition, every function associated with an external item in the functional architecture should have at least one input/output requirement traced to the function, as should every external item. Recall that all elements of the system's architectures are developed in increasing layers of detail, so the exit criterion for the functional architecture will be applied with each completion of a layer of detail.

7.2 DEFINING TERMINOLOGY FOR A FUNCTIONAL ARCHITECTURE

This section defines the concepts of system modes, states, and functions, followed by simple and complete functionalities. Modes and functionalities have long been thought to be critical to the establishment of an understanding of the logical aspects of a system.

A system *mode* is defined to be a distinct operating capability of the system during which some or all of the system's functions may be performed to a full or limited degree. Other authors [Wymore, 1993] define the modes of a system to be functions of the system; that is not the definition presented here. All systems have at least one standard or fully operational mode. Most systems have operating modes during which they are partially operational. For example, an elevator system has a maintenance mode during which one or more of the elevator cars can be stopped for maintenance, while the others continue in operation. Often systems have start-up and shutdown modes. A laptop computer, on which I am writing this paragraph, has several modes of operation that correspond to the power that is being supplied; all of the laptop's functions are available in each of these modes, but not with the same performance characteristics. Finally, systems often have a number of unwanted failure modes; car manufacturers have installed switches to enable the use of an extra gallon of gasoline to try to avoid the failure mode of no gas.

The *state of the system* is commonly defined to be a static snapshot of the set of metrics or variables needed to describe fully the system's capabilities to perform the system's functions. The system is progressing through a constantly changing series of states as time progresses. In other words, the state of a system is the values of a long list of variables, called *state variables*, at a specific point in time. This list of state variables contains all of the information needed to determine the system's ability to perform the system's functions at that point in time. The list of state variables does not change over time, but the values that these variables take does change over time. The variables can be continuous or discrete. As an example, the state variables for a laptop computer might include power input rate from the outside, power level of the battery, input rate for each input source (keyboard, modem, network), output rate for each output device (parallel port, serial port, modem, network, screen), central processing unit (CPU) usage, and free hard disk space.

A *function*, on the other hand, is a process that takes inputs in and transforms these inputs into outputs. A function is a transformation, including the possible changing of state one or more times. Every function has activation and exit criteria. The activation criterion is associated with the availability of the physical resources, not necessarily with the start of the transformation activity. The function is activated as soon as the resource for carrying out the function is available. When the appropriate triggering input arrives, the function is then ready to receive the input and begin the transformation process. The activation criterion for the function then is the combination of the availability of the physical resource and the arrival of the triggering input. The exit criterion of a function determines when the function has completed its transformation tasks.

Chapters 3 and 12 cover a number of behavioral modeling techniques that address issues related to the activation and deactivation of functions, both as the result of the natural transformation processes associated with functions as well as the control structure that controls the functional processing and causes the system to change modes. Included in Chapter 12 are behavior diagrams, finite-state machines (state-transition diagrams), statecharts, control flow diagrams, and Petri nets. Note that state-transition diagrams and statecharts are related to the definition of mode being used here rather than the definition of state.

Must a function represent a dynamic process? Can a function be used to represent a constant process? All of the functions that are shown in Appendix B for the elevator case study represent a dynamic function; that is, inputs enter the function over a given time period and some time later the outputs emerge. Does a pedestal that is holding a vase perform a function? The perspective taken here is that the pedestal does perform a function in this case; if the pedestal fails due to fatigue or an earthquake, then a dynamic process that the system is trying to prevent will occur (the vase will crash to the ground and be ruined).

A *functionality* is a set of functions that is required to produce a particular output. Now we define simple and complete functionalities:

Simple Functionality: an *ordered* sequence of functional processes that operates on a single input to produce a specific output. Note there may be many inputs required to produce the output in question, but this simple functionality is only related to one of the inputs. As a result the simple functionality may not include all of the necessary functional processes needed to produce the output. Nor does this simple functionality trace the only possible sequence of these functional processes. Note each simple functionality has a specific order associated with the functions that define the simple functionality; for this reason we cannot say that a simple functionality is an element of the power set of functional processes because there is no order associated with an element of the power set. Also we cannot say that this simple functionality is a mathematical function since a given input may be mapped into more than one output.

Complete Functionality: a complete set of coordinated processes that operate on all of the necessary inputs for producing a specific output. There is usually no specific order associated with the complete set of functional processes; however a *partial order* of the functional activities can be established because some functions will usually have to be activated and completed before some others. The complete functionality cannot be an element of the power set of functional processes because there is still some order information associated with the functions in the complete functionality. There is no order information in the sets of functions that comprise the power set of functions. There is a well-defined set of inputs, which is one element of the Cartesian product (or n-tuple) of inputs, and is uniquely associated with the output. This output is also an element of the Cartesian product, or m-tuple, of outputs.

A *functional architecture* can be defined at several levels of detail:

1. A logical architecture that defines what the system must do, a decomposition of the system's top-level function. This very limited definition of the functional architecture is the most common and is represented as a directed tree.
2. A logical model that captures the transformation of inputs into outputs using control information. This definition adds the flow of inputs and outputs throughout the functional decomposition; these items that comprise the inputs and outputs are commonly modeled via a data model (see Chapter 12). An IDEF0 model without any mechanisms is used as the modeling technique in this chapter to represent the functional architecture at this level of detail. Other modeling techniques in Chapter 12 for data and process modeling could also be used.

3. A logical model of a functional decomposition plus the flow of inputs and outputs, to which input/output requirements have been traced to specific functions and items (inputs, outputs, and controls).

An example of a functional architecture for the elevator case study can be downloaded from the following web site:
http://www.theengineeringdesignofsystems.com.

7.3 FUNCTIONAL ARCHITECTURE DEVELOPMENT

IDEF0 is used here as the graphical process modeling technique to represent the first elements of the functional architecture defined above. In Chapter 12 several alternate graphical process-modeling techniques are presented that can be used in place of or in addition to IDEF0. IDEF0 was chosen because IDEF0 has well-defined, standardized syntax and semantics that distinguish between the inputs to be transformed into outputs and the control information that guides the transformation process. In addition, IDEF0 has a place to represent the physical architecture, namely the mechanisms. Later the allocated architecture can be illustrated using the mechanisms within IDEF0.

It is possible to complete the functional architecture without resorting to any graphical techniques. Text and tables are sufficient to represent all of the information conveyed by any of the graphical techniques. However, Jones and Schkade [1995] provide convincing evidence that most systems and software professionals resort to graphical techniques during the system or software engineering process. The graphical techniques contain much greater information in a format that can be communicated more effectively and efficiently.

7.3.1 Functional Architecture Process Model

Figure 7.1 shows the IDEF0 model for the development of a functional architecture. See the full IDEF0 model for engineering a system in Appendix B. The approach shown in this figure begins by creating many function sequences, or simple functionalities, that satisfy the scenarios in the operational concept. These functionalities are created by shining a light into the black box of Chapter 6, thus turning the black box into a "white" box. Now the functions that are needed to transform system inputs into system outputs become visible.

Then the engineer synthesizes these many simple functionalities into a functional decomposition; this synthesis can be accomplished via a top-down decomposition or a bottom-up aggregation. Section 7.4 examines these two approaches in more detail. In practice this second step of defining the functional decomposition combines both aggregation and decomposition. The flow of inputs and outputs from outside the system are added, and the necessary internal items are added, creating a functional model. Before distributing this functional model widely for comment (step three) the scenarios

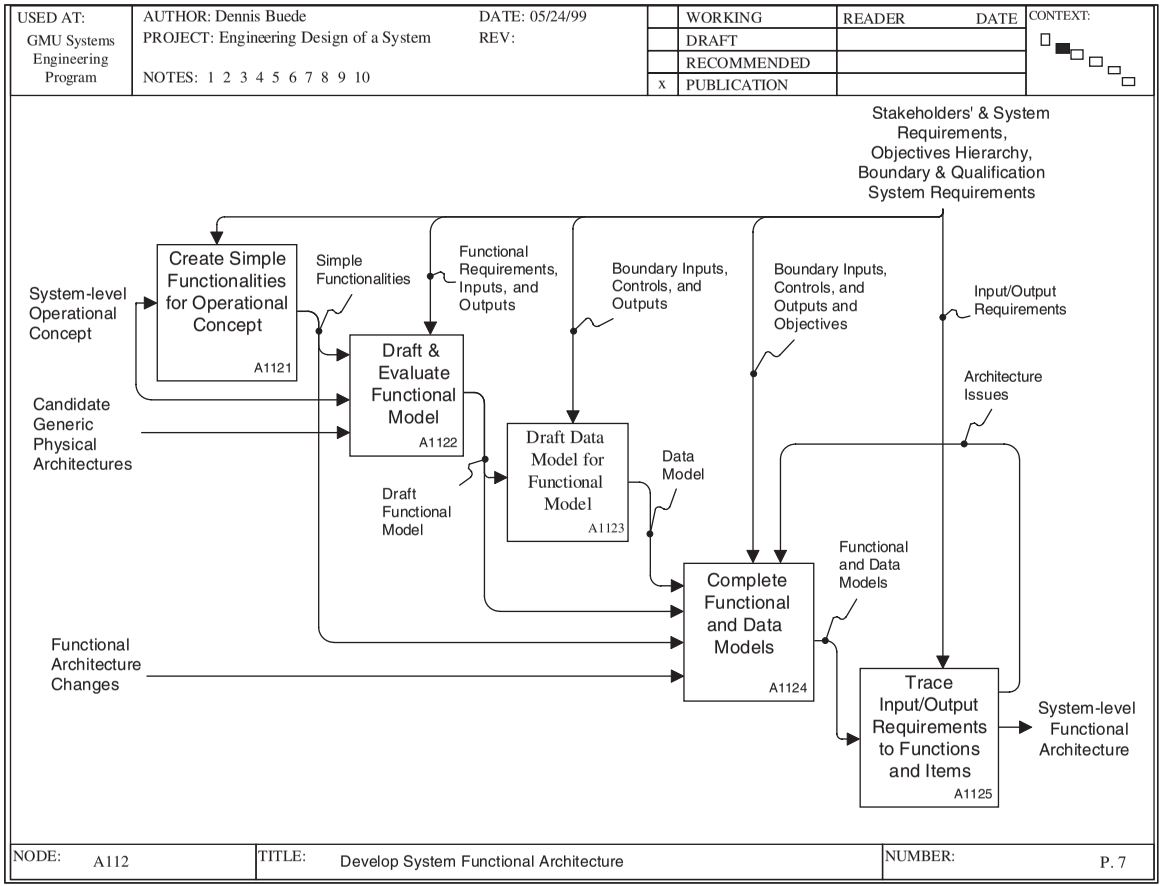

FIGURE 7.1 Process for developing a functional architecture.

from the operational concept are used once more to test the draft decomposition and ensure that the functional model is consistent with these scenarios.

The third step addresses the data or items that serve as inputs or outputs to the various functions of the functional architecture. For computer-intensive systems developing a data mode is critical (see Chapter 12) so that the relations among the various items flowing through the system are understood at the level needed for a successful design.

The fourth step is the solicitation of the opinions of other engineers and stakeholders about missing functions or alternate decompositions that are more meaningful than have been produced during the second and third steps. During this third step the allocated architectural activity, which combines the functional and physical architectures, is proceeding. Feedback from the development of the allocated architecture often causes changes to the functional model, changes that enable the functional model and the physical architecture to match more closely. (Chapter 9 discusses these issues in more detail.)

