Part 3

# Supplemental Topics

Chapter 12

# Graphical Modeling Techniques

## 12.1 INTRODUCTION

There are three categories of qualitative modeling approaches used as part of the development of functional and allocated architectures during the engineering of systems: data modeling, process modeling, and behavior modeling. A *data model* addresses the relationships among the inputs and outputs of a system. A *process model* defines the functional decomposition of the system function and the flow of inputs and outputs for those functions. A *behavior model* defines the control, activation, and termination of system functions needed to meet the performance requirements of the system. In addition, object-oriented engineering is becoming a major force in software engineering and is beginning to be employed in systems engineering; object-oriented engineering uses these three domains as well. Within each of these three approaches, as well as object-oriented engineering, there are a number of methods that are currently being used in systems and software engineering, as shown in Table 12.1. This table provides a subset of the modeling approaches currently in use. This chapter provides a description and sample model applications of each of the modeling techniques that comprise data, process, and behavior modeling. SysML and its modeling methods as well as IDEF0 (Integrated Definition for Function Modeling) were covered in detail in Chapter 3. As discussed in Chapter 9, balancing or aligning the elements of multiple modeling techniques is important in the development of the functional and allocated architectures.

**TABLE 12.1 Functions of the design process**

| Design Function | Major Inputs | Major Outputs |
|---|---|---|
| Define Problem To Be Solved | Concerns and Complaints by Stakeholders<br>Available Data from Stakeholders | Definitions of Measures of Effectiveness and Desired Ranges<br>Constraints |
| Develop and Evaluate Alternate Concepts for Solving Problem | Ideas for Concepts from All Interested Parties | Recommended Concept(s)<br>Objective Hierarchy & Value Parameters for Meta-System |
| Define System Level Design Problem Being Solved | Stakeholders' Inputs | Stakeholders' Requirements<br>Operational Concept |
| Develop System Functional Architecture | Stakeholders' Requirements<br>Operational Concept | Functional Architecture |
| Develop System Physical Architecture | Stakeholders' Requirements | Physical Architecture |
| Develop System Allocated Architecture | Stakeholders' Requirements<br>Functional Architecture<br>Physical Architecture<br>Interface Architecture | Allocated Architecture |
| Develop Interface Architecture | Draft Allocated Architecture | Interface Architecture |
| Develop Qualification System for the System | Stakeholders' Requirements<br>Systems Requirements | Qualification System Design Documentation |

## 12.2 DATA MODELING

There are many approaches to data modeling. This section describes two different modeling techniques. Entity–relationship (ER) diagrams are the oldest form of data modeling. Higraphs are the most formally based approach and offer the most power.

Two other approaches, IDEF1 and IDEF1X, were developed within the IDEF community but are not discussed in detail here. IDEF1 models data using entity classes and relations among entity classes. An entity class has attributes that describe the entity. The relations that are possible between classes come from entity–relationship diagrams and address mainly relationships that are one-to-one, one-to-many, and so forth. IDEF1 is an approach for modeling the structure of information as the information is maintained in an organization, including the business rules [Griffith, 1994]. IDEF1X also models data using entity classes and relations among the classes. IDEF1X allows for a fuller definition of subtypes and attributes in terms of their aliases, data type, length, definition, primary key, discriminator, alternate keys, and inversion entities than does IDEF1. Similarly, the relationships in IDEF1X may be defined on the arcs and include one-to-one, one-to-many, and so forth. IDEF1X is used for designing relational databases [Griffith, 1994]. The interested reader should see the FIPS PUB 184 [1993] on IDEF1X.

### 12.2.1 Entity–Relationship Diagrams

*Entity–relationship diagrams* model the data structure or relationships between data entities. Art entity is a class of real, similar items (e.g., people, books, computers). Entity types are shown in boxes; relationships are shown in diamonds or as labels on the arcs. If diamonds are used, the graph has no directed edges (with one exception). The relationship is usually read from left to right or from top to bottom, but this is not universal [see Yourdon, Inc., 1993]. When the edges are directed, the relationship is read in the direction of the edge. Figure 12.1 shows examples of both directed edges and diamonds.

The exception for directed edges when diamonds and undirected edges are being used is called an associative entity. The associative entity is important when there will be important data that is related to the relationship, as well as the entities connected with the relationship. For example, a bank may wish to keep data about each transaction (e.g., deposit, withdrawal). In this case, the relationship is placed in a box, like any entity would be, and the edge connecting the box housing the relationship to the diamond in which the relationship would have been placed becomes a directed edge, the direction of which can be in either direction [see Yourdon, Inc., 1993; Yourdon, 1989]. Figure 12.2 shows an example of an associative entity.

A unique relationship is that of supertype/subtype, which has become known as a class/subclass relationship and is shown in Figure 12.3. A common way to define a supertype/subtype relationship is by the relation “is-a.” An is-a relationship can be based upon a partition of an entity or a subdivision that is

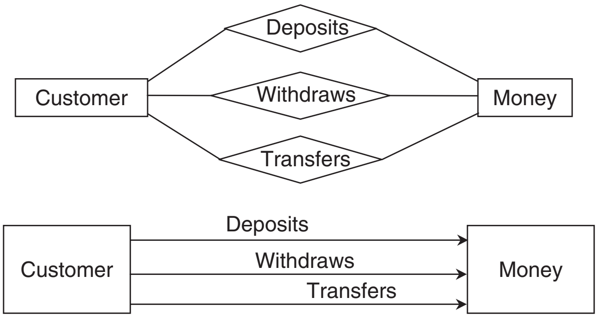


**FIGURE 12.1** Simple entity–relationship diagram.

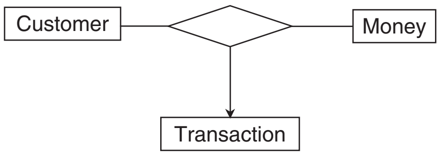


**FIGURE 12.2** Associative entity.

not a partition. For example, if there are only two types of accounts offered by a bank, the relation shown in Figure 12.3 is based upon a partition; if there was a third type of account, the relation is not based on a partition. Many of the entities and relationships associated with systems engineering that we have discussed so far are shown in Figure 12.4. Are the subtypes shown for requirement a partition or not?

Another type of relationship is called a binary relationship; this is exactly the same as the relations that we discussed in Chapter 4 and including both unary and binary relations. Unary relations are relationships among instances of the same object. These relationships can be reflexive. Figure 12.4 does not show any of these relationships because there are no instances of any entities shown. Binary relationships among instances to two different objects are binary relations and must be irreflexive. The relationship ‘‘built-from’’ is an example of a binary relationship. These binary relationships can be one-to-one, one-to-many, many-to-one, or many-to-many. Some ER methods make the finer distinction between one and zero-or-one, many, and zero-one-or-many.

## 12.2.2 Higraphs

Harel [1987] introduced higraphs as a generalization of Venn diagrams and ER diagrams. Figure 12.5 shows a higraph for a subset of the ER diagram of systems engineering shown in Figure 12.4. An entity is considered to be a set with multiple elements, called a blob. A blob is represented as an enclosed area; see system-wide requirement in Figure 12.5. Atomic sets are blobs with no other blobs contained within them; the only nonatomic blobs in Figure 12.5 are

