Chapter 1

Introduction to Simulation-Based Systems Engineering

COPYRIGHTED MATERIAL

Simulation-based systems engineering, discussed in this book, utilizes the concept of interacting concurrent processes as an integral part of the systems engineering process. Both discrete-event simulation and Markov models are applied to model interacting concurrent processes and resources. A key factor to understanding complex system behavior and structure is through the use of graphical timelines and simulation traces of the interacting concurrent processes and resources. Timelines and simulation traces explicitly show how these interactions affect system effectiveness and performance, contributing to understanding the systems design problem.

A commonsense approach to understanding, designing, analyzing, and evaluating complex systems is presented throughout this book. It begins by discussing traditional ways of thinking about systems and then shows how one of these views, the operational view of systems, can be best expressed using interacting concurrent processes. A graphical language, Operational Evaluation Modeling (OpEM), is introduced (details are in Chapter 2 and the Appendices) that provides a natural way to describe interacting concurrent processes and implement them using simulation or Markov models. A graphical discrete-event simulation library, Operational Evaluation Modeling for Context-Sensitive Systems (OpEMCSS), is discussed, throughout this book, which implements the OpEM graphical language and provides a means to experiment with complex systems and do simulation-based systems engineering. A large number of example models are described that illustrate how to use the OpEMCSS library blocks to model complex systems. These examples are presented so that a large spectrum of readers can understand them. Examples of complex systems abound. Consider

Simulation-Based Engineering of Complex Systems, Second Edition, by John R. Clymer
Copyright © 2009 John Wiley & Sons, Inc.

a distributed vehicle traffic control network located in a large city. Each major intersection has a vehicle traffic light controller to determine traffic light timing. In this system, each traffic light controller uses its perceptions about incoming traffic flow to optimize light timing, thus minimizing local vehicle waiting time. In Chapter 9, it is shown that the result of each traffic light controller adapting light timing to accommodate traffic flow coming from other intersections is to minimize the average waiting time in the entire network. Global minimization of traffic waiting time results as a consequence of the emergent behavior of this system, which is the self-synchronization of each traffic controller's light timing with other controllers.

As light timing control in the overall traffic grid evolves in the way discussed above, a complex but definite fractal pattern in network operation, north–south red-to-green transition times, emerges out of an initial random light pattern. The emergent behavior of the traffic grid cannot be explained through an understanding of each controller alone. Understanding only comes when we study the interactions of the controllers as they adapt their behaviors in response to perceived information about incoming traffic flow, achieving self-synchronization of all traffic light controllers in the network.

Most people possess, as common sense, the fundamental ability to understand such complex systems. Each one of us formulates goals for ourselves all the time to solve problems confronting us. Once we state our goal, we visualize a set of tasks or steps that takes us from our current situation to one satisfying our goal. If we can expect help from others to execute these tasks, we organize the tasks into a sequential and concurrent arrangement, more commonly called a plan. The execution of our plan by a group of people hopefully leads to goal satisfaction. In engineering management, military command and control, and business management organizations, computer simulation programs are often used to optimize plans and make them robust in the presence of varying contingencies.

In this chapter, general systems are defined and some of the more commonly used system models are described. Next, each reader is asked to analyze a goal-oriented activity he or she already knows how to do and to build a functional flow model of this activity. Functional flow models are an important part of the systems engineering process.

Simulation and artificial intelligence (AI) is introduced as a systems design approach where intelligent agents make decisions that give a system "requisite variety." Networks of agents are complex and can have emergent behaviors, making them complex adaptive systems (CAS). How the traditional systems engineering process must be modified to design CAS is provided in a discussion of expansionism versus reductionism in Section 1.1.4.

How simulation programs work is introduced in this chapter and discussed further in Chapter 2. To provide the necessary background for this discussion, an introduction to the OpEMCSS graphical simulation library and the OpEM graphical language is presented first. Next, how a sequential process model works is discussed, and the same model is used to introduce the idea of run output convergence. Next, an OpEMCSS simulation of a producer–consumer process

is used to explain how an interacting concurrent model works, and this model is also used to discuss the idea of sensitivity analysis using model output data.

The systems engineering process and life cycle for bringing complex systems into being are described. The concept of “levels of system description” is discussed (Section 1.3.2) in terms of the systems engineering life cycle. An OpEMCSS simulation of a systems development process is described to provide the reader with an explicit understanding of this process and to focus the chapter discussion on how simulation-based systems engineering can be used to answer system design questions early in the system life cycle when design changes have the least impact on project success. The chapter concludes with a summary.

1.1 DEFINITION OF COMPLEX SYSTEMS

In general, a system consists of a network of interrelated and interacting components, working together with the common objective of fulfilling some designated need. The components of a system include facilities, equipment (hardware and software), material, services, data, skilled personnel, and techniques required to achieve, provide, and sustain system effectiveness. A system can be organized into a component hierarchy where components at one level of the hierarchy can be broken down into a set of components at the next lower level. The rational for component decomposition is to simplify component interfaces by minimizing interactions with other components. A good component hierarchy is required to facilitate systems engineering management, including the development of the project work breakdown structure (WBS). We will show later that, for complex systems, a hierarchy of networks is better for system description, analysis, and evaluation.

For example, consider a transportation system that is decomposed into a set of top-level components that includes a vehicular system, airline system, and a railroad system. The airline system is further decomposed at the second level into a set of components that includes an airplane system. The airplane system is decomposed at the third level into a set of components that includes a propulsion system, a fuel system, and a communications system. Each of these components can be developed relatively independently by separate product development teams, which greatly facilitates systems engineering management. However, for a system of systems (SOS) design problem, where these top-level components (vehicular system, airline system, railroad system) interact, a network of systems is more usable. The systems engineering management problem is discussed further later in the chapter.

A system must have a purpose in that it must be functional and able to respond to an identified need. If a rock lying on the ground serves no apparent need, then it can clearly not be a component of a system by our definition. However, suppose the rock borders a garden. In this case, the rock serves an identified need, and it is an integral part of the garden “system.” But, is the garden border rock functional? The rock does not appear to actually do anything to respond to an

identified need, although it does have a useful purpose. According to Blanchard and Fabrycky (1998) the rock is a structural component of the system rather than an operational component that performs a function. The plants in the garden are the operating components because they transform air, water, and minerals into plant growth.

However, if we view this system from the outside looking in, we would see that the rock provides information to the environment defining the boundary of the system. Therefore, one could say that, since the rock border provides information as an output of this system, it is functional.

What do we mean by functionally interacting with other entities? How can a system component be functional? In mathematics, a function transforms inputs into outputs:

$$\mathbf{Y} = \mathbf{F}(\mathbf{X})$$

Let **X** be a vector of input signals, **Y** be a vector of output signals, and **F** a vector of equations:

$$\mathbf{Y} = \mathbf{F}(\mathbf{X}) = (f_1(X), f_2(X), \ldots, f_n(X))$$

For example, the rock border reflects light to define the boundary of the system. The input X to this function is light from outside the system that is reflected back from the rock border to again pass outside the system as output Y where it is transformed into information by an observer viewing the system, defining the system boundary for the observer.

Thinking about our functional rock, it can be seen that there are three different views used to visualize this system. One view is the traditional structural system model where interface mechanisms connect one component to another. The second view used is the functional model where inputs are transformed into outputs by each function in a network of functions. The third view is the external view that looks at the system from the outside and considers the interactions of the system with its external systems and the environment.

Traditional electronic circuits and analog control systems have inputs **X** that are transformed into outputs **Y** where all are continuously varying signals. Such systems are simulated using continuous-time models where time is incremented in very small, constant time steps. This kind of simulation model is not within the scope of this book; however, the commercial version of OpEMCSS called CASSim, which is a product of FORELL Enterprises Inc., includes a continuous system simulation block.

In this book, we consider discrete-event systems where system interactions occur at discrete points in time, having a variable time step. A message passing system, where system components send messages to each other at discrete points in time, is a discrete-time analogy to the electronic circuits or analog control systems discussed above. A message received by a component is analogous to an input signal **X**. The message X is transformed into output message Y that is sent to other components. In a message-passing system, messages can represent information, data, knowledge, energy, or material flowing through the network. The

message-passing model is a generalization of the traditional electronic circuits and analog control systems model.

The *structural* view of a message-passing system is an architectural diagram that describes the system architectural components, connecting links from component to component, and often the names of the message links required for component communication in the network.

The *operational* view of a message-passing system is a behavior diagram that describes the sequence of tasks that must be performed during a period of time in order to transform X to Y, and it represents all context-sensitive interactions among the concurrent tasks performed by a system component. This model also describes when to send the required messages to other components. One type of operational model that shows the flow of tasks required to achieve a goal is called a *functional flow* model.

The *external* view is a system context diagram that defines the boundary between the system and its external systems and environmental context. Also included in this view is the interface between the external systems and the system. The names of the message links required for external input/output communications are often shown.

The concept of a function, which is a major building block in functional flow models or the operational view as discussed above, is described as a transformation from input X to output Y such that a network of these functions describes system behavior. In order to design a complex system, the traditional functional flow model must be expanded into a context-sensitive system (CSS) model, discussed throughout this book, where the functions are represented as process threads. In complex systems, these process threads adapt to the current environmental or system state through inputs from and collaboration with other functions in order to achieve the requirements of the system.

In order to model CSS behavior, a state vector $\mathbf{Z}$ must be added to provide the function memory of what has happened in the past. Memory is required in order to control the generation of the function process thread. Function inputs plus messages from other functions combined with the state vector $\mathbf{Z}$ provide information that intelligent agents (Chapters 7 and 9) use to select among alternative process threads. Each alternative thread produces a particular mapping from X to Y.

Given these additional behavior modeling constructs, function F resembles a finite-state machine that can adapt to the changing environmental or system context. An additional model requirement is that when function process threads work in parallel to collaborate, the process threads must be synchronized to define the start and end of parallel operation. Further, sensor functions in the system must adapt to the environmental situation and communicate with other internal functions to collaborate in order for the system to respond successfully to any relevant environmental change. Systems that can adapt to the environment are more likely to have requisite variety, discussed later in this section, and thus greater system effectiveness.

To summarize, a system is a network of system components or subsystems where each component performs functions and sends messages to other components and the environment in order to achieve all system requirements. Messages can represent information, data, knowledge, energy, or material flowing through the network. If the functions performed and messages sent are allowed to adapt such that feedback paths in the network occur, then the overall system may exhibit emergent behaviors that cannot be produced by any simple subset of components alone. A model that describes a system as network of system components and focuses on the physical details is called *architectural*, as discussed above, and a model that describes a system as a network of functions and focuses on the behavioral details is called *functional flow*. The functional flow model is used during the conceptual system design phase of a system design project; and the architectural model, after functions have been allocated to the system components or subsystems, is used during the system design phase. Bringing complex systems into being is described later in the chapter where these models are discussed further.

1.1.1 Exercise: Model a Goal-Oriented Activity

Before we introduce how to design a complex system using systems engineering methodology later in the chapter, let us first consider how computer simulation can be used in everyday, commonsense problem solving. To simplify our discussion, think of a goal-oriented activity that you already know how to do. For example, you might think of fixing car brakes, baking a cake, sewing clothes, wood working, assembling a bicycle, or making a pizza. Make a list of all the tasks required to achieve your goal.

For example, in making a pizza, one must: (1) mix the flour, water, and yeast and let the dough rise, (2) spread the dough on a pizza pan, (3) cut up the pepperoni and the mushrooms, (4) grate the cheese, and (5) assemble and bake the pizza.

Next, suppose that you have some help executing your problem-solving plan. Organize the tasks of your plan into sequential and concurrent tasks and draw a functional flow diagram. Only tasks that depend on other tasks for input must be shown as sequential. Tasks should be shown as being executed concurrently unless they must be performed sequentially.

For example, given that one other person is available to help, pizza making tasks 1 and 2 can be done concurrently with tasks 3 and 4, but all these tasks must be completed before we can start task 5.

Figure 1.1 shows a functional flow model of the pizza making tasks. Use the diagram to visualize making a pizza by starting from the left side of the diagram and scanning right. The “and” on the left side indicates that tasks 1 and 2 can be executed concurrently with tasks 3 and 4. The “and” on the right side of the diagram indicates that all these tasks must complete before task 5 can be started.

A brief overview of the OpEM language and OpEMCSS graphical library blocks is provided next as background to the agent-based system architectures

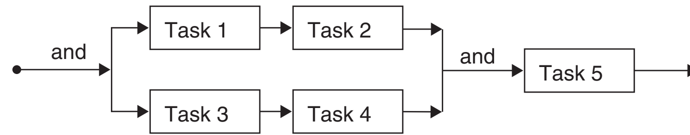

Figure 1.1 Functional flow model of pizza making.

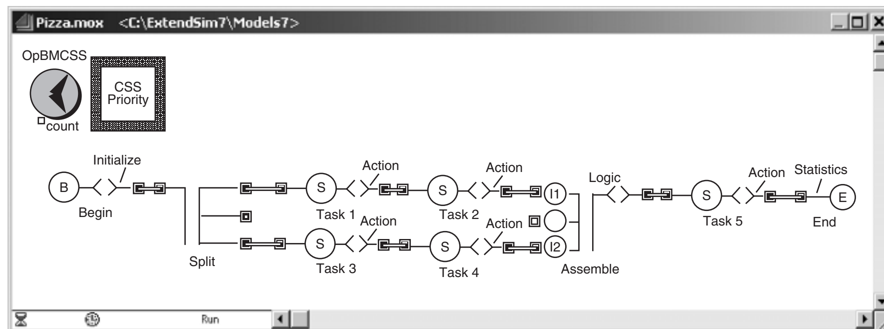

Figure 1.2 Simulation model of pizza making expressed as a parallel process.

discussion that follows. Also, the concept of timelines is introduced as a means to visualize concurrent process interactions.

An OpEMCSS simulation model of pizza making, expressed as a parallel process, is shown in Figure 1.2. A parallel process is defined as the set of all possible sequences of system states and events that represents the operational behavior of a system or organization, and, in this case, it describes all possible operational sequences that represent alternatives in how our pizza could be made. One can imagine that as the state reaction times of the tasks vary, the order of the events would change, resulting in different operational processes.

The blocks shown in Figure 1.2 implement the functional flow diagram described by Figure 1.1. The Begin Event block starts the simulation of making a pizza. The next block is a Split Action block that creates the two concurrent subprocesses T_1, T_2 and T_3, T_4 as shown. Four Reaction Time Event blocks follow, which model the time it takes to execute each task. The Assemble Event block waits for both concurrent processes to complete execution before task 5, modeled by a Reaction Time Event block, can start. The End Event block terminates each pizza making sequence and collects system performance data. See Appendix A for more about these model blocks.

