Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

OPTIMIZING PRODUCTION WORK FLOW USING OPEMCSS

John R. Clymer

Applied Research Center for Systems Science
California State University Fullerton
Fullerton, CA 92834, U.S.A.

ABSTRACT

A graphical discrete event simulation library is proposed for system simulation that is based on interacting concurrent processes. This library works with EXTEND (Imagine That Inc), an inexpensive yet capable and easy to use simulation software package, and it is called Operational Evaluation Modeling for Context-Sensitive Systems (OpEMCSS). Context-Sensitive Systems (CSS) is a systems theory, based on finite state machines, that can assist a systems engineering manager, business operations manager, or manufacturing production manager in understanding, evaluating, and optimizing production work flow, represented as communicating concurrent (parallel) processes. CSS theory can be expressed using the OpEM graphical modeling language. Thus, OpEMCSS is a graphical simulation library that can model systems based on CSS theory and the OpEM language. A simple part production problem is discussed that is an example of applying the Classifier Event Action block, which is a rule-based classifier contained in OpEMCSS, to discover optimal rules to manage the workflow.

1 INTRODUCTION

In order to optimize production workflow, it is necessary that the concurrent processes, used to represent the operation of each workstation as it performs production tasks, share information. Through either central or distributed decision making, workstations are allocated to production tasks based on shared information in order to maintain balanced work flow and on-time completion of tasks. In complex systems where each workstation can perform many different production tasks, it can be difficult to discover good decision rules to allocate workstations to production tasks. An OpEMCSS simulation of the production process can model the required information sharing and can be used to discover good decision rules to manage production workflow.

OpEMCSS is a graphical Discrete Event Simulation (DES) library that works with EXTEND, a relatively inexpensive yet powerful software product of Imagine That Inc. <http:\\www.imaginethatinc.com>. OpEMCSS can be put into perspective by comparing it with other DES views such as queuing models, Petri nets, and functional flow models. A comprehensive review of alternative systems modeling views, including OpEMCSS, can be found in (Bahill 1998).

1.1 Queuing Models

In a queuing model, transactions flow through a network of queues and servers. If queuing processes operate without any interactions or communications that modify their behavior, other than transaction flow, they are called context-free processes in this paper. In contrast, if transaction flow in one queuing process depends on what is happening in another queuing process, they are called context-sensitive processes. For example, the EXTEND DES library is based on context-free queuing processes. To apply this library to model context-sensitive process interactions in surface ship combat, attribute sharing and process synchronization blocks had to be added to the library. The SLAM graphical simulation language (Pritsker 1986) also describes transactions flowing through a network of queues and servers.

1.2 Petri Nets

In a Petri net model, tokens flow from place to place as defined by transitions. The significance of tokens can be as varied as the state of a process or data flow. One of the early goals of Petri net developers was to be able to predict mathematically, model characteristics such as consistency and deadlock. Since then, the Petri net model has been expanded to model process timing, and what tokens can represent has also been expanded by allowing tokens to have "Color." With these additions, the Petri net modeling approach has lost the ability to predict model characteristics mathematically and has evolved from an analytical to a simulation based method. Petri nets are capable of expressing moderately context-sensitive process interactions.

1.3 Functional Flow Models

In a functional flow model, such as IDEF0 (Buede 1999), transactions flow in a network of functional blocks. Each block receives transactions (i.e., data, knowledge, energy, or material) from other blocks, transforms input transactions into output transactions which are then sent to other blocks. Functional flow models are usually developed from the top-down, forming a hierarchy of models. OpEMCSS supports hierarchical functional flow models by using EXTEND's hierarchical blocks. In context-free systems these functional transformations do not adapt. If these transformations do adapt based on what other functions are doing, then such a system model is context-sensitive. Complex Adaptive Systems (CAS) are context-sensitive systems that have an emergent behavior for the overall system that cannot be achieved by any proper subset of the system. In a production CAS the desired emergent behavior is balanced workflow and on time tasks.

1.4 Overview

The concept of a parallel process is discussed first because this concept is key to understanding Context-Sensitive Systems (CSS) theory and thus Complex Adaptive Systems (CAS). A Finite State Machine (FSM) modeling view, that is based on transition rules, is presented next to explain CSS theory and the parallel process concept. The FSM modeling view is presented because this model has a formal basis in computation theory (Yeh 1976). In the FSM modeling view, the form of the transition rules determines if a system is context-free or context-sensitive and, thus, provides a formal definition of these concepts. Different kinds of transitions are possible which allow modeling of queuing systems operation, process-resource contention issues, functional flow, and production system architecture features as they affect overall systems effectiveness, which for a production system are task latency, resource utilization, and system throughput. A FSM expressed using transition rules, although instructive, is not easy to understand or communicate to others. Therefore, a graphical description of FSM transitions is presented using the OpEM graphical language in the remainder of the paper.

OpEM graphical language, that is an expression of interrelated communicating FSMs and the parallel process concept, and OpEMCSS library blocks, that implement the OpEM graphical language, are proposed as a basis for complex system simulation, including production systems. A simulation of a part production process that is built using these blocks is discussed as an example of an OpEMCSS model.