The final step in the development of the functional architecture addresses the tracing of input/output requirements to both the functions in the data model and the items (data elements) flowing through this functional model. Each input (output) requirement is traced to those functions that have been designated as receiving (producing) the respective input (output). Similarly, each input (output) requirement is traced to the item for which the requirement is defined. The functional requirements are traced to the top-level system function because this top-level function is responsible for accomplishing these subfunctions. Each external interface requirement is traced to each item that will be delivered to the system (or carried away from the system) by that interface. In addition, each external interface requirement is traced to the function that is receiving the input or sending the output that has been traced to that same external interface. This process of tracing input/output requirements often raises issues about the structure of the functional decomposition, leading to possible changes in this decomposition. By tracing the input/output requirements to functions and data in the functional architecture, these requirements are being "flowed down" so that the allocated architecture will have all requirements associated with the elements of specifications that are developed for individual system components.

7.3.2 Decomposition Versus Composition

Decomposition, often referred to as top-down structuring, begins with the top-level system function and *partitions* that function into several subfunctions. This decomposition process must conserve all of the inputs to and outputs from the system's top or zero-level function. By conserve, we mean use/produce all and add no new ones. Next, each of the several first-level functions is decomposed (partitioned) into a second level set of subfunctions. Note that not every function must be decomposed; only those for which additional insight into the production of outputs is needed should be partitioned.

The success of decomposition is predicated on having a sound definition of the top-level function of the system and the associated inputs and outputs, that is, a compete set of requirements. The benefit of having an external systems diagram is to achieve this complete set of requirements. A major difficulty of decomposition is the partitioning process to develop the subfunctions of the system is somewhat unguided. Section 7.4 provides some guidance for this decomposition. The best decomposition is usually one that will match the partitioning of the system's physical resources, the physical architecture. This way the flow of data and physical items that cross the internal interfaces between components will be clearly identified.

The opposite approach, composition, is a bottom-up approach. With composition one starts by identifying the simple functionalities associated with simple scenarios involving only one of the outputs of the system. Each functionality is a sequence of input, function, output–input, …, function, output–input, function, output–input, function, output. The functions in the functionality are all functions of the system and are relatively low-level functions in the functional hierarchy. These functions usually show up in third, fourth, or even lower levels of the hierarchy. For complex systems this initial step is a substantial amount of work. After the many functionalities have been defined, one begins the process of grouping the functions in the functionalities into similar groups. These groups are aggregated into similar groups; this process continues until a hierarchy is formed from bottom to top.

The advantage of the composition approach is that the composition process can be performed in parallel with the development of the physical architecture so that the functional and physical hierarchies match each other. Second, this approach is so comprehensive that the approach is less likely to omit major functions. The drawback is that the many functionalities must be easily accessible during the composition process so that all of this work can be successfully used; the simple functionalities are often pasted on the walls of a large conference room. The composition method dates back to the 1960s and 1970s when systems engineering was in its infancy; many systems engineers continue to prefer this approach. There is no empirical evidence that either the composition or decomposition approach is better than the other.

Ultimately, using a combination of decomposition and composition approaches is wisest. This is sometimes referred to as middle-out. Often, one makes use of simple functionalities associated with specific scenarios defined in the operational concept to establish a "sense" of the system. Then positing a top-level decomposition that is likely to match the top-level segmentation of the physical architecture is common before proceeding to do decomposition that is reinforced by periodic reference to the functionalities to assure completeness.

Decomposition is efficient and often successful when the system is an update or variation of an existing system. Composition is strongly recommended when the system is unprecedented or a radical departure of an existing system.

Before proceeding, it is important to discuss some valuable properties of the functional hierarchy. Besides the obvious design implications that are

embodied in this hierarchy, the hierarchy is also important as a communication tool. This communication is important for both other engineers and the stakeholders. For this reason, limiting the number of functions at each node in the functional tree to a number that enhances communication is advisable; large numbers of functions at a given level of a decomposition turn any graphical technique into a "bowl of spaghetti," where the functions are the meat balls and the arrows are the spaghetti.

7.4 DEFINING A SYSTEM'S FUNCTIONS

As discussed above, assigning functions in the functional architecture in total-to-one and only one resource in the physical architecture is best. Clearly, the functional and physical architectures cannot be developed independently and satisfy this property. In fact, there are times when the decision to allocate a particular function to one of several resources has substantial performance implications and is the subject of one or more trade studies. The bottom line is that the functional architecture may be revised several times as the allocated architecture is finalized. Therefore, focusing on getting the functional hierarchy right the first time is improper since this is an impossible task.

7.4.1 Approaches for Defining Functions

There are a number of keys one can use to partition a function into subfunctions. At the top of the hierarchy we would expect to see functions devoted to the system's operating modes, if there are any. For functions that have multiple outputs, we could partition the function into subfunctions that correspond with the production of each output. Similarly, we could key on the inputs and controls to find a partition of the function. More appropriate than either of these is to decompose on the basis of stimulus–response threads that pass through the function being decomposed. Finally, there is often a natural sequence of subfunctions for a particular function. For example, at the bottom of the functional architecture we would expect to see functions such as receive input, store input, and disseminate input or retrieve output, format output, and send output.

Hatley and Pirbhai [1988] developed an architectural template for representing the physical architecture of the system; Figure 7.2 shows the physical segments of the template. This template suggests the creation of a generic partition of six subfunctions, one for each of the Hatley–Pirbhai components. These six generic functions could be used in any functional architecture:

- Provide user interface: those functions associated with requesting and obtaining inputs from users, providing feedback that the inputs were received, providing outputs to users, and responding to the queries of those users

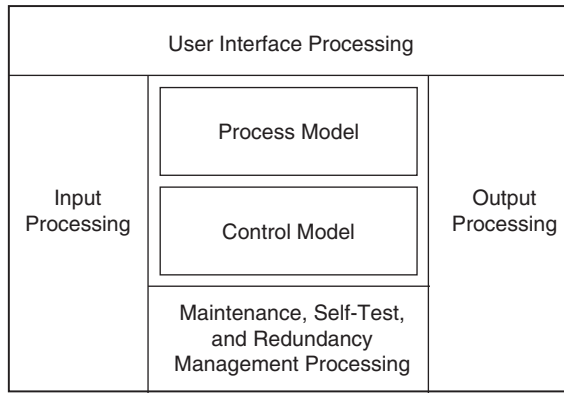

FIGURE 7.2 Architecture template of Hatley and Pirbhai [p. 195, 1988].

- Format inputs: those functions needed to receive inputs from external interfaces (nonhumans), and other nonhuman system components and to process (e.g., analog-to-digital conversion) those inputs to put them into a format needed by the system's processing functions
- Transform inputs into outputs: the major functions of the system
- Control processing: those functions needed to control the processing resources or the order in which these processing functions should be conducted
- Format outputs: those functions needed to convert the system's outputs into the format needed by the external interfaces or other nonhuman system components and then place those outputs onto the appropriate interface
- Provide structural support, enable maintenance, conduct self-test, and manage redundancy processing: those functions needed to perform internal support activities, respond to external diagnostic tests, monitor the system's functionality, detect errors, and enable the activation of standby resources

This partition is a very valid approach at the top of the functional architecture; the author has used this approach several times to initiate decomposition with success. Most systems would have all or nearly all of these functions as an initial partition. Figure 7.3 uses the Hatley–Pirbhai template to show the four top-level functions of the elevator case study, which can be downloaded from the following web site: http://www.theengineeringdesignofsystems.com.

As the decomposition of system functions proceeds, we would expect to find smaller subsets of these six generic functions being embedded within each of the higher level functions. Figure 7.4 renames the Hatley–Pirbhai [1988] partition

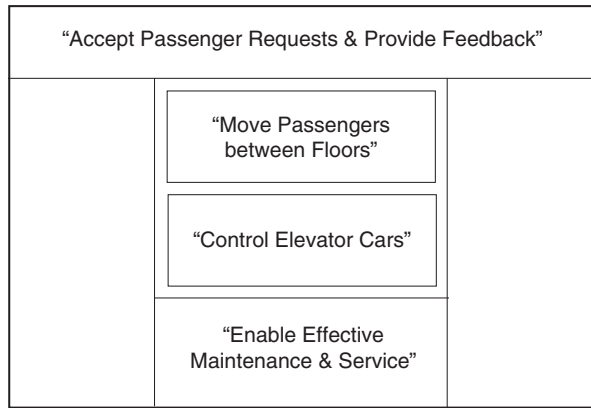

FIGURE 7.3 Elevator functions within the Hatley–Pirbhai template.

as functions and illustrates the functional decomposition by showing likely decompositions within the top level functions; the top-level decomposition of the system function is in the middle of the figure.

McMenamin and Palmer [1984] describe a system's functions as being composed of essential or fundamental activities and custodial activities. All but one of the functions implied by the Hatley and Pirbhai [1988] template are fundamental activities. The function, ''enable maintenance, conduct self-test, and manage redundancy processing,'' performs custodial activities. Additional custodial activities that could be embedded in this function are the provision of structural support, maintenance of information archives, provision of security services, and so forth. In addition, custodial activities maintain the system's memory so the system knows what it needs to know to perform its fundamental activities. This knowledge is called the essential memory of the system; examples include the storage of data items between the time they become available and the time they are used by the fundamental activities. McMenamin and Palmer [1984] recommend separating the custodial activities and the fundamental activities. This separation is not completely possible at the top-level with the taxonomy suggested by the Hatley–Pirbhai [1988] template, nor is this separation often desirable at this high level. However, achieving this separation at lower levels of the functional decomposition is possible and desirable.

Baylin [1990] provides a number of interesting insights into modeling the functional aspects of a system by focusing on the system's objectives. The purpose of any system is to achieve the objectives that have been defined for that system. As a result the engineer of a system would be foolish not to use the system's objectives as a guide for defining the top-level functions of the system.