Timelines and simulation traces describe the concurrent arrangement of process tasks as they are executed in time. They are analyzed to identify interactions among the concurrent processes that affect system performance and effectiveness. Suppose the state set of the top process is $Q_1 = \{B, T_1, T_2, T_5, I_1, E\}$ and the state set of the bottom process is $Q_2 = \{T_3, T_4, I_2, \Phi\}$. The Cartesian product

set is $Q = \{Q_1 \times Q_2\}$. A subset of the set Q that describes the possible system states that can occur is called the state space set S:

$$S = \{(B, \Phi), (T_1, T_3), (T_1, T_4), (T_2, T_3), (T_2, T_4), (T_1, I_2), (T_2, I_2), (I_1, T_3), (I_1, T_4), (T_5, \Phi), (E, \Phi)\}$$

The pizza making process begins as a sequential process in parallel state (B,Φ) where Φ represents the null state for process 2. The process splits into two concurrent processes that can be in any of the next eight parallel states (T_1,T_3), (T_1,T_4), (T_2,T_3), (T_2,T_4), (T_1, I_2), (T_2, I_2), (I_1, T_3), and (I_1, T_4). When both concurrent processes complete [parallel state (T_2,T_4) completes], the process synchronizes and becomes a sequential process again. State (T_5,Φ) represents executing task T_5 where Φ represents the null state for process 2. The last state of the process is (E,Φ), which occurs after task T_5 is complete. There are many alternative pizza making, parallel process outcomes depending on how long it takes to perform each task. Parallel states (T_1,T_3) and (T_2,T_4) always occur; however, the occurrence of states (T_1,T_4) or (T_2,T_3) depends on the relative timing of states T_1 and T_3. Timelines and simulation traces are both ways to visualize the order of execution of the parallel system states in order to evaluate process interactions that are affecting system performance and effectiveness.

The external, structural, and operational views are all required to completely describe a system. The operational view is expressed as a parallel process based on sequences of system states and events. Each system state includes the discrete state of each concurrent process. For example, pizza making system state (T_1,T_3) includes top process state T_1 and bottom process state T_3, and it only occurs when T_1 and T_3 are performed at the same time. Discrete states represent periods of time where tasks are being performed by resources or tasks are waiting for some logical condition to be satisfied. Wait states are discussed later in the chapter. Events are changes of system state that occur at discrete points in time. Each sequence of system states and events is called a parallel process instance or process thread. In the above example, subprocesses T_1, T_2 and T_3, T_4 are each concurrent subprocesses that interact through the split and assemble construct shown in Figure 1.2.

In applying the operational view, we are interested in representing process interactions such as synchronization where one process waits for another process before it can continue or resource contention where one process waits for a resource that another process is using. Synchronization can occur when a process waits for data to be sent from another process or waits for task completion in another process. Another important interaction involves a task changing what it does in response to a message from another task. The objective in applying the operational view is to optimize the operational performance and mission effectiveness of the system.

The structural view is expressed as a network of interconnected components or subsystems. In applying the structural view, we are interested in representing the interface for communication among the components. At the conceptual and preliminary systems design level, we are interested in what data or messages

are flowing from one component to another. We are also interested in the communication channel bandwidth in bytes per second that determines how long it takes to send each message. At the detailed hardware–software design level we are interested in exactly how these messages are sent, including the protocols used and the electrical characteristics of the link. The objective in applying the structural view is to optimize the structural performance of the system and to simplify the component interfaces.

The external view is expressed as a context diagram showing connections between the system and all relevant external systems contained in the system environment. In applying the external view, we are interested in determining the real environmental demands on the system. Systems context diagrams show the system boundary and interfaces with the external systems. The input/output interface between the concept system and all external systems is specified so that a dynamic model of the environmental demands on the system can be developed.

The system design and analysis methodology presented in this book generally begins with the external view to model the environmental demands on the system. The operational view is developed next in order to perform design trade-off analysis early in the system design process and then proceeds toward the structural view to optimize the physical system. We believe that this is consistent with our philosophy of designing a system from the top-down and building a system from the bottom-up.

1.1.2 Agent-Based System Architectures

A modern variation of the message-passing model discussed earlier in the chapter, which has been developed to deal with complexity, is called an "agent-based system architecture." Multiagent systems are discussed in more detail in Chapter 9; however, a brief introduction is presented here. The important concept of requisite variety is also discussed.

An agent-based system architecture is a network of intelligent agents that share facts with other agents and adapt their behavior in response to these shared facts; indeed, intelligent agents apply knowledge in the form of rules to transform input to output facts and to make decisions to adapt their behavior. Through such collaboration, an agent-based system achieves overall system goals, and it often exhibits emergent behavior. In some systems, agents are able to learn new rules and evolve new shared facts as discussed in Chapters 7 and 9.

In an OpEMCSS model of an agent-based system, facts are modeled as ExtendSim (a registered trademark of Imagine That Inc. of San Jose, California) attributes, where each attribute consists of a symbolic name combined with a numerical value. Attributes also model other state variables that expand the meaning of discrete states in order to model process control variables and provide memory of past actions. Each process instance in a system simulation can have many associated ExtendSim attributes.

OpEMCSS blocks introduced for the first time in the agent-based system architecture example shown in Figure 1.3 are: (1) Message Event Action block

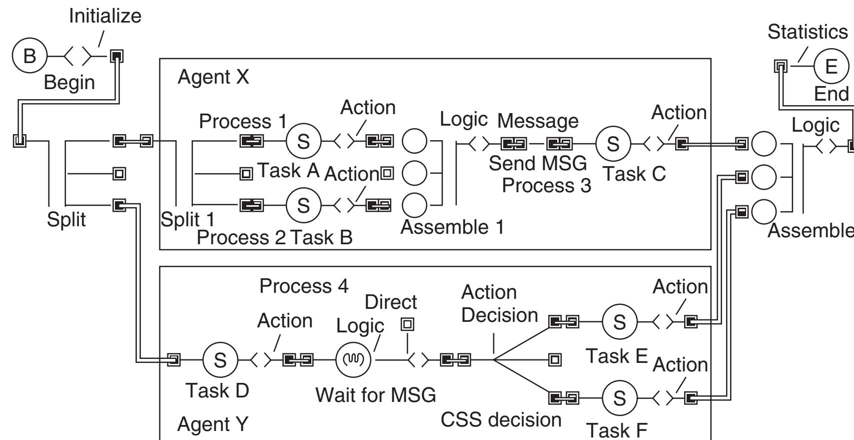

Figure 1.3 Agent-based system architecture.

that allows one process instance to send up to four attributes to other selected process instances, (2) Wait Until Event block that allows a process instance to wait until logic based on global attributes is true before continuing to the next state, and (3) Alternate Action block that allows attribute-based logic to decide which alternate subprocess to execute next.

Figure 1.3 shows an agent-based architecture expressed using the operational view. In this figure, an agent is shown as a collection of concurrent processes that are empowered to make decisions. Process 1 (task A) and process 2 (task B) are performed concurrently as indicated by the OpEMCSS language Split Action Split1 and Assemble Event Assemble1 blocks shown in the figure. When processes 1 and 2 are both complete, process 3 (task C) continues and processes 1 and 2 cease to exist.

During process 4, task D is performed before one of the alternative tasks E or F is selected. When process 3 of agent X continues after the assemble event, facts about processes 1 and 2 activities are sent to process 4 using the Message Event Action block. Process 4 waits in a Wait Until Event block until the message is received. The choice between tasks E or F, made by an Alternate Action block, depends on these shared facts. Therefore, process 4 modifies its behavior based on the activities of processes 1 and 2 of agent X. This is an example of an agent adapting its behavior in response to message facts received from another agent. Such adaptation can result in emergence.

The amount of knowledge sharing and the strength of feedback paths in a multiagent network is critical to achieving the desired emergent behavior. This is a basic principle of complex systems that is discussed in more detail in Chapters 7 and 9. If every agent shares with every other agent (and each agent adapts), the coupling may be too great and overall system behavior becomes chaotic. If too few agents communicate, the overall system may be static and unresponsive to a changing environment. However, if the right agents communicate just the

right knowledge facts with each other and the strength of each agent's response is just right, the system achieves the desired emergent behavior. This is called operating at the edge of chaos (Kauffman, 1995).

The problem for system designers is to determine the optimal amount of knowledge sharing and the required rules to make the decisions that are the most effective for the system. In the vehicle traffic control network discussed above, the offset traffic control signal based on incoming traffic flow patterns had to be just right for the intersection controllers to self-synchronize and minimize the average vehicle waiting time in the network. The desired emergent behavior for the traffic control system was achieved by experimenting with the feedback in the network. This was done by trying different sets of fuzzy control rules and control system gains.

1.1.3 Simulation and AI-Based System Design

A simulation and AI-based system design methodology is introduced that can assist a person in developing an intelligent, agent-based system. This methodology is discussed further later in this chapter and in Chapters 3 and 9.

Agent collaboration allows very complex systems to be designed and built. Suppose there are N agents in a system and each agent can exhibit one of M alternative agent process instances. An agent process instance is a sequence of one or more tasks performed during a period of time. The total number of possible system process instances for the system is M^N. Recall that in the pizza making example, there were two people (agents) working together to make a pizza. When we combined the two concurrent agent process instances describing the activities of the two people, there were two alternative state-event sequences or system process instances possible that represent making a pizza. Alternative system process threads result from relative task execution timing, as in the pizza system, and from alternative agent process threads resulting from shared communications as discussed here.

If agents can adapt their individual process threads to achieve system goals (collaborate) such that all M^N system process instances can actually occur, then system complexity, measured in terms of the number of total system process instances, increases exponentially as N increases. A network of relatively simple agents can be complex as measured in terms of the very large number of overall system process outcomes that can occur.

Almost 50 years ago Ross Ashby (1961) referred to this form of complexity as "variety." Based on the concept of variety, he formulated his law of requisite variety. The *law of requisite variety* states that a system can achieve its mission goals as long as the variety of the system (M^N system process instances) is greater than the variety of the environment with which the system is competing or contending. In order to maintain requisite variety and thus improve the chances of survival in a competitive world, the vector in naturally occurring, coevolutionary systems is generally in the direction of increasing complexity. I believe that natural selection works in favor of systems having requisite variety relative to

their environment. Thus, systems having requisite variety tend to survive better than those that do not. However, when all agents are seeking requisite variety, then the environment tends to become more and more complex. It is interesting to see the many different ways that agents achieve requisite variety in nature.

In order to maximize variety, it is necessary that most of the M^N system process instances can actually occur. Using distributed control, there seems to be more freedom to select the best overall system process thread for each situation than when using centralized control, which may unnecessarily constrain system operation. Another obvious problem with centralized control is that all feature facts needed to make the decision must be sent to a central node. After the decision is made, commands must be sent out to all nodes involved in executing the decision. An even worse communications bottleneck occurs if raw perception data is sent to the centralized decision node instead of feature facts. Further, the rules required for a centralized decision are usually much more complex than a distributed decision because the decision context is larger and more complex (Chapter 7). Breaking a problem into solvable parts has long been done to simplify the solution. Much current research is being done now in the area of network centric operations (NCO) and systems of systems (SOS) to reduce system complexity.

In a distributed decision, inferences required to transform raw perceptions into feature facts and form alternative decision hypotheses are spread over many nodes. The nodes communicate feature facts and alternative decisions to reach a consensus about what each node should do. Distributed decision making greatly simplifies node communication and allows the overall system to consider more alternative system process threads in its search to discover the best one in each situation. Modern business organizations and military C4ISR systems have already proved the benefits of distributed, self-synchronizing systems. These are discussed further in Chapters 7 and 9 where a Classifier Event Action block is discussed that discovers rules that produce optimal system process threads.

1.1.4 Expansionism Versus Reductionism

The purpose of using the OpEMCSS simulation library is to understand, design, analyze, and evaluate intelligent, multiagent systems and other systems by first applying an expansionist approach and then using the traditional reductionist methodology. First, applying expansionism, an optimal system concept is developed where the operation and interactions of agent process threads are optimized within the system operational environment to produce the best system behavior for each situation. Second, using reductionism, the system is decomposed into a hierarchy of agent networks to facilitate system engineering management and to optimize the design of the physical system.

Expansionism is characterized by:

Synthesis: Designing a system that is a collection of communicating process threads where each process thread is associated with an agent

Context-Sensitive Systems: Explaining the behavior of each process instance in terms of its interactions and context as well as task flow

System Theory: Understanding emergent system behavior in terms of process collaboration and adaptation from the top-down

In contrast, the traditional reductionist approach to system design is characterized by:

Analysis: Designing a system by reducing it to a hierarchy of relatively simple components having simple interfaces and minimal interactions with each other

Mechanistic Systems: Explaining that each component transforms inputs into outputs such that each transformation does not change as a function of the inputs

System Theory: Understanding overall system behavior in terms of the individual behaviors of its components from the bottom-up

A system can be reduced to independently operating components, but we cannot analyze each component independently and then integrate the analysis results to understand the system as a whole. The reason for this is that collaborating components transform each other, producing a much more complex system behavior than would be predicted by an independent study of each of the individual components.

Consider a double spring–mass system shown in Figure 1.4. This system can be reduced to two, single spring–mass components that interact. If each individual spring–mass system is studied independently, a simple, second-order harmonic response is observed for each. However, if the components are studied together, a more complex, fourth-order response is observed. This system of closely coupled components results in much more complex system behavior than

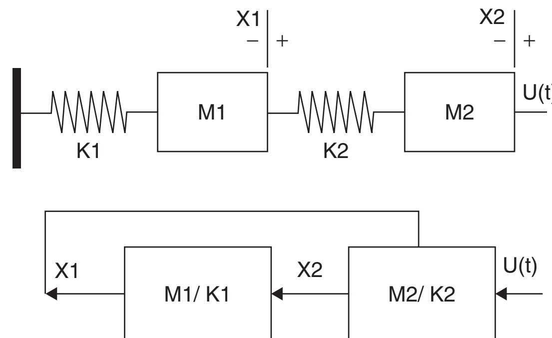

Figure 1.4 Problem with reductionist/mechanistic view.

a study of the individual component behaviors would suggest by themselves. This double spring–mass system is linear because the same input always produces the same output. Nonlinear transformations such as occur in CAS can make the system behavior much more complex because the same input does not always produce the same output. Thus, the traditional reductionist approach to system design must be modified when dealing with such complex systems.

Once a top-level concept of the whole system has been developed using the expansionist system design methodology, the system concept is decomposed into a hierarchy of components or agent networks suitable for hardware–software engineering and project management. From the previous discussion, we can see that what makes designing an intelligent, multiagent system difficult is discovering the facts and rules each agent needs to make decisions and which of the facts agents must share in order to collaborate. Once we determine the facts that must be shared, we can define the process-to-process communication link requirements. We can then consider alternative allocations of process instances to components or agents in order to minimize agent-to-agent communication message flow and the number of communication links required. Agent interface considerations can then be traded-off against other requirements such as technology, reliability, availability, and cost.

The point is that process-to-process communication requirements, needed to support decision making and collaboration, must be determined before the system is decomposed into a hierarchy of networks. The major steps of a simulation and AI-based system design and evaluation methodology for intelligent, multiagent systems are as follows:

1. Develop a parallel process model that describes desired system operation within the operational environment using the OpEMCSS library blocks.
2. Determine what facts and rules each process requires to collaborate by modeling decision making using the Classifier Event Action block in the OpEMCSS library to automate the search for rules.
3. Map process instances to components or agents in a way that simplifies the interfaces and communications as well as other requirements.
4. Model agent motion of system and environmental entity spatial interactions if required. Evaluate design trade-offs and optimize system design in terms of other considerations such as technology, reliability, availability, and cost.
5. Document systems analysis results and concomitant systems design specifications. These major steps of a simulation and AI-based system design and evaluation methodology discussed here will be expanded upon and added to during the remainder of the chapter until a detailed OpEMCSS simulation model is finally described. The end result is that the reader will come to understand how simulation and AI can be integrated into the modern systems design and development process.

The Classifier Event Action block, discussed above, is able to simulate the AI capability of agents to apply knowledge in the form of rules to transform process instance condition attributes into process instance action attributes. The action attributes can be used by an Alternate Action block to decide which alternative process thread to execute next; further, these attributes can also specify the functional message output Y of a component or agent as discussed above. The classifier block is discussed further in Chapter 7, and in Chapters 7, 8, and 9 it is applied in many decision-making examples.

