

SIMULATION-BASED Engineering of Complex Systems using EXTEND+MFG+OpEMCSS

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

ABSTRACT

A Complex Adaptive System (CAS) is a network of self-organizing, intelligent agents that share knowledge and adapt their operations in order to achieve overall system goals. Three things are needed to understand, design, and evaluate CAS. First, a mathematical model or way-of-thinking about CAS, called Context-Sensitive Systems (CSS) theory, is required to provide a solid foundation upon which to represent and describe the kinds of interactions that occur among the CAS agents during system operation. Second, a graphical modeling language is required that implements CSS theory in a way that enhances visualization and understanding of CAS. Third, a systems design and evaluation tool is required that makes it easy to apply CSS theory, expressed using a graphical modeling language, to understand, design, and evaluate CAS. As an example, an OpEMCSS model of two intelligent agents is discussed that learn rules and maximize their average reward in the prisoner's dilemma game.

1 INTRODUCTION

Simulation has become, in the last few years, a mandatory part of the systems engineering process, especially as applied to the engineering of Complex Adaptive Systems (CAS). Large military, manufacturing, and transportation systems are too complex and costly to be designed and evaluated without the use of simulation. Business organizations are currently being re-engineered, and simulation is a natural tool to design and evaluate them. The purpose of this paper is to describe a modeling and simulation tool called Operational Evaluation Modeling for Context Sensitive Systems (OpEMCSS) that can explicitly describe the kinds of interactions among system components and processes that occur during CAS operation.

Two kinds of systems engineering tools are typically applied to support the engineering of complex systems and the systems engineering process: (1) tools that automate the system development process and produce a design capture database such as CORE (Buede 1999) and (2) tools that are used for concept exploration and discovery such as EXTEND+MFG+OpEMCSS (Clymer 1999, 2000, 2001).

A system development database tool is used for complete automation of the systems engineering process from design team entry of requirements until generation of the systems specification documents occurs. Such a tool allows a central design database to be accessed by members of a design team and design description or decision-making information to be entered and shared. A design capture tool can be used to map the functional model onto the system architecture, and it can generate systems specification documents. Limited modeling capability is provided to check the consistency of a design; however, these tools are not suitable for concept exploration of CAS.

A concept exploration tool allows rapid motion through problem space to examine system design problems. It facilitates "out-of-the-box" thinking to discover the best system concepts that solve system design problems without focusing on a point design too soon. As each problem is solved, the solution is entered into the system design database. Thus, a system development database tool and a concept exploration tool are complementary in their support of the systems engineering process.

The OpEM methodology discussed in this paper has been successfully applied during concept exploration, as discussed above, in various CAS applications. For example, pre-OpEMCSS studies of CAS are discussed in: (1) a visual flying rules (VFR) air traffic control system where the aircraft are intelligent agents (Clymer 1992a, 1995), (2) a railroad system where the trains are the intelligent agents (Clymer 1992b), (3) a sonar array system where the sonar nodes are the intelligent agents (Clymer 1994, 2001), and (4) a vehicle traffic grid where the traffic light controllers are the intelligent agents (Clymer 1995, 1997, 2001). All of these models have been re-implemented in OpEMCSS and are available for your evaluation and experimentation. OpEMCSS is a graphical Discrete Event Simulation (DES) library that works with EXTEND+MFG, a relatively inexpensive yet powerful software product of Imagine That Inc located in San Jose, CA ("<http://www.imagethatinc.com>"). A free academic version of OpEMCSS is currently available for download.

1.1 Complex Adaptive Systems

A CAS is a network of self-organizing, intelligent agents that share knowledge and adapt their operations in order to achieve overall system goals. For example, consider a distributed, vehicle traffic control network located in a large city (Clymer 1997), mentioned above. Each major intersection has a vehicle traffic light control subsystem or controller to determine traffic light timing. In this system, each traffic light controller uses its perceptions about incoming traffic flow to optimize light timing, minimizing local vehicle waiting time. The result of each traffic light controller adapting offset light timing to accommodate traffic flow coming from other intersections is to minimize the average waiting time in the entire network. Global minimization of traffic waiting time results as a consequence of the emergent behavior of this system, traffic light collaboration.

