Proceedings of the 2001 Winter Simulation Conference
B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.

DISTRIBUTED SIMULATION AND CONTROL: THE FOUNDATIONS

Wayne J. Davis

Department of General Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801 USA

ABSTRACT

This paper seeks a new simulation and execution paradigm for the design and operation of complex systems. An expanded life cycle for a simulation model is first provided. It is assumed that complex systems can be represented as systems of interacting subsystems, which evolve by executing tasks upon objects. Care is taken to distinguish the real world where process execution occurs from the virtual world where planning is addressed. It is illustrated that the ideal model should be able to both evaluate and control the subsystem that it addresses. The advantages of such approach are discussed with relation to both validation and execution needs. In particular, it is demonstrated that a distributed-controller based paradigm could provide significant advantages in the evaluation of the system using distributed simulation. This form of execution is also contrasted to evolving on-line simulation requirements that will support the real-time distributed management of these systems.

1 INTRODUCTION

Computers and information technologies have accelerated the construction of systems of ever-increasing complexity. Moreover, society's needs and desires will continue to drive system development. The ability to analyze and manage the emerging complex systems has not kept pace with the system evolution unfortunately. Consider the recent concerns with air traffic control systems. The consensus is that the current traffic exceeds the operational capacity at several major airports. However, this consensus cannot be verified because one cannot determine capacity of the current system or project how that capacity is affected by disruptions. Moreover, one cannot predict if a planned response will mitigate the disruption or will amplify its consequences with positive feedback.

However, simulation modeling and analyses is the only alternative for assessing performance of such complex systems. Analytical approaches do not and probably will not ever exist. The scientific/engineering community has recognized the expanding chasm between the current simulation capabilities and those needed to design and manage the current complex systems. Presumably, this chasm should influence the evolution of future simulation technologies. The chasm between capabilities and needs continues to widen, however. Given this state of affairs, one must conclude that inertial barriers are constraining this evolution, and these inertial barriers must arise from the current paradigms underlying accepted modeling and analysis practices. It now appears that the existing paradigms may have taken us to a dead end from where there is no possibility of bridging the chasm between needs and capabilities.

If this assertion is correct, then one must seek other paradigms, and that is the intent of this paper. The paper adopts a green-field approach, ignoring all available simulation technologies and practices. The paper first defines needs and then seeks means for addressing them. The paper's intent is not to provide a new paradigm per se. Indeed, there may be more than one solution. Rather, the goal is to discuss the essential capabilities that an effective paradigm must provide.

2 THE LIFECYCLE OF A SIMULATION MODEL

A list of essential modeling requirements obviously depends upon the model's intended use. Existing simulation practices primarily address the off-line analysis of a proposed system or a modification to an existing system. Many models are never verified because most models support the design of a proposed system. After the designed system is implemented, there is limited, if any, future need for the simulation model, and the model is abandoned.

Let us assume that a singular simulation model can be employed during both the design and operation of a system. Figure 1 provides a proposed lifecycle for a simulation model. The lifecycle assumes that the system design is also dynamic and distinguishes two primary application phases: off-line (associated with system design) and on-line (associated with system operation). This lifecycle will initially be addressed as a serial process moving from system conceptualization through design to operation and

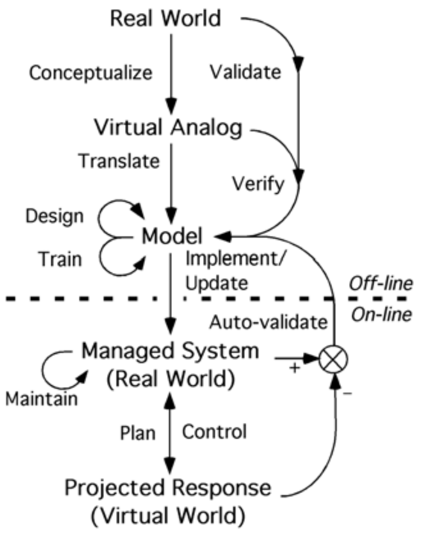

Figure 1: Expanded Life Cycle for a Simulation Model

maintenance. Later, design and operation will be considered as concurrent functions in order to allow the system to be modified even as it operates.

2.1 Off-Line Stages

2.1.1 Conceptualization

The first stage is conceptualization where the boundaries for the considered system are established. For example, one might consider the air traffic control system for managing all flights over the continental United States. Such complex systems are best viewed as a system of subsystems. The question arises as to what subsystems might exist, how each subsystem behaves and how the interactions among the subsystems might be coordinated. Although how one decomposes a complex system into its constituent subsystems may not be unique, basic rationales do govern the decomposition process, including temporal, spatial and functional considerations as well as a need to view the system at multiple levels of granularity.

Consider temporal relationships. The overall system as well as an individual subsystem can address multiple time domains. An airport's traffic control might be interested in the immediate runway operations, the projected number of operations that will occur during the next hour and the potential disruptions that might evolve if a forecasted storm occurs in the afternoon.

There are also spatial considerations. The continental United States is divided into several air control regions. Each of these regions is then subdivided into sectors both on a geographical and altitudinal basis. In addition, the airspace in the immediate vicinity of an airport is designated for dedicated control. Clearly, a management structure is required to coordinate the traffic within each region/sector/airport.

There are also functional relationships. Landings and takeoffs may be considered by different controllers at a busy airport. Certainly, runway operations will be segregated from ground operations. Each spatial subsystem can also contain numerous physical/system elements. At an airport, system elements include passengers, planes, airlines and their personnel, general aviation providers, and the controllers. There can also be multiple linkages among the similar system elements across the spatially defined subsystems.

Because different system elements within a given subsystem must address different functions on different time domains, it follows that each subsystem will consider the system state at different levels of granularity. One controller might view the arrivals to a given airport as simply an anticipated arrival pattern. Another controller might consider each plane on an individual basis, but not know the individual passengers traveling on the flight. However, the airline must have a list of passengers in each flight.

Requirement 1: A modeling paradigm must intrinsically support all modes of decomposition in order to specify the constituent subsystems that comprise the overall system and the fundamental relationships among these subsystems. The goal is to achieve a system-of-systems perspective for viewing the entire system.