1.1.5 Summary

This introductory section was intended to begin with commonly understood structural, operational, and external views of systems and then to introduce the concept of a parallel process as a way of thinking about interacting concurrent processes. It was shown that the parallel process concept provides a more detailed operational model than the functional flow model, which facilitates understanding, designing, analyzing, and evaluating complex systems. The parallel process concept was introduced using the OpEM graphical modeling language, which is discussed in more detail in Chapter 2. The expansionist systems engineering methodology, introduced in this section, is discussed further later in the chapter and in Chapters 3 and 9. In the next section, we will show how to apply simulation as part of a general problem-solving approach and expansionist simulation-based systems design, analysis, and evaluation methodology for complex systems.

1.2 USING SIMULATION TO UNDERSTAND COMPLEX SYSTEMS

Knowing how OpEM simulations work can help understand complex systems. An OpEM simulation is an abstract description of the structure and behavior of a complex system that is easier to understand than the system itself. Because an OpEM simulation is unambiguous and executable, it can be used to numerically, rather than subjectively, evaluate the impact of the structure and behavior of the system, as designed, on system performance and effectiveness. In this section, the ExtendSim user environment and OpEMCSS libraries are summarized briefly to provide an overview of their capabilities. A more detailed system development procedure is described and then applied to a space station system example. A purely sequential process model is used to introduce how simulations work and to discuss the concept of model output convergence. A simple parallel process is used as an example of how parallel process simulations work and to introduce the concept of sensitivity analysis. This section is concluded with a summary.

1.2.1 ExtendSim Discrete-Event Simulation User Environment and OpEMCSS Overview

The purpose of this book is to present concepts and methods needed to engineer complex systems using discrete-event simulation (DES) and artificial intelligence

(AI). Hands-on experiments using discrete-event simulation and Markov models have been included throughout the book in order for the reader to achieve a better understanding of complex systems and a greater utilization of these concepts and methods during the simulation-based systems engineering process. In order to do the hands-on experiments, facility with using ExtendSim and the OpEMCSS library is required.

Operational Evaluation Modeling (OpEM) for Context-Sensitive Systems (CSS), or OpEMCSS, is a graphical, DES library that expands the ExtendSim simulation software environment in order to model context-sensitive systems (CSS). A CSS has system state transitions that depend on the *global* system context rather than only one *local* process thread. For example, the two-agent system, shown in Figure 1.3, has a state transition in process 4 of agent Y that depends on information received from process 3 of agent X.

It is recommended that the reader read through Appendix A, in the back of the book, now and do the exercises in order to immediately (rather than incrementally) gain the required facility with using ExtendSim and the OpEMCSS library. Appendix A first discusses minimum requirements for successful CSS modeling languages and provides a modeling languages survey that includes: (1) Petri nets, (2) IDEF0 functional flow diagrams, and (3) ExtendSim queuing diagrams. Next, the OpEM graphical modeling language is compared to these other languages and the rationale for using OpEMCSS is explained. OpEMCSS familiarization exercises are described that include: (1) how to set up ExtendSim with OpEMCSS libraries and models, (2) ExtendSim environment overview, and (3) block familiarization exercises that will instruct the user as to what each of the basic OpEMCSS blocks contributes when building an OpEM simulation model in ExtendSim.

Throughout the book, the presentation of the concepts and methods needed to engineer complex systems is organized such that many of the OpEMCSS blocks are described in the chapters in order to have explicit examples that facilitate the desired hands-on experience. The basic idea is that understanding what the blocks do in a model facilitates the understanding of complex systems. This is because a simulation model is an abstracted description of a complex system that is designed to answer questions about that system. In addition to block descriptions in the text, Appendix B focuses on blocks that are not adequately discussed elsewhere, but it also provides a summary description of all the blocks.

OpEMCSS can be used to evaluate alternative component morphological instantiations of component algorithms and methods, discussed in the next section, in order to discover hidden synergisms among particular sets of algorithms that optimize overall system effectiveness and performance. This is something that OpEMCSS can do very well using ExtendSim's capabilities to develop special blocks that implement detailed component algorithms. More detailed and globally optimized component requirements can be determined and added to the system design specification. These additional derived requirements, which are subsequently imposed during hardware and software design, will allow a more globally optimized system design to be brought into being.

Appendix C discusses the Special Event Action block that is included in the OpEM auxiliary library (current version is OpEMAUX7.LIX). Whereas the OpEMCSS main library is protected (access to block MODAL code is denied), the OpEM auxiliary library MODAL code can be modified by the user to implement component algorithms. All event action blocks receive a process instance item index (an integer that indexes the item array), modify item attribute values (local and/or global), and then send the item index to the next block. In order to modify item attributes, a MODAL procedure ExecuteEventAction() obtains the values of local and global input attributes, computes the new output attribute values, and then modifies the value of each output attribute, either locally and/or globally. An example discussed in Chapter 2 provides more information about how an OpEMCSS model works.

1.2.2 Simulation Model Development Procedure

Simulation has become, in the last few years, a mandatory part of the systems engineering process, especially as applied to the engineering of complex systems. Large military, manufacturing, and transportation systems are too complex to be designed and evaluated without the use of simulation. Business organizations are currently being reengineered and intelligent enterprises are being designed, and simulation is a natural tool to design and evaluate them. Computer simulation and Markov models are used throughout this book to perform a numerical evaluation of system performance and effectiveness. Other models such as functional flow diagrams, network graphs, semantic networks, and hierarchy charts are used during the development of these computer models.

Two kinds of systems engineering tools are typically applied to support the engineering of complex systems and the simulation-based systems engineering process: (1) tools that automate the system development process and produce a system development database such as CORE (Buede, 1999) and (2) tools that are used for concept exploration and discovery such as OpEMCSS (Clymer, 1997).

A system development database tool is used for complete automation of the systems engineering process from design team entry of requirements until generation of the system specification documents. Such a tool allows a central design database to be accessed by members of a design team and design description or decision-making information, including issues and risks, to be entered and shared. A system development database can be used to map the functional model onto the system component architecture, and it can generate system design specification documents.

Limited modeling capability can be provided in design capture database tools to check the consistency of a design; however, these tools are not usually suitable for concept exploration and discovery and systems analysis and evaluation. A concept exploration and discovery and systems analysis and evaluation tool such as OpEMCSS mitigates system development problems that are caused by the failure to optimize the interoperability and synergisms among all component algorithms and methods at the overall system level. Further, the interactions of

the system with its external systems and the dynamic demands of the operational environment on the system must also be included in a system-level model. Design-capture database tools currently do not model detailed component algorithms or the dynamic demands of a coevolutionary operational environment. However, a design-capture database tool does support the overall systems development process discussed later in the chapter. Therefore, design-capture database tools and concept exploration and discovery and systems analysis and evaluation tools such as OpEMCSS are complementary and, thus, both required to perform simulation-based systems engineering. In this section, a simulation model development procedure is described and an example system design problem is presented, and a few steps of this procedure are highlighted. In the introduction to this chapter, we developed a functional flow model of a goal-oriented activity that we already knew how to do. This exercise allowed us to focus on the graphical modeling tools rather than the problem itself. However, our ultimate goal is to be able to explore a problem space to develop the best solution concept to solve a customer problem that we do not already know how to do. Generating many alternate concepts that solve the problem in its operational environment, each expressed as a simulation model, and then evaluating each solution to find the best one summarizes our simulation-based systems design methodology. The simulation model development procedure that follows in this section involves additional steps not needed for the previous design exercise because now we want to solve a system design problem that we do not already know how to do.

For each system design problem to be solved, the following steps are recommended:

1. Define the system problem to be solved and describe system scenario or context. This step produces a context diagram or external model.
2. Define the missions and mission objectives of the system and measures of effectiveness. This step defines system effectiveness and performance parameters to be estimated by the model.
3. Define objectives for study of each system solution using a model. What questions a model must answer often dictates model details.
4. List tasks that must be performed to achieve each mission objective. What must the system do in order to achieve each mission objective?
5. Subdivide tasks into periods of time represented by states where: (a) a task is being performed by a resource and (b) a task is waiting for a logical condition to be satisfied.
6. Group purely sequential tasks into the same processes and tasks that can be performed concurrently into separate processes. Always allow for maximum permissible parallelism: Tasks are sequential only if they must be done that way.
7. Manually develop timelines of concurrent states that describe system operation and assist visualization of system operation. Show process states on the timeline and indicate when one process sends data to another process or

releases a resource to another process. Identify all context-sensitive interactions as they occur in the timeline.

8. Develop a directed graph model diagram on the computer screen as discussed in Appendix A. Start with task and wait states, then add state variable attributes to model interactions and physical details as well as data collection for model output. The model is visualized from the timeline, then generalized for all cases.
9. Generate timelines or traces from simulation output as discussed above to verify and validate the model and modify the directed graph model diagram until satisfied.
10. Operate the model to achieve the model objectives and report analysis results.

This simulation model development procedure is executed within and receives input from the general systems development procedure. The interface between the simulation model development procedure and general systems development procedure is discussed in the next major section of the chapter.

Example applications of this procedure will be presented during the remainder of this chapter and Chapter 3. Suppose a system consists of a space station factory in low Earth orbit having (ND) docks at the space station, (NS) shuttle vehicles, and (NR) refurbishing facilities for the shuttles. Further, suppose that the overall capability of the space station system to process shuttle missions requires evaluation.

The mission of the space station is to dock with each arriving shuttle, unload and load cargo and passengers, and undock the shuttle. Shuttle missions are generated and arrive periodically at the space station. The system design problem to be solved is to determine under what conditions shuttle missions can be accommodated by the space station with little mission delay and high space station and shuttle resource utilization.

A shuttle mission consists of: (1) mission planning, PLN; (2) load cargo aboard shuttle and boost shuttle to low Earth orbit, arriving at the space station, LDB; (3) move to dock, unload, and load cargo and passengers, MUL; and (4) deorbit, land, and unload cargo and passengers, DLU. When a shuttle finishes a mission, it must be refurbished, RFB, before it can be assigned to another mission. A mission is generated every 48 hours. Mission times are PLN, 24; LDB, 18; MUL, 24; DLU, 18; and RFB, 48 hours.

System operation can be visualized as two processes in parallel—a mission generator process and a mission execution process thread for each shuttle mission in the system. A timeline, as mentioned in the above procedure, is a graphical display of a simulation trace. An example timeline of a shuttle mission is shown in Figure 1.5. The vertical dotted lines indicate process interactions. Each mission generated is connected to the beginning of an instance of a shuttle mission execution process. When mission execution is complete, refurbishment follows.

An OpEMCSS simulation model of space station system operation is shown in Figure 1.6. This model was developed using the simulation model development

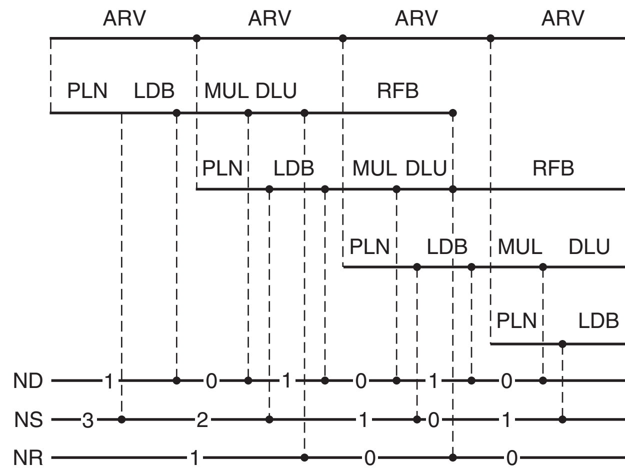

Figure 1.5 Example timeline of a shuttle mission.

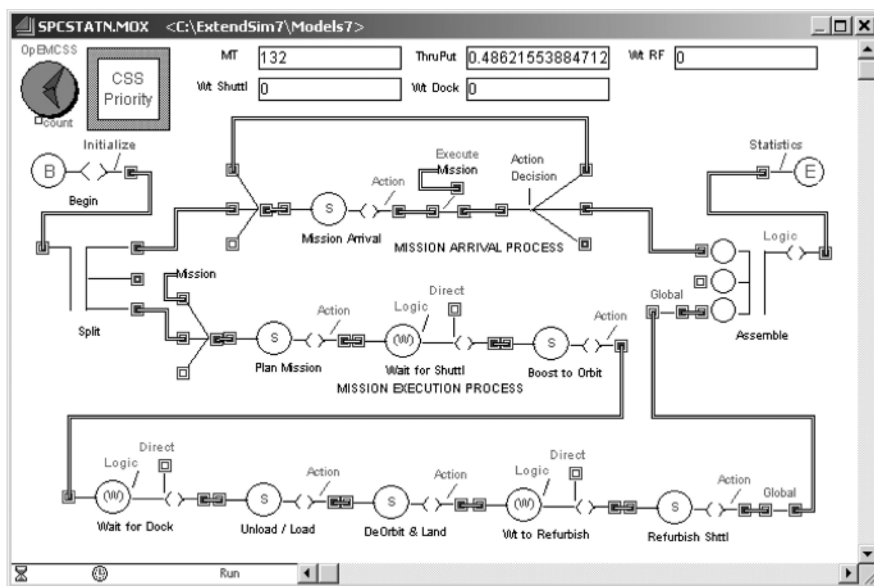

Figure 1.6 Simulation model of space station system operation.

procedure described in this section. The Wait Until Event blocks were added to the model so that shuttle missions could wait when space station system resources are busy.

As an exercise, run the model with the mission interarrival time first constant ($k = 0$) and then exponentially distributed ($k = 1$). See Appendix A for an

explanation of change parameter k. Explain why the two timelines generated are so different. What basic principle of system operation is represented here?

1.2.3 Simulation Programs: How Serial and Parallel Process Models Work

A sequential process model, the thief of Baghdad process, is discussed to introduce how sequential process simulations work and to discuss the concept of model output convergence. A simple parallel process model, a producer–consumer process, is used as an example of how parallel process simulations work and to introduce the concept of sensitivity analysis.

Thief of Baghdad Process The thief of Baghdad scenario is as follows. A thief is arrested and placed in a prison cell. Three hidden tunnels lead away from the cell. Two of the tunnels emerge secretly into the prison courtyard. If the thief takes one of these tunnels, he is immediately captured when he emerges into the courtyard, and he is returned to his cell. It takes 3 hours to crawl through the first tunnel and 1 hour to crawl through the second tunnel. The third tunnel leads to freedom, and it takes 1 hour of crawling time to escape. Each time the thief emerges into the courtyard, he is so severely beaten that he cannot remember the tunnels he has already taken. Thus, the probability of choosing each tunnel remains the same each time a tunnel is chosen. The objective of the study is to answer the question: What is the average time required for the thief to regain his freedom?

An OpEM-directed graph model of the thief of Baghdad process is shown in Figure 1.7. The model consists of 6 states and 5 events. The circles represent states, periods of time required to perform the escape process. The states are: (1) begin, B; (2) cell, C; (3) tunnel 1, T1; (4) tunnel 2, T2; (5) tunnel 3, T3; and (6) end, E. The brackets < > in the process diagram represent changes in state or events where the process transitions from one state to the next. Mission attributes are parameter values that control sequential process outcomes, and they are inputs to the simulation model that are often varied to observe their affect on model outputs. Mission attributes for the thief of Baghdad process are reaction times

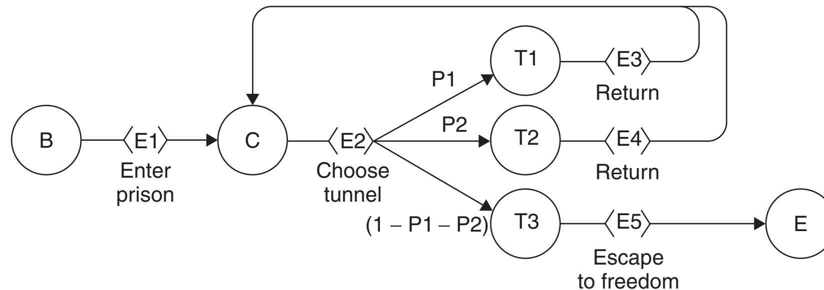

Figure 1.7 Thief of Baghdad process.