2 PARALLEL PROCESS

Considering the three modeling views that are discussed above, the queuing theory model is probably the one most familiar to a business operations manager or manufacturing production manager. An OpEM parallel process model can represent queuing processes by including a wait until state that represents a transaction waiting in a queue. When the wait logic is true, the transaction goes to a workstation where the task is executed. Task execution is represented by a reaction time state that models a period of time.

For example, consider a simple production system that consists of three tasks: cut parts, mill parts, and drill parts. Assume that the sequence of these tasks in the production process is fixed as listed and that there are five workstations that can perform either mill or drill. Also, assume that a sixth workstation cuts the parts. Thus, when a part is cut it goes to the mill queue to wait for one of the workstations to perform the milling task. When milling is complete, the part goes to the drill queue to wait for a workstation to perform the drilling task. Three concurrent processes can describe this queuing system: (1) part cutting described by the "Cut Part" state, (2) part milling represented by the "Wait for Mill" and "Mill Part" states, and (3) part drilling described by the "Wait for Drill" and "Drill Part" states.

A parallel process is defined as the collection of all possible sequences of system states and events that represent process flow and interactions for a system or organization. Each system state consists of the discrete state of each process instance, where the number of process instances may vary, plus zero or more state variables. The discrete state portion of an example system state for the part production process, discussed above, is (Cut Part, Wait for Mill, Mill Part, Drill Part) that describes four concurrent process instances at a particular point in time. Thus, discrete states represent periods of time where either (1) a task is being performed by a resource(s) such as "Mill Part" or (2) a task is waiting for a specified logical condition to be satisfied before it can continue such as "Wait for Mill," discussed above. Also, included in the system state are zero or more state variables. State variables have values that identify conditions, other than what task is currently being performed, such as parameters that are used to control execution of the process activities. For example, for the part production parallel process there is a state variable R, that indicates how many workstations are currently available, that is used in the wait until state logic to decide when to allocate a workstation to a part task.

A Finite State Machine (FSM) is defined as a set of states and a set of transitions between these states. For example, DEVS (Zeigler 1999) uses a FSM model to describe the operation of each system component such that components can communicate with each other and can be formed into hierarchies. FSM models are often expressed

either in terms of state transition tables, sets of transition rules, or using state transition graphs. We will use transition rules in the following discussion because the form of these rules determines whether a system is context-free or context-sensitive (Yeh 1976). Each of the rules we use in this paper describes the operation of a subset of FSMs (one or more FSMs) where each FSM in the subset represents one sub-process. Such a rule can describe either one FSM or several FSMs operating together. This allows us to explicitly describe both context-free operation and context-sensitive interactions among concurrent sub-processes using our transition rules.

In the FSM modeling view discussed next, system operation is represented by a set of concurrent processes where each process instance is a described by a FSM. Thus, each process instance in the set requires a discrete state dimension and zero or more state variable dimensions, as discussed above. A process instance occurs each time that a sub-process diagram in an OpEM directed graph model is duplicated one or more times, as discussed in the next section. Dimensions for one or more process instances can form subsets of the system state vector ($D_1D_2...D_N$) that contains dimensions for all process instances currently in the system model. As in the production system example discussed above, the size of the system state vector can vary as process instances are created and destroyed.

For example, let ($D_iD_j...D_k$) dimensions be a subset of system state and event sequence $E_aE_b...E_n$, implementing a transition, indicate value changes in these dimensions. An example system transition is described by the rule:

$$(D_i\ D_j...D_k) \text{ -> } E_aE_b...E_n\ (D'_i\ D'_j...D'_k)$$

This rule indicates that a transition from subset dimensions ($D_iD_j...D_k$) to new values ($D'_iD'_j...D'_k$) occurs after event sequence $E_aE_b...E_n$ is executed. Each event E_i is associated with an individual sub-process process instance, and it may cause a change in one or more dimensions of the subset when it is executed. The dimensions of each sub-process process instance are grouped together, and one or more sub-process instances can be represented in a rule. A transition (event sequence) occurs and persists for zero time; a state may persist for a non-zero time. The transition rule notation is commonly used in computation theory (Yeh 1976) to define formal languages or finite state machine behavior as strings of symbols.

Taking the operational view, state transitions provide a parallel process model of operation for the overall system. The OpEM graphical language and its OpEMCSS implementation describe the behavior of parallel processes, as they interact in time, and the dependence of each process instance execution on subsystem attributes and data flow.

Taking either the functional flow or architectural view, subsystem behavior is described by a collection of one or more processes. Here, a state dimension containing a state variable can represent transactions. Flow of transactions from one subsystem to another in the network can be described by transition rules.