Next, the operational capabilities for each subsystem must be described. Each subsystem might include several system elements. Some of these elements are acted upon. For example, passengers must board the plane before it departs from the gate. The plane must also be serviced and inspected. Some of the elements will act upon the others. For example, one or more airline personnel assist with the boarding operation. The pilot performs the final inspection and commands all plane operations between the departure from a gate and the arrival at the next gate. In order to model any subsystem, one must be able to define what tasks or operations can be executed, what system elements are involved and the specific procedures for executing a given operation. Furthermore, executing a singular task may involve several distinct subsystems at different times. For example, as a given flight travels from Washington National/Reagan Airport to Chicago O'Hare, it crosses several air traffic control regions and sectors. Clearly, one must also specify how tasks/operations can be transferred from one subsystem to another or coordinated. A task execution at one subsystem might also cause future tasks to occur at other subsystems. For example, a takeoff at one airport necessitates that a future landing will hopefully occur at another airport.

Requirement 2: Systems evolve as they execute procedures or tasks. A modeling paradigm must address the task execution process within each subsystem and *the associated mechanisms for coordinating task execution among the subsystems.*

The goal is to expand the scope of model application in order to support the management of the operational system. Defining the management structure becomes a critical component of the design and task execution process. Obviously, the operational constraints arising from the included management structure must be described within the model.

2.1.2 Translation

The next step in the life cycle is to construct the model. In some instances, a physical model or prototype may be constructed. More often, simulation studies employ computational models. One might employ a commercial simulation package to create the computational model in such cases. However, few available packages can support the modeling requirements discussed above. Alternatively, one might employ a general programming language as C, C++ or Java. Simulation models are often object-oriented, and it may be beneficial to adopt an object-oriented programming language. However, such an adoption is not essential.

While generating the model, one must also consider how the model will be executed. Today, two basic execution modes are employed: execute the model on a single processor or distribute the model across several processors. The decision to employ a distributed computational environment is often made after the model has been specified. Unfortunately, such an approach ignores the possibility that different modeling paradigms might provide models that are easier to distribute. Moreover, the conventional (single-threaded) processing and distributed (multi-threaded) processing are not the only two options.

Before seeking other execution modes, let us reconsider the model from a response perspective. The computational model itself can be viewed as a description of all possible responses that could occur when it is executed. For stochastic systems, it is impossible to delineate every possible state trajectory. Nevertheless, the model might permit one to characterize summary features of the contained responses. One typically employs the computational model to sample a collection of state trajectories within an experiment in order to predict statistically the system's performance for a particular assignment of values to the included design variables. One can also consider a singular trial. This latter execution mode often arises in training situations, where the trainee interacts with the system as its simulated state trajectory evolves and takes action that influence the system's future response.

The above execution modes are associated with off-line applications. With respect to the latter two cases, either single-threaded or multi-threaded (distributed) computational methods might be employed. Other execution modes also exist for on-line applications. In the second case described above, one projected the conditioned response of the system derived from a given assignment of design variables. In an on-line simulation, one might project the conditioned near-term response, given the current system state and the selected control policy for managing the considered subsystem. Given that each individual component subsystem must be managed, each subsystem requires its own subsystem model in order to project its future performance.

That is, any subsystem might perform a dedicated on-line simulation for its near-term response as it continues to operate in real-time. Given that different systems consider different time horizons, the dedicated on-line simulations would be customized to project the response for an appropriate time-period. In addition, a given subsystem might perform more than a single on-line simulation in order coordinate its response with the other subsystems with which it interacts. In particular, the outputs from the detailed on-line simulations of one or more subsystems could statistically characterize the initial state from which another subsystem performs another less detailed on-line simulation over an extended horizon. It is critical that one distinguishes the latter situation from the typical distributed simulation scenario. A conventional distributed simulation addresses a singular experiment with a given model. On-line applications require multiple experiments to be conducted concurrently across a collection of subsystems or within a given subsystem. Moreover, the interaction among subsystems may require an on-line simulation by one subsystem to be concurrently coordinated with another on-line simulation within a different subsystem.

Requirement 3: The simulation paradigm must support, if not facilitate, the various execution modes that can occur in both off-line and on-line applications. In particular, the paradigm should employ intuitive state definitions that will permit the simulation trial to be easily initialized to a measured or projected system state and simulated responses to be shared among interacting subsystems.

2.1.3 Verification

Verification represents a feedback mechanism that insures that model specifications have been faithfully incorporated within the computational model. Conventional simulation tools often force the modeler to modify one or more specifications in order to allow the model to be described with the tool's included objects. Some modeling specifications are nearly impossible to achieve within a given modeling paradigm. For example, stochastic queuing networks generate system responses as a collective set of local responses occurring at the included nodes. It is difficult to model and assess the performance constraints associated with a proposed control architecture using this paradigm.

2.1.4 Design

Several processes can be initiated after the model is created. Using off-line analysis, the designer can explore alternative assignments for the included design variables in order to enhance the system's expected performance. Given that the model should also consider the control architecture, the designer might also explore different controller specifications for managing individual subsystems and the interactions among the subsystems.

2.1.4 Training

Most complex systems also require one or more humans to interact with various subsystems while the system operates. In this regard, the control structure must first provide accurate information in order to assist the operator in selecting and implementing an appropriate course of action. Training often involves the trainee interacting with a singular simulated trajectory as it evolves in real time. It may be difficult to conduct comprehensive training sessions where all operators concurrently participate because a complex system can employ many operators. Therefore, the overall model should provide submodels that can be effectively employed to train an individual operator in the management of a particular subsystem as it would evolve while interacting with the other subsystems, even though their operators are not present.

The review process that follows a training exercise is also critical. Here, an instructor might query the trainee regarding a particular course of action that s/he selected at a given point during a training session. Ideally, the instructor would desire to return the simulation exercise to state where the trainee elected a questionable course of action and assess the consequences that might have evolved if an alternative course of action was adopted. In order to have this capability, the modeling paradigm should provide a temporal state representation that can be easily stored and replayed as desired. Any stored state should also provide an initial point from which a subsequent training exercise can be initiated and stored as another trajectory. This may appear to be unrealistic request, but remember that one can now store feature length movies on a single DVD. The proposed replay mechanisms would also assist the verification process.