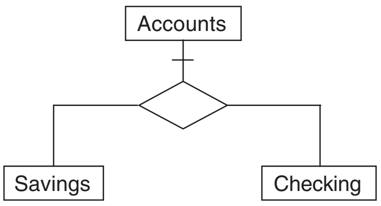


**FIGURE 12.3** Class/subclass relationship diagram.

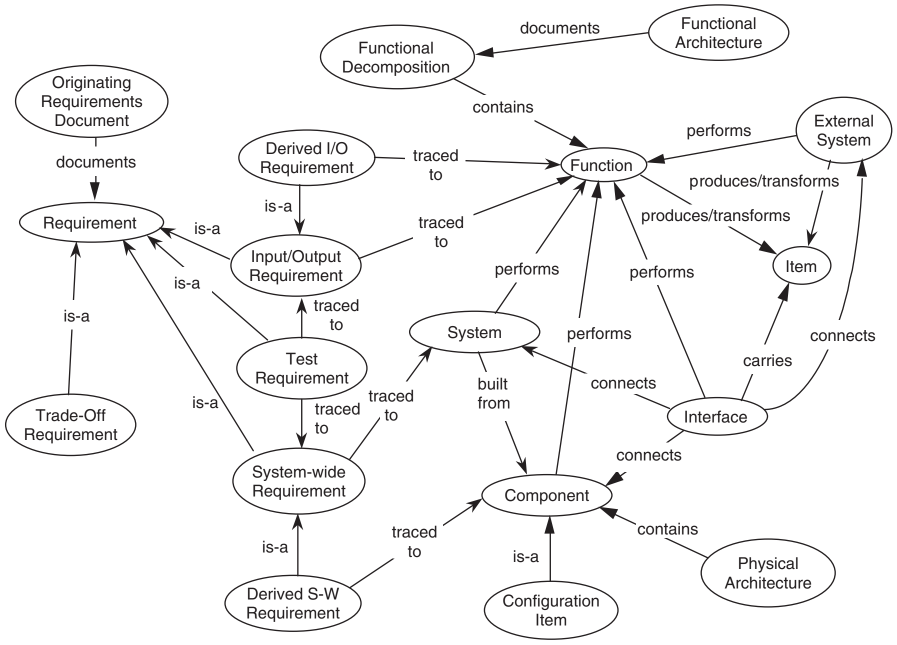


**FIGURE 12.4** Complex ER diagram of systems engineering.

requirements, time, and components. (To be correct we should have placed blobs inside the eight intersections of stakeholders' and derived requirements with input/output, system-wide, trade-off, and test requirements. However, this would have compromised the readability of the figure.) The is-a relationship from ER diagrams is replaced by representing one entity as a subset of another. Cartesian products (unordered n-tuples) are shown by placing a dashed line between blobs inside a larger blob representing the n-tuple. See the time blob, representing a four-tuple of year-month-day-hour in Figure 12.5. This concept is not in Figure 12.4.

In higraphs the relation is shown in diamonds with an undirected line entering the diamond and an arc leaving the diamond to indicate which way the relation is read.

## 12.3 PROCESS MODELING

This section addresses data flow diagrams and $N^2$ charts.

### 12.3.1 Data Flow Diagrams

Data flow diagrams (DFDs) are one of the original diagramming techniques, popular primarily with the software and information systems communities.

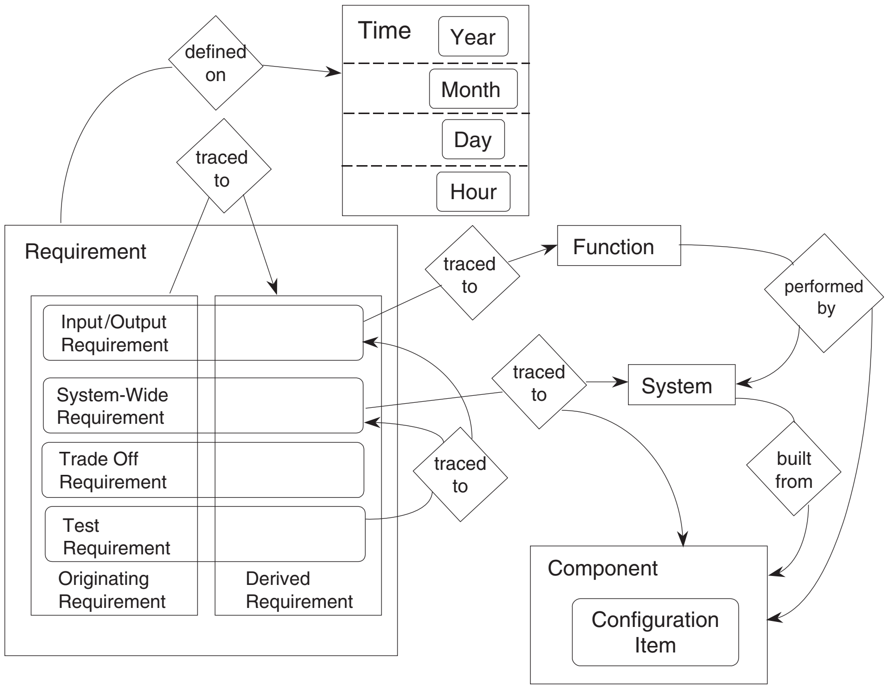


**FIGURE 12.5** Partial higraph representation of the systems engineering ER diagram.

The basic constructs of data flow diagrams, shown in Figures 12.6–12.9, are the (1) function or activity, (2) data flow, (3) store, and (4) terminator.

The circle is the most standard representation for a function. Arcs again represent the flow of data or information between functions, or to and from stores. Double-headed arcs are allowed; these represent dialog between two functions, for example, a query and a response. The labels for an arc are placed near each arrow. Branches are allowed and are depicted as forks. Branch labeling conventions in data flow diagrams are the same as those for IDEF0; see Figure 12.7. Joins are also permitted [Hatley and Pirbhai, 1988].

A new concept is introduced: the store or buffer, a set of data packets at rest. Again there are several legal representations of a store, as shown in Figure 12.8. In fact, a store is a physical solution based upon a number of problems; for example, unreliable hardware, different programmers implementing software that uses the same data, or growth potential for future enhancements. There is no need for a store in a representation of "the *essential* requirements of the system" [Yourdon, 1989, p. 151]. Stores are typically only shown on the level one functional decomposition [Hatley and Pirbhai, 1988].

The final syntactical element of data *flow* diagrams is the terminator, or external system in the language of Chapter 6. In fact, an ancestor diagram that

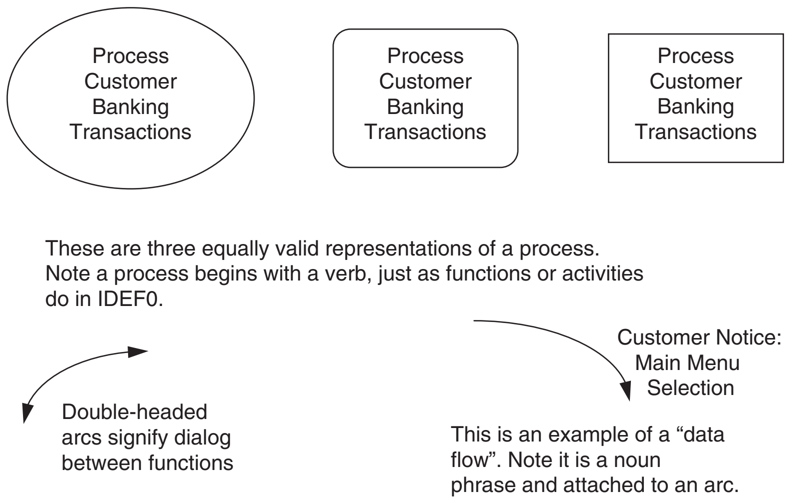

Process Customer Banking Transactions
Process Customer Banking Transactions
Process Customer Banking Transactions
These are three equally valid representations of a process. Note a process begins with a verb, just as functions or activities do in IDEF0.
Double-headed arcs signify dialog between functions
Customer Notice: Main Menu Selection
This is an example of a "data flow". Note it is a noun phrase and attached to an arc.

**FIGURE 12.6** Semantics of data flow diagram.

shows the interaction between the external systems, or terminators, and the system being designed or analyzed, are standard in data flow diagrams (see Figure 12.9). Terminators are shown in boxes with the system being placed in an oval.

Yourdon's guidelines for constructing DFDs are focused toward both correctness and communicability:

1. Choose meaningful names for the processes, flows, stores, and terminators.
2. Number the processes.
3. Redraw the DFD as many times as necessary for aesthetics.
4. Avoid overly complex DFDs.
5. Make sure the DFD is internally consistent and consistent with any associated DFDs.

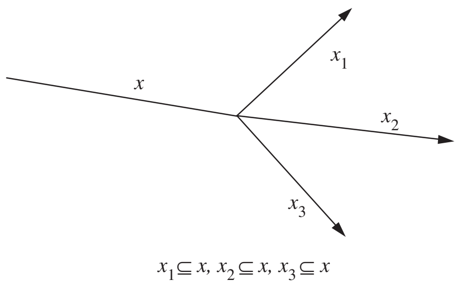


**FIGURE 12.7** Branches in data flow diagrams.

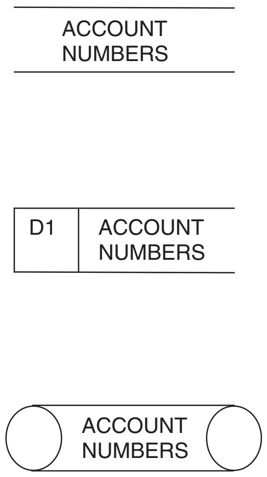


**FIGURE 12.8** Alternate representations of a store or buffer.

Note that process names are verb–object phrases and are usually capitalized. Flows are noun phrases and are not capitalized. Hierarchical numbers are recommended along with the use of *leveled* DFDs in order to avoid complex DFDs. Leveled DFDs follow many of the guidelines of IDEF0 decomposition.