Context-Sensitive Systems (CSS) are systems with at least one transition rule based on a context (i.e., the left side of a rule) involving two or more process instances. For example, a system becomes context sensitive when the subset dimensions ($D_iD_j...D_k$) forming the left side of a transition rule represent two or more process instances. This definition permits many context-sensitive interactions, which may include:

- A transition for a single sequential process that depends on state dimensions of two or more process instances and results in a change in discrete state. A process is permitted to adapt its behavior based on knowledge or data obtained from other processes. The OpEMCSS alternate action block models this interaction.
- A synchronized transition for two or more process instances that depends on discrete state dimensions of each process instance and results in a change of discrete state for each process instance. Processes can coordinate begin and end of tasks using the OpEMCSS split action and assemble event blocks to model this interaction. These blocks allow one subprocess to split into one or more sub-processes; further, each subprocess in the split can be duplicated forming multiple concurrent process instances.
- Functional flow transition rules that can depend on state variable dimensions (data or control) for one or more process instances and result in a change in one or more state variable dimensions for one or more process instances. These rules model the functional transformations and transaction flows found in the functional-flow model. OpEMCSS context-sensitive, message, memory, local, and global event action blocks model this interaction.

Because of the split action and assemble event pairs in a system process model discussed above, process instances can come into and out of existence as a function of time. Such variable instantiation of process instances is similar to the Object-Modeling Technique (OMT) where instances of object classes are created and deleted as the program executes (Rumbaugh 1991).

3 OPEM DIRECTED GRAPH LANGUAGE

Use of the Operational Evaluation Modeling (OpEM) directed graph language to develop a model and analyze a problem requires an in-depth understanding of the OpEM

Figure 1: Four Kinds of Discrete States

parallel process language. In this section, each language element is defined and rules for combining elements to form process diagrams are provided.

3.1 System State

A parallel process is the set of all sequences of system states and events that represent system operation. The system state is the discrete state of each process instance and the value of each state variable. Discrete states of parallel processes represent periods of time and are circles on an OpEM directed graph. There are four kinds of discrete states: (1) reaction time, (2) wait, (3) semi-continuous, and (4) idle. These are shown in figure 1 on the previous page.

A reaction time state represents the length of time a resource is performing a particular process instance task. Often a random variable generator computes a reaction time. Data to determine the distribution of the random variable may be determined by field observation or experimentation. Thus, a random variable can represent details of system operation that occur at lower levels of the system description hierarchy, allowing us to focus on complexity issues at the current level (Clymer 1990).

A wait until state represents the time a process instance waits for a logical condition to be satisfied in order to perform a task. Logic that activates the event may be in the event itself or elsewhere. If in the event itself, the state itself is identified with the logic on the graph. If 'passivated', awaiting external logic to be satisfied the event following the wait until state is identified as a direct execution path. Events are depicted as "< >" in a model. The two kinds of wait states are shown in figure 2.

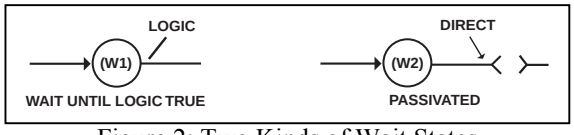

Figure 2: Two Kinds of Wait States

A more efficient technique, from the point of view of computer time required, is to use the 'passivated' event approach. Logic is checked by another process instance only when necessary and the wait event is executed by this process instance, using a direct execution path, when this logic is satisfied.

If the logic is located internally, logic usually is tested at each discrete time in the simulated sequence of states and events until it is satisfied. Logic can involve values of both discrete and continuous state variables. In a detailed model, it is sometimes necessary to compute values of state variables prior to testing logic.

A semi-continuous state approximates the continuous behavior of a detailed model of system operation. State variables associated with this kind of state are updated repeatedly with a constant time step to model a process that varies continuously. In contrast, discrete time events occur at irregular time intervals. Combined discrete event and continuous processes often occur in hardware-in-the-loop simulations. A semi-continuous state is shown in figure 3.

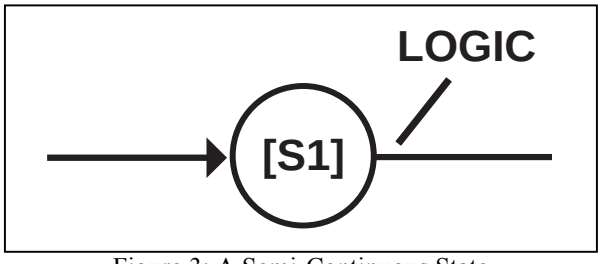

Figure 3: A Semi-Continuous State

A semi-continuous state is indicated by square brackets around the state name inside the circle. A detailed model that updates state variables is a part of the logic associated with the event following the state. The logic decides when the continuous process ends.

An idle state represents a period of time a subprocess is waiting for one or more other subprocesses to be completed before an assemble event can occur. In contrast, a wait state requires logic to be satisfied. Some of these subprocesses may be duplicated into multiple process instances. An assemble event combines one or more subprocesses and process instances into a single subprocess. The idle state will be discussed further in the context of the assemble event.

State variables represent data, knowledge facts used in inferencing, process control variables, entity position and velocity, and many other useful model attributes. In general, they represent process conditions other than the discrete process states discussed above.

3.2 Events

Events signify changes in system state, and are represented by directed line segments connecting the states in a

directed graph model (figure 4). Near the center of the line segment is a pair of brackets "< >."