Many engineers involved in developing systems have read and suggested Miller's [1978] classic titled *Living Systems* as a guide for defining the functions of a system. Miller examines seven levels of systems that range from a cell through a supranational system, and include an organ, an organism, a group, an

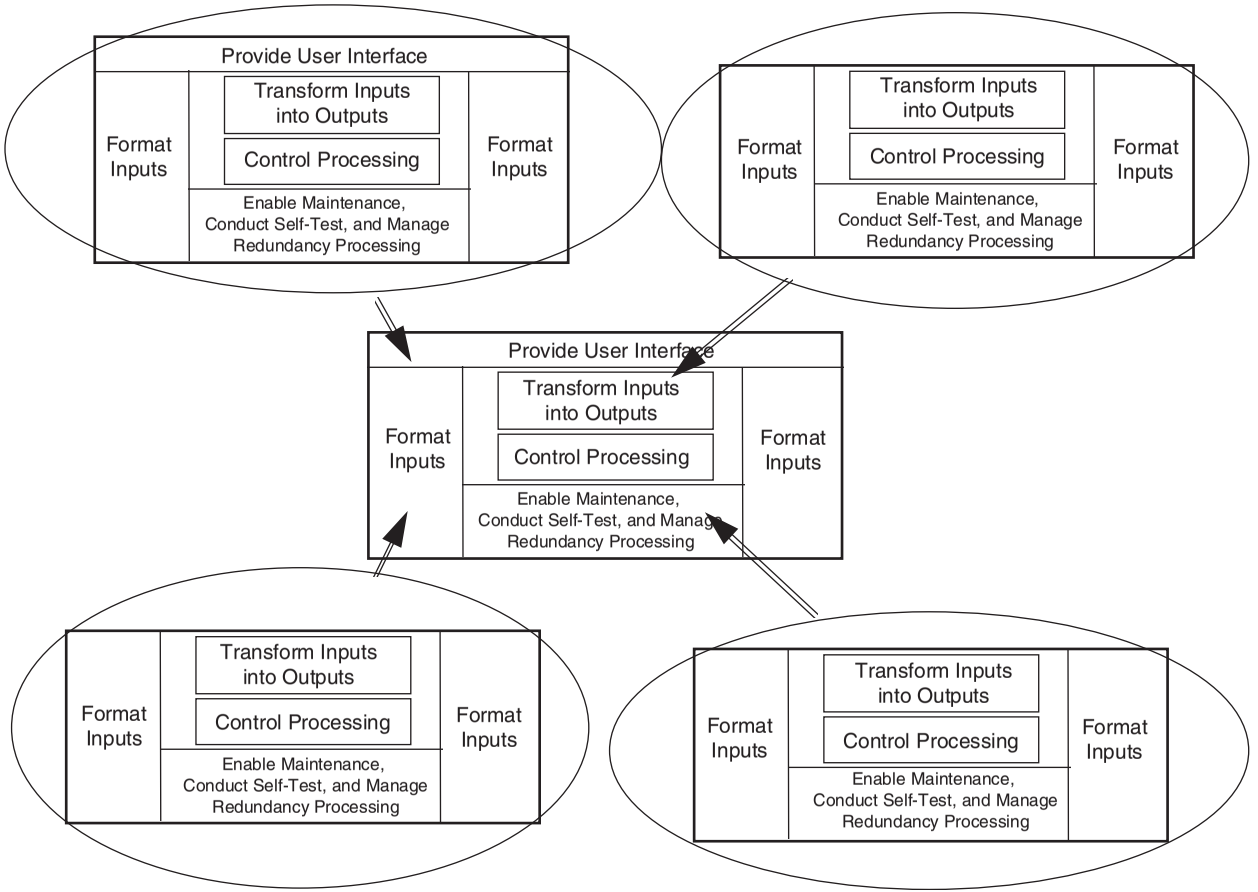

FIGURE 7.4 Exemplary functional decomposition.

organization, and a society. One of Miller's claims is that there are 19 subsystems that must be part of any of these living systems. In fact, Miller defines these 19 subsystems in terms of the function that each performs (see Table 7.1); leading the reader of *Living Systems* to believe that it is the 19 functions that are most useful to engineers of human-designed systems. One key to Miller's study or living systems is his assertion that these systems either process matter-energy or information or both. The top two functions in Table 7.1 address the processing of both matter-energy and information. The functions on the left half of Table 7.1 process matter-energy; while those on the right process information. There are blanks left in the table so that functions on the left and right that are similar can be opposite each other. This assertion is key to understanding the two columns of subsystems and related functions in Table 7.1.

The common concepts for defining a partition of a function are system modes, function outputs, function inputs and controls, system objectives, stimulus-response threads, and the functional template based upon the Hatley–Pirbhai [1988] architecture template.

7.4.2 Typical Functional Decompositions by Life-Cycle Phase

This section suggests functional hierarchies and segments of functional hierarchies for the development and the manufacturing phases of the system's life cycle. The previous section dealt with the operational phase of the life cycle.

Duffy and Buede [1996] suggest structuring the management portion of the *development phase* into three major activities—formulate the development strategy, execute the development strategy, and evaluate the results of the development activity. Formulating the development strategy has as many elements of a development strategy as needed. Common elements of the development strategy are the procurement, engineering or technical, financing, communication, technology development, and testing strategies. Other elements may include the regulatory and risk mitigation strategies. The IDEF0 model of the systems engineering design and integration process in Appendix B demonstrates the execution of the engineering elements of the development strategy.

Dietrich [1991, p. 886] defines *manufacturing* as "using resources to perform operations on materials to produce products." A *manufacturing system* is a "set of resources used to manufacture some product, together with the associated information system and any behavioral requirements imposed by the owners of the resources." The products being produced are the primary outputs of this phase; inputs are defined to be bulk material; internal items are called work-in-progress (WIP). WIP is material upon which some value-added operations have been performed. Seven types of generic manufacturing functions are defined, based upon the types of bulk material, WIP, and primary outputs:

- *Bulk Operation:* manipulate bulk material to produce other bulk material.
- *Kitting Operation:* transform one or more bulk materials into one or more units of WIP.

TABLE 7.1 Subsystems and Functions of Living Systems [after Miller, 1978]

Subsystems which Process Both Matter-Energy and Information

1. *Reproducer*, the subsystem which is capable of giving rise to other systems similar to the one it is in.
2. *Boundary*, the subsystem at the perimeter of a system that holds together the components which make up the system, protects them from environmental stresses, and excludes or permits entry to various sorts of matter-energy and information.

Subsystems which Process Matter-Energy	*Subsystems which Process Information*
3. *Ingestor*, the subsystem which brings matter-energy across the system boundary from the environment.	11. *Input transducer*, the sensory subsystem which brings markers bearing information into the system, changing them to other matter-energy forms suitable for transmission within it.
	12. *Internal transducer*, the sensory subsystem which receives, from subsystems or components within the system, markers bearing information about significant alterations in those subsystems or components, changing them to other matter-energy forms of a sort which transmitted within it.
4. *Distributor*, the subsystem which carries inputs from outside the system or outputs from its subsystems around the system to each component.	13. *Channel and net*, the subsystem composed of a single route in physical space, or multiple interconnected routes, by which markers bearing information are transmitted to all parts of the system.
5. *Converter*, the subsystem which changes certain inputs to the system into forms more useful for the special processes of that particular system.	14. *Decoder*, the subsystem which alters the code of information input to it through the input transducer or internal transducer into a "private" code that can be used internally by the system.
6. *Producer*, the subsystem which forms stable associations that endure for significant periods among matter-energy inputs to the system or outputs from its converter, the materials synthesized being for growth, damage repair, or replacement of components of the system, or for providing energy for moving or constituting the system's outputs of products or information markers to its suprasystem.	15. *Associator*, the subsystem which carries out the first stage of the learning process, forming enduring associations among items of information in the system.

(*Continued*)

TABLE 7.1. Continued

Subsystems which Process Both Matter-Energy and Information	
7. *Matter-energy storage*, the subsystem which retains in the system, for different periods of time, deposits of various sorts of matter-energy.	16. *Memory*, the subsystem which carries out the second stage of the learning process, storing various sorts of information in the system for different periods of time. 17. *Decider*, the executive subsystem which receives information inputs from all other subsystems and transmits to them information outputs that control the entire system. 18. *Encoder*, the subsystem which alters the code of information input to it from other information processing subsystems, from a "private" code used internally by the system into a "public" code which can be interpreted by other systems in its environment.
8. *Extruder*, the subsystem which transmits matter-energy out of the system in the forms of products or wastes.	
9. *Motor*, the subsystem which moves the system or parts of it in relation to part or all of its environment or moves components of its environment in relation to each other.	19. *Output transducer*, the subsystem which puts out markers bearing information from the system, changing markers within the system into other matter-energy forms which can be transmitted over channels in the system's environment.
10. *Supporter*, the subsystem which maintains the proper spatial relationships among components of the system, so that they can interact without weighting each other down or crowding each other.	

- *Fabrication Operation:* fabricate a WIP from another unit of WIP and bulk material.
- *Assembly Operation:* assemble two or more units of WIP and bulk material into a subassembly (higher level WIP).
- *Byproduct Operation:* transform two or more WIPs of different types into two or more WIP types that are not identical to the input WIPs).

- *Distribution Operation:* divide one or more units of a single WIP into two or more units of possibly different types of WIP.
- *Consumption Operation:* consume one or more WIPs yielding bulk, dissipated, or useless material. (Note shipping finished products and stockpiling subassemblies are considered consumption operations.)

7.4.3 Feedback and Control in Functional Design

It is important to emphasize the use of feedback in the design of the system. *Feedback and control* is the comparison of the actual characteristics of an output with desired characteristics of that output for the purpose of adjusting the process of transforming inputs into that output (see Sidebar 7.1). Open-loop control processes may or may not make this measurement, but in either case make no adjustments to the process once started. See Figure 7.5. The heating and air-conditioning systems in all but the most expensive cars allow the driver to set the output temperature of the heater and the fan speed; this is an example of an open-loop control system. The driver serves as the feedback process that adjusts the heat and fan speed when a deviation from the desired temperature is noticed. Closed-loop control processes use measurements of the output as feedback for the purpose of adjusting or controlling the transformation process. Heating and air conditioning systems in most houses have a thermostat for setting the desired temperature; this thermostat adjusts the length of time that the heating or air conditioning is left on in order to reach the desired temperature. This is an example of a closed-loop control system.

SIDEBAR 7.1: HISTORY OF CONTROL SYSTEMS

Mayr [1970] traced the earliest example of a control system to the second century BC; this control system was a water clock that operates on the same principles as current flush toilets and is not dissimilar to numerical integration on a digital computer.

In about 1620 Cornelis Drebbel, a Dutch mechanic and chemist, designed a system to control the temperature in a furnace used to heat eggs in an incubator.

About 1787 Thomas Mead invented a centrifugal governor, which was adapted about a year later by Matthew Boulton and James Watt, who invented a fly ball governor to control the rotation speed of a grinding stone for a wind-driven flour mill. The first study of feedback control and the stability of such systems was described in a paper titled "On Governors" by J.C. Maxwell in 1868.

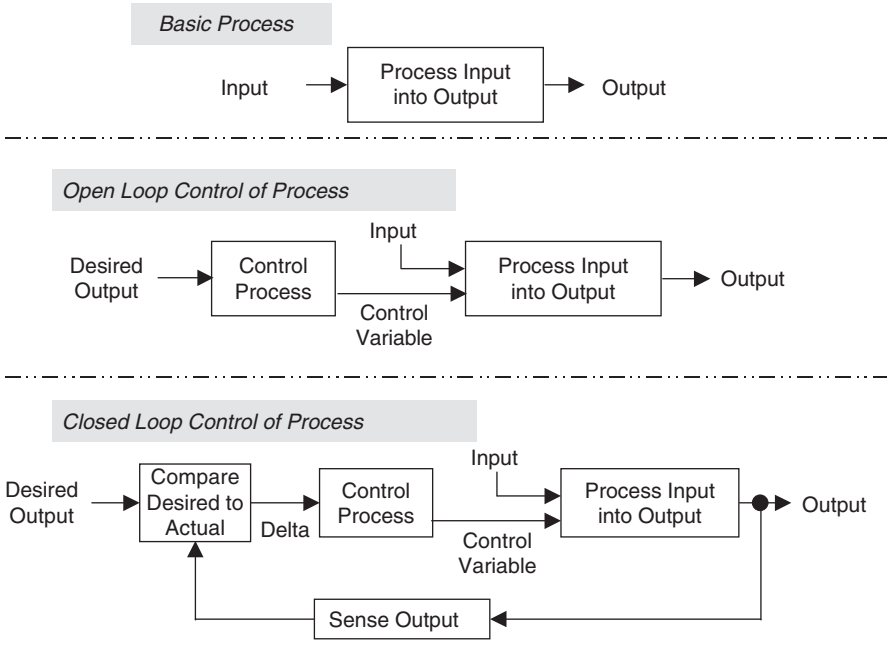

FIGURE 7.5 Open and closed loop control processes.