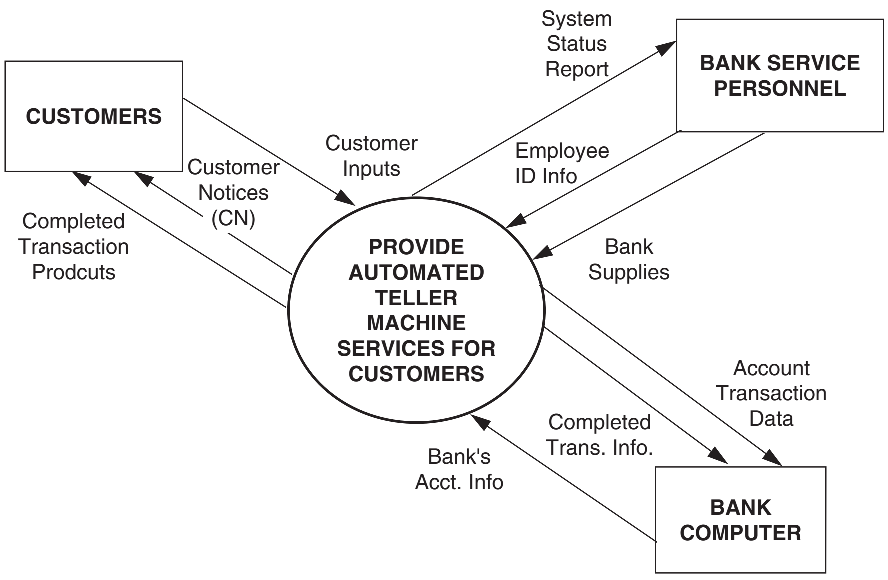


**FIGURE 12.9** Context diagram using a data flow diagram.

Finally Yourdon [1989] recommends avoiding processes and stores that are sinks and sources and labeling all flows and processes.

### 12.3.2 N-Squared ($N^2$) Charts

Systems engineers [Laws, 1990b] created $N^2$ charts in the 1960s to depict the data or items that are the inputs and outputs of the functions in the functional architecture. The $N^2$ elements provide the same description of a hierarchical decomposition of the system's functions as does IDEF0 and data flow diagrams. The $N$ functions that are a partition of some higher level function are displayed along a diagonal of the diagram with $N$ rows and $N$ columns (see Figure 12.10). Each function is shown in a rectangle with a numerical box across the top. In the off-diagonal elements are roundtangles (rectangles with rounded corners) that contain the names of the items being sent from the box in the associated row to the box in the associated column. The charts (sometimes called interface diagrams) are called $N^2$ because the chart contains $N^2$ boxes to show the flow of items within (or internal to) the $N$ functions. Every item that exits the first function and enters the second function is in the box to the right of the first function and above the second function. Items exiting the second function and entering the first function are shown to the left of the second function and below the first function. In general, items flowing from the ith function to the jth function are in the ith row and jth column. Additional boxes along the top and down the right are added as an option to show the flow of external items into and out of the set of $N$ functions, respectively. The $N^2$ charts provide the same information as IDEF0 and data flow diagrams with the exception of stores in data flow diagrams and control items in IDEF0. Ancestor diagrams are used to show the items being exchanged between the system and its external systems. Branches and joins are not used; rather, items are defined at the lowest level of decomposition relevant to a particular diagram and are then repeated as often as necessary. See, for example, the item "sensed malfunctions" in Figure 12.10.

As can be seen in the $N^2$ chart in Figure 12.10, the most obvious value of this technique is the information concerning where there is *no* interaction between functions. Systems engineers have used the $N^2$ chart to allocate functions to components such that there is minimal interaction among the components; the order of the functions is modified so that the interactions among the functions are all grouped close to the diagonal.

## 12.4 BEHAVIOR MODELING

This section addresses modeling techniques that are used to explore the dynamics of the system: behavior diagrams, finite-state machines, statecharts, control flow diagrams and Petri nets. These modeling techniques address

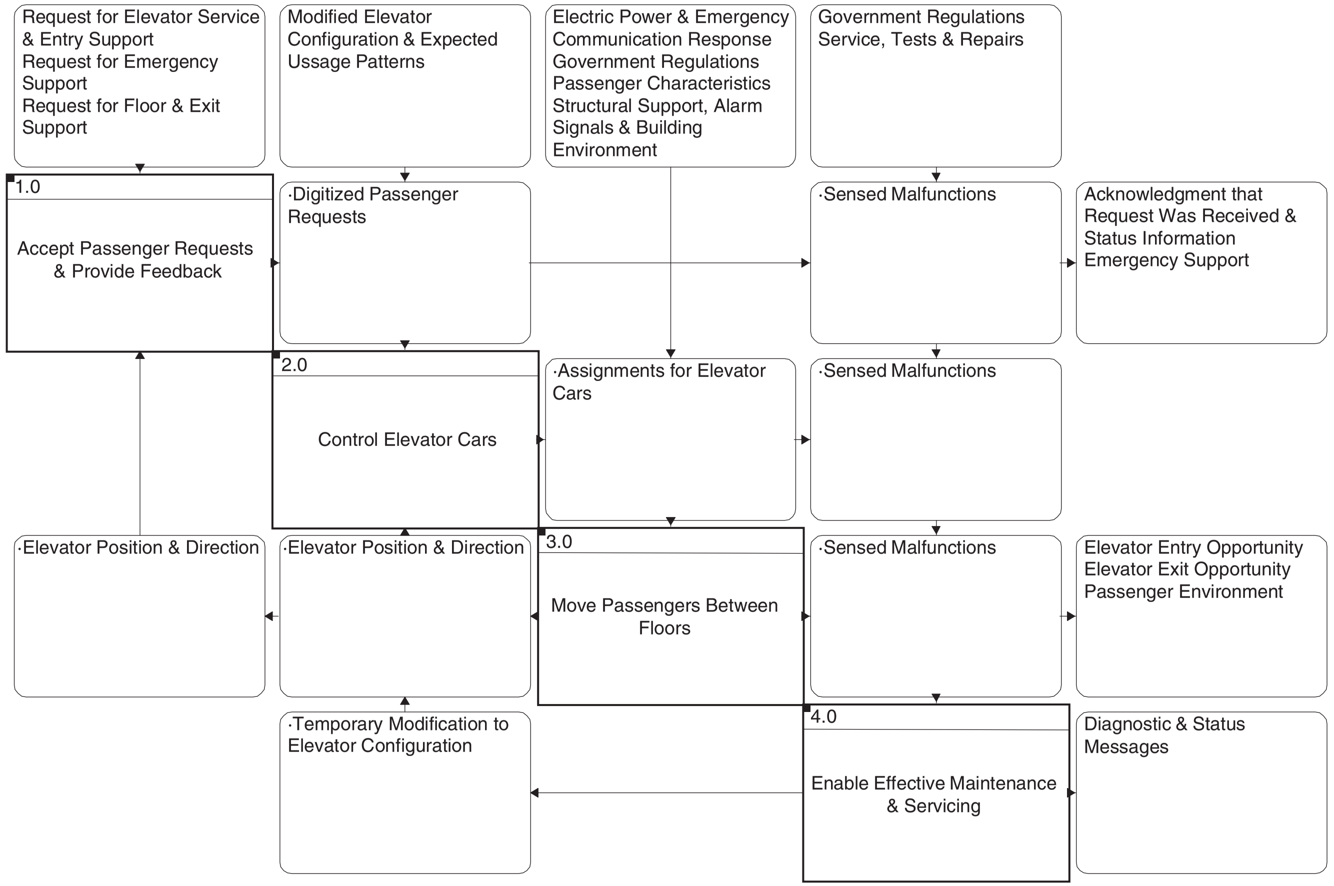

Request for Elevator Service & Entry Support
Request for Emergency Support
Request for Floor & Exit Support
Modified Elevator Configuration & Expected Usage Patterns
Electric Power & Emergency Communication Response
Government Regulations
Passenger Characteristics
Structural Support, Alarm Signals & Building Environment
Government Regulations
Service, Tests & Repairs
1.0
Accept Passenger Requests & Provide Feedback
·Digitized Passenger Requests
·Sensed Malfunctions
Acknowledgment that Request Was Received &
Status Information
Emergency Support
2.0
Control Elevator Cars
·Assignments for Elevator Cars
·Sensed Malfunctions
·Elevator Position & Direction
·Elevator Position & Direction
3.0
Move Passengers Between Floors
·Sensed Malfunctions
Elevator Entry Opportunity
Elevator Exit Opportunity
Passenger Environment
·Temporary Modification to Elevator Configuration
4.0
Enable Effective Maintenance & Servicing
Diagnostic & Status Messages

**FIGURE 12.10** An $N^2$ chart.

discrete-event behavior, which is behavior that is triggered by the occurrence of specific events.