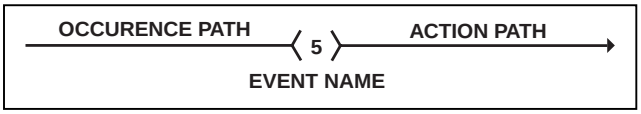

Figure 4: Events Shown as Directed Line Segments

Below these brackets is the event name, a short description of the event. To the left of the brackets is the 'occurrence path' that connects the event to the prior state. To the right is the 'action path' that connects the event to the following state. As discussed above, an event represents a change in one or more process instance dimensions of the system state vector. Event action implements state vector dimension changes, controls process flow, directly executes events in other sub-processes, and collects simulation report data.

Figure 5 shows an exit event from a wait state. The event has two alternate occurrence paths. Path 1 has logic specified and path 2 is a direct execution path from another event. An event may have alternate action paths as well.

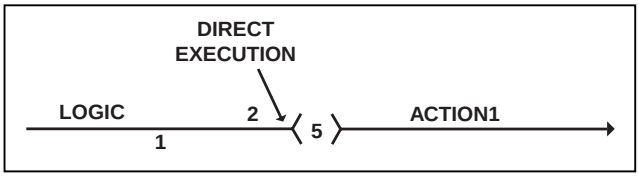

Figure 5: Exit Event from a Wait State

Figure 6 shows an event having two action paths. Only one path can occur each time that the preceding event is executed. ACTION1, associated with both paths, is executed first, then logic chooses the action to perform. In the example shown, if LOGIC is true, action two is performed, otherwise action three occurs.

Two parallel vertical lines to the right of the brackets "< >" indicate a split event (figure 7). Action one, preceding both parallel paths, is performed first. The sub-process then splits into two parallel subprocesses, both action two and three being performed. A split event models a context-sensitive transition where two FSMs have a synchronized start of operation, as discussed above, causing them to begin concurrent operation together.

Multiple process instances can occur two ways: (1) a split event creates multiple sub-processes and process instances as discussed above or (2) a generator process can create process instances and directly execute each process start event as shown in figure 11.

An assemble event (figure 8) has two parallel vertical lines preceding the brackets "< >." Assemble logic is specified to the left of the brackets next to the parallel lines. The numbers below each occurrence path to the left of the parallel double lines are path numbers that define the path that has been completed.

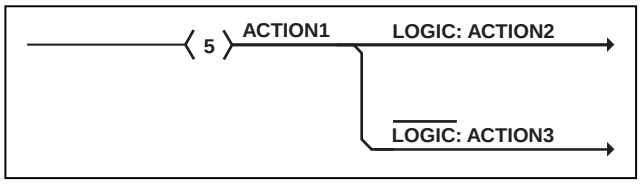

Figure 6: An Event having Two Action Paths

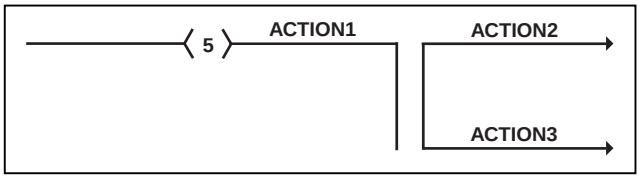

Figure 7: A Split Event

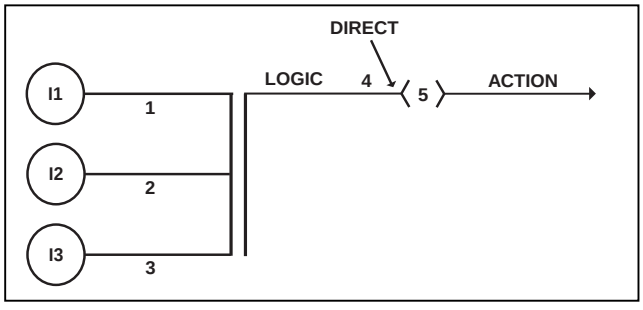

Figure 8: An Assemble Event

When a sub-process, process instance ends, assemble logic is tested. The path number associated with that occurrence path determines that the process that has been completed. The event "<5>" occurs only when the assemble logic is satisfied. An example of assemble logic is (1 * 2 * 3). The * is a logical AND. This means that sub-process paths one and two and three must be completed before these processes are assembled. Another example logic is ((1 * 2) + 3) which means that sub-process paths one and two or three must complete before these processes assemble since the + is a logical OR.

When assemble event "<5>" occurs, three sub-process diagrams are terminated and a single sub-process diagram continues. What is represented here is that all process instances, created for each sub-process diagram involved in the assemble event, must be deleted before continuing. In particular, assemble logic that includes a logical OR will definitely require process instances to be deleted, and this is very difficult for Petri nets to model because some process instances must be found and destroyed. However, the OpEMCSS Assemble Event block can model OR logic easily and automatically destroys the appropriate sub-process, process instances.