A *negative feedback process* attempts to close the gap between the current output and the desired output, thus striving for a stable process. A *positive feedback process* attempts to increase the difference between current output and the desired output, usually creating an unstable situation. In the engineering design process, feedback and control enable the comparison of the current state of the system with the desired state for the purpose of repeating parts of the generation of the current state to obtain a current state that is closer to the desired state. The concept of feedback comes from the engineering of control systems, which has been the training ground for many systems engineers.

Closed-loop control processes contain at least four subprocesses: comparison of current and desired output characteristics; control adjustments to the process based upon the comparison; the transformation process for turning inputs into outputs; and a sensing process for turning the output into measured dimension(s) that can be compared to the desired output. The first element is the comparison process in which current values of key variables are compared with desired values of those variables. The comparison process requires definition in advance for what elements of the state of the process are going to be compared. This comparison inevitably introduces a time lag into the process. This element of the feedback process is trivial, but at the same time is the cornerstone. The second element is the control process for deciding what to do about the difference between the current value of the output and the desired

value of the output. The third element of the feedback process is the transformation process that is being controlled by the feedback process. This process dictates how a successful feedback process should be created and is often adapted by the feedback process as part of the correction activity. Sensing the output of the process being controlled is the final element of the feedback process.

While most examples of feedback control systems are in lower level elements of complex systems, there is no reason why such a concept will not also work at higher levels of abstraction. An example is the "Develop System Allocated Architectures" function of the IDEF0 model of the process for engineering a system in Appendix B and repeated in Figure 7.6. There are three feedback and control loops. The first involves the first and second functions, "Allocate Functions & System-wide Requirements to Physical Subsystems" and "Define & Analyze Functional Activation & Control Structure." Here the second function performs the measurement, comparison, and control function based upon the output, functional allocations to components, of the first function.

The measurement, comparison, and control (decision making) in the second loop are done in the third function "Conduct Performance & Risk Analyses" for the output of the second function, alternative system-level allocated architectures. The analysis process determines whether the system-level allocated architectures contain one that is "good enough" to be the finalized design and then proceeds with documentation. If the decision is that there is not an allocated architecture that is good enough, then analysis results are passed to the first two functional processes as controls for making refinements. The intention here is that the analysis results could be passed to either of these two processes or the combination of them. The smallest refinements would conclude with passing analysis results and guidance only to the second process ("Define & Analyze Functional Activation and Control Structure"). Large refinements would require passing results and guidance to both processes.

There is a final feedback loop during documentation, which is when many questions arise. In this case the third function is reactivated if questions arise that cannot be answered using the current documentation and analysis results. If the issue deals with performance and risk analysis, the answer can be generated and the result passed back to the documentation activity. However, if the issue has implications for the allocation of function, tracing of requirements, or activation and control structures, then the initial feedback loop discussed above is reenergized.

Besides the feedback control loops that are designed inside the system, the engineer of the system has to be cognizant to design feedback control for the system using the external systems. The most common example of such feedback control occurs when a human is one of the external systems and closes a feedback loop to improve the system's performance. The driver of an automobile adjusts the car's speed and direction to achieve safe travel; there are numerous output devices at the driver's station of the automobile to enhance the driver's ability to serve as the controller of the car.

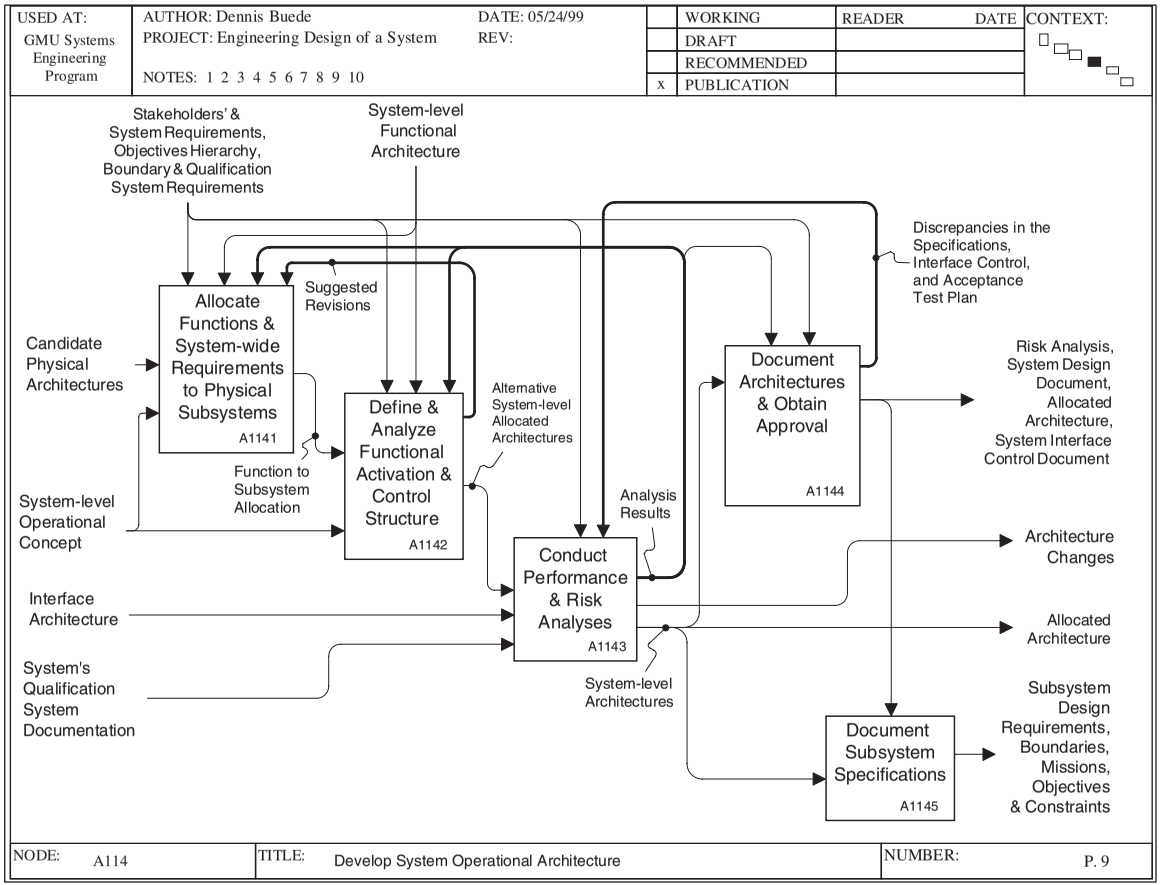

FIGURE 7.6 Illustration of feedback control in the development of the system allocated architecture.

More detailed literature on feedback control can be found in Dickinson [1991], Dorny [1993], Franklin et al. [1994], and Van de Vegte [1994]. A graph-theoretic approach for analyzing control systems has been developed called signal flow graphs. Signal flow graphs are used to transform a set of processes with feedback into a single, composite process.

7.4.4 Evaluation of a Functional Hierarchy

A functional architecture can be evaluated for shortfalls and overlaps. A *shortfall* is the absence of a functionality that is required to produce a desired output from one or more inputs. Shortfalls can be divided into the following categories: absence of the proper functionality for some set of inputs, inability to produce a desired output, and insufficient feedback control to produce the desired output.

Recall the definition of a function from Chapter 4. A function maps all elements of the domain to some element in the range and does not map any element of the domain into two distinct elements of the range. Whenever there are potential inputs to the system with which the system's functionality cannot deal, the engineer of the system did not create a system function but rather a system relation. A relation in Chapter 4 includes functions but also includes those entities that fall short of a function. In fact, the most common types of shortfall are the absence of or inappropriate functional responses to unexpected inputs and to failure modes within the system. For example, the elevator system must be able to respond properly when a fire alarm sounds. Less obvious unexpected inputs might be the need for a user to stop the elevator immediately. Therefore the systems engineer must always enumerate all possible inputs, including those inputs that are not wanted but can arrive. In the mathematical terms of Chapter 4, a Cartesian product of possible inputs must be formed for each function in the functional model of the functional architecture. This is only necessary for the lowest level functions in the functional decomposition. The Cartesian product of inputs for a function uses each category of input shown in the functional model for a specific function. For each of these categories there are usually several possible input states, some of which are not desired. For example, if there were three possible input categories to a given bottom-level function and each input category had three possible states, there would be three-tuple formed by taking the Cartesian product of these three input categories. The three-tuple would have 27 ($3 \times 3 \times 3$) different combinations. The functional definition of this bottom-level function must account for every one of these 27 possible combinations.

The second category of shortfall is the inability to produce a needed output. This type of functionality will be obvious if all of the system's outputs have been defined. This is a major benefit of the external systems diagram in Chapter 6 and the functional architecture discussed in this chapter. Evaluating for this category of shortfall is not always possible without constructing an overall functional architecture.

The final shortfall addresses the quality of the outputs produced. Often this quality falls short of that desired by the stakeholders because the engineers have not incorporated sufficient feedback control, either internally to the system or inclusive of the external systems. Missing needed feedback is a common mistake made in the functional architecture. This is true not only for the functional architecture of the system being designed for the operational phase of the life cycle, but also for the functional architectures of the developmental and manufacturing systems.

An *overlap* is a redundancy in functionality that is not needed to achieve additional performance, for example, reliability. Functional overlaps, unlike physical overlaps for redundancy, are not needed and therefore can only cause problems.

A common technique for identifying shortfalls and overlaps is to follow each scenario in the operational concept (Chapter 6) through the functional architecture. Each scenario in the operational concept begins with a single input to the system from one of the external systems and continues with a sequence of inputs to and outputs from the system to various external systems. Each scenario was developed by treating the system as a black box. Now is the time to shine a light into that black box (producing a white box) and see what functions the system is performing to transform the inputs into outputs. Start with the first input to the system for a given scenario (see Fig. 7.7); color the line in the context diagram (A-0 page or node) for that input green (or whatever color you choose). Find an interesting output of the system in the scenario and color that output on the context page green also. In Figure 7.7 the input selected was ''Request for Elevator Service & Entry Support'' by a potential passenger, which is shown as a dotted-dashed line since color is too expensive for a text book. The output selected was ''Elevator Entry/Exit Opportunity'' when the elevator arrives at the potential passenger's floor; this output is also shown as a dotted-dashed line.

Now move to the AO page (node) and color these same two lines green; see Figure 7.8 for the dotted-dashed lines. Now go to the function on the AO page that received that input (the Al function in Fig. 7.8) and find the appropriate output of the function that is needed to get to the output on the context page and color the line associated with that output green. ''Digitized Passenger Request'' is shown with a dotted-dashed line in Figure 7.8. Proceed to this next function on the AO page and find the most appropriate output to color. This is like looking through a house for clues to a mystery, searching room by room, finding a clue in each room that leads to the next room, until finally the room is found with the already identified path outside. In Figure 7.8, ''Digitized Passenger Request'' led to the A2 function, ''Control Elevator Cars.'' The appropriate output of this function was ''Assignments to Elevator Cars,'' leading to A3, ''Move Passengers Between Floors,'' which is where ''Elevator Entry/Exit Opportunity'' was found.