2.1.5 Validation

The above steps (Conceptualization, Translation, Design, Training and Verification) only address off-line analyses preceding the implementation of the system. That is, these steps can occur even if the physical system does not exist. Presumably, the desired outcome of the design process is the construction and operation of the designed system.

Recall that the verification process sought to insure that model specifications had been faithfully addressed within the computational model. Verification cannot check the validity of the model specifications defined during the conceptualization. The validity of these specifications can only be checked by comparing the simulated performance projections against the actual system performance. The goal of validation is then to refine the model so that it correctly replicates the system behavior. Generally, validation is also addressed as an off-line procedure.

The need for model validation is sometimes questioned after the design system has been implemented. However, if one seeks to improve the system's design further, then one should first improve the model's accuracy. Thus, with a more accurate model, one can initiate the next design cycle for improving the system. Such improvements might be directed solely toward a more efficient execution of the current tasks the system can address. The redesign process might also seek to expand the capabilities of the existing system in order to permit it to perform additional tasks. The redesign process still represents an off-line analysis of a proposed system that currently does not exist even though this redesign process can be addressed while the actual system operates .

Uses of the simulation model within the on-line operation of the existing system will now be discussed. Before addressing on-line applications, however, one first must distinguish the real world from the virtual world. The operation of the real-world system provides a real response that can be observed and measured. The simulation exercise creates a virtual response projecting what could happen. The validation process contrasts that virtual response against the actual response for a given set of conditions.

2.2 On-Line Considerations

On-line applications necessitate an immediate interaction between the real and virtual worlds. Davis (1998) discusses the on-line simulation process where the simulation trial is initialized to the current state of the system and its future response from that state is then projected under a specified set of operating conditions. He continues to discuss how one might employ on-line simulation analyses to compare alternative strategies/courses of action for execution given the current state. His discussed approach is an elaboration of prior applications of simulation to scheduling tasks where the future performance is projected and compared against the actual performance. The deviations between the predicted (virtual) and real-world response are monitored under these prior scheduling applications. Whenever these deviations become significantly large, the simulated virtual projection is then updated (i.e. the model is re-simulated). Davis (1998), on the other hand, advocated that the on-line simulations occur constantly as the system evolves. That is, one should not wait for the deviations to grow in order to justify further simulation.

Unfortunately, none of these past approaches adequately support the on-line management of a complex system. Prior approaches have focused upon the future evolution of a given subsystem. Complex systems are comprised of a set of subsystems that must be coordinated. Addressing a single subsystem's problem independently of the other subsystems with which it interacts is insufficient for the management of the overall system. In many cases, it is actually counterproductive. *One must consider the coordinated response of the entire set of included subsystems as they interact under the coordination of the included control architecture*.

In order to define the requirements for supporting the on-line system operation, one needs to return to the fundamental principles under which the system operates. The overall system evolves as it included subsystems perform tasks. In most cases, the execution of tasks is goal-oriented. That is, the system (with its subsystems) has a purpose or reason for changing its state. Consider the air transportation system. The flights do not occur simply to move planes from one location to another. Rather the flights transport customers (passengers) or cargo between locations. Similarly, a manufacturing system makes product to sell.

Let us assume that the subsystems perform their tasks upon other objects or entities. Because real-world systems are being considered, let us assume that these objects are real and reside in the physical world. On the other hand, most planning and coordination occurs in the virtual world where one seeks to assess the consequences of a course of action before it is implemented.

If one assumes that subsystems exist to execute tasks, then their tasks must be assigned in a manner that is consistent with the overall goal for the system. Let us assume that every subsystem can interact with a set of other subsystems from which it receives tasks for execution. After receiving an assigned task, a given subsystem may further decompose the assigned task into subtasks and then seek the assistance of other subsystems in executing the generated subtasks. Eventually, this proposed task (re)assignment process provides a collection of subtasks that can be immediately executed upon a real object. When such a situation occurs, no further task decomposition is needed. Rather, the assigned task is executed upon the real object by a real-world process.

Clearly, one must be able to model the state evolution of the real-world objects as they are processed in the real world. This evolution is necessarily constrained by the processing plan that defines which tasks will be performed upon the object. In Figure 2(a), the process plan is graphically depicted. At the left there is a circle representing the real-world object that is to be processed. A sequence of arrows is included to the right of the object, each representing the subtasks that are to be performed upon the object. Each indicated task arrow include subtasks that can be executed at a single given process. Therefore, in order for the considered object to progress along the processing plan from one task arrow to the next, the object must physically move from one process to another.

The goal in processing the object is to consume the process plan. As each processing task is finished, the state of the considered object changes. In a sense, the remaining processing steps provide a manner for one to predict how the object will evolve as it is processed by the system. Hence, the object in Figure 2(a) represents the original object before any processing is initiated. As each task is executed, the circle advances to right. The tasks to the right of the object represent the remaining processing plan. The executed tasks are horizontally flipped and placed to the immediate left of the object as they are executed. The sequence of left-facing arrows to the left of the object represents its prior processing history. Hence, the object in Figure 2(b) denotes the object at its current state after the first two tasks have been executed. If the object moves forward along the remaining process, its future state changes can be projected. On the other hand, if one moves backward along the object's processing history, one can presumably backtrack the executed tasks in order to return to the original object shown in Figure 2(a). In Figure 2(c), the case where all of the tasks have been executed is represented.

Figure 2 depicts the case where the process plan is defined before the object enters the system. Other situations also exist including process plans with alternative paths, process plans with loops to allow rework, process plans that evolve at the time of execution. Unfortunately, there is insufficient space to discuss these other situations.