### 12.4.1 Behavior Diagrams

Behavior diagrams [Alford, 1977] originated as part of the Distributed Computer Design System of the Department of Defense. System behavior is described through a progressive hierarchical decomposition of a time sequence of functions and their inputs and outputs. Functions are represented as verb phrases inside boxes. There is a control structure represented by lines that flow vertically, from top to bottom, through the boxes. The control structures (see Figure 12.11) are identical to that described for FFBDs above. The control lines have only one entry path into a function, but may have multiple-exit control paths. Input and output items are represented in boxes with rounded corners; their entry to and exit from functions is depicted by arcs that enter and exit the boxes, respectively.

Specific control structures for sequence, selection, iteration, looping, concurrency, and replication have been defined within behavior diagrams, just as they have been in FFBDs. A sequence of functions is connected via a vertical straight line. A selection function is denoted by a function with two or more control lines emanating from the bottom of the function. The emanating control lines must be labeled to denote the exit criterion associated with each control line.

The multiple control lines must also be joined lower in the diagram at a select node, a small circle with a + inside. Figure 12.11 shows a selection function on the top middle.

An iterate control structure is set off on a control line by two nodes. Each node is a circle with an @* inside. There is an arc from the bottom iterate node to the top iterate node with a DomainSet label that defines at what frequency or how many times the functions inside the iterate structure are to be exercised; see the bottom left of Figure 12.11.

An exit loop control structure uses a selection function to determine the point at which the repetition of a function (or set of functions) should be terminated. The exit loop control structure is set off by two vertically placed nodes (circles with an @ inside) that are connected with an arc going from the bottom node to the top node. The selection function that is responsible for ending the repetition has multiple exit control lines, one of which ends at an G node or circle with G inside. An exit loop control structure is shown in the top right of Figure 12.11. When the exit criterion for the G node is satisfied within the function, control emanates out the control line with the G node and then drops below the bottom iterate node to the L node.

The control structure denoting that functions can he executed concurrently (see the bottom middle of Figure 12.11 and Figure 12.12) is depicted by two vertically placed nodes designated by circles with & inside. In this special control structure all of the control lines below the first concurrent node are activated when control hits this first & node. The control line below the bottom