4 OPEMCSS LIBRARY BLOCKS

The basic OpEMCSS blocks are organized by categories:

1. Begin Event, End Event, and Evolutionary Algorithm blocks that define a system process instance (EXTEND calls these "runs");
2. Split Action and Assemble Event that define the begin and end of concurrent (parallel) processes;
3. Global Reaction Time Event, Reaction Time Event, and Wait Until Event that model the time spent in a discrete state;
4. Alternate Action, Classifier Event Action, Context-Sensitive Event Action, Event Action, Global Event Action, Initialize Event Action, Input Event Action, Local Event Action, Message Event Action, and Reward Event Action that perform event actions; and
5. Executive Block that sequences events in simulated time and Context-Sensitive Priority that updates the priority of each process instance at each event.

A brief summary of the OpEMCSS library blocks used in the part production model is provided next. For a more detailed evaluation, download the library from <http// ecs.fullerton.edu/~jclymer.>

4.1 Category 1

The Begin Event block generates an initial process instance item and initializes its attributes to start a simulation run. In EXTEND, attributes are of the form "**AttributeName= NumericalValue"** and are used to implement the OpEM model state variables discussed above. An Evolutionary Algorithm block can be the action of a begin event to search for optimal process control parameters. A Split Action block, that creates a set of sub-process, process instances, usually follows a Begin Event block and its initialization actions.

Since each parallel process instance in the model uses the attributes initialized by the Begin Event block, these attributes can provide global communication among all process instances. An example is a resource counter R that is used to decide if a resource is available for a process instance. If a resource is taken, the resource counter is decremented globally to communicate this to all process instances. When a process instance is finished with the resource, the resource counter is incremented globally to communicate to all process instances that the resource is available.

The End Event block deletes the final process instance item of a simulation run. This block can obtain parameter values from up to five blocks. These values are accumulated to produce an average value for each selected parameter based on a sample of simulation runs. The End Event block also sends the parameter values obtained from other blocks to each Evolutionary Algorithm block for computation of population member fitness that is used to control the search.

4.2 Category 2

The Split Action and Assemble Event blocks, working in pairs, allow sub-process diagrams and associated process instances at the same level in the system process to be synchronized according to a user supplied logic equation. Split Action and Assemble Event blocks allow a process instance to come into existence and operate concurrently with other process instances for a period of time, ceasing to exist when assemble logic is satisfied.

In an object-oriented model of a system, variable numbers of objects come into existence, exist for a time, then go out of existence (Rumbaugh 1991). An OpEMCSS process diagram, including one to three split-assemble levels, is similar to an OMT model in that each OpEMCSS sub-process diagram can define a variable number of duplicated process instances as discussed above. This contrasts with basic timed Petri net models that require a diagram to be duplicated for each process instance.

4.3 Category 3

Each block in Category 3 models the time required for each sub-process process instance to perform a task or wait for a logical condition to be satisfied to continue.

The Wait Until Event block can have its time duration determined by a logical equation. The wait until logic equation can be a function of up to eight process instance attributes, specified in the block dialog shown in figure 9, plus built in parameters to achieve pre-emptive, priority resource allocation and agent motion interaction events. Event actions allowed in the block are modifications of up to two global process instance item attributes using a "+=" operation. For example, when a shared resource is allocated to a process instance, the attribute R, representing the quantity of this resource, must be decremented. The operation R += -1 decrements R globally so all process instances that share this resource are notified. In this case, the += operation is equivalent to R= R-1.

A process instance item arriving on the "Direct" input connector of the Wait Until Event block bypasses the logic equation and is sent directly to the output connector. Direct input allows a "passivated" wait event to be modeled as discussed above. Such a process waits until another process "wakes it up" with direct execution.

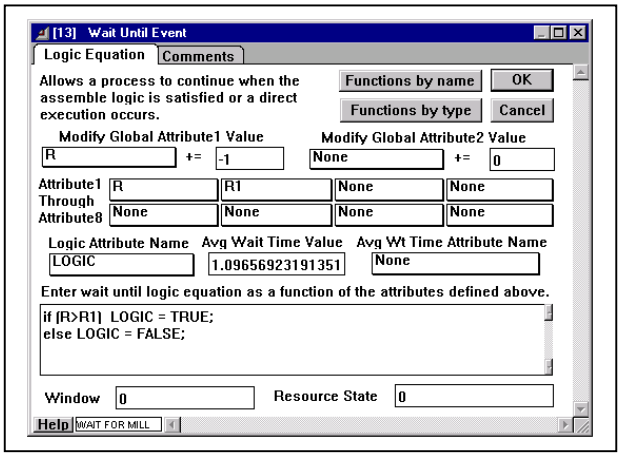

Figure 9: Wait Until Event Block Dialog

The Reaction Time Event block has a Gamma distributed reaction time specified in the block dialog shown in figure 10. Event actions permitted are modifications of up to two global process instance item attributes, using the "+=" operation, and one local process instance item attribute based on an equation. The global resource attribute R, discussed above, is incremented by this block when the resource is no longer needed.