Given this assumed structure for a processing plan, the modeling of an individual subsystem can be addressed. At any given time, the subsystem has a set of assigned tasks to be executed by elements within its control domain. The subsystem receives these tasks from other subsystems, called its Assignors. In executing an assigned task, the subsystem usually decomposes the task into subtasks (using prescribed procedures) and then reassigns the generated subtasks to other subsystems, which are termed its Acceptors. The proposed relationship is depicted in Figure 3. In the depicted general case, a subsystem executes tasks by reassigning them as subtasks. However, if the subsystem is a real-world process, it physically can perform a physical task upon a real object. Moreover, as the complex system operates in real time, its physical state evolves through the task executions occurring at the real-world processes. Processes can have no Acceptors because

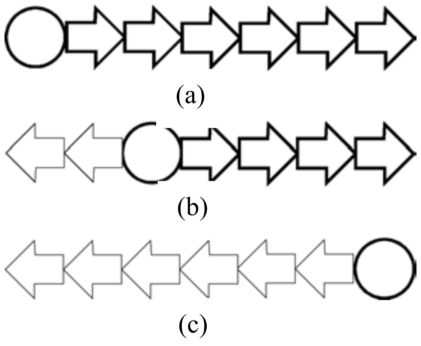

Figure 2: A Processing Plan and its Execution

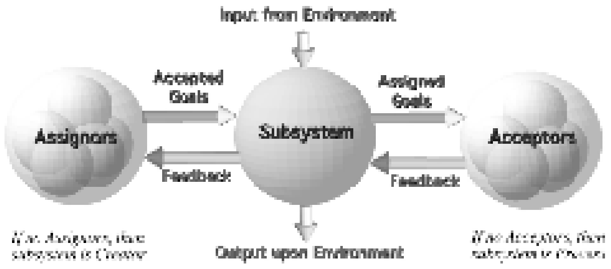

Figure 3: Basic Relations among Subsystems

the act within the real world in real time. They must execute the tasks that they are assigned.

On the other hand, the more general subsystem relies upon its Acceptors to execute its assigned tasks. Consequently, it physically cannot act in the real world or in real time. Rather, it considers a virtual world of what could happen as its assigned tasks are executed by its Acceptors in the future. It must also rely upon its Acceptors to describe *its* current state because they are responsible for executing its reassigned subtasks, which will cause *its* state to change. One can also assume that the given subsystem has other assigned tasks, which it has not yet decomposed or reassigned. The subsystem's controller must seek a plan for the reassigned execution of these remaining tasks. Using the feedback information from its Acceptors, the subsystem's controller then employs on-line simulation to project its statistical performance should it adopt a given reassignment strategy. Observe, however, that this on-line simulation is being initialized to the projected future state for its Acceptors as they execute their currently assigned tasks. Consequently, the subsystem's simulated trajectory resides entirely within the virtual world of the future. Its on-line simulated state trajectory cannot consider the current time because only real-world processes can act in real time.

Let us now view each subsystem as a pipeline that receives tasks from its Assignors and transports them as subtasks to its Assignors. A subsystem's primary on-line simulation projects the flow for the tasks that are currently in its pipeline or have been reassigned to its Assignors. If new tasks are not accepted, then the subsystem's pipeline becomes empty. Therefore, a subsystem must constantly seek new tasks from its Assignors as inputs to its pipeline. Each Assignor also represents another pipeline containing its accepted tasks that are to be reassigned as subtasks to its set of Acceptors, to which the considered subsystem belongs. Each Assignor is also performing its own on-line simulation that is initializes to projected future state of its Acceptors (that includes the considered subsystem) as they execute their currently assigned task.

Usually, the granularity of an Assignor's on-line simulation is less detailed than that of any Acceptor, including the considered subsystem. Hence, when the considered subsystem interacts with an Assignor, the Assignor expects the considered subsystem to provide a more detailed assessment of the probable outcome should a given task be assigned to the considered subsystem. In this regard, the considered subsystem interacting with the particular Assignor performs another on-line simulation to assess the consequences of making a proposed assignment before the assignment actually occurs.

The considered subsystem also acts as an Assignor to its Acceptors' interface between the considered subsystem and its Acceptors. A recursive nature results where each subsystem performs an internal on-line simulation pertaining to the execution of the tasks currently in its pipeline and two shared on-line simulations: one in conjunction with its Acceptors and the other in conjunction with its Assignors. Moreover, these three on-line simulations can occur *concurrently* within any given subsystem.

The subsystem employs the same model in implementing all three on-line simulations. In fact, the on-line simulations differ only in the set of considered tasks. One on-line simulation projects the outcome of the current assigned and reassigned tasks. Another projects the outcome if additional tasks are accepted. The latter projects the outcome if additional subtasks are reassigned. Given the multiple uses, the subsystem's model must explain the task acceptance, decomposition and reassignment processes. In short, the subsystem's model must depict the behavior of its controller as it moves tasks through the subsystem's pipeline. Ideally, the model should be the controller for the subsystem. If this situation could occur, then the collection of subsystem models immediately comprises the set of controllers that forms the control architecture; and more importantly, this feature significantly simplifies the validation process because the set of system models and controllers are one and the same.

The remaining components within the overall system model are the process models. These processes exist in the real world and can be modeled using traditional engineering approaches. For example, one can physically model the dynamics of a robot, a milling machine, an aircraft and so forth. Each of these physical processes has its dedicated controller and instruction set for interacting with process. In fact, many physical processes are now shipped with dedicated software emulators that permit one to verify that a generated set of instructions provides a desired response or outcome.

Knowing what subsystem models are needed, the next step is to define the system's architecture and its concept of operation governing the interactions among the included subsystems. Let us begin with the included collection of real-world processes. The task execution of these processes causes the real-time state evolution of the overall system. However, these processes must be connected to the control architecture in order to receive tasks. Therefore, each process must have at least one Assignor

Most subsystems, including all processes, will have Assignors. The Assignors for one or more subsystems may reside outside the environment of the considered system because one must arbitrarily define the boundary of the considered system. These external Assignors provide the exciting force upon the addressed system. Hence, external Assignors must exist and are essential.