This process continues for every other page of the functional model. Figures 7.9–7.12 show this trace of the input and output from a given scenario

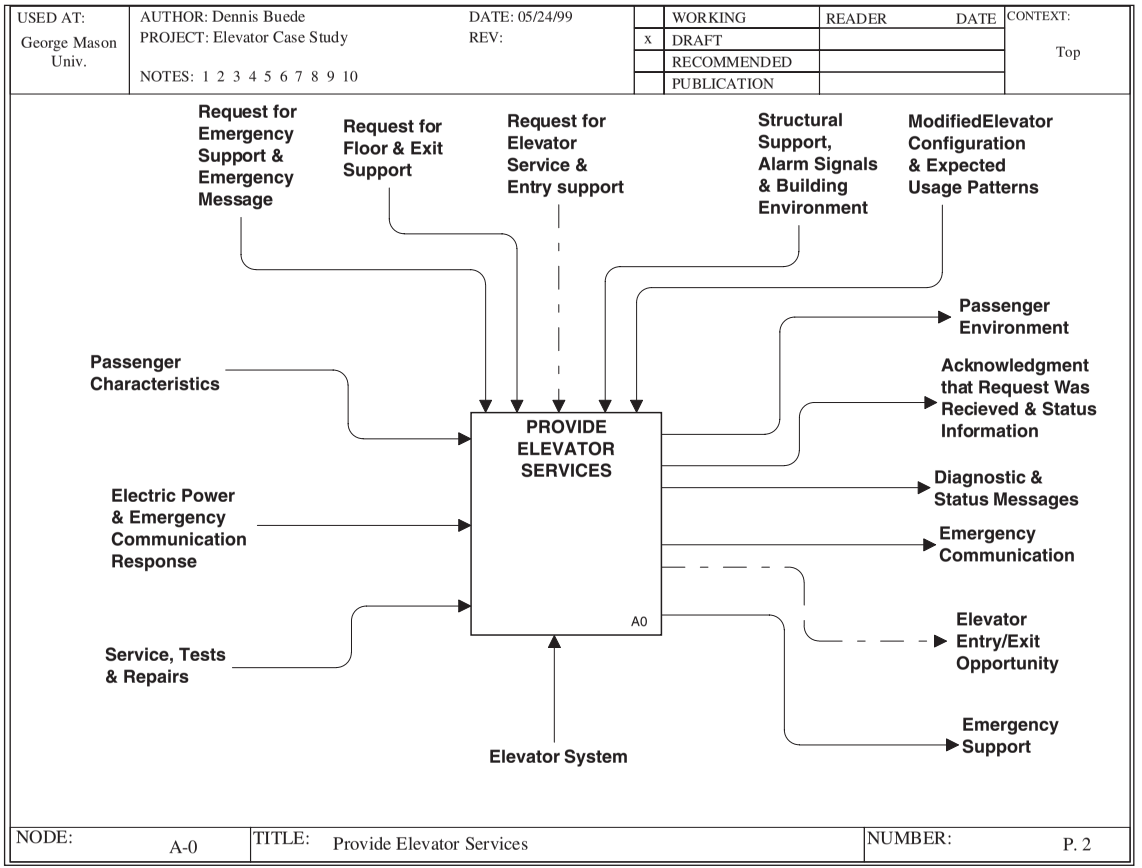

FIGURE 7.7 Scenario trace on the context page.

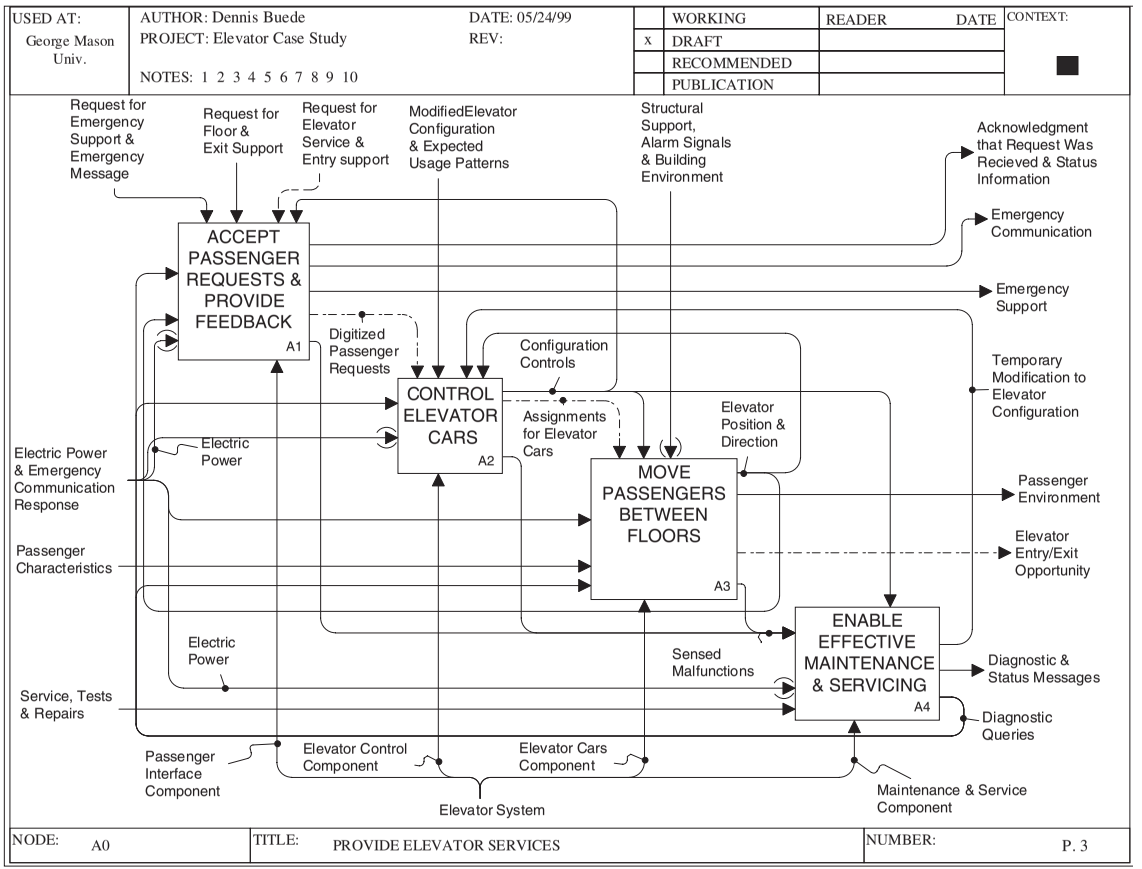

FIGURE 7.8 Scenario trace continued in the AO diagram.

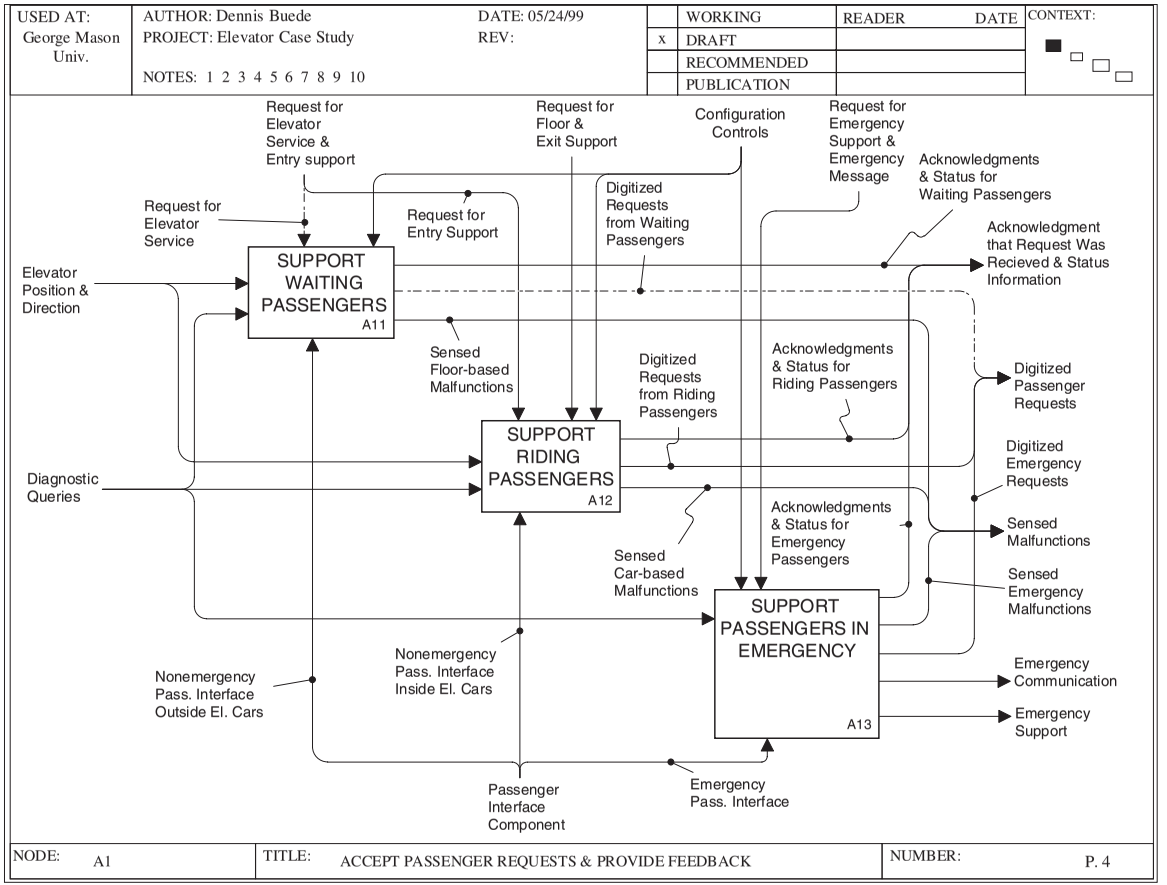

FIGURE 7.9 Scenario trace continued in the AI diagram.