Another feature of CAS is emergent patterns of behavior that can actually be observed during system operation. For example, as light timing control in the overall traffic grid evolves in the way discussed above, a complex but definite pattern in network operation, north-south red to green transition timing, emerges out of an initial random light pattern. The emergent behavior of the traffic grid cannot be explained through an understanding of each controller alone. Understanding only comes when we **study the interactions** of the controllers as they adapt their behaviors in response to perceived information about incoming traffic flow.

CAS are not, as yet, well understood; although, they have been the subject of intensive research during the last ten years by the Santa Fe Institute (Complexity Journal). However, we do know that the ability of a CAS to achieve the desired emergent behavior depends heavily on the right knowledge being shared among the agents and correct decision rules being applied (Clymer 1992a, 2001).

1.2 Needed to Understand CAS

Three things are needed to understand, design, and evaluate CAS. First, a mathematical model or way-of-thinking about CAS, called Context-Sensitive Systems (CSS) theory, is required to provide a solid foundation upon which to represent and describe the kinds of **agent interactions** that occur among the CAS agents during system operation. Second, a graphical modeling language is required that implements CSS theory in a way that enhances visualization and understanding of CAS. Third, a systems design and evaluation tool is required that makes it easy to apply CSS theory, expressed using such a graphical modeling language, to understand, design, and evaluate CAS. Fundamental to understanding CAS are the **agent interactions**.

Three kinds of agent interactions are required to represent and describe CAS operation:

- SYNCHRONIZATION - One or more process instances must wait while another process completes a task;
- RESOURCE CONTENTION - One or more process instances must wait while another process uses a resource; and
- COMMUNICATION AND ADAPTATION - One process instance sends a message to another process that is used to decide an action.

Figure 1 shows a functional flow model for a user job.

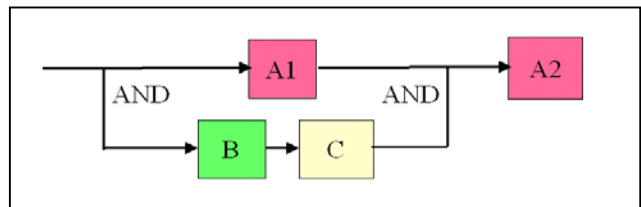


Figure 1: Functional Flow Model of a User-Job

A new job enters the system every 8 seconds. Each job has a concurrent task A1 in parallel with tasks B and C. When both tasks A1 and C complete execution, the parallel process synchronizes and task A2 is executed.

Figure2 shows a timeline of the arrival and execution of three user jobs. Task A1 requires 10 seconds, task B-6 seconds, task C-7 seconds, and task A2-3 seconds. The parallel vertical lines signify the split and assemble of task A1 in parallel with tasks B and C.

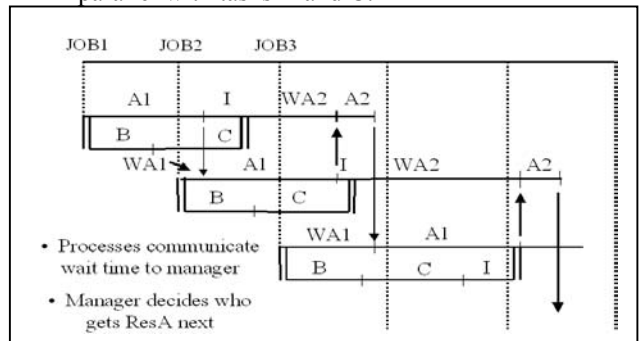


Figure 2: Timeline of System Operation

For job 1, the top process waits in idle state I while task C completes execution. This is an example of **synchronization**. After job 2 arrives, task A1 must wait because resource A is busy with job 1. This is an example

of **resource contention**. When task A1 of job 2 completes, there is a conflict concerning which task is allocated resource A next. Do we allocate resource A to task A2 of job 1 or to task A1 of job 3? This is an example of where the processes must **communicate and adapt** in order to collaborate to improve system effectiveness: minimum wait time and maximum throughput. In this example, one decision policy could be to allocate resource A to the task that has the most jobs waiting or another policy could be to allocate resource A to the task that has been waiting the longest.