Now consider any subsystem within the system that is not a process. Through its Acceptors, it can indirectly assign tasks to a subset of the included processes. One could derive more general connectivity principles, but let us simply assume that an exciting task assignment from the environment will be sequentially decomposed into a sequence of subtasks that ultimately will be executed at the processes. Moreover, if the decomposition results in any subtask that cannot be executed with the included processes, then the task cannot be executed by the overall system.

Obviously, the next step is to define what tasks can be executed by the considered system. Because the tasks that determine the system's physical evolution in real time are executed upon other physical objects at the system's included processes, one must define first the set of objects that will be acted upon and what tasks will be performed. Because each of these tasks occurs at a physical process, one can write the instruction set for the task in the dedicated instruction language for the appropriate process. Also, observe that the execution of any task upon an object at a process may require additional objects or resources. In a manufacturing setting, one might need an operator to install the object into the machine and then monitor the machine's operation. Often the machined object will be placed into a fixture (another resource) prior to this installation. A given machining operation may require a tool, yet another resource.

In some cases, the output from the processing of one object might become the input for the processing of another object. For example, when one assembles an automobile, numerous components are sequentially attached to the assembled object. Each of these components must either be manufactured or assembled from other components.

Given the set of tasks that will be assigned to the system from its environment, the essential task decomposition schemes must be defined in order to allow each external exciting task to be executed using predefined subtasks that can be executed at the included processes. An incoming task is then treated as an object (or order) to which a sequence of remaining processing steps is attached. Assuming that a given controller (other than a process controller) is now managing a particular order, the controller determines the next process that the order should visit and selects an appropriate Acceptor through which that process can be accessed. The selected Acceptor is then asked to *Accept* the order and execute one or more remaining process steps. This iterative assignment process continues until the Acceptor is the required next process.

When an Acceptor accepts a task, it must take physical control of the object upon which the task is to be performed. In the real world, this implies additional supporting/enabling tasks may be required for the transfer of physical control to occur. In a manufacturing setting, a material handler might retrieve the involved object from one workstation and deliver it to another. Observe, however, that the material handler is also a physical process that can change the physical location of the object.

The basic conversation among the controllers is very simple. The Assignor first requests an Acceptor to *Accept* the object and then to *Execute* a specified task upon the accepted object. After the Acceptor accepts the object with its assigned task, it provides feedback information pertaining to when or if the task has been completed. This feedback information is generated when the Acceptor performs an on-line simulation pertaining to the planned execution of its assigned tasks. There is one concern, however. Ideally, the Assignor desires to know if the Acceptor can execute that task before the actual assignment occurs. Therefore, a provision should be included to allow the Assignor to *Pre-qualify* the Acceptor before the Assignment occurs. In the performing the *Pre-qualification*, the Acceptor addresses the following concerns:

- Insures that the current state of the process will permit it to execute the task
- Determines what additional resources must be assimilated in order for the task to be executed and
- Projects when the task might be finished, assuming that the needed resources will be delivered by a specified time.

This *Pre-qualification* requires the Acceptor to look ahead in order to determine how its managed subsystems' state might evolve should the task be assigned. This look ahead is addressed through its on-line simulations and its subsequent requests of its own Acceptors to Pre-qualify any subtask that could evolve from the decomposition of the proposed task assignment.

In general, the Assignor conducts a sequential communication with its appropriate Acceptor with the instructions to *Pre-qualify*, to *Accept* and to *Execute* each subtask derived from the decomposition of an assigned task. The feedback information from the Acceptor's *Pre-qualification* determines what might happen if the task is assigned and when the task will be completed after the task has been assigned. Since this feedback information projects a future outcome, it represents planning within the virtual world. The principal exception involves the process because their controllers typically do not have an installed planning capability. Rather, a process simply *Accepts* the object with the assigned task (in its instructional language) and then *Executes* the task. The process then notifies the Assignor when the task is completed.

2.2.1 Implementing Model as a Distributed Controller

This paper previously advocated that the model for a managing subsystem and the code that implements its controller should be the same. One must now distinguish the execution of the collection of management subsystems during the typical simulation of the overall system's response versus the on-line operation of the system where the same models manage the systems response. The on-line management application will be considered first. Managing subsystems must behave both as an Acceptor and an Assignor while functioning as pipelines for the execution of tasks. Observe, however, that the interaction among managing subsystems does not change the physical state of any physical entity. Rather, it changes which entities a managing subsystem controls and the assigned tasks it has agreed to execute.

The managing subsystems interact by sending messages as described above. Each managing subsystem knows its Assignors and Acceptors. Hence, it knows from which subsystems it can receive messages, the type of messages it will receive and how it should respond. Whenever a managing subsystem receives a message, it triggers the execution of its model that serves as its controller. The message processing is a singular-threaded execution that causes an internal change of the controller's state within its virtual world. The message process may also cause messages to be sent to other controllers either now or at some future time. When the controllers are managing the real system, they will likely transfer their messages via a communication network. However, distinct controllers need not be situated upon distinct computers nor do all the controllers need to be onsite. It is only necessary that one controller can send messages to another controller in an expeditious manner.

Physical processes must be managed because the system is operational. Each of these processes usually has its own dedicated controller implemented upon a microcomputer processor or a programmable logic controller. These process controllers must communicate with their managing controllers, Assignors. Often, serial communication means such as RS-232 are employed, but it is sometimes possible to communicate with these specialized process controllers over a conventional Ethernet LAN. These process controllers must receive their tasks in their dedicated instructional language from their assigned Assignor. Often, the specialized process controller will be attached directly to the computer where its Assignor resides. This is particularly true when serial communication is employed.