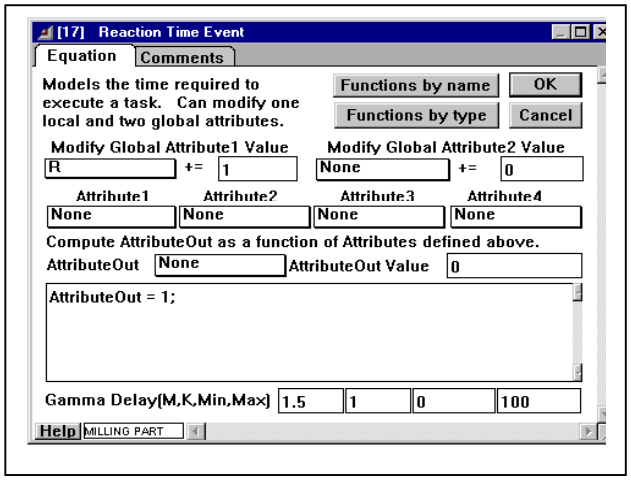

Figure 10: Reaction Time Event Block Dialog

For the Global Reaction Time Event block, reaction time is computed by an equation. The reaction time equation can be a function of GAMMADELAY, a Gamma distributed random variable, plus up to four attributes. Up to two global attributes can be modified using a "+=" operation.

The Global Reaction Time Event block works with the Wait Until Event block to achieve pre-emptive, priority resource allocation. Details of how to accomplish this are given in the "help" section of each block's dialog.

The Executive and Wait Until Event blocks work together to ensure that any time a global state variable is changed anywhere in the model, all wait logic is checked again before time is advanced. This ensures that no improper wait time is accumulated.

4.4 Category 4

Blocks in this category perform an event action after a category 3 block is completed. As discussed above, an event represents a change in one or more process instance dimensions of the system state vector. A Reaction Time Event block, Global Reaction Time Event block, or Wait Until Event block implements a change in discrete state for a process instance. Event action blocks change attribute values, decide process flow, directly execute events in other sub-processes, and compute simulation report attributes.

Each event block maintains a linked list, ordered by time, to store process instance items currently associated with the state represented by the block. When an event occurs, a process instance item is sent from an event block and passes through zero or more action blocks to the next event block. Event blocks have only a limited capability to perform event actions; action blocks expand the kinds of event actions that can be performed when an event occurs.

Alternate Action blocks allow one of three alternate transition paths to be selected, after an event has occurred, based on a decision equation. The DECISION value can equal 1, 2, or 3, which specifies the top, middle, or bottom output connector of the block; respectively. The decision equation can be a function of up to eight attributes, specified in the block dialog.

Classifier Event Action blocks each contain a forward chaining inference engine that is used to transform process instance attributes, for an item passing through the block, into other process instance attributes that represent rule actions. If several different actions are implied by the input process instance attributes (i.e., several rules are eligible to fire in a context), the best action is selected based on either the maximum BID value or a probability. The BID is a function of rule strength, specificity, and condition support such that a more specific rule has a higher BID. The rule selection probability is a function of rule strength and specificity such that a more specific rule has a higher probability of being selected to fire. Probability of rule selection is required for rule learning, but the maximum BID can be used once all rules have been determined.

An Event Action block can modify two global process instance attributes, using the "+=" operation, and one local process instance item based on an equation. The equation can be a function of up to four attributes, specified in the block dialog, plus "RandomNum", a uniformly distributed random number, and "CurrentTime."

A new process instance item can be created by an Event Action block and sent to an input connector of an Event Occurrence block placed before either a reaction time or wait until event block. This feature can represent a generator process that creates new process instance items

as a function of time. For example, the arrival of cut parts into the part milling sub-process during a period of time is modeled as shown in figure 11. This feature can also allow one process instance to execute another process instance directly. For example, a "passivated" process can be "waked up" by direct execution as discussed for the Wait Until Event block. Direct execution of events makes many types of complex, context-sensitive transitions possible.

A Global Event Action block can modify two global attributes based on an equation. Each global process instance attribute, having the proper process number, is modified by an attribute equation that can be a function of up to eight process attributes, specified in its dialog, plus "RandomNum", a uniformly distributed random number, or "CurrentTime". If process number in the dialog is zero, all system process instance items are modified. Otherwise, only process instance items with "Process", an element of the process identifier, equal to process number are modified. If the "local" box is checked in the dialog, the process instance item passing through the block is also updated.

The Reward Event Action block is used to compute a Classifier Event Action block reward payoff value based on an equation. The reward payoff value, that is sent to all Classifier blocks via a message, is computed by a reward payoff equation that can be a function of up to eight process instance item attributes. The payoff attribute name, specified in the dialog, is also sent in the payoff message. Classifier blocks with the message attribute name equal to "PayoffAttributeName," specified in the Classifier block Learning Dialog, can accept the message. A process number, specified in the dialog, is also sent in the payoff message. If this process number is zero, the rest of the process identifier sent is ignored. Otherwise, the duplicate process numbers, included in the process identifier sent, must be correct for a decision to be rewarded. If a duplicate process number is zero or compares with the duplicate process number for a process decision, then that process instance decision can be rewarded. This allows a sequence of decisions for a process instance to be rewarded.