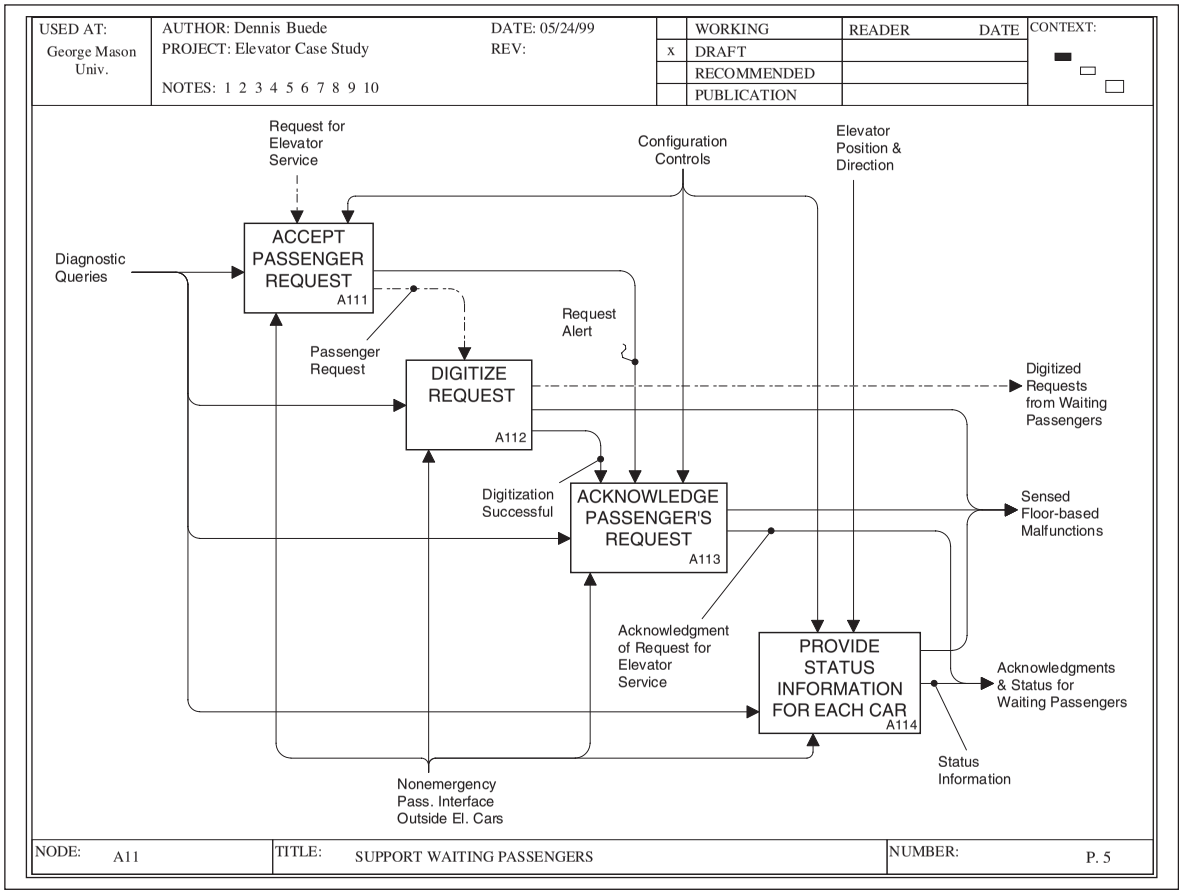

FIGURE 7.10 Scenario trace continued in the Al1 diagram.

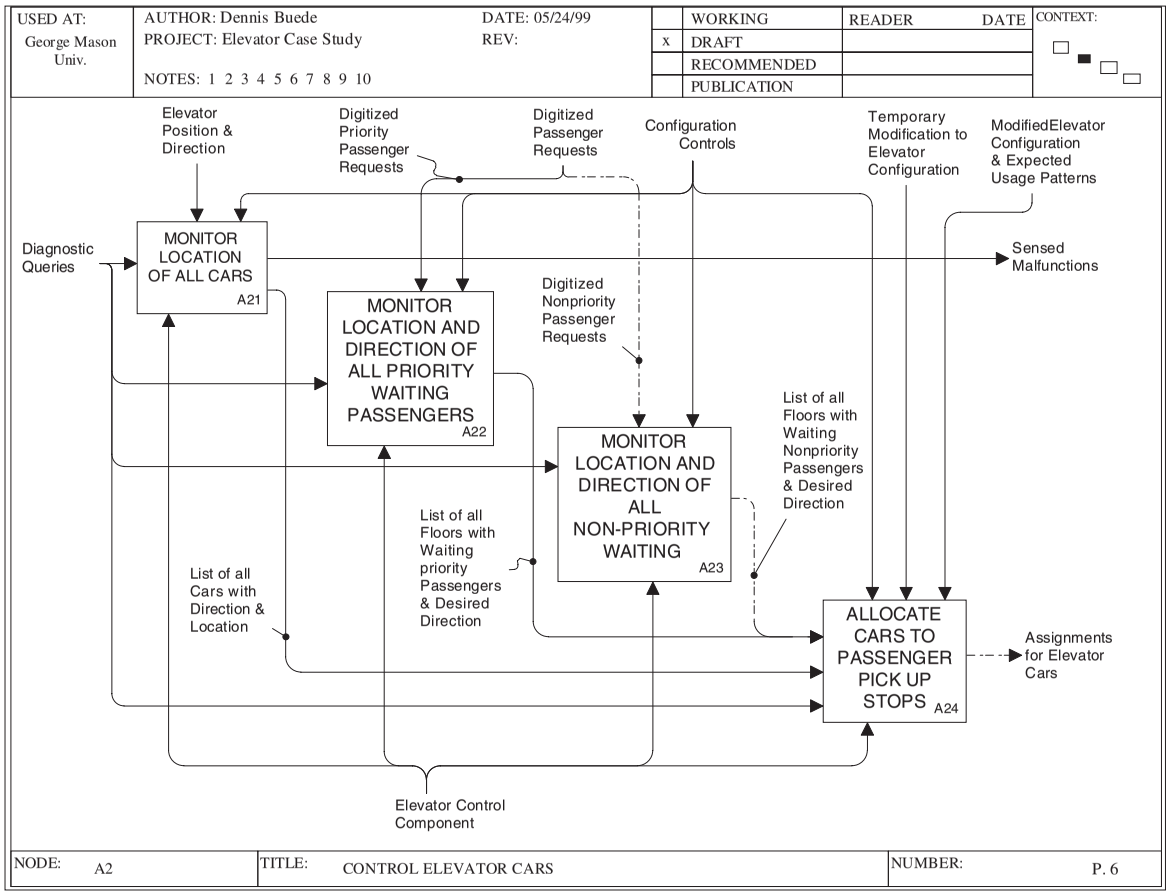

FIGURE 7.11 Scenario trace continued in the A2 diagram.

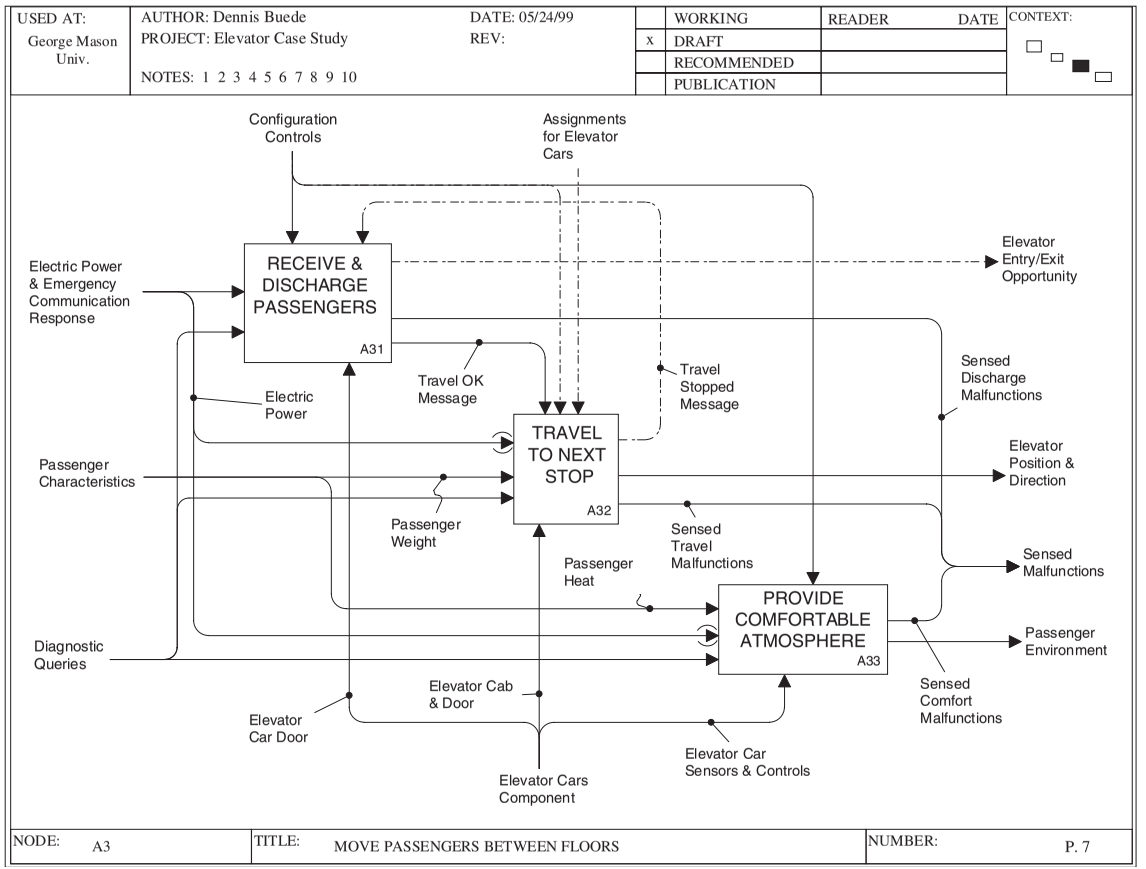

FIGURE 7.12 Scenario trace completed in the A3 diagram.

throughout the entire functional model of the elevator system in the case study that can be downloaded from http://www.theengineeringdesignofsystems.com.

In addition, defining failure modes for the system and creating *error detection and recovery functionalities* within the common operating modes as well as the failure modes is critical. These functionalities for error detection and recovery are critical for stakeholder usability. How often has your computer shut down with no warning and little support for saving open files? The more mature an operating system is, the more functionality the operating system commonly has for saving open files as part of the crash, and the more unlikely such crashes are. Details on functionality for addressing error detection and recovery are covered later in the chapter.

7.5 DEVELOPMENT OF THE FUNCTIONAL DECOMPOSITION

The literature [Marca and McGowan, 1988] surrounding the structured analysis and design technique (SADT), which became IDEF0, suggests the following activities for creating a functional decomposition with inputs, controls and outputs:

- Determine the purpose and viewpoint.
- Generate a data list, based upon the system's boundaries (the external systems diagram).
- Generate an activity list.
- Define the AO diagram, and the level 1 functional decomposition.
- Draw the context diagram, A-0 (this has already been done, based on the external systems' diagram).
- Continue this process while decomposing the level 1 functions.

The purpose and viewpoint define the issues that the IDEF0 model will address. The purpose for systems engineering applications is straightforward, namely to depict the functional activities of the system in a particular phase of the system's life cycle; as can be seen in the elevator case study (available on the author's web site) there is a separate IDEF0 model for each phase. Similarly, the viewpoint is the systems engineering team; this team is creating the functional architecture, of which the IDEF0 model is a part, for the purposes of designing the system. Typically, there are a number of stakeholders with a somewhat diverse set of opinions that are concerned about each phase of the life cycle; the systems engineering team should include representatives of these stakeholders and has ultimate responsibility to integrate these opinions.

The data list of inputs, controls, and outputs for the system's top-level function should already be available from the external systems' diagram. Nonetheless, this is an excellent time to review and critique the data list to determine if there are any missing or redundant items.