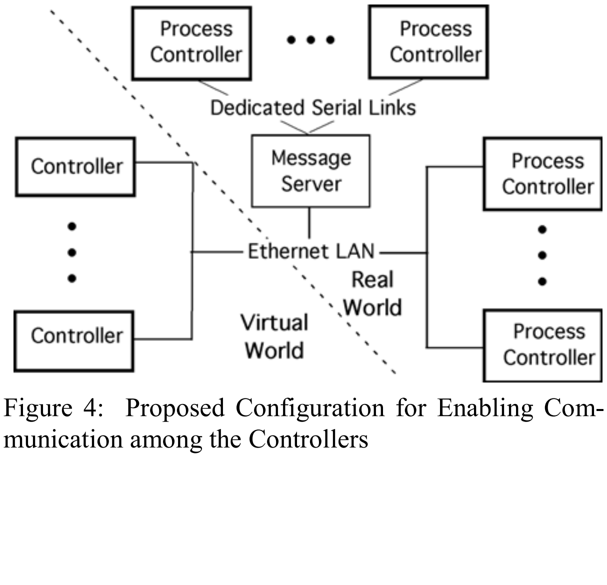

Figure 4: Proposed Configuration for Enabling Communication among the Controllers

However, experience with real-world systems has demonstrated that the configuration depicted in Figure 4 is more versatile. In this scenario, a message server is included that receives all transmitted messages and then routes them to their appropriate addresses where the recipient controller's code is executed. The address for a managing controller is typically an IP node. The process controllers are usually attached to the message server if serial communication is employed or assigned an IP node if they can receive their messages from an Ethernet LAN. Observe that Figure 4 includes a partition between the virtual world addressed by the managing controllers and the real world within which the process controllers act.

Let us focus on the message server's operation. Incoming messages are received and routed to their destination with minimal delay. The recipient controller processes the message and then routes its responses to other controllers through the message server at the appropriate time.

The proposed message server's operation further provides the basis for simulating the entire system using the same set of subsystem models as shown in Figure 5. In implementing a simulation of the entire system, each real process is replaced with the computer code (often supplied by vendor) that emulates its processing. These emulators should employ the same instruction set as the actual process. The subsystem's models can be located upon a single computer because every subsystem model executes as a singular-threaded computation. A software implementation of the message server models the communication among the controllers and controls the simulation as follows:

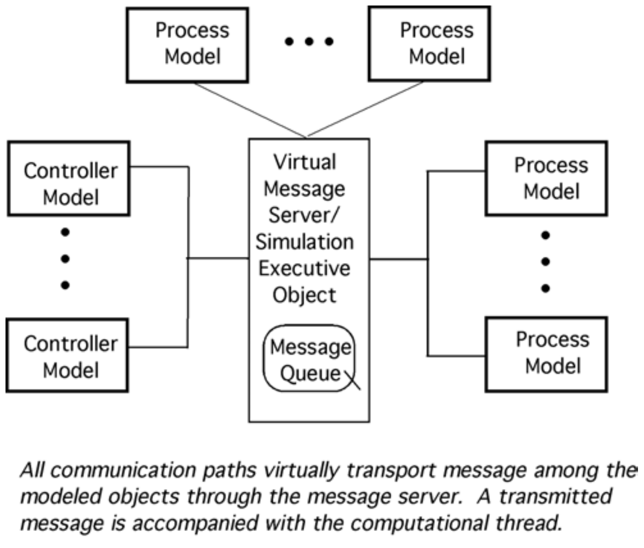
Process Model
Process Model
Controller Model
Virtual Message Server/ Simulation Executive Object
Process Model
Message Queue
Controller Model
Process Model
All communication paths virtually transport message among the modeled objects through the message server. A transmitted message is accompanied with the computational thread.

Figure 5: Configuration for Simulating System where Controllers Interact via a Virtual Message Server

The queued messages are stored in chronological order based upon their desired delivery time. After a message is processed, the simulation time is advanced to desired delivery time for the next message in the queue. The message with the computational thread is then forwarded to the recipient controller whose included model processes the message and generates the appropriate response(s) to specified recipient(s) and at a specified delivery time(s). These messages, along with the computational thread, are returned to the message server, which then inserts the new messages into its chronological ordered message queue. The next message is then removed from the queue and the simulation time is advanced to its delivery time. The message and the computational thread are then passed to the next designated recipient controller.

2.2.2 Implementing Model as a Distributed Simulation

The proposed operation is very similar to the conventional processing of a scheduled event queue in most current simulations. Thus, it should also be possible to implement the simulation in a multi-threaded distributed environment using available distributed simulation techniques because of this similarity. Fujimoto (1999) provides an excellent discussion of the current techniques.

In order to demonstrate how a modeling architecture might enhance one's ability to perform distributed simulations, let us consider the operation of the multi-level system pictured in Figure 6. The overall system is comprised of seven subsystems. Subsystem 1 serves as the Assignor to the accepting subsystems 2 and 3. Subsystem 2 manages or serves as the Assignor to processes 4 through 7. One can assume that subsystem 3 also manages processes. However, its processes have not been included in order to simplify what is already a complex figure.