RC, RT1, RT2, and RT3 and probabilities P1 and P2. These mission attributes are initialized by event E1 to start a replication of the escape process. For example, reaction time mission attribute RT1 defines the time (3 hours) that the thief spends in the tunnel 1 state. Reaction time mission attributes RC, RT2, and RT3 each define 1 hour spent in the other states; respectively. The B state represents the time before the thief enters prison, and the E state models the time after the thief escapes from prison. Thus, each escape process replication begins with a transition from the B state to the C state (event E1) and ends with a transition from the T3 state to the E state (event E5).

The C state represents a 1-hour period when the thief is in his cell and searching for a tunnel. When the reaction time for the C state is complete (defined by mission attribute RC), a transition out of the C state occurs (event E2). The thief chooses tunnel 1 with probability P1, tunnel 2 with probability P2, or tunnel 3 with probability 1 − P1 − P2. Because of thief's memory loss, the probability distribution used does not change based on past choices. A Markov model that allows the thief to have memory of past choices is described in Chapter 4. How to make a selection among several alternate event actions based on a discrete probability density distribution is described in Chapter 2.

Mission attribute reaction time values RC − 1, RT1 − 3, RT2 − 1, and RT3 − 1 and mission attribute probabilities P1 − 0.1 and P2 − 0.1 are used in a run of the model. Two event-state replication traces generated by the simulation model are

B − C − T1 − C − T1 − C − T1 − C − T1 − C − T3 − E > escape time 18

B − C − T1 − C − T3 − E > escape time 6

The simulation run outputs are the average escape time (12 units) and the percent of the process time spent in each state C (29%), T1 (63%), T2 (0%), and T3 (8%), computed over all replications of the process. From these values we can determine where the thief spends most of his time during an escape sequence. Of course, these results depend on the mission attribute value inputs (key performance parameters, KPPs) used to characterize the process.

Figure 1.8 shows a flowchart of the simulation program that generates the event-state replications discussed above and then calculates the average escape time and percentage of time the process spends in each state. The flowchart consists of Start, Input KPPs, Initialize, Execute Next Event, More Events?, Another?, Output Run Report, and Stop blocks. The simulation begins by imputing mission attribute values (RC, RT1, RT2, RT3, P1, and P2), run options, and initializing all data collection counters needed to compute the simulation run outputs. The run options are the number of run replications (event-state traces) and activation of event-state trace output. Studying event-state trace outputs is required to debug a model; however, computing simulation run outputs based on many replications requires no trace output. Data collection counters C, T1, T2, and T3 (accumulates the total time in each state), TOTIME (accumulates total escape time), and the seed of the uniform random number generator U(0,1)

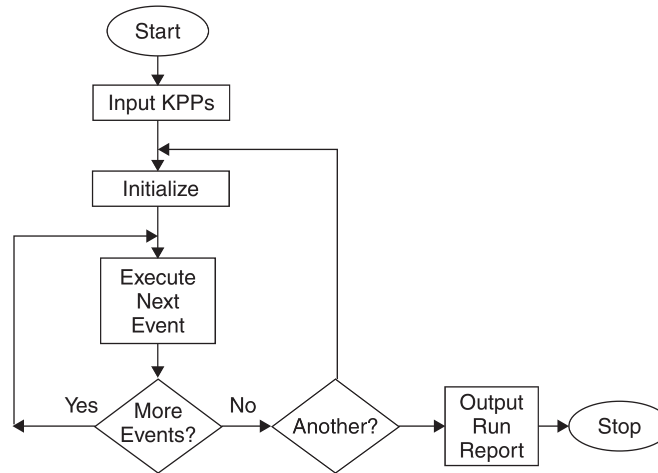

Figure 1.8 Flowchart of sequential simulation program.

described in Chapter 2 are initialized to start a simulation run of one or more event-state replications.

A set of event-state replications is generated by the iteration loop of blocks consisting of Initialize, Execute Next Event, More Events?, and Another? The number of escape sequences generated by this loop is defined by parameter TOTMIS (TOTal MISsions), which is specified as a run option.

An escape, event-state replication begins in the B state as discussed above. The Initialize block initializes model state variables and executes event 1 of the process. If the Boolean run option variable TRCFLG (TRaCe FLaG) was set by the Input KPPs block, then the event-state replication trace is printed as a simulation output.

A second iteration loop consisting of Execute Next Event and More Events? generates the desired event-state sequence for a single replication. The Execute Next Event block performs data collection and schedules the next event in the sequence. For example, when event 2 is executed, mission attribute RC is added to the data collection attribute C. Attribute RC is also added to replication time variable TNOW (Time NOW), which maintains the current time during an event-state sequence. For each event, the event-state trace is printed if that option is selected. The remaining events follow a similar pattern.

Event 2 is unique in the thief of Baghdad simulation in that it uses a discrete probability distribution to select a tunnel. How to do this will be described in Chapter 2.

The More Events? block determines when the event-state sequence is complete for a single replication. For the thief of Baghdad simulation, the event-state replication (process thread) ends when event 5, and escape to freedom occurs.

An important task when applying simulation to solve systems design problems is sensitivity analysis, which requires convergence of simulation output to a

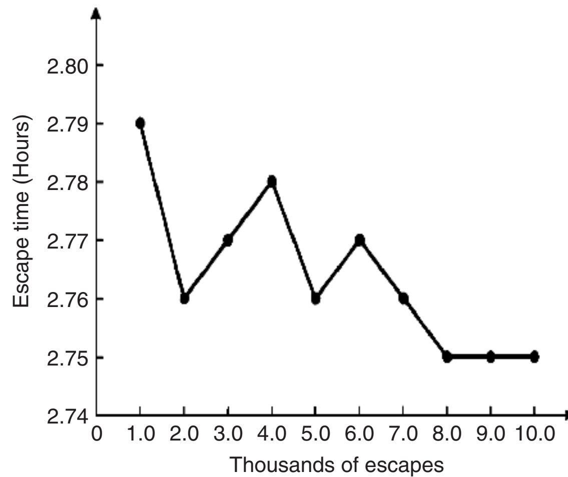

Figure 1.9 Convergence of thief of Baghdad process.

steady-state value. Figure 1.9 shows average escape time for the thief of Baghdad process as a function of number of escapes. As the number of simulated escape sequence replications increases, a larger sample of process outcomes is used to calculate the average escape time. Figure 1.10 shows a run listing for 100,000 escapes with no event-state trace. The converged average escape time shown is 2.75 hours, which is confirmed in Chapter 4 by a Markov model of the thief of Baghdad process. As shown in Figure 1.9, the average escape time converges to the true value of 2.75 hours after about 7000 escapes. Run the OpEMCSS thief of Baghdad simulation TFBAGDAD.MOX for 10,000 replications and observe where convergence occurs. Use the Pause/Resume ||> button to pause the run so you can see the number of replications at the bottom of the ExtendSim model

```
INPUT REACTION TIMES FOR STATES C, T1, T2, T3
1,3,1,1
INPUT PROBABILITIES P1 AND P2
.1,.1
INPUT 1 FOR EVENT STATE TRACE, ELSE 0, THEN TOTHIS
0,100000

REPORT OF THIEF OF BAGHDAD PROCESS SIMULATION
AVERAGE ESCAPE TIME IS  2.75
PERCENT IN STATE C IS    .46
PERCENT IN STATE T1 IS   .14
PERCENT IN STATE T2 IS   .04
PERCENT IN STATE T3 IS   .36
```

Figure 1.10 Run report for 100,000 escapes with no event-state trace.

window. Note how the average escape time varies ±2.75 as the number of escapes increases. The variation is large at first but decreases as the number of replications increases. Convergence of the average escape is a predicted by the law of large numbers discussed in Chapter 2.

Application of OpEMCSS to a Calculator Factory The calculator factory simulation model describes two concurrent process threads. The simulation program described by the flowchart in Figure 1.8 must be modified to simulate concurrent process threads. The program must now use a time-ordered list to keep track of all pending events in the model that are associated with the concurrent event-state sequences. In the sequential process simulation program, simulation time TNOW was advanced after each event was executed. However, in the parallel process simulation program, simulation time is only advanced when all events scheduled for the same current time have been executed. An additional modification of the parallel process simulation program is execution of Wait States, which use logic based on state variable values to decide when an event occurs. The logic for each pending wait state event must be evaluated at each discrete simulation time after all the reaction time events have been executed. The flowchart for the parallel process simulation program is discussed further in Chapter 2.

The parallel process simulation program discussed above is called *event oriented*. The ExtendSim + OpEMCSS parallel process simulation program is called *object oriented*. In ExtendSim + OpEMCSS, the process state-to-state transitions are modeled by an item record where the item pointer moves from block to block in the process diagram. In event-oriented simulation programs, a sequence of event procedures is executed to generate the same event-state trace.

The ExtendSim + OpEMCSS version of the thief of Baghdad process is compared and contrasted with an event-oriented version to demonstrate how parallel process simulations work. In Chapter 2, how event-oriented and ExtendSim + OpEMCSS models work for interacting concurrent processes will be discussed further.

Scenario Producer–consumer process, shown in a factory scenario in Figure 1.11, provides an example of a parallel process simulation where the processes interact. Workstation 1 assembles hand calculators. Major parts of the calculator are the printed wiring board with integrated circuits mounted, the plastic case, and the battery. The worker at station 1 can assemble a calculator in an average of 60 seconds. After finishing a calculator, he places it on a conveyor belt. The calculator moves toward workstation 2. The packer at workstation 2 removes the calculator from the belt and packages it in a suitable container for shipping. Packing the calculator for shipment takes 90 seconds on average. When finished, the packer places the calculator package on a stack and begins packing the next calculator.

To ensure the safety of the operation, the conveyor belt can be accessed by only one workstation at a time. Therefore, while the assembler places a calculator on the conveyor and moves it toward the packer, the conveyor cannot be accessed

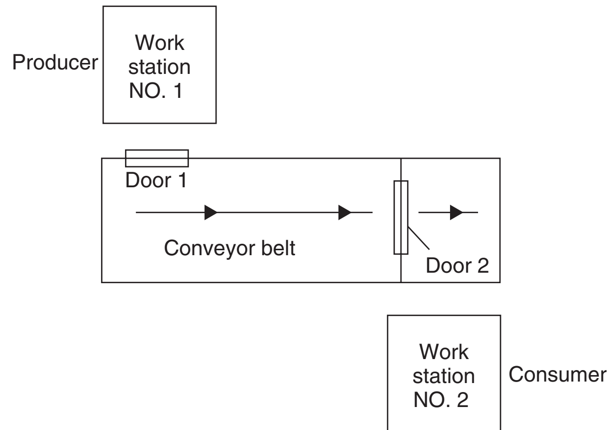

Figure 1.11 Producer–consumer process shown in factory scenario.

by the packer. Likewise, when the packer moves the conveyor belt to obtain a calculator at the end of the belt, the assembler cannot access the conveyor. For this reason, two doors have been added to the conveyor belt. When a calculator has been assembled and the conveyor is not being used by the packer, the assembler opens door 1 and places the calculator on the belt. He closes the door and moves the conveyor until a calculator reaches door 2, where the conveyor automatically stops. Since the conveyor can hold up to 10 calculators between doors 1 and 2, one to nine time units are required to move the conveyor, depending on how many calculators are already on the belt. If the conveyor is not being used by the assembler, and the packer needs a calculator, he opens door 2, moves a calculator one position toward the end of the conveyor belt, and removes it from the belt. Only one time unit is required to move a calculator the one position from door 2 to the end of the conveyor belt, where the packer removes it.

The conveyor belt is a potential bottleneck that cannot be eliminated because it passes through a large fire wall that separates the manufacturing area from the warehouse area. Four wait conditions can slow the production and packaging of calculators. First, the conveyor belt may be full of calculators; so the assembler must wait for the packer to remove one. Second, the packer may be accessing the conveyor belt; so the assembler must wait. Third, the assembler may be accessing the conveyor belt; so the packer must wait. Fourth, the belt may be empty; so the packer must wait until a calculator is assembled.

The relevant measure of effectiveness (MOE) for the calculator factory system is the average number of calculators produced and packed per minute. Besides the conveyor belt logical wait constraints, factory operation is affected by how much time is required for the assembler and the packer to perform their various work tasks. The logical wait constraints are required for safety and are out of our control; however, task reaction times can be changed. The objective of this simulation project is to balance the logical wait times and maximize the throughput of the calculator factory system.

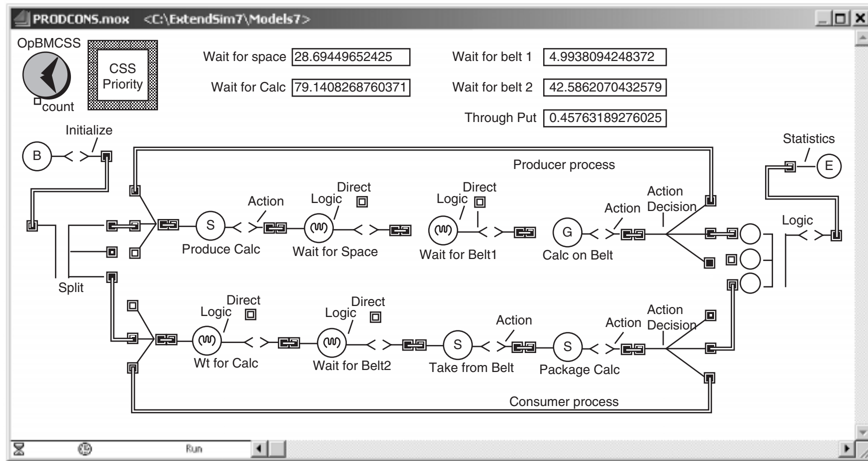

Figure 1.12 Directed-graph model of producer–consumer process.

OpEMCSS Model The parallel process model shown in Figure 1.12 was originally described in Dijkstra ((1968)) for two computers sharing a common memory buffer. One computer produces an output called a "portion" and places it in a buffer cell in memory. Another computer takes a portion from the buffer and processes it. This process is known as the producer–consumer process. The problem, discussed by Dijkstra ((1968)), is to coordinate buffer accesses made by the two processes to ensure they do not access the same buffer cell at the same time. The coordination method must allow the processes to operate independently, except when both need the buffer. In his study, Dijkstra describes the producer–consumer process using a modified ALGOL-type language; it is represented here using the OpEM directed-graph language expressed using some of the OpEMCSS library blocks.

Discrete states of the producer process are: (1) Produce Calc, PRD, time to produce a calculator; (2) Wait for space, WTS, time waiting for an empty space on the conveyor belt to become available; (3) Wait for belt 1, WB1, time waiting by the assembler for the belt to become free when the belt is in use by the packer; (4) Calc on belt, PCB, time to place a calculator on the conveyor belt and move it toward the packer; and (5) the time the assembler is idle after the last calculator has been produced while waiting for the packer to empty the conveyor belt.