Next, we have the first of many decomposition decisions. *How should the top-level system function be decomposed?* Spending some time gathering information and brainstorming about system functions for each phase is always a good idea, in addition to creating an activity list from which to choose or synthesize the functional decomposition. For the operational phase of the life cycle a previous section presented the options of starting with the operational modes of the system or alternatively with the functional taxonomy derived from the Hatley–Pirbhai [1988] architecture template. At this point in time the systems engineering team certainly has not finalized the definition of operating modes for the system. In fact, the functional decomposition will inevitably be modified over time as the performance of the allocated architecture is evaluated. Figure 7.3 depicted the elevator's top-level functional decomposition for the operational phase in terms of the Hatley–Pirbhai template.

There are many ways to gather information:

- Review documents, but watch for viewpoint changes.
- Observe operations, but be careful about the details that you do not know well enough to recognize and the need to make major changes from the current system to the system under development.
- Conduct interviews; questionnaires can be used but have very limited value (be sure you get the right experts).
- Invent a strawman for the experts to critique.
- Create several alternate decompositions and create a composite strawman based on the best features of each after some critical discussion (this creativity technique is often called the ''gallery'').

Once a working version of the functional model is created, the functional model should be reviewed by individuals that have substantial knowledge and varying perspectives about the system's functioning in a given life-cycle phase. This review process should:

- Try alternate decompositions.
- Disaggregate the functions differently.
- Bundle and unbundle arrows differently.
- Reevaluate functional dominance in terms of feedback and control.
- Catch interface errors.

As part of this review process creating a data model of the inputs, controls, and outputs using an entity–relationship–attribute or higraph model would be wise. These techniques are discussed in Chapter 12. The data model often introduces critical design issues that have been overlooked in the functional or process model.

How far should the functional decomposition be carried out? Generally speaking, the functional decomposition should proceed to the second, third,

or fourth level. At this point the physical and allocated architectures should be developed and analyzed. The more detailed the operational concept the more reliably the functional architecture can be developed to the fourth level. Defining the system's functions to line up with the physical components is best so that the inputs, controls, and outputs clearly line up with external and internal interfaces. The level of detail should be appropriate with the viewpoint and purpose, that is, the stakeholders and specified phase of the system's life cycle. Be sure to eliminate details if they are not helping create the allocated architecture. Also, see Sidebar 7.2 for a list of common mistakes made in the development of a functional architecture.

SIDEBAR 7.2: COMMON MISTAKES IN DEVELOPING A FUNCTIONAL ARCHITECTURE

1. Including the external systems and their functions. The functional architecture only addresses the top-level function of the system in question. The external system diagram establishes the inputs, controls, and outputs for this function. A boundary has been drawn around the system to exclude the external systems and their functions.
2. Choosing the wrong name for a function. The function name should start with an action verb and include an object of that action. The verb should not contain an objective or performance goal such as maximize, but should describe an action or activity that is to be performed.
3. Creating a decomposition of a function that is not a partition of that function. For example, a student once decomposed "AO: Provide Elevator Services" into "Al: Transport Users," "A2: Evaluate System Status," and "A3: Perform Security & Maintenance Operations." "Al: Transport Users" was then decomposed as follows: "All: Provide Access to Elevator," "Al2: Transport Users," and "A13: Provide Emergency Operations." Al2 cannot be a child of itself. The sub-functions of a function should all be at the same level of abstraction [Chapman et al., 1992].
4. Including a verb phrase as part of the inputs, controls, or outputs of a function. Verb phrases are reserved for functions.
5. Violating the law of conservation of inputs, controls, and outputs. That is, every input, control, and output of a particular function must appear on the decomposition of that function, and there can be no new ones.
6. Trivializing the richness of interaction between the functions that decompose their parent. Consider many possible simple functionalities that comprise the children of a parent function and then

develop the inputs, controls, and outputs that enable these simple functionalities to exist, including the necessary feedback and control.

7. Creating outputs from thin air. The most common mistake is to define a function that monitors the system's status but that does not receive inputs about the functioning or lack of functioning of other parts of the system.

7.6 FINISHING THE FUNCTIONAL ARCHITECTURE

Two key areas of the functional architecture that need to be addressed before the job is finished are (1) defining system errors and the failure modes that result and inserting the functionality to detect the errors and recover and (2) inserting the appropriate functionalities for some combination of built-in self-test (BIST) and external testability. The functionalities described here are typically not part of the initial drafts of the functional architecture because they depend to a significant degree on the physical architecture; as a result these functions are often added once the allocated architecture is taking shape.

Fault tolerance is a laudable design goal, meaning that the system can tolerate faults and continue performing. In fact, the design goal of every systems engineering team is to create a system with no faults. However, faults like friction have to he tolerated at best, even after our best efforts to eliminate them. This discussion on fault-tolerant functionality depends greatly on understanding several key terms; see Jalote [1994] and Levi and Agrawala [1994]. Figure 7.13 provides a concept map based on these definitions.

System: an identifiable mechanism that maintains a pattern of behavior at an interface between the system and its environment. [Anderson and Lee, 1981]

Failure: deviation in behavior between the system and its requirements. Since the system does not maintain a copy of its requirements, a failure is not observable by the system.

Error: a subset of the system state which may lead to a failure. The system can monitor its own state, so errors are observable in principle. Failures are inferred when errors are observed. Since a system is usually not able to monitor its entire state continuously, not all errors are observable. As a result, not all failures are going to be detected (inferred).

Fault: a defect in the system that can cause an error. Faults can be permanent (e.g., a failure of system component that requires replacement) or temporary due to either an internal malfunction or external transient. Temporary faults may not cause a sufficiently noticeable error or may cause a permanent fault in addition to a temporary error.

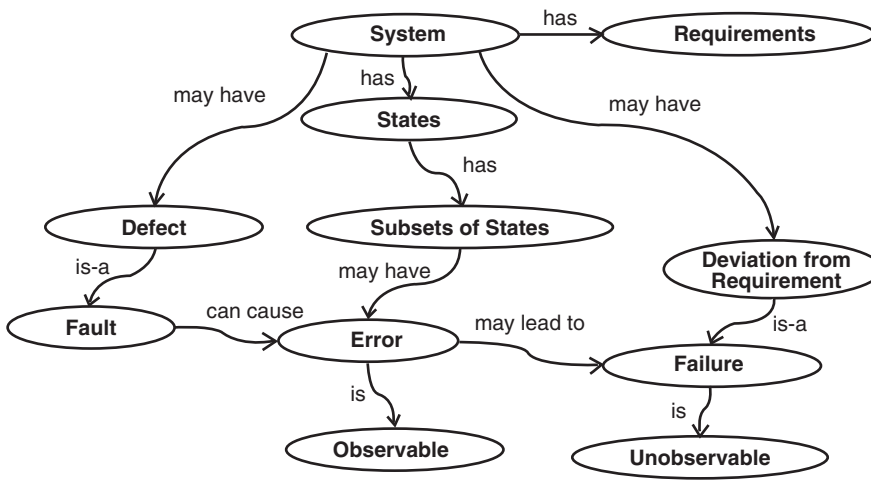

FIGURE 7.13 Concept map for fault tolerance terms.

First, note the difference of the definition of system in the fault tolerance literature and that discussed in Chapters 2 and 6 of this book, which represent the systems engineering community. The fault tolerance community is focused on inferring failures by detecting errors. The notions that are central to this focus are the system's requirements (or specifications), the boundary between the system and the system's environment at which the state of the system is defined, and the interface that connects the system to its environment. The fact that a system has objectives, as defined by the stakeholders, and functions (or tasks), as defined by the systems engineers, is not relevant to the fault tolerance community and is therefore not found in their definition of a system.

Achieving fault tolerance in a system means using both the designed functions and physical resources of the system to mask all errors (deviations between actual system outputs and required system outputs) from the system's environment. Fault tolerance can only be achieved for those errors that are observed. The generic system functions associated with fault tolerance are (1) error detection, (2) damage confinement, (3) error recovery, and (4) fault isolation and reporting. The design of physical resources needed for fault tolerance is discussed in the next chapter.

Error detection is defining possible errors, deviations in the subset of the system's state from the desired state, in the design phase before they occur, and establishing a set of functions for checking for the occurrence of each error. Just as with requirements development, defining error checking to be complete, correct, and independent of the design of the system is desirable. Unfortunately, this is not yet possible so error detection will be imperfect. The most frequent error detection involves errors in data, errors in process timing, and physical errors in the system's components. The most common checks for data

errors include type and range errors. Type checks establish that the data is the right type, for example, Boolean versus integer. Range checks ensure that the value of the data is within a specified range. Knowing the correct values of the data is not possible so type and range checks are approximations of the checking that would be most effective if the truth were known. Semantic and structural checks are also possible on data elements. Semantic checks compare a data element with the state of the rest of the system to determine whether an error has occurred. Structural checks use some form of data redundancy to determine whether the data is internally consistent. A structural check used in coding is to add extra bits to the data bits; these added hits take on values that depend on the values of the data bits. Later these extra bits and the associated data bits can be checked to ensure an appropriate relationship exists; if not, an error is declared. Similarly robust data structures in software use redundancy in the data structures to check for data errors. Timing checks are used in real-time or near-real-time systems. Timing checks assume the existence of a permissible range for the time allotted to some process being performed by the system. A timer is activated within a process to determine whether the completion of the process is within an appropriate range; if not, an error is declared. Hardware systems typically detect timing errors in memory and bus access. Operating systems also use timing checks. Finally physical errors in a component of the system are the province of BIST and will be discussed in the next chapter.

Damage confinement is needed in fault tolerance because there is typically a time lag between the occurrence of failure and the detection of the associated error. During this time lag the failure or the implications of the failure may have spread to other parts of the system; error recovery activities are dangerous without having knowledge about the extent of damage due to a failure. As soon as the error detection functionality has declared an error, damage confinement functionality must assess the likely spread of the problem and declare the portion of the system contaminated by the failure. The most common approach to damage confinement is to build confinement structures into the system during design. ''Fire walls'' are designed into the system to limit the spread of failure impacts. With these predesigned fire walls declaring that a failure is limited to a specific area of the system when an error is declared is possible. A more sophisticated approach is to reexamine the flow of data just prior to an error to determine the possible spread of errors due to a failure; this sophisticated approach requires not only that error detection functionality be designed into the system but that functionality to record a time history of data be added so that this information exists when the information is needed.

Error recovery functionality attempts to correct the error after the error has been declared and the error's extent defined. If the error concerns data in the system, backward recovery is typically employed to reset the data elements to values that were recorded and acceptable at some previous time. These values may not be correct in the sense that they are the values the system should have generated. Rather, these values are acceptable in the sense of type, range, and

semantics discussed above in error detection. The purpose of backward recovery is to keep the system from a major failure, not to restore the system to the correct state. As a result, the system's users are typically notified as part of the error recovery process that a failure occurred and are given the chance to attempt to recover the correct data or restart at an appropriate place to generate the correct data. Forward recovery is an attempt to guess at what the correct values of the data should have been; this is dangerous but sometimes justified in real-time systems where backward recovery and user notification is not possible. Timing errors are handled by ending a process that is taking too long and asserting a nominal or last computed value for the process output. Physical errors are handled by either graceful termination of the system's activities or switching to redundant (standby) components when they are available. In recovering from physical errors, capturing the last available values of the system's data structure prior to termination or component switching is critical.

Fault isolation and reporting functionality attempts to determine where in the system the fault occurred that caused the failure that generated the error. To isolate faults the components of the system must be providing information about their current status.

BIST for a specific component incorporates the functionality to test defined functionality and provide feedback about the results. These types of BIST are common during system start-up and routine operation.