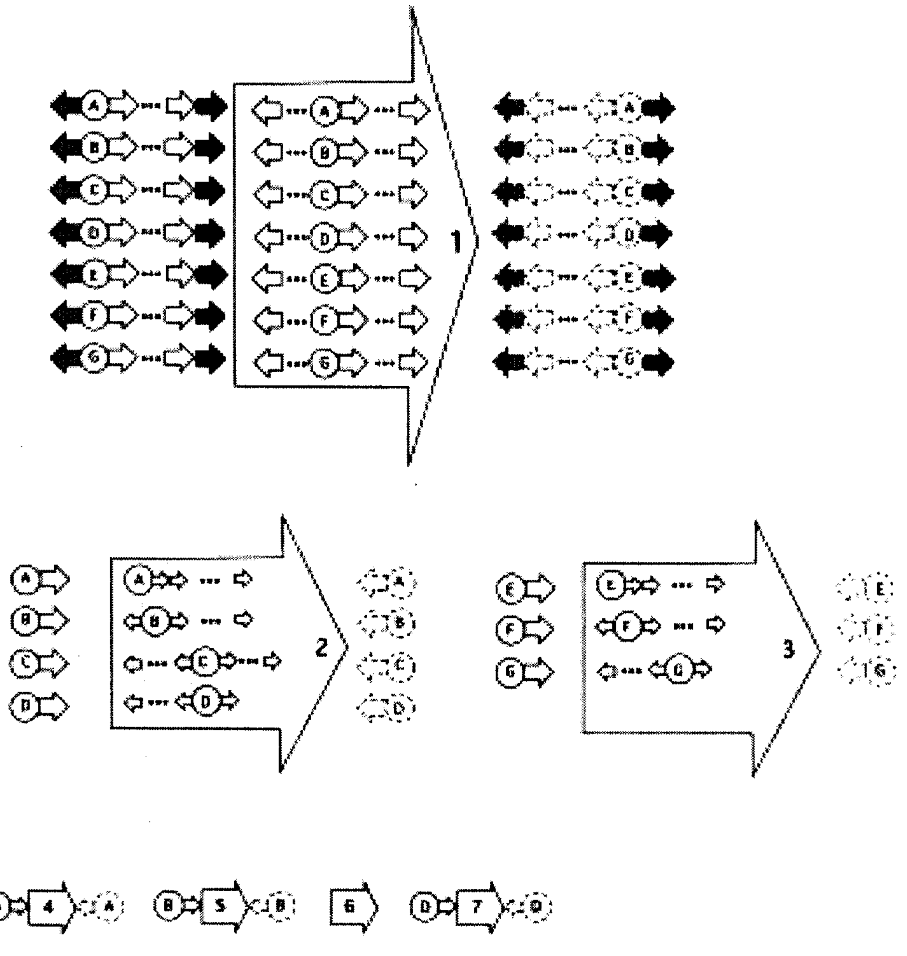

Figure 6: Proposed Hierarchical Task Decomposition as a Foundation for Distributed Simulation

Currently, objects A through G are being processed by the system. These objects are depicted o the left of subsystem 1, as they were initially assigned. The included white arrows represent the sequence of tasks, with supported tasks included, that are to be executed by the processes under the control of subsystems 2 and 3. The shaded arrow before the circle represents the processing history that occurred before the object arrived at subsystem 1. Hence, the circle for the object at input represents its state at the time it arrived at subsystem 1. To the right of the large arrow that represents subsystem 1 is the desired goal state for each object where all the assigned tasks have been executed. Note in this case, all the original task arrows have been flipped to represent their completion and the object state is represented by a circle to the right of the completed tasks. The rightmost shaded arrow for any object represents the tasks that will remain after the currently assigned tasks are executed.

Within the large arrow that represents subsystem 1, the current state of each of the processed objects is represented. This case assumes that a portion of the assigned tasks have been executed upon each object and the arrow immediately to the right of the circle represents the current task that is being executed. In order to simplify the discussion, let us assume the objects have been labeled such that the next task for objects A through D will involve processes under the control of subsystem 2 while objects E through G have a next task that requires processes managed by subsystem 3. Let us also assume that the next task has been assigned for objects A through D to subsystem 2, which is indicated by drawing a dashed rectangle around their next tasks and projecting that rectangle onto the input of subsystem 2. Similarly, the next tasks for objects E through G are indicated as the inputs to subsystem 3.

Now let us focus upon subsystem 2. The desired output for subsystem 2 requires the completion of the next assigned task as indicated to the right of the subsystem 2's representative arrow. Within the arrow representing subsystem 2, the current state of subsystem 2 is indicated as it processes its assigned tasks. Observe, however, that the assigned task for each object has been further decomposed into a set of subtasks that will be executed at processes 4 through 7 that it manages. For object A, the first subtask is currently being addressed by process 4. For object B, the first subtask has been completed, and the second subtask is being addressed by process 5. Several subtasks have already been executed upon object C and the next subtask is waiting to be assigned. One can assume that this next subtask must not require process 6, which is currently idle. The last subtask upon object D is being addressed by process 7. Again, these assignments are depicted by the dashed rectangles that are projected onto the input side of the appropriate process. Observe also that a similar reassignment of subtasks for objects E through G should have occurred at the processes managed by subsystem 3, which have not been included in the figure.

Now let us look at the processes. Each process is single threaded and can only process a single object at a time. On the other hand, subsystems 1 through 3 are multi-threaded and can manage several objects concurrently. Remember also only the processes can act in real time and this is where the physical changes to processed objects occur. Given that one knows when each process was assigned its current processing task, one can sample the future time when the current task will be completed. At this point, the physical state of the involved object will cease to change until its next subtask is assigned to the appropriate process. With respect to objects A, B and D, subsystem 2 has two options. Allow the assigned processing subtask to be completed or interrupt (preempt) a current processing task. Let us postpone the preempting option for the moment. In this case, subsystem 2 knows when each of the ongoing processes will be completed because each process samples its completion time and then posts it with subsystem 2.

Subsystem 2 is primarily interested in two future events. The first event is when a new processing thread can be initiated upon another object. The second event is when all the subtasks for a given object have been completed, implying that it has completed the next primary task upon the given object as assigned by subsystem 1. Let us consider some special cases:

> Case 1: Object C is waiting for process 4. On the other hand, process 4 is scheduled to complete its current subtask upon A before the other processes complete their current tasks. Therefore, as soon as process 4 completes its current task, it can immediately initiate the next processing subtask on C.
>
> Case 2: Objects A, B and C will all need process 7 to address their next subtask. Let us assume that processes 4 and 5 will complete their current subtasks before process 7. Hence, the next time a new processing subtask can be initiated is when process 7 finishes its current effort upon object D.
>
> Case 3: After process 7 finishes its current task upon object D, subsystem 2 will have completed the entire sequence of subtasks for executing the next primary task for object D. Therefore, when process 7 finishes, subsystem 2 can notify subsystem 1 that it has completed its assigned task upon object D.

Subsystem 2 can compute the minimum time at which it can initiate a new processing subtask at one of its managed processes or the time it will finished an assigned task on one of the assigned objects because it knows the sampled completion time for each process. Let us refer to the minimum time that either of these situations will occur as subsystem 2's next event time. Subsystem 2 computes its next event time and posts it with its Assignor or subsystem 1. If the next event corresponds to the completion of an assigned primary task upon a given object, the posted next event time is accompanied with the message that subsystem 2 uses to notify subsystem 1 that it has completed an assigned task upon the given object. In a similar fashion, subsystem 3 also computes its next event time.