Discrete states of the consumer process, shown in Figure 1.12 are: (1) Wt for Calc, WTC, time waiting for a calculator to be placed on the conveyor belt by the assembler when the belt is empty; (2) Wait for belt 2, WB2, time waiting by the packer for the conveyor belt when the belt is in use by the assembler; (3) Take Calc from Belt, TCB, time to move a calculator through door 2 to the end of the belt and remove it; (4) Package Calc, PAK, time to package a calculator; and (5) a transient idle state that occurs after the last calculator has been packaged.

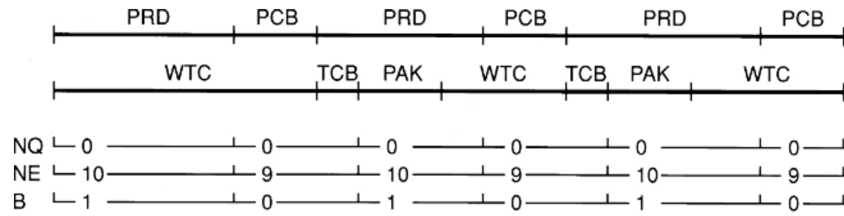

Figure 1.13 Timeline for visualizing system operation.

The OpEMCSS model shown Figure 1.12 describes all possible sequences of states and events that represent the operational behavior of the calculator factory. The timeline of Figure 1.13 is only one particular sequence. As shown in the timeline, three process control state variables are used to control access to the belt in order to place and remove calculators. These attributes are: (1) NQ, current number of calculators on the belt; (2) NE, current number of available spaces on the belt, and (3) B, belt is free or in use. The fact that NQ in the timeline is always zero is a clue to the cause of low calculator throughput of the factory. Run the OpEMCSS producer–consumer model PRODCONS.MOX in trace mode to confirm this timeline.

When two or more processes being performed concurrently share a common buffer, they may both attempt to access the buffer at the same instant. If one process is reading the contents of a buffer cell, and a second process interrupts this task to change the buffer cell contents, a system error will result. In the calculator factory system, simultaneous access of the conveyor belt by the assembler and packer is a safety hazard, resulting in the required logical wait constraints discussed above.

In his classic study, Dijkstra ((1968)) presents a series of solutions to this problem that lead to the conclusion that a set of uninterruptible primitive operations is needed. These are called P and V operations here. They operate on process control, state variables. He calls these semaphores to provide error-free communication between processes. The P operation tests the value of a semaphore. If zero, the process waits until it is positive again. If positive, it is decremented, and the process continues. The V operation increments the value of the semaphore. The testing and decrementing of the P operation and incrementing of the V operation cannot be interrupted to avoid simultaneous access of a semaphore.

In the producer–consumer model, state variable B controls access to the conveyor belt, NE signals that the belt is full, and NQ signals that the belt is empty. In general, the P operation defines occurrence logic of the Wait for space and Wait for belt 1 states of the producer process and of the Wt for calc and Wait for belt 2 states of the consumer process. The V operation increments state variables used in signaling when the appropriate action is complete for each state variable or semaphore. Semaphores NQ and B are incremented when state PCB completes, and NE and B are incremented when state TCB completes.

What do you think happens if the order of the Wait for space and Wait for belt 1 states is reversed on the directed graph? Swap the wait state blocks in the producer and consumer processes and run the revised model in trace mode to observe what happens. This is an important principle of complex systems.

1.2.4 Sensitivity Analysis

Two main types of management information are obtained from an OpEMCSS simulation model. These are measures of effectiveness (MOEs) and measures of performance (MOPs). Measures of effectiveness provide information about how well the system achieves its mission objectives. Measures of performance measure system efficiency and resource utilization, and they provide clues about where bottlenecks are in the system. These clues lead to recommendations to improve system effectiveness. In the assembler–packer process, the MOE is system throughput, and there are four MOPs representing the average wait time for each wait state in the model. A simulation model produces MOEs and MOPs as a function of the values of mission attributes. A mission attribute or KPP is a model input parameter that characterizes the operation of people and equipment as they perform a particular mission function or task. To obtain the value of each mission attribute, the attribute must be precisely defined. Precise definition allows the value of each mission attribute to be obtained from an operational test or from a detailed model of system or component operation.

Some process-level mission attributes are availability, operational reliability, capability, personnel ability, and reaction time. *Availability* is the probability that a collection of equipment required to perform a function is not in a failure state at the start of a test. *Operational reliability* is the probability that the collection of equipment required to perform the process will not fail when called upon to perform the process, given that the equipment was available at the start of test. *Capability* is the probability that the collection of equipment required to perform the process will not cause the process to fail to achieve its objectives, given that the equipment is available and operationally reliable. *Personnel ability* is the probability that the people required to perform the process will not cause the process to fail to achieve its mission objectives, given that the equipment required is available and operationally reliable. *Reaction time* is calculated from the start of the test to either the time the objective is achieved (for successful outcomes) or the end of test (or abort event) for an unsuccessful outcome.

It is important to include failures that result in task abort times in your simulation model because, even though the task is unsuccessful, resources are being used. Therefore, probabilities of failure must be included in the simulation model. These KPP probabilities allow numerical evaluation of how sensitive MOEs and MOPs are to possible system failures. After a failed task aborts, failure recovery processes must be designed and simulated, and it must be determined under what failure conditions system requirements can be met.

A mission process model provides management information as a function of mission attributes or KPPs. Figure 1.14 shows such a graph. Evaluation of the

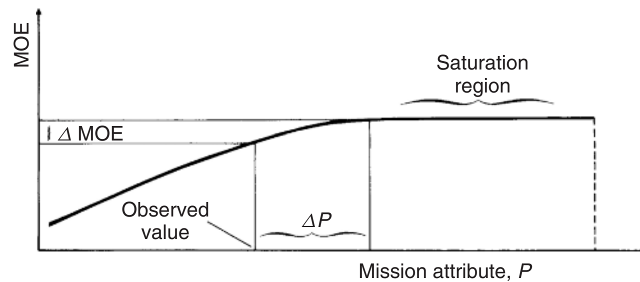

Figure 1.14 Sensitivity curve of mission attribute vs. MOE.

graph reveals the potential improvement $\Delta MI/\Delta P$. This measures how sensitive management information (MI) is to changes in mission attribute or KPP (P). Mission attributes with the largest gain are the best candidates for improving system effectiveness. A sensitivity curve is developed for each mission attribute to identify all high-gain areas for improvement. Each sensitivity curve also shows where no further gain is possible, as indicated by the saturation region on the figure. Team members can use the directed-graph model of the mission process to discuss the high-gain areas identified by sensitivity analysis and share ideas about how to improve the process.

Any design team member may run the program to change mission attribute values and produce a variety of output formats, such as a sensitivity curve. Changes in the value of mission attributes are accomplished simply by opening the appropriate OpEMCSS block dialogs and changing values in the proper parameter or equation boxes.

As an exercise, run the Prodcons.mox simulation and develop some sensitivity curves. Run the simulation repeatedly and vary each reaction time PRD and PAK from 10 to 90 seconds in steps of 10 for each run. Plot the five model scoreboard values as a function of these reaction time values. Run simulation repeatedly for each reaction time PCB and TCB for mean values of 10 and 5 for each run. Plot the model scoreboard values as a function of these reaction time values. For each MOE/MOP plot, reset each mission attribute to its nominal value, repeat this procedure for the next MOE/MOP plot. What values of the mission attributes optimize the throughput of the calculator factory? How can we balance manufacturing system work flow using only minimal changes to the mission attributes?

1.3 BRINGING COMPLEX SYSTEMS INTO BEING

Systems engineering is concerned with two activities: (1) determining what is the real problem and (2) documenting the best solution. Systems that can be designed

using systems engineering methodology are as varied as a military command and control system, transportation network, manufacturing production line, or business organization. The modern practice of systems engineering makes use of integrated product development teams that include engineers, scientists, marketing people, manufacturing specialists, product support personnel, stakeholders, and others. These multidisciplinary teams often start with a vague statement that "things could be better" and proceed through dialog with the stakeholders to identify the problem and then find the best solution.

In complex systems, the statement of the problem—called the system requirements—and the description of the problem solution—called the system design specification—can be quite voluminous. Therefore, computer databases, such as CORE, are now used to manage the system requirements and the system design specification as they evolve during the system engineering life cycle. These databases have replaced traditional methods that used paper documents because: (1) databases can be made much less ambiguous in describing the requirements and the system design specification than ordinary English text documents and (2) computer databases can be maintained current while being shared by the integrated product development team as the requirements and system specifications evolve.

Simulation models are a good way to evaluate alternative system concepts and designs in order to determine the "best" way to satisfy the system requirements. Simulation models can also be used to allocate top-level system requirements to lower level system descriptions. In previous sections of this chapter, we discussed the nature of complex systems and how the OpEM graphical modeling language provides a formal, explicit representation of operation, and structure that is needed for complex systems. In this section we will describe the systems engineering, systems development process, and life cycle, discussing how simulation can assist in design of complex systems.

1.3.1 Definition of Systems Engineering

According to Blanchard and Fabrycky (1998), systems engineering is the application of scientific and engineering efforts to: (1) transform an operational need into a description of systems performance parameters and a preferred system configuration through the iterative process of functional analysis, synthesis, simulation, optimization, design, test, and evaluation; (2) integrate related technical parameters and assure compatibility of all physical, functional, and program interfaces in a manner that optimizes the total system design and definition; and (3) integrate functional design, reliability, maintainability, human factors, system support, safety, security, and other related specialties into the total engineering effort.

According to Buede (1999), engineering of a system is an engineering discipline that develops, matches, and trades-off requirements, functions, and alternative system resources to achieve a cost-effective, life-cycle balanced product based on the needs of the stakeholders.

Stakeholders include users, owners, manufacturers, maintainers, and trainers to name a few. Progress is achieved through iteration of the system development process that proceeds from operational needs to system-level requirements to component and configuration item (CI) specifications. In general, the system development process concerns are: what the system does, how well the system must do what it does, and how the system should be tested to assure all requirements have been met.

A member of an integrated product development team (IPDT) must possess the following characteristics: (1) think in terms of the entire system life cycle (conceptual design, preliminary system design, detailed hardware–software design, production/construction, maintenance and support, retirement, and disposal); (2) recognize the importance of reliability, maintainability, human factors, supportability, producibility, disposability, product quality, economic, and related factors at program inception; (3) utilize computer-aided systems engineering (CASE) tools, such as CORE; and (4) understand the various phases of a program, different organizational activities and their interfaces, and the communication processes that need to exist for effective systems engineering management. IPDT members include stakeholders, system architects, discipline engineers, systems analysts, systems engineering process engineers (i.e., CORE experts), and project management.

How does systems engineering relate to systems science and systems analysis? Systems science is concerned with systems theory and formal models that lead to insights into first principles of complex systems operation and structure. Systems science provides a concise "way of thinking" about complex systems. Context-sensitive systems (CSS) theory, discussed throughout this book, is an example of systems science. Systems analysis is the iterative process of applying various analytical methods (such as simulation, Markov models, and optimization techniques) in the evaluation and optimization of system requirements, alternative system concepts, and component architectures including interfaces. Systems engineering incorporates system scientific principles and employs systems analysis methods to attain its objectives.

A summary of the systems engineering, systems development process includes the following tasks: (1) dialog with the stakeholders to acquire knowledge about the problem to be solved; (2) translate stakeholder problem description into top-level technical (originating) requirements that are measurable and observable; (3) define the functions required for the system to accomplish all mission objectives and organize these functions into concurrent processes that collaborate; (4) develop alternative operational concepts each expressed as a parallel process and evaluate them, and select concept that best satisfies the originating requirements; (5) allocate each functional process to a component within a network of system components, called the system architecture, and evaluate alternative allocations in order to discover the best system design in terms of all requirements; (6) determine system requirements at lower levels of system decomposition (component and configuration item levels) needed to optimize top-level system process and structural requirements using simulation; (7) decompose system design into

a network of networks from the top down until a level is reached where each system component or configuration item can be developed by an independent design team, and (8) use a computer database to document the conceptual and preliminary system design in terms of system requirements, including all data and rationale for all design decisions. Simulation is used throughout the systems development process to overlap design and verification activities in order to find system deficiency problems as early as possible during the systems engineering life cycle.

1.3.2 Levels of System Description

Levels of system description, shown in Figure 1.15, are listed from top to bottom: (1) mission, (2) function, (3) parallel process, (4) resources, and (5) physical system, plus an interface between the process level and the resource level. In top-down system design, a system or organization is defined starting at the mission level and proceeding to the physical level. In contrast, during detailed hardware–software design, system components are designed, built, and tested from the bottom-up.

The mission level is concerned with overall objectives of an organization or system and its operational environment. Mission effectiveness is defined as the ability of an organization or system to perform its mission and achieve its mission objectives while operating within its operational environment. The ability of a system or organization to perform its mission can be measured in several ways; thus, mission effectiveness is often a vector of measures. The most common measures deal with the probability of achieving each mission objective. Each MOE must be defined so as to be observable and measurable. Performance measures relate to user satisfaction, system efficiency, response time, or use of resources. Measures of performance are concerned not only with whether the objectives were achieved or not but also with how well the system performed its mission.

At the function level, precisely what tasks must be done to achieve each mission objective is determined. Each function is defined in terms of required

Mission level	Mission, mission objectives, user environment, measures of effectiveness	
Function level	Functions, interactions among functions and environment, functional flow diagrams	
Parallel process level	Parallel process, sequences of states and events, directed graph model	
Mission attribute	User	What
Interface	Producer	How
Resources level	Resource sets, equiment Dependency diagrams	
Physical system level	Physical system	

Figure 1.15 Levels of system description.

inputs, the task to be performed, and the outputs produced. Interactions among the functions and with the environment are described by flow diagrams. Functional flow diagrams describe the sequence in which the functions are performed, as well as the connection between inputs and outputs among the functions and with the environment. Often, a hierarchy of functional flow diagrams is developed, the level of detail increasing as each level is expanded. The diagrams also specify functions that can be performed simultaneously. Thus, they specify the sequential and parallel arrangement of functions defined by the input/output relationships. Flow diagrams are useful for specifying exactly what a system or organization is to do without considering how the functions will be carried out. They allow the user to express his or her need in terms of what is required, rather than in terms of a proposed solution. Functional flow diagrams are often used by CASE tools such as CORE (Buede, 1999), to specify what the system does first before considering how it will be done. This avoids converging to a physical component architecture design before the stakeholder problem is understood adequately.

At the parallel process level, an OpEMCSS model is developed that describes operational behavior of an organization or system. A parallel process model can also be developed as a hierarchy from the top down using ExtendSim's hierarchical blocks. To develop a parallel process model, each function in the flow diagram is subdivided into periods of time called states, when either: (1) a function is being performed by a resource or (2) a function is either waiting for a resource to become available or for some other logical condition to be satisfied before it can continue. The OpEMCSS simulation model of the Buede system development process, discussed in the next section, is an example of translating ambiguous and unexecutable IDEF0 diagrams into an unambiguous and executable OpEMCSS simulation.

The physical system is represented as a collection of resources, each of which is a group of people and equipment needed to perform particular system functions. In Chapter 5, the relationships between resources and functions are represented using equipment dependency diagrams, and they are used to compute reliability and availability. Reliability failures and system recovery processes must also be included in the system design model as well as resource contention discussed previously.