The functional architecture must be expanded during the final development of the allocated architecture to include functions for error detection, damage confinement, error recovery, and fault isolation and reporting. In accordance with the fault tolerance community, these functions should be defined for every state variable of the system, which includes the system's outputs. In addition, including error trapping for many of the inputs to the system is important. Error trapping includes functions for error detection, damage confinement, and error recovery for user inputs; the system must monitor system inputs to detect unacceptable inputs and alert the user that a given input is unacceptable and to reenter a correct input. For example, the system is expecting the user to input a number as part of a menu selection or data entry task. However, the user, due to inattention or typing error, enters a letter instead. Most older software would immediately crash, sometimes crashing the entire computer system. However, more recent, well-designed software will monitor the input for such an error and alert the user that this error has been made and request a new input.

7.7 TRACING REQUIREMENTS TO ELEMENTS OF THE FUNCTIONAL ARCHITECTURE

There are two elements of the functional architecture that should have input/output requirements traced to them: the functions and the external items

(inputs and outputs). Both of these tracings can be accomplished in systems engineering tools such as CORE. All elements of the set of input/output requirements should be traced to appropriate functions that have been defined in the functional decomposition. Tracing input requirements and output requirements to functions should be done throughout the functional decomposition as is shown in Figure 7.13; this tracing is guided explicitly by the association of inputs and outputs with functions in the functional architecture. For example, since "calls (requests) for up and down service" is an input of "Support Waiting Passengers," all of the requirements related to this input should be traced to the function "Support Waiting Passengers" and that function's predecessors in the functional decomposition. Similarly, external interface requirements should be traced to the function that is associated with receiving the input or sending or output, respectively. For example, the phone line (external interface) transmits and receives items that are associated with the function "Support Passengers in Emergency"; therefore the external interface requirement to use a phone line to communicate via the building with maintenance personnel should be traced to this function. Each external interface requirement should also be traced to the predecessors of this function. Finally, all of the functional requirements should be traced to the top-level system function. As discussed in Chapter 6 a preferred convention for the functional requirements is to list the functions in the top-level functional decomposition that define the system function. This tracing of input/output requirements to functions is illustrated in Figure 7.14 for a sample of functions and requirements from the elevator case study, which can be downloaded from http://www.theengineeringdesignofsystems.com.

The logic for tracing input/output requirements to functions is as follows. The ultimate product of the systems engineering team is a set of specifications for each CI. Intermediate products are specifications for the intermediate components that comprise the system and are built from the CIs. Each of these specifications will contain requirements that are derived from the system-level requirements that are derived from the stakeholders' requirements. In addition, each of these specifications will contain a functional architecture that is relevant to the component or CI of interest. This functional architecture for a component or CI will be a subset of the system's functional architecture and will contain input/output requirements traced to these functions at the system level. These input/output requirements should be contained in the specification. Tracing system input/output requirements to functions is a method for ensuring that the appropriate input/output requirements are contained in each specification that has to be developed during the design process.

In addition, tracing input/output requirements to functions serve as a consistency check. Does each function have requirements traced to it for each input and output? Is each input/output requirement traced to at least one function?

The input and output requirements are also traced to the external item elements. This tracing is made explicit in the set of input and output

Functions	Input/Output Requirements (A Sample)					
	Input Requirements		Output Requirements		Functional Requirement	External Interface Requirement
	The elevator system shall receive calls for up and down service from all floors of the building.	The elevator system shall receive passenger activated fire alarms in each elevator car.	The elevator system shall provide adequate illumination.	The elevator system shall open and close automatically upon arrival at each selected floor.	The elevator system shall control elevator cars efficiently.	The elevator system shall use a phone line from the building for emergency calls.
0 Provide Elevator Services	X	X	X	X	X	X
1 Accept Passenger Requests + Provide Feedback	X	X				X
1.1 Support Waiting Passengers	X					
1.2 Support Riding Passengers						
1.3 Support Passengers in Emergency		X				X
2 Control Elevator Cars						
3 Move Passengers between Floors			X	X		
3.1 Receive + Discharge Passengers				X		
3.2 Travel toNext Stop						
3.3 Provide Comfortable Atmosphere			X			
4 Enable Effective Maintenance and Servicing						

FIGURE 7.14 Tracing a sample of input/output requirements to a sample of functions.

requirements for the operational phase of the elevator, as shown in Appendix B. The rationale for tracing the input and output requirements to external items is that the external interfaces need to satisfy these requirements. The internal items of the functional architecture will also have the relevant input and output requirements traced to them later in the design phase so that the internal interfaces of the system will have derived requirements that they must meet. This tracing can provide a valuable consistency check: Does each item have at least one requirement traced to it? Also, does each requirement trace to some item? If either of these questions is negative for any requirement or item, there has been a breakdown in the requirements development process. Finally, an item will be "carried by" a link, which "comprises" an interface. The item will have one or more input/output requirements traced to it. In addition, the link will ultimately have derived system-wide requirements traced to it. The interface specifications will be built from the requirements that are traced to the items being carried by the links comprising the interface as well as the system-wide requirements that ultimately are traced to the interface.

7.8 SUMMARY

The functional architecture of a system, as defined in this chapter, contains a hierarchical model of the functions performed by the system, the system's components, and the system's CIs; the flow of informational and physical items from outside the system through the system's functions and on to the waiting external systems being serviced by the system; and a tracing of input/output requirements to both the system's functions and items.

This chapter introduces quite a few terms that are key to understanding and developing a functional architecture. A system mode is an operational capability of the system that contains either full or partial functionality. A state is a modeling description of the status of the system at a moment in time. A function is an activity that the system performs in order to transform an n-tuple of inputs into an m-tuple of outputs. These concepts are key to the development of a functional architecture. The system's modes and functions should be part of the functional architecture, while the system's state should be definable by a set of parameters in any operational mode while performing any set of functions. The parameters that comprise this state may vary based on the operational mode and the functions being performed.

Other key terms addressed in this chapter include failure, error, and fault. Failure is a deviation between the system's behavior and the system's requirements. An error is a problem with the state of the system that may lead to a failure. A fault is a defect in the system that can cause an error. To achieve the desired level of fault tolerance, the system must perform the functions of error detection, damage confinement, error recovery, and fault isolation and reporting.

A method for developing a functional architecture was defined in this chapter. Defining the functional architecture is not easy and is a modeling

process that the engineer of a system must learn. The modeling process uses a combination of decomposition and composition. The concepts of feedback and control are critical to defining the system's functions.

The engineering of a system has to rely upon more than the physical design of the system. The functions or activities that the system has to perform are a critical element of the design process and the design of these functions needs to be given an equal importance to the physical design by the engineers. The designs of functions and physical resources for the system are not independent; they must both be done, usually in parallel.

PROBLEMS

7.1 What are the operating modes of your car's stereo system?

7.2 For the ATM of the Money Mart Corporation:

i. As part of the systems engineering development team, use IDEF0 to develop a functional architecture. The functional architecture should address all of the functions associated with the ATM. This functional architecture should be at least two levels deep and should be four levels deep in at least one functional area that is most complex. Note that you will be graded on your adherence to proper IDEF0 semantics and syntax, as well as the substance of your work.

ii. Pick three scenarios from the operational concept and describe how these scenarios can be realized within your functional architecture by tracing functionality paths through the functional architecture. Start with the external input(s) relevant to each scenario and show how each input(s) is(are) transformed by tracing from function to function at various levels of the functional decomposition, until the scenario's output(s) is(are) produced. Highlight with three different colored pens (one color for each scenario) the thread of functionality associated with each of these three scenarios.
If your functional architecture is inadequate, make the appropriate changes to your functional architecture.

iii. As part of the systems engineering development team for the ATM, update your requirements document to reflect any insights into requirements that you obtained by creating a functional architecture. That is, if you added, deleted, or modified any input, controls, or outputs for the system, modify your input/output requirements. Also update your external systems diagram if any changes are needed.

7.3 For the OnStar system of Cadillac:

i. As part of the systems engineering development team, use IDEF0 to develop a functional architecture. The functional architecture should address all of the functions associated with OnStar. This functional

architecture should be at least two levels deep and should be four levels deep in at least one functional area that is most complex. Note that you will be graded on your adherence to proper IDEF0 semantics and syntax, as well as the substance of your work.

ii. Pick three scenarios from the operational concept and describe how these scenarios can be realized within your functional architecture by tracing functionality paths through the functional architecture. Start with the external input(s) relevant to each scenario and show how each input(s) is(are) transformed by tracing from function to function at various levels of the functional decomposition, until the scenario's output(s) is(are) produced. Highlight with three different colored pens (one color for each scenario) the thread of functionality associated with each of these three scenarios.

iii. If your functional architecture is inadequate, make the appropriate changes to your functional architecture.

iv. As part of the systems engineering development team for OnStar, update your requirements document to reflect any insights into requirements that you obtained by creating a functional architecture. That is, if you added, deleted, or modified any input, controls, or outputs for the system, modify your input/output requirements. Also update your external systems diagram if any changes are needed.

7.4 For the development system for an air bag system:

i. As part of the systems engineering development team, use IDEF0 to develop a functional architecture. The functional architecture should address all of the functions associated with the development system for an air bag. This functional architecture should be at least two levels deep and should be four levels deep in at least one functional area that is most complex. Note that you will be graded on your adherence to proper IDEF0 semantics and syntax, as well as the substance of your work.

ii. Pick three scenarios from the operational concept and describe how these scenarios can he realized within your functional architecture by tracing functionality paths through the functional architecture. Start with the external input(s) relevant to each scenario and show how each input(s) is(are) transformed by tracing from function to function at various levels of the functional decomposition, until the scenario's output(s) is(are) produced. Highlight with three different colored pens (one color for each scenario) the thread of functionality associated with each of these three scenarios.

If your functional architecture is inadequate, make the appropriate changes to your functional architecture.

iii. As part of the systems engineering development team for the development system for an air bag, update your requirements document to reflect any insights into requirements that you obtained by creating a

functional architecture. That is, if you added, deleted, or modified any input, controls, or outputs for the system, modify your input/output requirements. Also update your external systems diagram if any changes are needed.

7.5 For the manufacturing system for an air bag system:

i. As part of the systems engineering development team, use IDEF0 to develop a functional architecture. The functional architecture should address all of the functions associated with the manufacturing system for an air bag. This functional architecture should be at least two levels deep and should be four levels deep in at least one functional area that is most complex. Note that you will be graded on your adherence to proper IDEF0 semantics and syntax, as well as the substance of your work.

ii. Pick three scenarios from the operational concept and describe how these scenarios can be realized within your functional architecture by tracing functionality paths through the functional architecture. Start with the external input(s) relevant to each scenario and show how each input(s) is(are) transformed by tracing from function to function at various levels of the functional decomposition, until the scenario's output(s) is(are) produced. Highlight with three different colored pens (one color for each scenario) the thread of functionality associated with each of these three scenarios.

If your functional architecture is inadequate, make the appropriate changes to your functional architecture.

iii. As part of the systems engineering development team for the manufacturing system for an air bag, update your requirements document to reflect any insights into requirements that you obtained by creating a functional architecture. That is, if you added, deleted, or modified any input, controls, or outputs for the system, modify your input/output requirements. Also update your external systems diagram if any changes are needed.