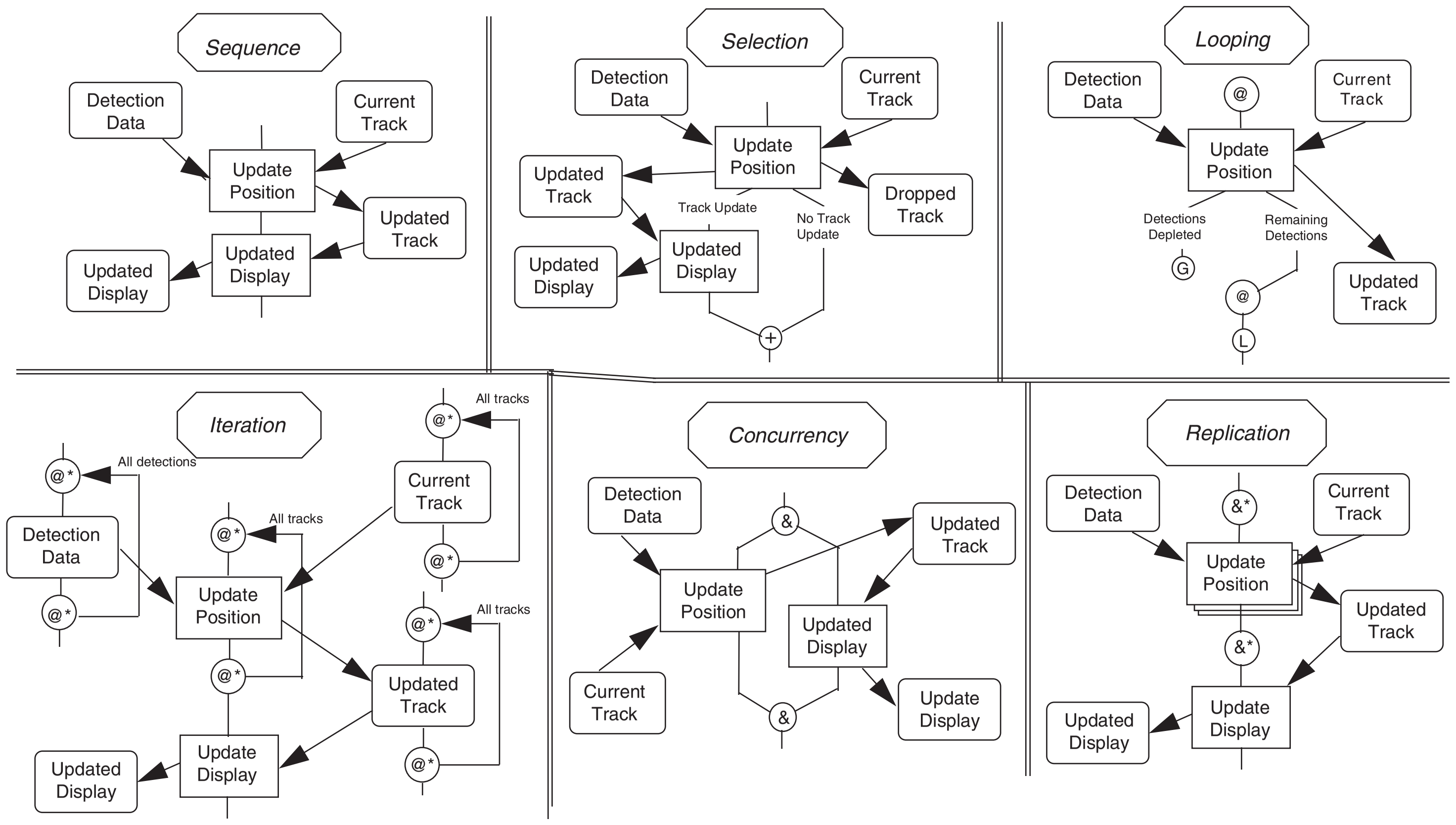


**FIGURE 12.11** Control structures for behavior diagrams.

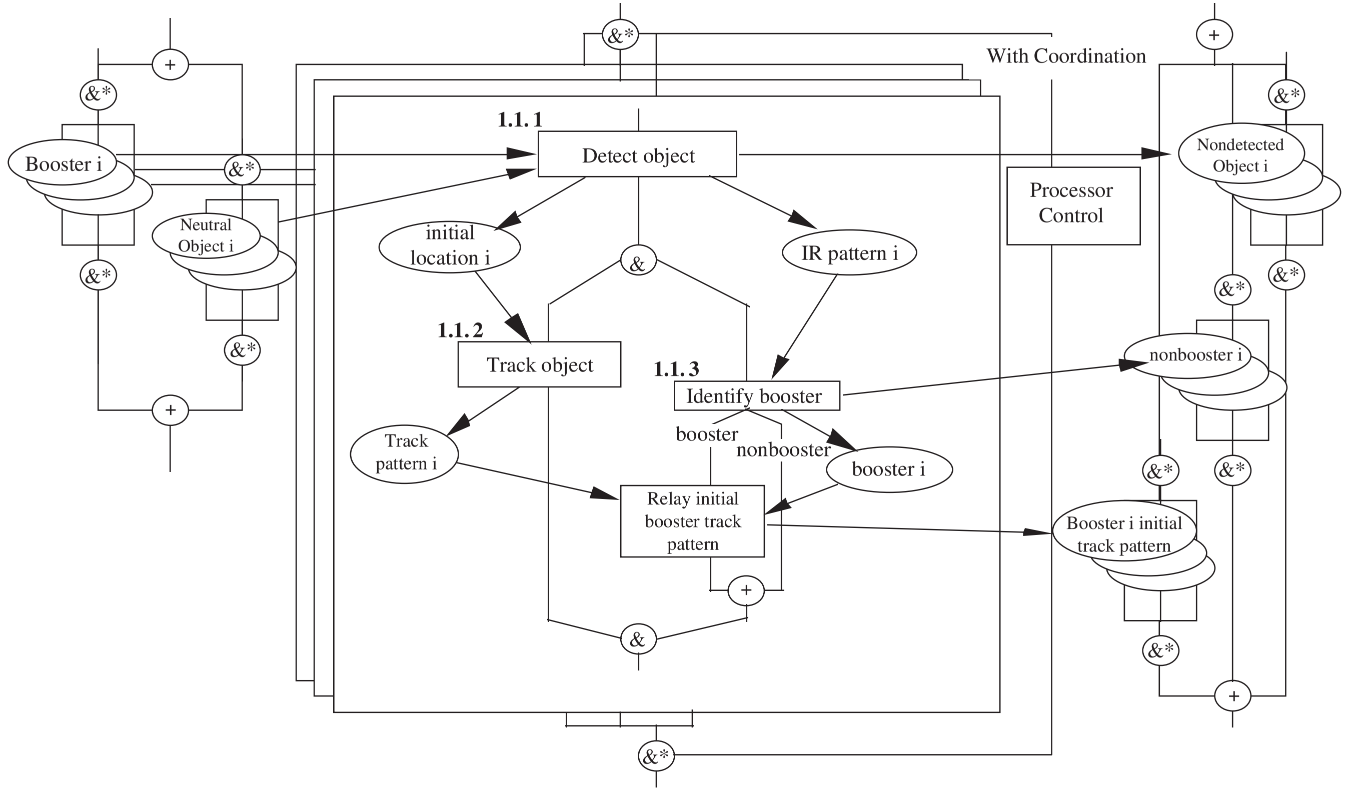


**FIGURE 12.12** Concurrent control structure.

concurrent node cannot become active until all of the functions on the concurrent control lines are finished executing.

Two vertically placed nodes with &* inside denote a replication control structure, which is a special case of a concurrent control structure. In this case an identical function is executed concurrently, presumably by multiple copies of the same resource. A DomainSet on a line that connects the upper and lower replication nodes labels the number of concurrent resources. The fact that there are multiple resources executing the same function is made visual by the symbol for a "stack of papers" on the main control line between the upper and lower replication nodes. There may be a Coordination function on the line with the DomainSet label.

Definition of the items within the behavior diagram is equally important. First, it is possible to use the sequence, concurrent, and replication control structure to organize the items (or inputs and outputs) associated with functions. Second, there are various categories of items. An item that enters the system from outside or is produced by the system for outside consumption is called an *external item*; all other items are called internal items. The roundtangle for an external item is larger than that for an internal item. All items can be hierarchically decomposed just as functions can. An item that is decomposed is called a *time item* and is represented by a clear box with a solid little square in the upper left corner. An item that is at the bottom of a decomposition is called a *discrete item*; a discrete item is represented in a shaded roundtangle. Discrete items are classified as either message, state, temporary, or global items. A *message item* is sent from a function on one control line (or process) to a function on a different control line (or process) and the message item triggers the receiving function to execute as soon as the function is enabled by the control structure. *Global items* do not trigger the receiving function to execute. *State items* are input to and output from functions on the same control line and are therefore always internal items. A state item is not a trigger. *Temporary items* are for special purposes.

### 12.4.2 Finite-State Machines and State-Transition Diagrams

Machines, a modeling domain for dynamic systems, are partitioned into finite-state and continuous. Finite-state machines (FSMs) [Denning et al., 1978] have only discrete-valued inputs, outputs, and internal items. Continuous machines allow continuous and discrete inputs, outputs, and internal items. Continuous machines are sometimes called analog machines. When digital computers became more popular than analog computers, FSMs became the major focus of attention in engineering due to the finite-state nature of digital computers. *Even* so continuous and discrete signals are usually handled very differently by a digital computer. The continuous variable (e.g., speed or internal temperature of the elevator car) is represented by a word that typically contains many more bits than the variable has significant digits. On the other hand a digital variable (e.g., operating mode such as fully operational or partially operational or not operational, and direction of a specific elevator car such as up or down) is usually represented by a symbolic word that has a relatively few number of states, say less than 10.

Finite-state machines are usually divided into sequential and combinational; see the machine partition in Figure 12.13. The focus here is on the sequential FSM, as represented by a state-transition diagram (STD). A combinational FSM is one in which its current outputs are characterized only by its current inputs, a condition of having no memory that is often not met. The sequential FSM allows past inputs to play a role in the determination of the current outputs, thus enabling the FSM to have a memory. There is a formal mathematical theory for an FSM, providing some interesting theoretical results and simulation capability.

The STD models the event-based, time-dependent behavior of a system. Recall from Chapter 7, the *state* of a system is defined to be its status, as defined by as many variables as needed to determine the system's ability to meet its missions. The *mode* of a system is its operating condition, such as off, idling, or moving for an automobile. It is the mode of a system that should be modeled by an STD. However, as shown in Figures 12.14 and 12.15, there is a fine line between the modes of a system and the functions of a system.

Boxes (or ovals) and arcs are the syntactical elements of STDs; the boxes represent system modes and the arcs represent the direction of mode change. Typically the arcs are labeled to show both the input stimulus (or event that triggers the mode change) and the action or output taken by the system in response to the event. The event and output are typically separated by a slash or horizontal line: event/output. Figure 12.14 shows a partially completed STD for an automatic teller machine. This STD is incomplete because the transitions to the four customer choices are not labeled; the transitions from the four customer choices are not depicted via arcs. It is possible that each might be completed successfully or canceled. The withdrawal might be denied. In each case the customer can choose another transaction or not. Figure 12.15 shows an STD for an elevator car (this figure is a modification of one found in Gomaa [1993]).

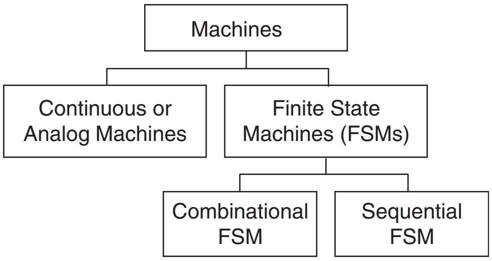


**FIGURE 12.13** Partition of machines.

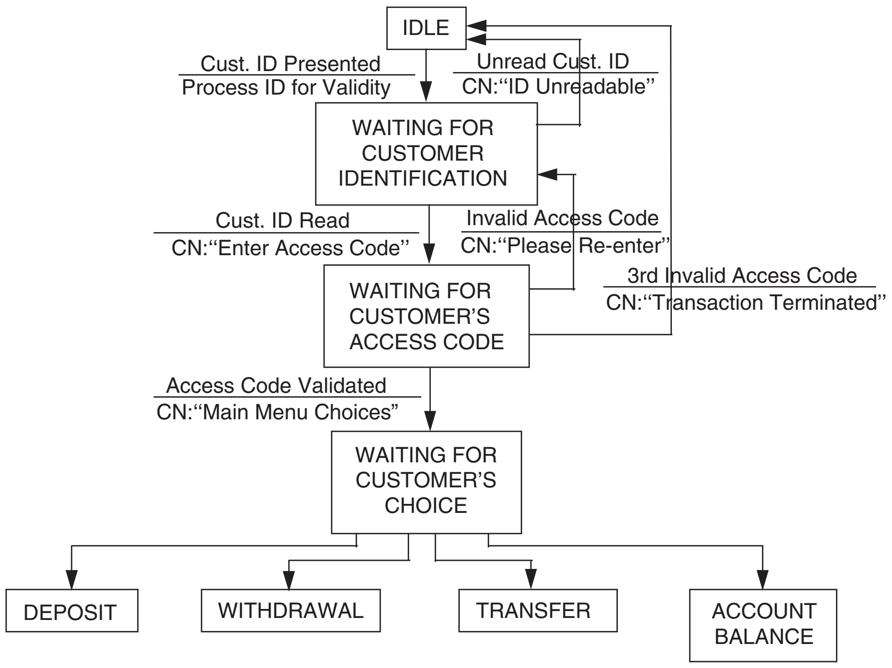


**FIGURE 12.14** State-transition diagram for an ATM.

It is important to note differences between the view provided by an STD and the view provided by one of the process models (DFD, IDEF0). The STD makes no attempt to provide a functional partition of the top-level system function or any function that is part of its partition. Rather the STD focuses on key triggering events that will cause the system to transition from one operational mode to another and identify any key system outputs produced as a result that transition. Similarly process models are not required to capture the system's operating modes. In Chapter 7 the functional architecture was defined to capture the system's operating modes as the initial decomposition of the system's functions.