After states of the process have been identified, an OpEM directed-graph description of the process is developed that defines the sequences of states and events allowed for the process. In the directed graph, functions that always occur in the same time sequence are grouped to form a process. Functions that can be performed concurrently are grouped into separate processes, forming parallel processes. A parallel process model always allows for the highest degree of parallelism possible, independent of the physical system. Program logic specifies the particular level of parallelism achievable for a given architectural design. Examples in later chapters show how logic can specify the degree of parallel operation achievable for a given architectural design. The behavior of the environment in which the system operates is also represented by process threads

included in the system model to simulate the dynamic demands of the environment on a system. When a system model includes the environmental and external system processes, it is called an *operational model*.

A mission attribute, as discussed above, characterizes the operation of a collection of people and equipment performing a particular mission function. Mission attributes, also called key performance parameters (KPPs), are used to aggregate system details in order to simplify a model. However, we must make sure that the individual aggregations do not interact. We want to reduce system details in the model without masking system complexity issues. Some attributes are probabilities of alternate actions; decision-making rules; task reaction time; and probabilities of failure that result from resource availability, operational reliability, capability, and personnel ability. Mission attribute, or KPP, values are obtained from the objectives hierarchy document. A resource is a collection of people and equipment that performs a function or mission task. It represents the aggregate structure of the physical system prior to the definition of the physical system component architecture. Consider an elevator system in an apartment building. If there are three elevators, there are three elevator processes in parallel. Each process can consist of four states: (1) idle, (2) move one floor, (3) open door, and (4) close door. For the elevator process to perform these functions, an associated elevator resource must be available and operating. See Chapter 9 for a detailed elevator model.

The physical system level is the detailed physical description of how the system operates. This includes the physical system component architecture and all the algorithms and methods needed to achieve all system performance, cost, and schedule requirements.

1.3.3 Systems Engineering Life Cycle

One version of the systems engineering life cycle is shown as a timeline diagram in Figure 1.16, and it is described in more detail in Blanchard and Fabrycky

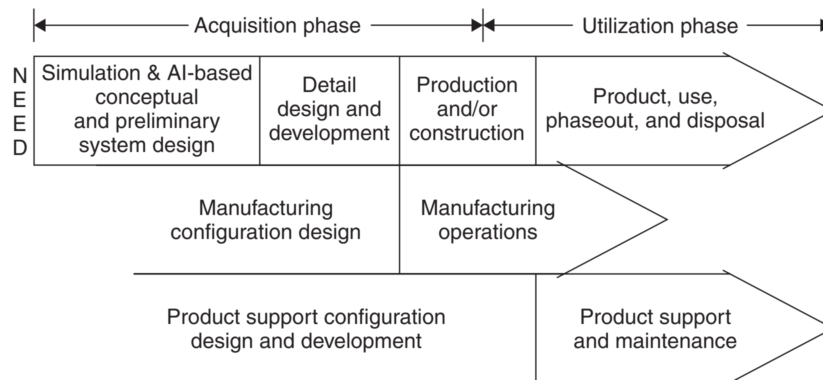

Figure 1.16 Systems engineering life cycle.

(1998). The detailed phases shown are conceptual design, preliminary system design, detailed hardware–software design, production/construction, maintenance and support, retirement, and disposal. Manufacturing and product support configuration design and development are shown concurrent with system design (conceptual, preliminary, and detailed system design) in order to emphasize the multidisciplinary nature of the design team.

The phases of the systems engineering life cycle according to the Defense Systems Management College (DSMC) are mission area analysis, concept evaluation, program definition and risk management, engineering and manufacturing development, production and deployment, operations and support, and disposal.

Mission area analysis is the "front end" of the systems engineering process where the "real" problem is defined and the top-level requirements are specified in the database. Scenario descriptions of what the system will do to solve the stakeholder's problem are specified. As discussed in the summary of the model development procedure, system mission, mission objectives, environment and contextual interfaces, and measures of effectiveness are defined. According to Buede (1999), requirements can be organized into a hierarchy that consists of mission, originating, derived, and system requirements. Mission requirements are obtained from the stakeholders and define the stakeholder's problem. Originating requirements define the system design problem as constrained by the mission requirements. Derived requirements are obtained by simulating alternative operational concepts in order to identify trades leading to more detailed requirements that further constrain the system design problem. System requirements are mission, originating, and derived requirements stated in precise engineering terms.

During concept evaluation, the design team identifies all feasible system concepts to achieve the system missions and mission objectives and evaluates each concept using the specified operational scenarios. Mission objectives are decomposed into functions, interactions among functions, interfaces with the environment, and functional flow diagrams. In the model development procedure discussed above, functional flow diagrams are translated into OpEMCSS process diagrams that describe sequences of states and events, message passing among processes, and process synchronization and resource contention. The OpEMCSS process diagrams allow simulation of alternative system concepts and evaluation of trade-offs including the dynamic demands that the operational scenarios place on the system. Alternative technologies and processes are considered, and processes are allocated to top-level, architectural components and evaluated. The design team attempts to simplify component interfaces and communications required for collaboration and, at the same time, maximize system effectiveness (mission effectiveness, reliability, availability, etc.) and minimize cost.

The conceptual design phase culminates in the: (1) systems design specification (type A) that describes top-level system requirements and, at least, one feasible solution stated in terms of functions, processes, top-level architectural components and interfaces, and design of the system qualification system; (2) systems engineering management plan (SEMP), which discusses the organization and work required for the system acquisition phase; (3) test and evaluation

master plan (TEMP), which defines the demonstration and validation modeling and testing to assure all requirements have been met; and (4) conceptual design review where the decision to continue is made by the stakeholders.

If the stakeholders accept the conceptual design, then the preliminary system design (demonstration and validation) phase can begin. The system design specification (type A) provides operational and maintenance requirements (a statement of the problem) and a very top-level description of one or more feasible solutions. Given these results from the conceptual design phase, the systems engineering process is again applied. However, now team efforts focus on system functional analysis, synthesis and allocation of requirements to subsystems, simulation and optimization, and production of detailed component specifications (type B, C, D, E).

The systems engineering process is iterated until a level of system description is reached where detailed hardware–software design can be effectively and efficiently performed. The complexity of each component or CI, defined by a detailed component specification, is appropriate for a single, independent, design team of about 7–12 people to handle. Further, the requirements for each component at this level have been specified such that, when the component is built, it can be integrated into the system and the overall system will satisfy all top-level requirements. Computer simulation of the system, modeled as a set of concurrent processes allocated to a component network, is a cost-effective way to make sure that top-level system requirements are decomposed properly all the way to the component level required for a successful detailed design effort. Further, simulation can evaluate alternative component architectures in terms of systems effectiveness and cost, and using the OpEMCSS special blocks, alternative detailed algorithms and methods can be evaluated.

In large, complex systems, the architecture is expressed as a component network of networks in order to simplify the detailed hardware–software design problem. Thus, the top-level requirements must be decomposed to provide component and CI requirements. For example, suppose the top-level availability (probability that the system will be available when called upon to function) is required to be 0.999999. If the system is decomposed into four components at the next level down, what must the availability of each of these components be in order to achieve the top-level system availability? We will consider this problem further in Chapter 5.

The preliminary systems design phase culminates in a system design review where the decision is made whether or not to continue. If a decision is made to continue, then the detailed hardware–software design and development (full-scale development) phase can begin. During detailed design, each system component is designed, based on its detailed component specification, and one or more prototypes are built. Prototypes usually evolve from ad hoc, breadboard models to engineering models to manufacturing prototypes that confirm that all manufacturing processes and product documents are ready for the production/construction phase.

Prototype system components are designed, built, and tested from the bottom-up. Therefore, improper flow down of top-level requirements to system components may not be identified until system integration and test, which occurs at the end of the detailed design phase, where correction becomes very costly. These errors could require changes in the manufacturing and product support configurations, product documentation, and product system. Thus, proper flow down of requirements during conceptual and preliminary systems design is critical to project success. Simulation of the system as a network of networks allows virtual verification of correct requirements flow down to be done early in the project.

The detailed design phase results in product, manufacturing, and product support specifications. Production of manufacturing prototypes occurs to test the manufacturing system and provide for operational tests. A critical design review decides if the system will go to full production.

In summary, simulation can be used to determine and specify detailed component requirements, including detailed algorithms and methods, necessary to achieve top-level system requirements. This must be done during the conceptual and preliminary systems design phases when it is much cheaper to make design changes. Further, at each stage of system design and development, it is necessary to evaluate the proposed design in terms of system requirements. During conceptual design, discrete-event simulation (DES) is used to evaluate the design and perform the requirements flow down. Next, during preliminary systems design, combined continuous and DES is performed to evaluate alternative technologies and software algorithms. As part of risk management, alternate implementation approaches for high-risk functions can be included in the simulation until a clear choice can be made. During detailed hardware and software design, real-time simulations including hardware/software/man-in-the-loop techniques are used to evaluate prototype components (hardware or software) within a simulated operational environment.

Finally, system integration occurs and the complete system is subjected to a rigorous test and evaluation procedure to verify and validate that all requirements have been satisfied. Simulation may continue to be used during production and product use phases of the life cycle to evaluate enhancements or the affect of changes that occur during the system operation phase.

1.3.4 Simulation of the System Development Process

The system development process is a goal-oriented activity that can be decomposed into a functional flow of sequential and concurrent tasks just like the pizza making process discussed at the beginning of this chapter. An OpEMCSS process diagram (PM_SM_Duality.MOX) of an organization that performs system development projects, as described in Buede (1999), is shown in Figure 1.17. The model consists of three top-level processes: (1) project arrival, (2) technical management, and (3) systems design project.

The interaction with the organization's environment is modeled by the project arrival process. Projects arrive periodically and enter the organization where they

Figure 1.17 OpEMCSS process diagram of an organization that performs system development projects.

are passed on to the technical management process. The first task of technical management is project planning where the complexity of the project, resources required, and schedule is estimated. After project planning is complete, a project is passed to the systems design project process. Technical management continues to monitor the project and take corrective action when the project deviates from the planned schedule.

The systems design project process is shown in Figure 1.17 as six, concurrent hierarchical boxes: (1) define system-level design problem, (2) develop system functional architecture, (3) design system physical architecture, (4) develop system operational architecture, (5) develop interface architecture, and (6) develop qualification system. First, the Define system-level design problem process is performed that produces a set of approved problem definition documents. Next, the Develop system functional architecture process is performed that produces a system-level functional architecture description. Functional design is followed by the Design system physical architecture process. A candidate generic physical architecture is selected, and the functional architecture design is reworked to be compatible with the physical architecture. While the functional architecture is being updated, a candidate physical architecture is specified that includes detailed algorithms and methods. When an approved system problem definition, system-level functional architecture, and candidate physical system architecture descriptions are available, the Develop system operational architecture process is performed. During this process, performance and risk analysis is performed using a system design simulation model (system emulation) that may result in requirement changes, functional architecture changes, or physical architecture

changes. Given that system performance requirements are satisfied and satisfactory project risk is determined, stakeholder approval for the system design is sought. An approved operational architecture and an approved system problem definition allow the Develop interface architecture and Develop qualification system processes to be performed concurrently with the documentation of subsystem specifications.

Define System-Level Design Problem The Buede functional flow IDEF0 model for the Define System-Level Design Problem is shown in Figure 1.18 translated into OpEMCSS processes. Each function is modeled by two Wait Until Event blocks, a Global Reaction Time Event block, and one or more Local Event Action and Message Event Action blocks. First, a function process waits for all required artifacts to be available to start the function process. Second, a function process waits for all people and equipment resources needed to perform the function to be available. Third, the time to perform the function is modeled by the reaction time block. Fourth, global attribute counters are updated to model all artifacts produced by the function for the current project. A Message Event Action block is used to share these artifact attributes with all function processes for the project. The six Define system-level design problem processes are as follows.

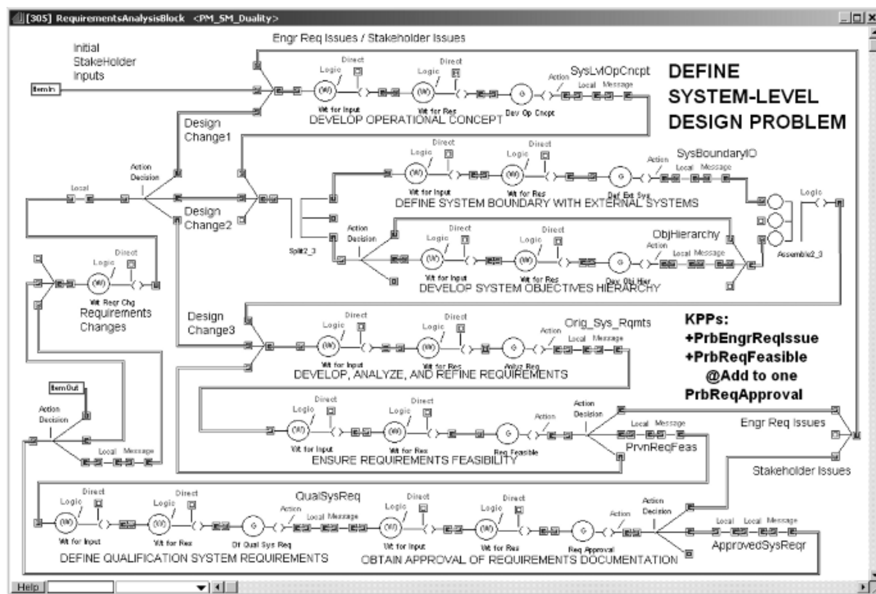

Figure 1.18 OpEMCSS process diagram of the Define System-Level Design Problem process.

Develop Operational Concept (Mission Analysis) The operational concept consists of the external systems diagram, a collection of operational scenarios, and mission requirements. The external systems diagram defines the system as it interacts with its environment, and it is defined in very general terms during mission analysis. A collection of dynamic operational scenarios is developed that define system use and the dynamic demands of the external systems and system environment on the system. Mission requirements are obtained from stakeholder inputs and are stated to maximize design flexibility. The operational concept covers the entire life cycle of the system and completely defines the stakeholder problem using common language.

Define System Boundary with an External Systems Diagram The external systems diagram shows the system boundary and interfaces with the external systems, and it is developed through analysis and evaluation of the operational concept. For example, timeline analysis can be used to make input/output requirements between the external systems and the system more explicit to facilitate drawing the system boundary and needed interfaces.

Develop System Objective Hierarchy The objective hierarchy forms the value system used to define the stakeholder's satisfaction with the system. Objectives are divided into program and operational. Included in operational objectives are required values for MOEs that measure system performance from outside the system and MOPs that measure system performance from within the system. MOPs include system response latencies, wait times, accuracy, system RMA (reliability, availability, maintainability), user satisfaction, and system quality. Program objectives include costs and schedule.

Develop, Analyze, and Refine System Requirements The originating requirements document (ORD) is stated in the language of the stakeholders, and it is derived from the operational concepts, external systems diagram, mission requirements, and stakeholder objectives. The ORD is formed into a hierarchy of individual requirement statements that can later be associated with the system functions and architectural components including external and internal interfaces. There are four categories of originating requirements: (1) input/output (external systems and context), (2) technology and systemwide (technology, suitability, cost, and schedule), (3) trade-off (value algorithms), and (4) system qualification (methods and plans). A requirements database assures that each system requirement is traceable to the implementation of the requirement and includes the stakeholder level of need (how bad do they want it) in order to facilitate design trade-offs relative to cost and schedule.

Define Qualification System Requirements Given the system requirements, the qualification system requirements are specified in order to assure verification (build the system right?) and validation (build the right system?) is achieved. The qualification system requirements are used to specify the hardware and software systems used during component, subsystem, and system tests.

Ensure Requirements Feasibility The set of requirements can include conflicting goals or not fit within the scope of the system's boundary and fundamental mission objectives. The requirements may imply deviation of development cost, schedule, or risk from that originally planned.