Let us now consider the case where the next event time for both subsystem 2 and 3 corresponds to a simple initiation of a next processing subtask. No message will be sent to subsystem 1 in this case. Subsystem 1 responds by pulling the next event time from either subsystem 2 or 3, depending upon which subsystem has posted the smallest next event time. At this point, the subsystem whose posted next event time was pulled is authorized to perform any system updates up to the posted event time. Observe that this could necessitate several processes would update their state also. The updating subsystem (2 or 3) would tell the processes to update their state to the completion of their current task by issuing the command to return the involved object at its sampled completion time. After all the updating has occurred at the involved processes, the updating subsystem (2 or 3) computes its next event time and posts it with subsystem 1.

Now let us investigate the case where the next event to be considered by subsystem 1 corresponds to the completion of a primary subtask that it assigned either to subsystem 2 or 3. In this case, an appropriate completion message will be transmitted with the posted next event time. Subsystem 1 will respond to the appropriate subsystem with a return request at the posted next event time. At this point, the recipient subsystem can update its state to that posted event time. Again, several other processes might need to update their state to reflect the completion of any processing subtask that occurred before the posted event time.

Subsystem 1 also has the option to assign a new task to either subsystem 2 or 3. For example, after subsystem 2 completes the current task upon object D at process 7, subsystem 1 can assign object D's next primary task either to subsystem 2 or 3. Also observe that subsystem 1 will eventually complete its assigned tasks upon objects A through G. If no new tasks are assigned to subsystem 1, then it and its managed subsystems become idle. Therefore, one can assume that subsystem 1 has an Assignor, which is not illustrated in Figure 6.

That same assignor may manage several subsystems that have similar capabilities to the illustrated subsystem 1. If this is the case, then subsystem 1 must post its next event time, which represents the minimum next event time for subsystems 2 and 3, with its Assignor. Additionally, if this next event represents the completion of the assigned sequence of tasks upon a given object, then it will submit the appropriate completion message to its Assignor. In this regard, subsystem 1 interacts with its Assignor in a manner that is similar to the way that systems 2 or 3 interact with subsystem 1. In addition, subsystems 2 and 3 assume the role of subsystem 1's processes.

Most systems are driven by external inputs. Moreover, at least one subsystem within the modeled system must have no Assignors. In traditional simulation approaches, one excites the system by creating entities. Whenever a creation occurs, the external creator posts its next creation time. Therefore, any subsystem that interacts with a creator has both a minimum next event time among its Acceptors and a next creation time. The involved subsystem then selects the minimum between these two times. If the minimum time corresponds to the next event time, then the involved subsystem pulls the posted next event time for the appropriate Acceptor, as discussed above. If the minimum time is the next creation time, then it accepts the created object and reassigns it to the appropriate Acceptor.

Any Acceptor can update its simulated time either to the time associated with a pulled next event or the time attached to an incoming message from its Assignor. Whenever this Acceptor updates its state, it triggers a chain reaction among its Acceptors and their Acceptors. This recursive updating continues until processes are reached. When a single-threaded process updates it state after completing a subtask, it remains idle until another subtask is assigned. When the process receives its next assignment, it samples its completion time, which represents the process's next event time. Whenever any Acceptor posts its next event time, its Assignor determines if a next event time has been posted for each of its active Acceptors. If so, then the Assignor posts its next event time. This process continues until a top-level Assignor is reached. A top-level Assignor then determines whether a creation or next event will occur as described above and the entire process repeats.

The following additional observations can be made:

- First, an Acceptor can have more than one Assignor. In this case, it posts its next event time with any Assignor from which it has an active assignment.
- Second, preempting causes no problem because the message to preempt must come as a command from the original Assignor. But that same Assignor controls which Acceptor can next update its state. For consistency, however, no Assignor should be permitted to issue a command with an associated time that is prior to the current simulation time.
- In order to allow more parallel processing to occur, if one Acceptor to a given Assignor is permitted to update to a given simulated time, then all of the Acceptors to a given Assignor should be permitted to update to the same time. Observe, however, that the posted next event time will be changed only for the Acceptor that receives the

next assignment or has its next event time pulled. Using this fact, the proposed paradigm should be able to employ numerous processors without needing to worry about time-warping or other synchronization procedures.

2.2.3 Autovalidation and Maintenance

A major concern in employment of a simulation model within an on-line control application is providing a model that reflects the true system behavior. Most complex systems are time variant and require continuous update of key modeling parameters such as task durations and reliabilities. In addition, one also desires to provide new tasks for the system while removing former tasks. One might also desire to change the organization of the control architecture as well as the manner in which individual controllers select their next task for execution.

Employing a modeling paradigm based upon controller interactions, significantly simplifies model maintenance. In particular, one can demonstrate that it is possible to provide on-line environments that allows the modeler to update the model easily and, in many cases, autonomously. The same paradigm also permits one to easily define new tasks for the systems or to remove tasks that are no longer essential.

Unfortunately, space constraints will not permit a detailed consideration of these issues. These considerations are real and must be addressed by future modeling paradigms.

3 CONCLUSIONS

This paper has attempted to define the specifications for new simulation modeling and execution paradigms for the design and management of complex systems. The goal was not to provide a solution for these specifications. Rather it was to assess what is needed.

The preliminary list specifications was compiled during many years of experience with complex systems. It is expected that more specifications will evolve as one attempts to provide the described capabilities for real-world systems. Because this list is preliminary, the intent was not to provide a solution, even though a solution is known. The paper’s intent is to generate discussion of what is needed rather than how best one might meet the essential requirements.

REFERENCES

Davis, W. J. 1998. On-line Simulation: The Need and the Evolving Research Requirements. In the *Simulation Handbook*, ed. J. Banks, 465-516. New York: John Wiley and Sons, Inc.

Fujimoto, Richard. 1999. *Parallel and Distributed Simulation Systems*. New York: John Wiley and Sons, Inc.,

AUTHOR BIOGRAPHY

WAYNE J. DAVIS is a professor of General Engineering at the University of Illinois at Urbana-Champaign. His research addresses the distributed intelligent control architectures for complex systems. To support this research, he has developed several new modeling paradigms and on-line simulation approaches.