### 12.4.3 Statecharts

Statecharts are a generalization of higraphs by Harel [1987] to extend the notions of STDs. This generalization of an STD is based on fonnal mathematical principles and leads to theoretical results and simulation models.

A major criticism of STDs has always been that the entire diagram must he contained on one level, meaning that an STD for a large system quickly becomes unintelligible and unmanageable. Statecharts, by exploiting the subset

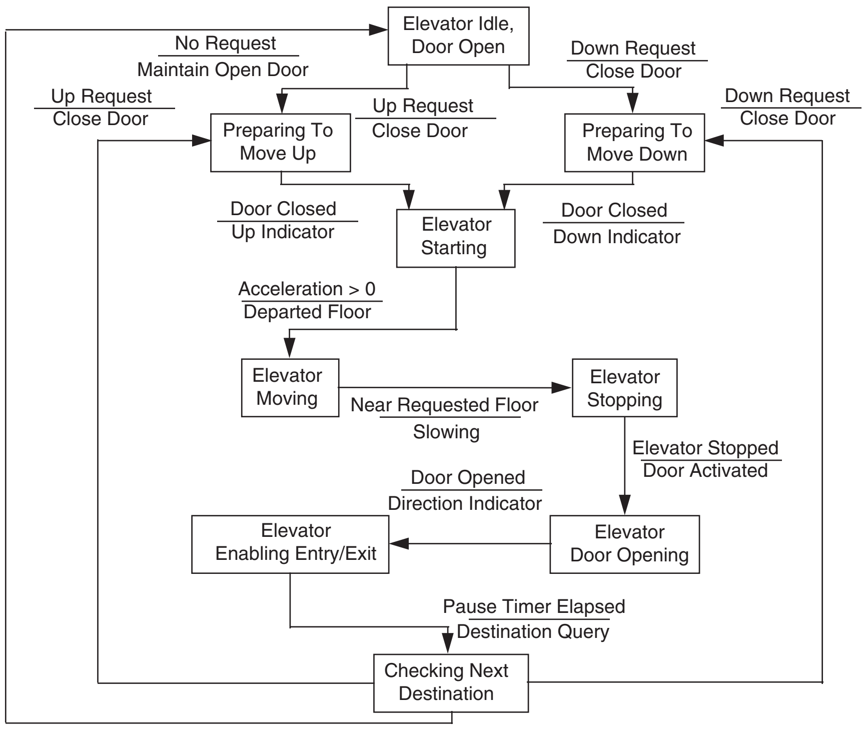

Elevator Idle, Door Open
No Request / Maintain Open Door
Down Request / Close Door
Up Request / Close Door
Up Request / Close Door
Down Request / Close Door
Preparing To Move Up
Preparing To Move Down
Door Closed / Up Indicator
Elevator Starting
Door Closed / Down Indicator
Acceleration > 0 / Departed Floor
Elevator Moving
Near Requested Floor / Slowing
Elevator Stopping
Elevator Stopped / Door Activated
Door Opened / Direction Indicator
Elevator Enabling Entry/Exit
Elevator Door Opening
Pause Timer Elapsed / Destination Query
Checking Next Destination

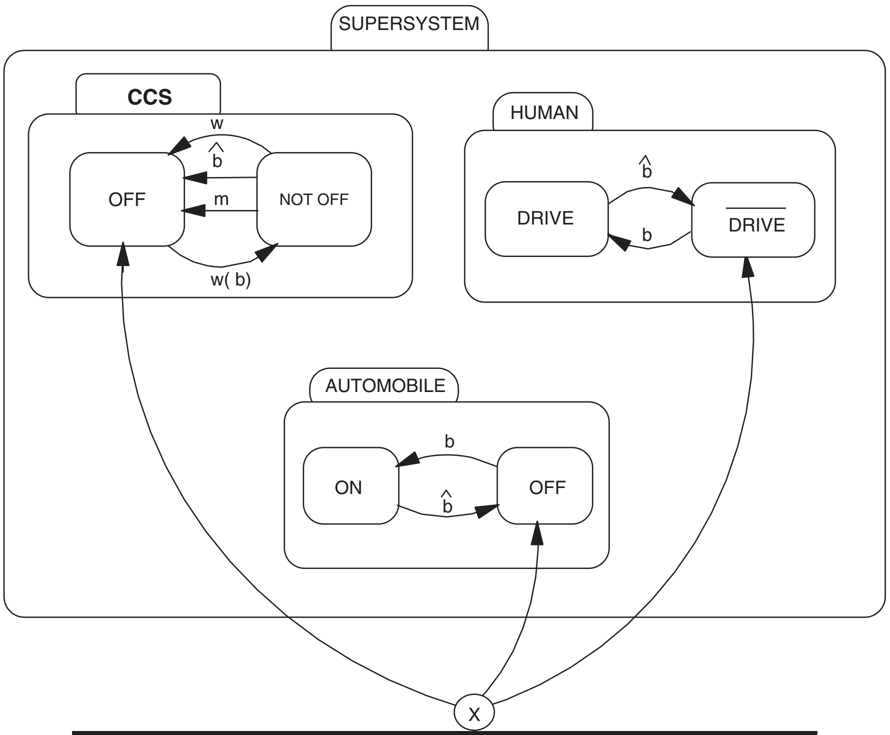
**FIGURE 12.15** State-transition diagram for an elevator car.

properties of higraphs, provide a means to develop hierarchical STDs. The atomic blobs in a statechart are singleton, or atomic, states.

Figure 12.16 presents an external system representation of a cruise control system (CCS) [Charbonneau, 1996]; the human operator and the remaining components in the car are the external systems. Noting how the action "b" and "b hat" affect all three subsystems by causing simultaneous state transitions with a single event demonstrates an extension by statecharts over the STD. The states to which the X label is connected indicate the initial condition or state for the three systems. Note that inside state ON for the automobile are the states of acceleration, deceleration, and maintain speed.

Arcs in statecharts are labeled, just as they in STDs. Inside the system the initial state is identified by finding the arc that emanates from a black dot; the state that this arc enters is the initial state of the system; see Figure 12.17.

Figure 12.17 presents the decomposition of the NOT OFF state of the CCS. The OFF state was not decomposed. Recall from the discussion on higraphs that the vertical dotted line indicates a Cartesian product. The INDICATOR and the SYSTEM STATUS blobs are independent, defining a Cartesian product. Both INDICATOR and SYSTEM STATUS have two states. The state DEAD for the INDICATOR is not decomposed.

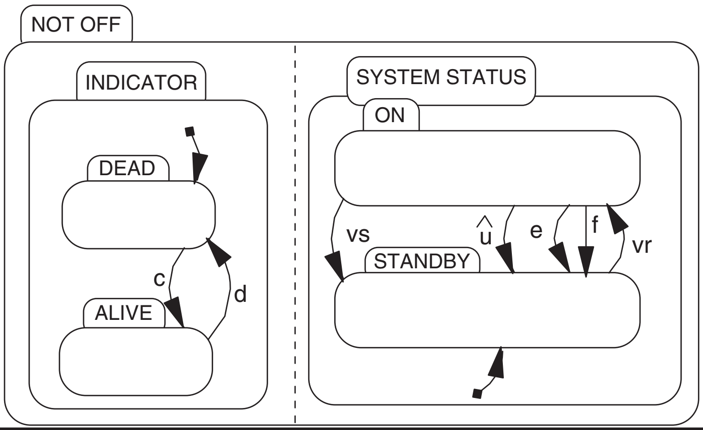


| ARC LABEL | DEFINITION |
|---|---|
| b | turn on car |
| b hat | turn off car |
| m | accident occurs |
| w | depress on/off button |

**FIGURE 12.16** External system statechart for a cruise control system (after Charbonneau [1996]).

The ability to represent unordered *n-tuples* in higraphs enables statecharts to depict states as being the orthogonal composition of elements from sets of states. When the initial state is an n-tuple, there must be *n* initiating arcs to define which element of the set of *n-tuples* is the initial state. Similarly, when there is a transition from (to) a state that is part of an n-tuple to (from) one that is not, the arc must be joined by an arc from (must branch to) $n - 1$ other arcs from other elements of the *n*-tuple.