4.5 Category 5

An OpEMCSS Executive block sequences events in simulated time. A Context-Sensitive Priority Block computes a priority for each process instance item at each discrete time based on an equation and prints process identifier, discrete state, and state variable values for each process instance at the end of each discrete time. The Executive and Context-Sensitive Priority Block work together to print a state trace, if selected, at each event in simulated time.

4.6 Summary of OpEMCSS Block Categories

An important feature of the OpEMCCS graphical simulation language is that a sub-process diagram can describe one or more process instances without having to duplicate the sub-process diagram for each one. This is especially

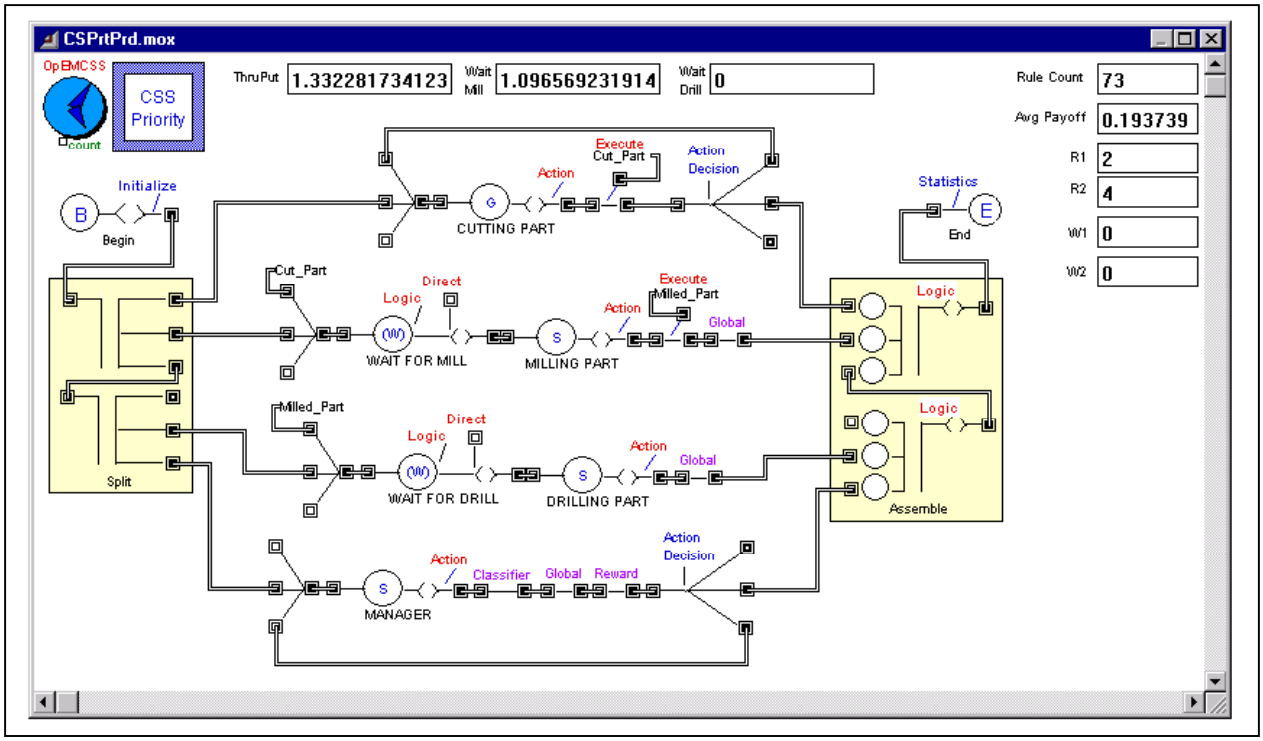

Figure 11: OpEMCSS Directed Graph Model of a Part Production System

important when modeling systems where the number of process instances is variable in simulated time and changes as the model executes. For example, the number of concurrent part milling process instances or part drilling process instances varies throughout a simulation run.

5 PART PRODUCTION MODEL

The part production model, shown in figure 11, is an example of applying the OpEMCSS library to model a simple process. Even for complex models, it is easy to place the blocks on the screen and connect them. Blocks are placed on the screen by clicking your mouse on a block name from the list opened from the EXTEND OpEMCSS library menu and dragging the icon to where you want it. When several blocks are on the screen, click your mouse on an output connector (little box with a black interior) and drag the mouse symbol to an input connector (little box with a white interior) of the next block. Next, you double click your mouse on a block icon and the block dialog, such as shown in figures 9 and 10, appears. You set up the dialog for proper block action, click your mouse on "OK," and the dialog closes. A model like the part production model can be built in less than an hour.

The part production model shown in figure 11 has four concurrent sub-process diagrams. These four sub-processes begin with a Split Action block that generates a process instance for the top and bottom process diagrams and ends with an Assemble Event block where logic synchronizes all process instances in the model when the last part has been completed for a simulation run.