1.3 Interacting Concurrent / Parallel Processes

What is an agent and what is a process? An agent is a physical component of the system. A parallel process describes what an agent or collection of agents does: the sequential-concurrent execution of a set of related tasks such as shown in Figure 2. A system can be modeled from the point of view of each agent and what it does or from the point of view of what the system does given the agents are performing their tasks. An important point is that OpEMCSS can model a system from either point of view.

A parallel process is defined as the collection of all possible sequences of system states and events that represent the operation of a system or organization. For example, suppose that three processes describe a system where the weapon (WPN) process is in the "PreFire Evaluation" state, the Fire Control System (FCS) process is in the "Tracking" state, and the Launcher (LNCH) process is in the "Aim" state. When the LNCH process goes to the "Ready" state, a message is sent to the WPN process, that has been waiting, which causes a transition of the WPN process to the "Fire" state. Each system state of a parallel process model includes the discrete state of each parallel sub-process. For example, FCS Tracking, LNCH Ready, and WPN Fire describe one particular system state for the three process model discussed above. Thus, discrete states represent periods of time where either (1) each functional activity or task is being performed by a resource(s) or (2) each functional activity or task is waiting for a specified logical condition to be satisfied (message received) before it can continue, as discussed above. Also part of the system-state may include zero or more state variables for

each sub-process. State variables have values that identify conditions, other than what functional activity or task is currently being performed, such as parameters that are used to control execution of the process. For example, a state variable of the launcher process is how many weapons are remaining in the magazine. Events define points in time where a change of system state occurs.

1.4 Paper Overview

The paper first provides an overview of the OpEM graphical language, that is a graphical expression of interacting concurrent processes such as shown in Figure 2, and the OpEMCSS library blocks, that implement the OpEM language. These are described as a basis for simulation-based engineering of complex adaptive systems. A simulation of the prisoner's dilemma, that is built using these blocks, is discussed as an example. This model describes a co-evolutionary CAS where each agent adapts its decision-making rules to achieve its goals. The agents discover that collaboration leads to the maximum payoff (Axelrod 1984). The paper concludes with a summary procedure that describes how OpEMCSS is used during the simulation-based engineering (design and evaluation) of complex adaptive systems as a concept exploration tool and a discussion of future research.

2 OPEM DIRECTED GRAPH LANGUAGE

Use of the OpEM directed graph language to develop a model and analyze a problem requires an in-depth understanding of the parallel process language. In this section, each language element is defined and rules for combining elements to form processes are provided.

2.1.1 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 a directed graph. There are four kinds of discrete states: (1) reaction time, (2) wait, (3) semi-continuous, and (4)

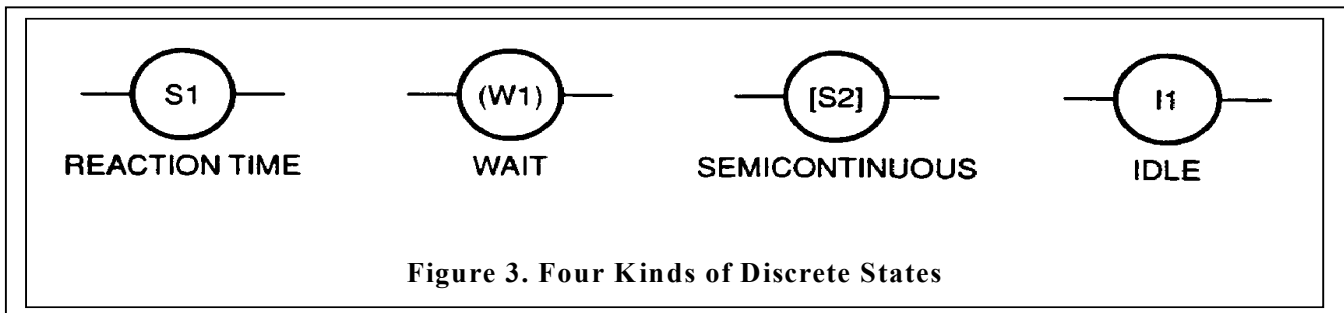


Figure 3. Four Kinds of Discrete States

idle. State symbols are shown in Figure 3 above.