Figure 12.18 shows the three states for ALIVE in Figure 12.17 that are associated with the INDICATOR. The "w" activity in this third-level chart is the same "w" in the supersystem top-level chart. This single activity, "w,"



| ARC LABEL | DEFINITION |
|---|---|
| c | circuit closed (good bulb or fuse) |
| d | circuit open ( bad bulb or fuse) |
| e | brake depressed |
| f | clutch depressed |
| u | wheel revolutions > 7920/ (pi*r) where r is the wheel radius in inches |
| vr | push button to resume / set |
| vs | push CCS button to standby |

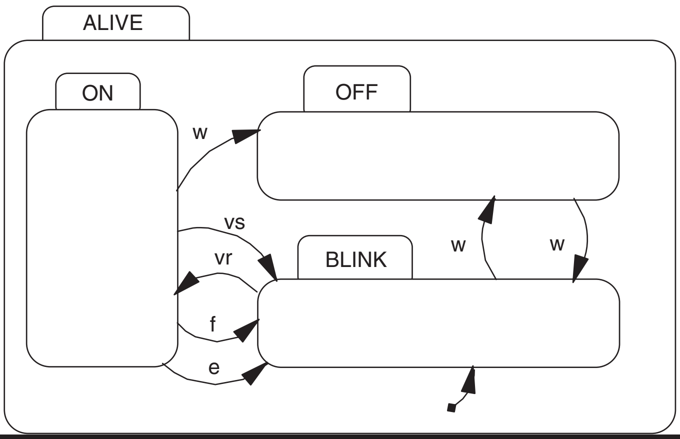
**FIGURE 12.17** Decomposition of the "not off" State (after Charbonneau [1996]).

causes state transitions both in depth (all sublevels) and in breadth (all subsystems).

Figure 12.19 depicts the decomposition of ON in Figure 12.17. The circled "H" is the only new concept introduced in this diagram. When the ON state is entered from the STANDBY state, it automatically reverts to the conditions it was in before it transitioned to STANDBY. The circled "H" is read as *Historical*. If the ON state is entered from the NOT ON state, it defaults to maintain because there is no historical reference.

Figure 12.20 integrates the statecharts (Figures 12.17–12.19) for the CCS with the additional decomposition for the STANDBY state shown in Figure 12.17 for SYSTEM STATUS.

When an event such as an interrupt causes a transition from many states to a single state, an STD implements this with many arrows to depict the effect of a single event. In a statechart an arrow can go from a state (blob in higraphs) containing several atomic states (blobs). As a result an interrupt can be shown with a single arrow from an aggregate state, demonstrating how the number of



| ARC LABEL | DEFINITION |
|---|---|
| e | brake depressed |
| f | clutch depressed |
| vr | push button to resume / set |
| vs | push CCS button to standby |
| w | depress on/off button |

**FIGURE 12.18** Decomposition of the alive state for the indicator (after Charbonneau [1996]).

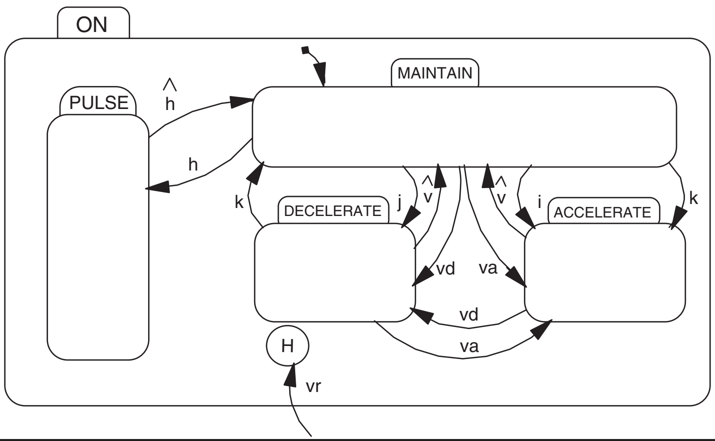
these arrows can be reduced with statecharts. See the transitions between NOT OFF and OFF in Figure 12.20.

Another extension of statecharts is the ability to nest transitions by using labels such as a/b. This means that transition ‘‘a’’ will cause another transition ‘‘b.’’ located elsewhere in the statechart, to occur. Harel [1987] calls this broadcasting because one event can broadcast a trigger that generates a chain reaction of one or more transitions throughout the statechart.

## 12.4.4 Control Flow Diagrams

Control flow diagrams (CFDs) are used in conjunction with data flow diagrams and model changes in the system's operating mode, thus turning on or off or restructuring sets of the system's functions. As defined by Hatley and Pirbhai [1988], the control structure of a system receives status information from external systems and sends such information about the system to these external systems. Control flows are typically discrete variables that can be modeled symbolically.

Control flow diagrams mimic DFDs in syntax and semantics, except for one additional symbol. In fact, the functional decomposition of the two should be



| ARC LABEL | DEFINITION |
|---|---|
| h | non drive wheel RPM not equal to drive wheel RPM |
| h(hat) | non drive wheel RPM equal to drive wheel RPM |
| i | wheel RPM decrease from set speed |
| j | wheel RPM increase from set speed |
| k | wheel RPM match to set speed |
| v(hat) | release the CCS button |
| va | va = push CCS button to accelerate |
| vd | push CCS button to decelerate |
| vr | push button to resume / set |

**FIGURE 12.19** Decomposition of the ''on'' state for the INDICATOR (after Charbonneau [1996]).

identical. These two types of diagrams could be superimposed to form a single diagram; some authors recommend this. There is a context diagram of control that shows the relationship of the system with the external systems, for example, the passing of status information concerning the changing of modes. The control arcs are typically shown as broken lines to distinguish them from data flow. The additional symbol is a bar or solid line, shown either vertically or horizontally. All of the bars on a particular diagram represent an FSM behavior for the functional element being decomposed by the functional elements shown on the joint DFD/CFD diagram.

### 12.4.5 Petri Nets

Petri nets (PNs) are based on a rigorous mathematical definition leading to an executable simulation model and having formal mathematical properties. Petri