The top process diagram is a part cutting process that generates part process instances that are sent to the part milling sub-process. Each part process instance can have attributes that distinguish it from other parts. The top sub-process diagram has a Global Reaction Time Event block that models the part cutting time for each part. The Alternate Action block continues sending the part cutting process instance back for another reaction time until the last part has been generated. When part generation is complete, the Alternate Action block sends the part cutting process instance to the Assemble Event block where it remains idle until synchronization occurs.

The second sub-process is a part milling process that models a part waiting in the part milling queue until a workstation is available to mill the part. The part milling queue is modeled by a Wait Until Event block whose dialog is shown in figure 9. When a workstation becomes available, resource attribute R is decremented and the part milling process instance item is sent to a Reaction Time Event block. This block models the time it takes to mill the part. When milling is complete, the Reaction Time Event block increments resource attribute R, as shown in figure 10, and the part process instance item is sent to an Event Action block. This block sends a part process instance item to the part drilling sub-process, as shown in figure 11, and the part milling process instance item is passed to a Global Event Action block. This block sends the current workload attribute W1 to the bottom, manager sub-process diagram as a global attribute. When the last part has been milled, the part milling process instance is sent to the Assemble Event block to remain idle until synchronization occurs.

The third sub-process is a part drilling process that models a part waiting in the part drilling queue until a workstation is available to drill the part. The sub-process diagram is similar to the second sub-process discussed above. For both the second and third sub-processes, the number of process instance items moving through the blocks varies throughout the simulation run. This is an example of a single sub-process diagram modeling multiple process instances. In a traditional Petri Net model, a sub-process diagram would be required for each concurrent process instance.

The fourth sub-process is a manager process that periodically updates two control attributes, R1 and R2, that are used in the wait until logic of the part milling and drilling processes; respectively. Figure 9 shows how R1 is used in the part milling process. Values for attributes R1 and R2 are decided using a Classifier Event Action block that has workload attributes W1 and W2 as its input.

A Classifier Event Action block contains a forward chaining inference engine that uses condition-action rules to transform condition attributes into action attributes. The condition attributes are obtained from a process instance item passing through the block. After the inference algorithm is complete, action attributes are added to the process instance item passing through the block before the item is sent to the output connector. These action attributes are used to control the system.

The Classifier Event Action block also has an evolutionary rule induction capability. In the part production model, this block receives a payoff attribute from a Reward Event Action such that a sequence of decisions is rewarded when they result in a balanced workload.

The rules that were discovered tend to decrease R1 and increase R2 as W1 increases so that more workstations are made available for milling parts when needed by the part milling process. The rules tend to maintain a balanced work-flow and maximize the throughput of parts moving through the system.

6 SUMMARY

A graphical discrete event simulation library is described that is based on interacting concurrent processes. This library is called Operational Evaluation Modeling for Context-Sensitive Systems (OpEMCSS). CSS, as discussed in this paper, is a systems theory, based on finite state machines, that can assist a manufacturing or production manager in understanding, evaluating, and optimizing production work flow, represented as communicating

concurrent (parallel) processes. A Finite State Machine (FSM) is defined as a set of states and a set of transition rules that describe transitions between these states. It was shown that if transition rules describe transitions for several FSMs working together, then these rules can describe context-sensitive process interactions. OpEM is described as a graphical simulation language that can model several FSMs working together and, thus, context-sensitive process interactions. A simple production problem is discussed that is an example of applying the Classifier Event Action block to discover optimal rules to manage the workflow for a part production process.

REFERENCES

Bahill, A.T., et al, 1998. The Design-methods comparison project, *IEEE Transactions on Systems, Man, and Cybernetics-Part C Applications and Reviews*, Volume 28(1): 80-103 .

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

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

Clymer, J. R. 1997. Expansionist/context-sensitive methodology: engineering of complex adaptive systems, *IEEE Transactions on Aerospace and Electronic Systems*, 33(2): 686-695.

Clymer, J. R. 1999. Simulation-based engineering of complex adaptive systems, *Simulation*, San Diego, CA: The Society of Computer Simulation International, 72(4): 250-260.

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

Rumbaugh, J. et al. 1991. *Object-oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice-Hall Inc..

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

Zeigler, B.P., and H. Praehofer. 1999. *Theory of Modeling and Simulation*, 2nd edition, Academic Press.

AUTHOR BIOGRAPHY

JOHN R. CLYMER is a professor of electrical engineering at California State University Fullerton (CSUF) and consults in the area of systems engineering, simulation, and artificial intelligence. In addition to consulting, he presents intensive short courses at various locations around the United States and abroad. His teaching assignments have included computer engineering, system control, continuous systems simulation, operational analysis and DES simulation, optimization and mathematical programming, and artificial intelligence (fuzzy logic and control, neural networks, and expert systems). Dr. Clymer's current research interests are focused in the area of intelligent, complex adaptive systems, applying integrated simulation, artificial intelligence, and evolutionary programming methods to study such systems. He is a founding member of the Applied Research Center for Systems Science at CSUF. He is a member of IEEE, SCS (SIMULATION journal associate editor), and INCOSE. His email address is <jclymer@fullerton.edu>.