Obtain Approval of Requirements Documentation The stakeholders must approve the ORD. The ORD defines the stakeholder problem. The Ensure requirements feasibility process terminates with three alternative paths: (1) begin Define qualification system requirements process, (2) return to Develop, analyze, and refine system requirements process to correct feasibility problems, or (3) return to top of model for major changes to the operational concept, system boundary, or system objectives due to engineering requirements issues. The Obtain approval of requirements documentation process has two alternative paths: (1) (ORD okay) continue with the functional system design or (2) (ORD has stakeholder issues) return to top of model for major changes to the operational concept, system boundary, or system objectives. Currently, these alternative action paths in the model are decided by probabilities that are a function of project complexity and organization technical competencies. The KPPs used to decide these alternate process paths are shown on the right-central area of Figure 1.18.

Develop System Functional Architecture The Buede functional flow IDEF0 model for the Develop System Functional Architecture is shown in Figure 1.19 translated into OpEMCSS processes.

According to Hatley and Pirbhai 1 (1988), a functional architecture consists of: (1) input processing, (2) process model (transform inputs to outputs), (3) output processing, (4) control model, (5) maintenance model, and (6) user interface processing. Design methods include both top-down, bottom-up, and a combination. The 10-step simulation development methodology presented in this book is a combination of top-down and bottom-up. Functions required to achieve each mission objective are listed and then combined into sequential and parallel arrangement. Systems analysis is performed as the model is expanded top-down to include more and more functional details.

Create Simple Functionalities for Operational Concept The operational concept includes a description of all external systems as they interact with the system. The inputs from external systems and outputs to external systems are specified. Simple functionalities describe the functions required to receive an external input and generate the appropriate external output.

Draft and Evaluate Functional Model Simple functionalities are combined to produce a draft functional model. Several inputs may be required to be combined to produce an output.

Draft Data Model for Functional Model Data items represent data, material, or energy that flow through the system from inputs to outputs.

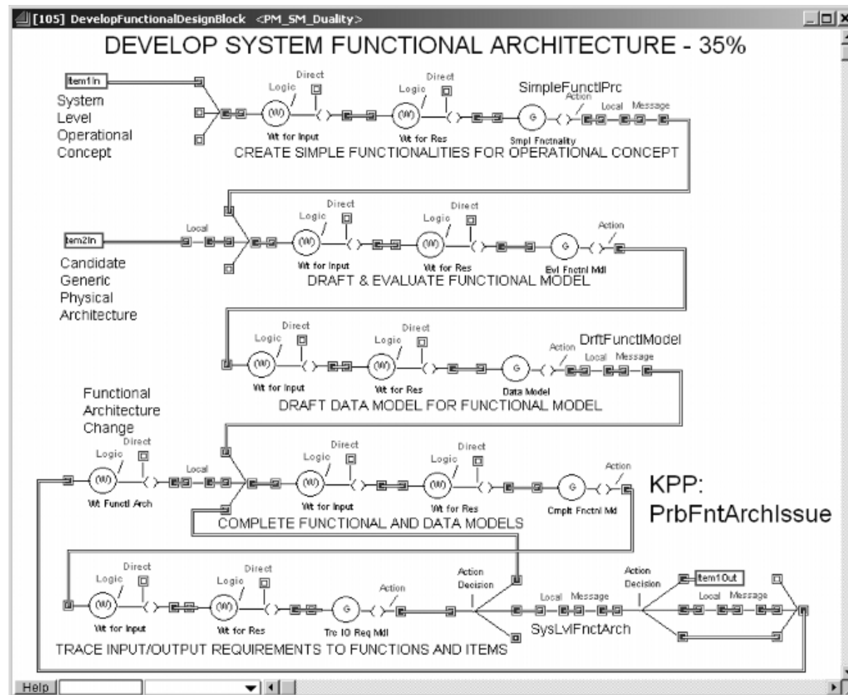

Figure 1.19 OpEMCSS process diagram of the Develop System Functional Architecture process.

Complete Functional and Data Models The draft functional and data models are combined to complete the functional architecture. Control items that are internally generated data items and define when functions can be executed are added to the model. User interface processing and system maintenance and failure recovery functions are added also.

Trace Input/Output Requirements to Functions and Items Input/output requirements are linked to the functions and data items. This ensures that all functions and data items are required. Any that do not have a link are either removed or derived requirements are added and linked to the function. During functional design, if a function is added that is considered required but has no originating requirement, a corresponding derived requirement is added subject to stakeholder approval.

Design System Physical Architecture The Buede functional flow IDEF0 model for the Design System Physical Architecture is shown in Figure 1.20 translated into OpEMCSS processes. The physical system architecture defines specifically how the system-level functional architecture is implemented. If the functions can be specified to have a one-to-one relationship with a physical

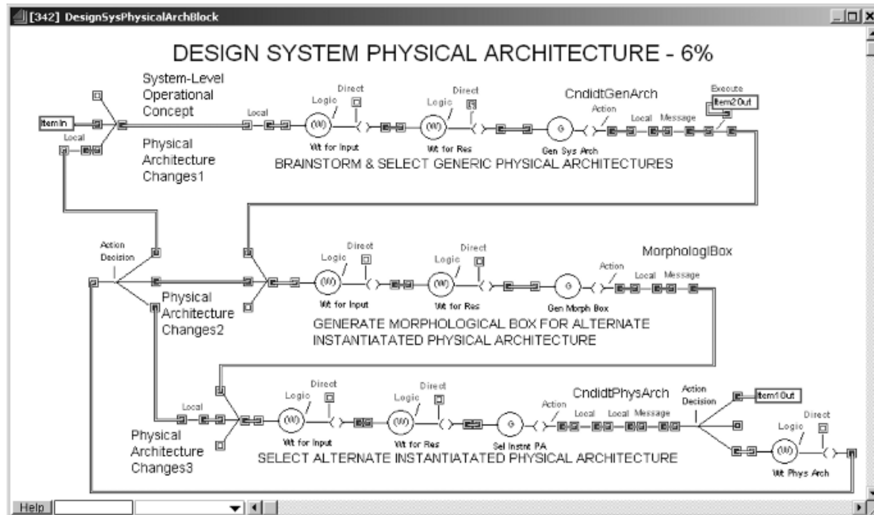

Figure 1.20 OpEMCSS process diagram of Design System Physical Architecture process.

component that performs the function, then the next project function is greatly simplified.

Brainstorm and Select a Generic Physical Architecture Given the system-level functional architecture, alternative generic physical architecture concepts are defined and evaluated. The functional architecture is designed to optimize mission effectiveness that is the probability that the system will achieve all mission objectives when called upon to do so. The physical architecture is designed to optimize mission performance that includes various performance measures, reliability, availability, survivability, social acceptance, and others. A system design simulation model can be used to select a candidate generic physical architecture, and the system functional architecture definition is then updated to accommodate the specifics of the candidate generic physical architecture selected.

Generate a Morphological Box for Alternate Instantiated Physical Architecture A generic physical architecture defines the structure of the components but not what algorithms or methods the components use. The morphological box is a way to define the collection of all alternative sets of component algorithms and methods to be evaluated where each alternative is an instantiated physical architecture.

Select Alternate Instantiated Physical Architecture An operational simulation must be used to evaluate these alternative instantiated physical architectures and select a candidate that optimizes mission effectiveness and mission performance.

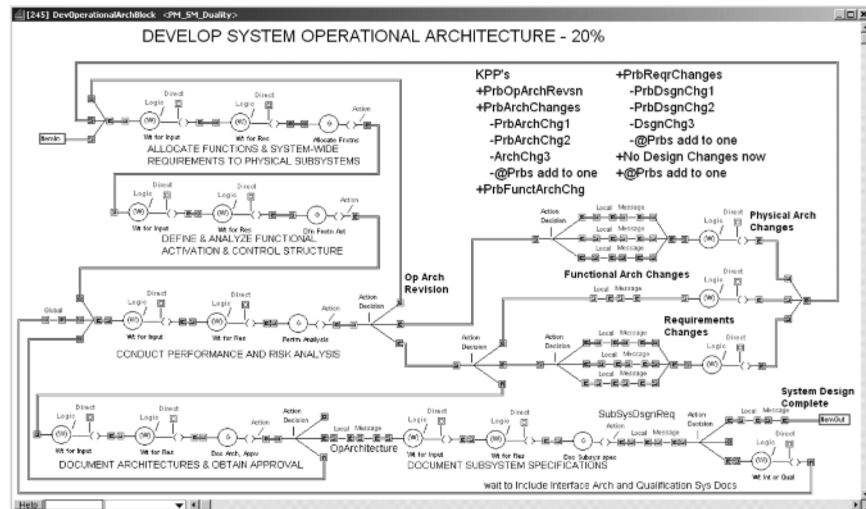

Figure 1.21 OpEMCSS process diagram of Develop System Operational Architecture process.

Develop System Operational Architecture The Buede functional flow IDEF0 model for the Develop System Operational Architecture is shown in Figure 1.21 translated into OpEMCSS processes. The system-level functional architecture and an associated instantiated physical architecture are combined to produce a system operational architecture, bringing the entire design together (including interfaces and qualification system), documenting it, and obtaining stakeholder approval. The processes are as follows.

Allocate Functions and Systemwide Requirements to Physical Subsystems This is the most critical design step because it affects whether the system meets all mission objectives or not. Functions must be allocated to hardware, software, or people resulting in flexibility versus speed trade-offs. Further, allocation of functions to components affects greatly the complexity of the component interfaces. Systemwide requirements such as reliability and failure recovery must be allocated to each component resulting in additional trade-offs.

Define and Analyze Functional Activation and Control Structure Each function requires certain input artifacts to be available in order to begin execution, defining precedence relationships among the functions. Further, functions often collaborate to produce output artifacts, requiring behavior synchronization. The activation and control structure must be evaluated to assure that no deadlocks can occur and the behavior is consistent.

Conduct Performance and Risk Analyses Operational simulation must be used to evaluate each candidate architecture to verify all requirements are met, including mission effectiveness and mission performance. As shown in Figure 1.21,

performance and risk analysis can result in requirements, functional architecture, or physical architecture changes.

Document Architectures and Obtain Approval As shown in Figure 1.21, failure to obtain stakeholder approval can also result in requirements, functional architecture, or physical architecture changes.

Document Subsystems Specification Once stakeholder approval has been obtained for an architecture, the interface architecture and qualification system are developed. At the same time component specifications for the architecture are written. These documents are provided to hardware/software detailed design teams.

Develop Interface Architecture The processes are: (1) Define interface requirements, (2) Evaluate and select high-level interface architecture, (3) Develop functional architecture for interface, (4) Develop physical architecture for interface, and (5) Develop operational architecture for interface. Any of these five processes may require rework.

Develop Qualification System The processes are: (1) Document subsystem specifications, (2) Define qualification system design, (3) Develop functional architecture of qualification system, (4) Develop physical architecture of qualification system, (5) Develop operational architecture of qualification system, (6) Develop interfaces of qualification system, and (7) Define models for qualification. Any of the first three processes may require rework.

1.3.5 Simulation-Based Systems Engineering

The solution space for a complex system is astronomical in size and, thus, requires automated design search methods in order to achieve optimal mission effectiveness, system performance, cost, and schedule.

Simulation-based systems engineering using the ExtendSim library OpEMCSS allows the automation of the search for the optimal combination of component algorithms and methods that defines an instantiated physical architecture. Also, the development of functional control rules can be automated. In fact, the optimal combination of an instantiated physical architecture and concomitant functional control rules can be determined.

In OpEMCSS, special blocks can be created that implement alternative component algorithms and methods in the model that are selectable by an attribute. This can be done for each component. An Evolutionary Algorithm (EA) block can be used to search the solution space by selecting the optimal combination of these attributes (algorithms and methods) as defined by an evaluator function added to the system design simulation. The solution space is defined by the collection of all alternative sets of component algorithms and methods. See Chapter 3 discussion on optimizing an inventory system.

Further, using OpEMCSS, a Classifier Event Action block can be used to learn the function control rules as well as overall system strategic decision-making rules, allowing intelligent interactions between the system and its environment. See discussions in Chapters 7–9 on intelligent decision-making systems.

Another idea is to use a Classifier Event Action block as a design agent that operates independently of the system model but part of the simulation program. This design agent takes system evaluations as input and makes changes to the design in a search for the best design that meets all stakeholder requirements. The design agent rules would be developed manually by systems engineers to oversee the design process.

Finally, an OpEMCSS system design simulation could be linked to the design capture database tool to receive KPPs for model inputs and for automatic update of the design database once an optimal design has been discovered. The system development process, discussed in the previous section, would be modified to allocate requirements to components after the optimal instantiated physical architecture and functional control rules have been discovered. The system development functions Select alternate Instantiated Physical Architecture, Define and analyze functional activation and control structure, and Conduct performance and risk analyses would be combined into a single function implemented by the system design optimization model.

1.4 SUMMARY

The concept of a parallel process can be used to improve the stakeholder–prod ucer dialog. The stakeholders of a system or organization are the people with a problem to be solved or other interest in the system (manufacturing, maintenance, training, etc.). The producer is the team of design engineers who try to satisfy the stakeholder's needs. The dialog between stakeholders and producer consists of the stakeholders expressing their needs and the producer determining the best alternative to satisfy these needs. The stakeholders and producer view system operation differently and speak different languages.

An OpEMCSS graphical simulation model is a means of expressing the stakeholder's view of system operation and structure in a language both sides can understand. Performance requirements of the stakeholders, called key performance parameters (KPPs), can be expressed as a set of mission attributes that are the inputs of the system design model. An OpEMCSS graphical simulation model of the system provides the structure of states and events that can be used to define system evaluation criteria (needed to design the evaluation system) that is derived directly from stakeholder requirements. The MOEs and MOPs displayed in the model scoreboard and the run report, the outputs of the system design model or emulation, are derived from stakeholder requirements and are understandable by stakeholders. Finally, animation of process flow, message passing, and agent motion make it easy for stakeholders to understand the proposed system solution, expressed by the system design model, and visualize design trade-offs.

A system model can be verified by team members and the stakeholders. Both can be involved in the development of the OpEMCSS graphical simulation model, either directly (by selecting and connecting the blocks) or indirectly (through review and comment). Team members can review event-state traces (timelines) to verify and validate that the system design expressed by the OpEMCSS simulation model is correct. They can further validate the model using a statistical report, based on sufficient replications for convergence, by simulating a scenario in which the outcome can be predicted by other means. Involvement by the stakeholders and team members builds confidence in the system definition model and greatly increases the ability of model users to understand system model results.

The OpEMCSS graphical simulation language can be used to evaluate a system design as defined by a design specification and requirements database such as CORE discussed in Buede (1999). The language describes explicitly functional decomposition, concurrent interactions among parallel functions, and message flow. It can represent multiple instantiation of processes and their synchronizations. Processes can come into existence, interact concurrently, and go out of existence using explicit language blocks for expression. Processes can send and receive messages, and processes can wait for messages before continuing with the next function in order to model either data or control flow.

OpEMCSS can represent different views that are commonly used to simulate systems. These views are queuing systems, process and resource, and functional flow. The physical architecture as a network of communicating components can also be modeled.

Queuing systems is a view that models a network of queues and servers. The OpEMCSS Wait Until Event block works with either of the two reaction time blocks to model a queue and server. The Priority.mox model, discussed at the end of Chapter 2, is an example of preemptive, priority scheduling in a single-queue single-server system that is difficult to represent in some simulation languages.