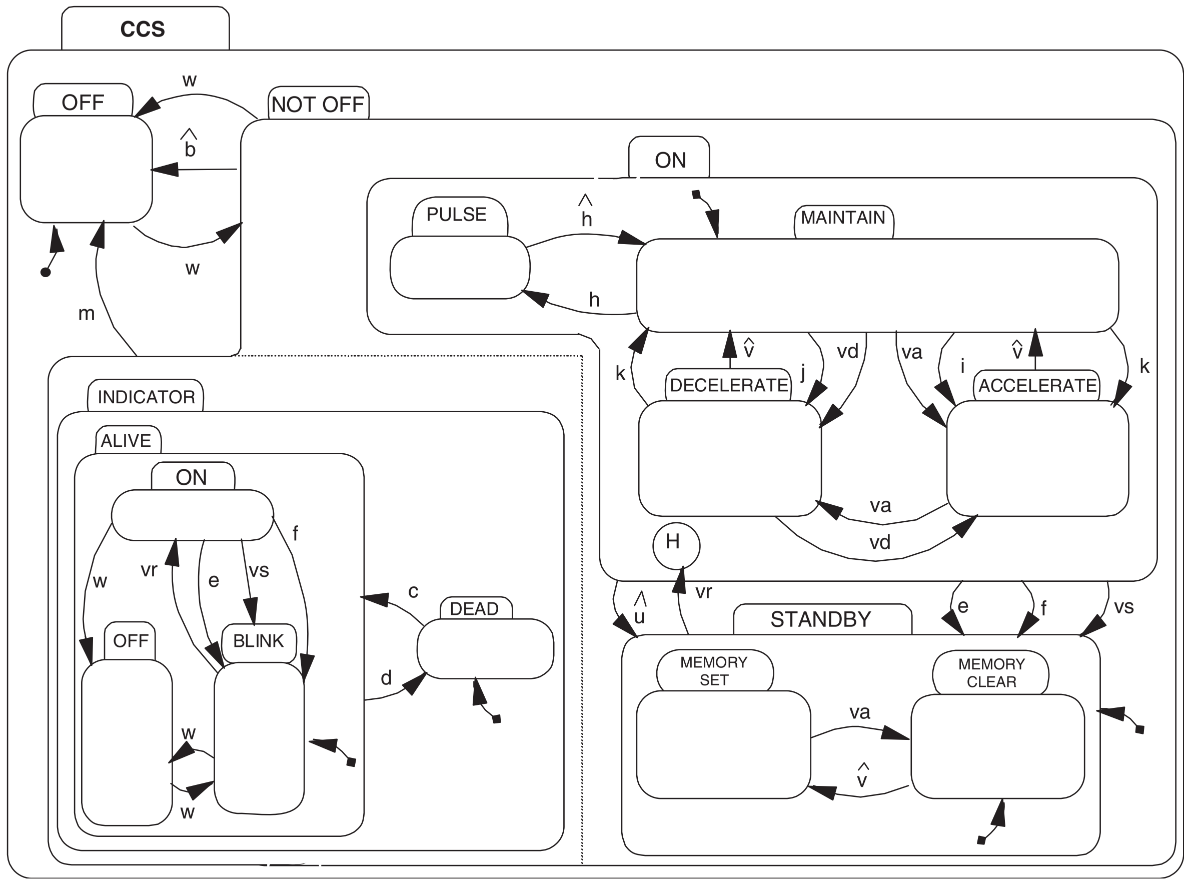


**FIGURE 12.20** Statechart for a cruise control system (after Charbonneau [1996]).

nets capture the precedence relations and structural interactions of potentially concurrent and asynchronous events.

Mathematically, a PN is a bipartite directed multigraph. The two node types are the place (depicted by a circle) and the transition (depicted by a bar or rectangle); see Figure 12.21. The arcs are restricted to connect places to transitions or transitions to places. In addition, PNs contain markings or a mapping of tokens to places. A transition can fire when a token is present in each of the places that have arcs entering the transition. So $t_1$ can fire in the top half of Figure 12.21; after the firing the transition places one token in each place that has an arc from the token.

A Petri net is defined a four-tuple, or four sets:

$P = \{p_1, p_2, \ldots, p_n\}$, the set of places,
$T = \{t_1, t_2, \ldots, t_m\}$, the set of transitions,
$A = \{P \times T\} \cup \{T \times P\}$, the set of input and output arcs,
$M = \{m_1, m_2, \ldots, m_n)$, the net's initial markings (drawn as dots).

The state of the PN is defined by the marking. In ordinary PNs, the tokens are indistinguishable. The existence of one or more tokens at a place indicates the availability of a resource for the fulfillment of a condition that is associated with a transition. Figure 12.22 provides two examples of simple systems for concurrent processing and a simple communications protocol.

There are many extensions of ordinary PNs. Colored PNs allow more than one type of token; timed PNs allow varying times for the transitions to occur; and stochastic PNs allow stochastic transitions. See Murata [1989] for a good overview of this topic.

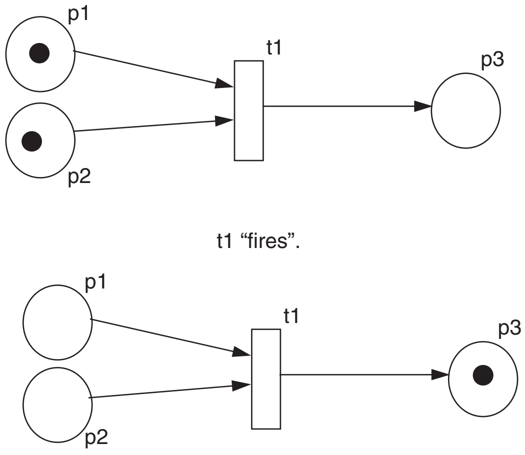


**FIGURE 12.21** Simple Petri net example.

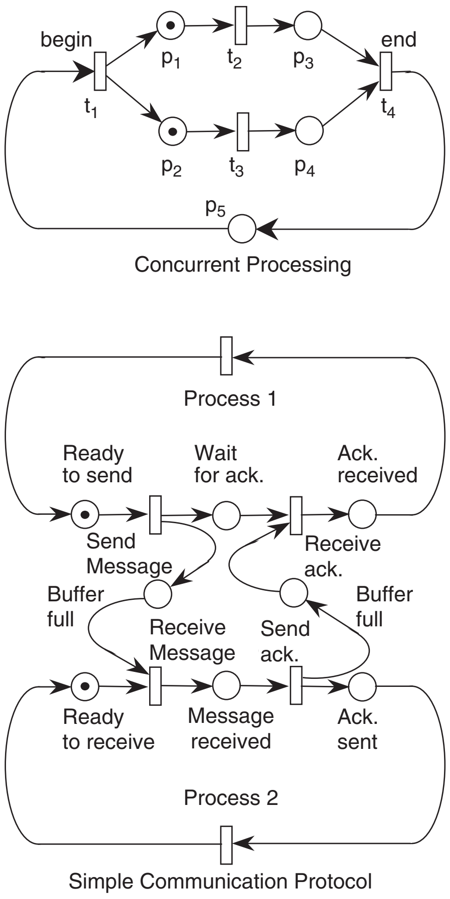


**FIGURE 12.22** Petri net models of simple system architectures (after Murata [1989]).

## 12.5 SUMMARY

The complete model-based examination of a system requires at least the use of data, process, and behavior modeling. When using multiple approaches to model a single system, balancing or aligning the elements of the multiple models is critical. Several approaches for each of these model categories were presented in this chapter.

Data modeling is the specification of data entities and relationships between pairs of entities at a minimum. In addition attributes of each data entity can be developed. Entity relationship diagrams provide the basic data modeling capability and are probably the most widely used of the data modeling techniques. Higraphs extend the data modeling of ER diagrams by adding the representation of subset and cross-product relationships among entities.

The three process modeling techniques covered in this book are IDEF0 (see Chapter 3), data flow diagrams, and $N^2$ charts. Each of these techniques captures the relationship among functions in the functional decomposition by representing the transformation of inputs into outputs. The $N^2$ charts are the simplest but least graphical representation of a process model. Data flow diagrams are widely used but least standardized of all of the modeling techniques discussed in this book. IDEF0 was quite standardized since it was created in the 1970s; the National Institute of Standards and Technology (NIST) has created a FIPS for IDEF0, thus making an IDEF0 model easy to read and comprehend. Distinctions between these techniques are that IDEF0 defines at least one control item for each function while the other techniques treat control items as inputs or ignore them. IDEF0 also includes the construct of a mechanism to represent the resources that execute the function, making it the only process modeling technique general enough to represent portions of the allocated architecture of the system. The control could be a trigger to activate the function or policy instructions for implementing the function. Data flow diagrams contain the concept of a data store that is useful during design to define which data elements will be contained in a specific database.

Five modeling techniques for behavior modeling were described in this chapter. FFBDs were described in Chapter 3. Control flow diagrams are the simplest and by far the least useful. Control flow diagrams add the concept of transitions to data flow diagrams, which suggests that the system modes and functions are identical. While this assumption may be useful in simple systems and software products, it is very limiting in most real systems of hardware, software, and other resources. Behavior diagrams come from the systems engineering discipline and add FFBD control structures on top of a process model to represent serial, concurrent, repetitive, and replicated process execution as well as the rule-based selection of functional outputs. While no formal mathematical model has been published to define these control structures, they have been implemented in software, suggesting that such a formal model exists and could be specified. Finite-state machines and state-transition diagrams are used in other engineering disciplines, but are not sufficiently general to capture the rich behavior possible in a complex system, for example, concurrent processing. Statecharts are a generalization of state-transition diagrams that enable many of these limitations to be overcome but still provide a limited semantics and syntax for modeling complex systems. Petri nets are the only behavior modeling technique with an underlying mathematical model that defines what can be done and provides analytical results without simulation. Unfortunately, Petri net models are quite sophisticated and are not likely to be employed on a widespread basis in the engineering discipline for systems until their potential benefits are much better justified and become widely known.

## PROBLEMS

12.1 Expand the ER model in Figure 12.4 to be a complete representation of the entities and classes discussed in Chapter 2 for the systems engineering process.

12.2 Create a higraph that is a complete version of Figure 12.4.

12.3 Create a complete behavior diagram model of the process of engineering a system based upon the IDEF0 model of the engineering of a system in Appendix B.

12.4 Create a statechart for the functioning of the air bag system from the time the driver turns the car on until an accident occurs that activates the air bag or the driver turns the car off.