Process and resources approach is a view that models the operation and structure of a system. The space station and the producer–consumer model, discussed in this chapter, are only two examples. More are described in Chapters 2, 3, 7, 8, and 9.

Functional flow is a view that models a network of functions that send messages to each other. Messages are transformed from input to output as one action of a function. As discussed above, processes can pass messages (process instance item attributes) among each other. Also, a set of OpEMCSS blocks have been provided so that agents can pass messages (message items) among each other. The agents are represented as hierarchical blocks in ExtendSim that are connected to each other using explicit communication link blocks. Thus, communication channel capacity can be modeled explicitly. A combined process and resource and message flow model is discussed in Chapter 9.

The OpEMCSS library blocks can represent queuing, concurrent process interactions and synchronization, message passing and functional flow, and component network communications. It is a very rich language with which to express system designs as well as to facilitate team dialog. How can you learn to understand

and design complex systems using simulation-based systems engineering (SBSE) methodology that uses combined descriptive and executable models? We believe that system complexity is directly related to the degree of context sensitivity of state transitions required to model the system. Thus, in this book we begin with context-free, queuing systems such as the space station, continue with moderately context-sensitive systems such as the assembler–packer model, and then proceed finally to highly context-sensitive systems such as the traffic grid model discussed in Chapter 9.

Chapter 2 discusses basic simulation concepts. In particular, the statistical aspects of simulation are summarized briefly, the OpEM graphical language is described in more detail, and the Petri net model is compared to OpEM graphs in order to better understand context-sensitive systems theory. How simulations of interacting concurrent processes work is discussed, comparing both event-oriented procedural and object-oriented OpEMCSS implementations. Several context-sensitive systems are described that use the basic OpEMCSS blocks. The second model is a very simple example of a context-sensitive system that can be difficult to model using some simulation languages.

Chapter 3 provides a series of examples that demonstrate the system design and model development procedure using the basic OpEMCSS blocks. It also discusses advanced applications using some of the more advanced features of the basic blocks to model preemption and direct execution of events. The best way to learn to understand and design complex systems is to do a class project. Some suggested projects are discussed in the Chapter 3 problems section.

We have found that after a person gains a basic understanding of the queuing system and process and resource views through the study of example simulation models, a concise mathematical view can be meaningful. In Chapter 4, Markov models are presented that assist in understanding context-sensitive systems theory by providing an explicit mathematical representation of interacting concurrent processes. The state space is mapped onto the set of positive integers and a state transition matrix defines all state-to-state transition rules. A Markov model is analytical and provides faster model results than a simulation. Further, reliability and availability, discussed in Chapter 5, and queuing theory, discussed in Chapter 6, are based on Markov models. In complex systems, components share knowledge and adapt their behaviors in order to collaborate to achieve overall system objectives. As a result of such collaboration, the system often exhibits one or more emergent behavioral patterns. Determining the facts and rules required for collaboration is not easy, but it is greatly facilitated using an OpEMCSS graphical simulation model. The Classifier Event Action block, discussed in Chapter 7, implements a forward chaining, crisp or fuzzy expert system controller that applies rules to process instance attributes to make decisions. The Classifier Event Action block also includes a rule generation capability to facilitate knowledge engineering. The rules can be used to transform input facts into output facts that are sent to other processes. Some of these output facts can specify which alternative behavior a process should exhibit in order to collaborate.

We have found that when modeling physical systems, the motion and the spatial interactions of agents can be difficult to program. In Chapter 8, a set of OpEMCSS blocks is described that manage agent motion and interactions using a linear model based on discrete events. Continuous time models of agent motion can be very slow to run. When modeling agent motion using discrete events, a simulation runs much faster. As an example of agent motion modeling, a sonar array simulation where ships move in relation to sonar sensors is described. This process uses the Classifier Event Action block to decide what kind of ship has been detected.

In Chapter 9, we discuss multiagent systems. Context-sensitive systems (CSS) theory, agents, and agent interactions are discussed first to introduce the chapter. Next, the California State University Fullerton (CSUF) engineering building elevators are modeled as a multiagent system where each elevator is an agent that collaborates with the other elevator to reduce the waiting time for people wanting to use an elevator. A distributed vehicle traffic light system simulation is discussed next as a more complex example of a multiagent system as well as an SOS. In this model, each traffic light controller is an agent that adapts its behavior to achieve a highly desirable emergent behavior: The average vehicle waiting time in the network is reduced relative to an uncontrolled system. To conclude Chapter 9, OpEMCSS blocks are discussed that model various forms of agent communications, provide for separate local agent memories to store agent knowledge, and implement global agent "blackboards" used to facilitate collaboration during group problem solving and planning. An example of a multiagent model is discussed that features the use of these blocks.

Appendices A, B, and C collectively provide a user's manual for OpEMCSS model development.

PROBLEMS

1. Think of a goal-oriented activity that you already know how to do. For example, you might think of fixing car brakes, baking a cake, sewing clothes, wood working, or assembling a bicycle. Make a list of all the tasks required to achieve your goal. Next, assume you have lots of help. Organize your tasks into a functional flow diagram showing the sequential and concurrent relationships among the tasks. Use the basic OpEMCSS blocks to create a parallel process representation of your activity.
2. In a communications system, messages are divided into blocks and transmitted one block at a time. After each block is sent, a series of bytes called the block check are sent to allow the receiver to check the accuracy of information in the received message block. If an error is detected, the receiver requests that the block be transmitted again. The communications process diagram is shown in Figure 1.22.

   Mission attribute P1 is the probability that a valid message is received (no block check error), and (1 − P1) is the probability that the block must be

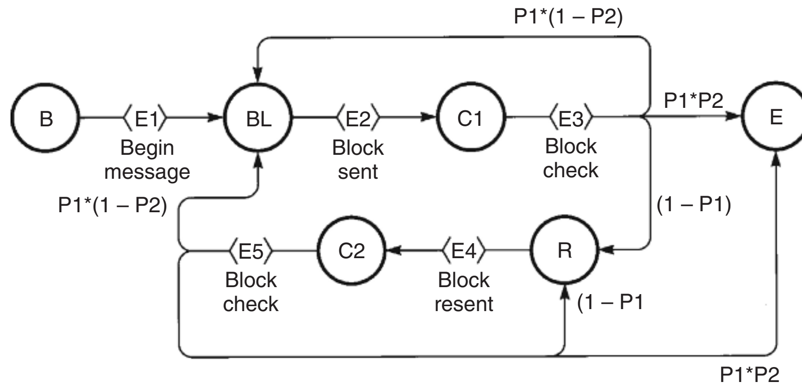

Figure 1.22 Communications system.

retransmitted. Attribute P2 is the probability that the last block of the message has been sent, and P2 equals the inverse of the number of blocks required to send a message. These probabilities are combined as shown in the figure to provide probabilities for the three alternate action paths out of events E3 and E5. P1 is 0.9, and a message is divided into 10 equal-sized blocks for transmission. It takes 10 time units to send a block and one time unit to perform the block check.

What percent of the total time required to send a message is the system in the BL state and doing useful work? What is the expected number of time units required to send a message? Develop an OpEMCSS simulation model similar to the thief of Baghdad model to answer these questions. Determine how many messages must be sent to obtain a large enough sample of outcomes to achieve convergence of model statistics by developing a convergence plot.

3. It is well known among gamblers that the odds favor the house in the game of craps. The rules of the game are as follows: On the first roll of a game, if the gambler rolls a 7 or 11, he wins $100, and the game is over. If he rolls a 2, 3, or 12 on the first roll, he loses $100, and the game is over. If on the first roll he rolls anything else, say N, he rolls again and continues the game. On any subsequent roll of a game, the gambler must roll the same value as the first roll, N, to win $100 and conclude the game. If he rolls a 7 on any subsequent roll, he loses $100, and the game is over. If he rolls anything other than N or 7, the game continues and he rolls again.

If a gambler plays craps long enough, he will lose all his money. If a gambler goes to Las Vegas with $1000 to start and bets $100 per game (where each roll of the dice requires 60 seconds, on average), how many hours are required for the gambler to lose all his money? What is the average probability of the gambler winning a game? A process diagram describing a trip to Las Vegas to play craps is shown in Figure 1.23. Write a simulation program similar to the thief of Baghdad model to answer these questions. Determine the number of

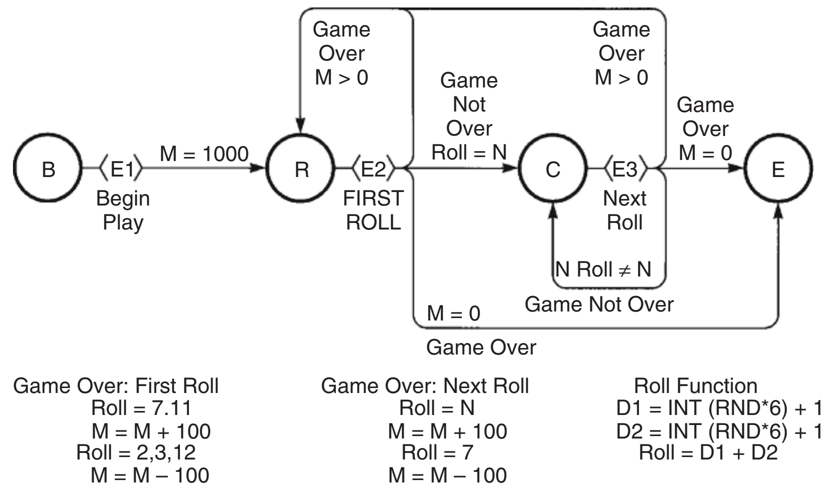

Figure 1.23 Las Vegas trip to play craps process.

trips to Las Vegas that must be simulated to obtain a large enough sample of outcomes to achieve convergence of model statistics. Develop a convergence curve for both the probability of winning a game and the average number of games required to lose all your money. Explain the difference in convergence rates between these two statistics.

4. Run the OpEMCSS simulation (PRODCONS.MOX) and develop sensitivity curves. Run the simulation and vary each reaction time Produce Calc, PRD, and Package Calc, PAK, from 10 to 90 seconds in steps of 10. Plot the five-model scoreboard values as a function of these reaction time values. Run simulation for each reaction time Calc on Belt, PCB, and Take Calc from Belt, TCB, for mean values of 10 and 5. Plot the model scoreboard values as a function of these reaction time values. For each MOE/MOP plot, reset each mission attribute to its nominal value, repeat this procedure for the next MOE/MOP plot. What values of the mission attributes optimizes the throughput of the calculator factory? How can we balance manufacturing system work flow using only minimal changes to the mission attributes?
5. Read the chapter and answer the following questions.
   a. What is it that distinguishes a complex adaptive system (CAS) from an ordinary network of system components?
   b. Given a system has two or more concurrent processes being executed, what is a context-sensitive interaction among these processes? What is it about system operation that makes a system context-sensitive as defined in this book?
   c. Explain the difference between reductionism and expansionism as they apply to system design and evaluation.
   d. Describe each of the phases of the systems engineering life cycle.

e. Describe each task of the systems engineering process and explain how these tasks change during the systems engineering life cycle.
f. Explain how the distributed traffic control system is a CAS. What is the emergent behavior of this system?
g. What are the three main views of the system used in systems analysis?
h. In the operational view there are two conceptual models commonly used to describe system operation: process and resource model and queuing theory model. What is the difference between these two?
i. In the structural view there are two conceptual models commonly used: data and control flow (DCF) model and component architecture diagram. What is the difference between these two?
j. What are the levels of system description that are developed throughout the systems engineering life cycle?
k. What does a split event do in a process model?
l. What does an assemble event do?

REFERENCES

Ashby W. R. *An Introduction to Cybernetics*. London: Chapman & Hall, 1961.

Blanchard, B. S. and W. J. Fabrycky. *Systems Engineering and Analysis*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1998.

Buede, D. M. *The Engineering Design of Systems: Models and Methods*. New York: Wiley-Interscience, 1999.

Clymer, J. R. Expansionist/Context-Sensitive Methodology: Engineering of Complex Adaptive Systems. *IEEE Trans. Aerospace Electronic Syst.*, 33(2), April, 1997, pp. 686–695.

Dijkstra, E. W.in *Co-operating Sequential Processes, Programming Languages*, F. Genuys, ed. New York: Academic, 1968, pp. 43–112.

Hatley, D. J. and I. A. Pirbhai. *Strategies for Real-Time System Specification*. New York: Dorset House, 1988.

Kauffman, S. *At Home in the Universe: The Search for the Laws of Self Organization and Complexity*. New York: Oxford University Press, 1995.

BIBLIOGRAPHY

Axelrod, R. M. *The Evolution of Cooperation*. New York: Basic Books, 1984.

Bahill, A. T., Alford, M., Bharathan, K., Clymer, J., Dean, D. L., Duke, J., Hill, G., LaBudde, E., Taipale, E., and Wymore, A. W. The Design-Methods Comparison Project. *IEEE Trans. Syst., Man, and Cybernetics—Part C Appl. Rev.*, 28(1), February 1998.

Capra, F. *The Web of Life*. New York: Bantam Doubleday Dell, 1996.

Clymer, J. R. Induction of Fuzzy Rules for Air Traffic Control, in Proceedings 1995 IEEE International Conference on Systems, Man, and Cybernetics, Vancouver, British Columbia, Canada, October 22–25, 1995, pp. 1495–1502.

Clymer, J. R. *Systems Analysis Using Simulation and Markov Models*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

Clymer, J. R. System Design Using OpEM Inductive/Adaptive Expert System Controller. *IASTED Int. J. Modeling Simulation,* 10(4), 1990, pp. 129–136.

Clymer, J. R., D. J. Cheng, and D. Hernandez. Induction of Decision Making Rules for Context Sensitive Systems, *Simulation*, 59(3), September 1992, pp. 198–206.

Clymer, J. R, P. D. Corey, and H. Bandukwala. Induction of Classification Rules for a Sonar System. *Simulation*, Volume 62(4), April 1994, pp. 256–267.

Clymer, J. R., P. D. Corey, and J. Gardner. Discrete Event Fuzzy Airport Control, *IEEE Trans. Syst., Man, Cybernetics,* 22(2), March–April 1992, pp. 343–351.

Clymer, J. R., P. D. Corey, and N. Nili. Operational Evaluation Modeling, *Simulation*, December 1990, pp. 261–270.

Corey, P. D. and J. R. Clymer. Discrete Event Simulation of Object Movement and Interactions. *Simulation*, March 1991, pp. 167–174.

Kelly, K. *Out of Control: The New Biology of Machines, Social Systems, and the Economic World*, Menlo Park, CA: Addison-Wesley, 1994.

Maier, M. W. On Architecting and Intelligent Transport Systems, *IEEE Trans. Aerospace Electronic Syst.*, 33(2), April 1997, pp. 610–625.

Oliver, D. W. Engineering of Complex Systems with Models, *IEEE Trans. Aerospace and Electronic Syst.*, 33(2), April 1997, pp. 667–685.

Pritsker, A. A. B. *Introduction to Simulation and SLAM II*. New York: Wiley, 1986.

Rumbaugh, J. R., Blaha, M. R., Lorensen, W., Eddy, F., and Premerlani, W. *Object-oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Yeh, R. T. *Applied Computation Theory: Analysis, Design, and Modeling*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

Warfield, J. N. *A Science of Generic Design: Managing Complexity through Systems Design*. Salinas, CA: Intersystems, 1990(two-volume set); second edition published by Iowa State University Press, Ames, IA, 1994.

Zeigler, B. P. and H. Praehofer. *Theory of Modeling and Simulation*, 2nd ed. San Diego: Academic, 1999.