A reaction time state represents the length of time a resource is performing a particular functional activity or 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. A wait state represents the time a system component waits for a logical condition to be satisfied in order to perform a functional activity or 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 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 4.

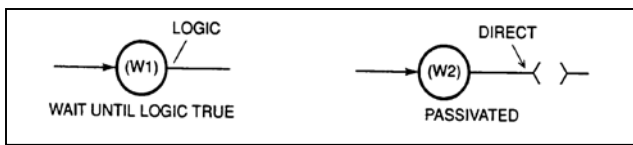


Figure 4: Two Kinds of Wait States

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 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.

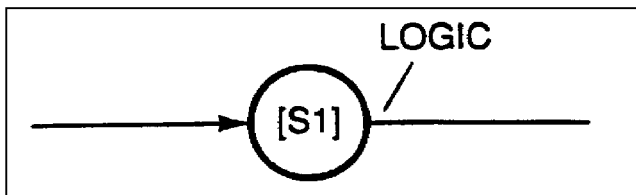


Figure 5: A Semi-Continuous State

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 5.

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 sub-process is waiting for one or more other sub-processes to be completed before an assemble event can occur. Some of these sub-processes may be duplicated into multiple process instances. An assemble event combines one or more sub-processes and process instances into a single sub-process. The idle state will be discussed further in the context of the assemble event.

2.1.2 State Variables.

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 states discussed above.

2.1.3 Events.

Events signify changes in system-state, and are represented by directed line segments connecting the states in a directed graph model (Figure 6). Near the center of the line segment is a pair of brackets "< >." 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 sub-process 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 6. Events are shown as Directed Line Segments

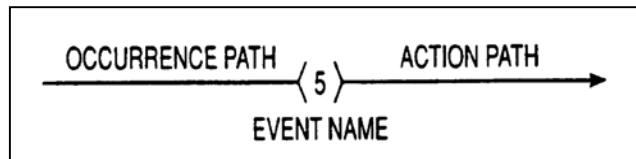


Figure 7 shows an exit event from a wait state. The event has two alternate occurrence paths. Path 1 has logic specified and path 2 is a direct execution path from another event. An event may have alternate action paths as well.

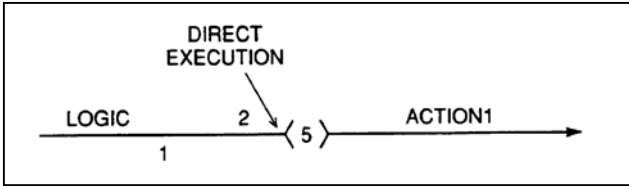


Figure 7: A Wait State Event having Alternate Occurrence Paths

Figure 8 shows an event having two action paths. Only one path can occur each time that the preceding event is executed. ACTION1, associated with either path, 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.

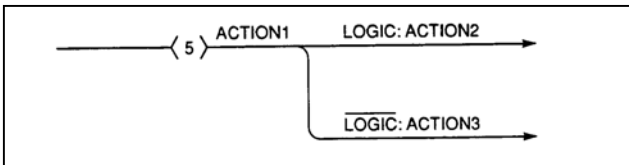


Figure 8: An Event that has two Action Paths

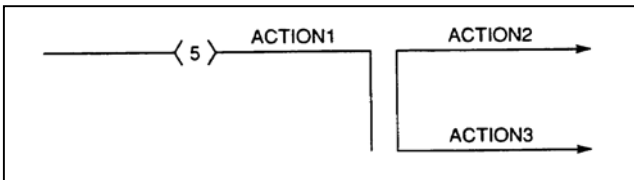


Figure 9: A Split Event

Two parallel vertical lines to the right of the brackets "< >" indicate a split event (Figure 9). Action one, preceding both parallel paths, is performed first. The sub-process then splits into two parallel sub-processes, both action two and three being performed.

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.

An assemble event (Figure 10) 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.

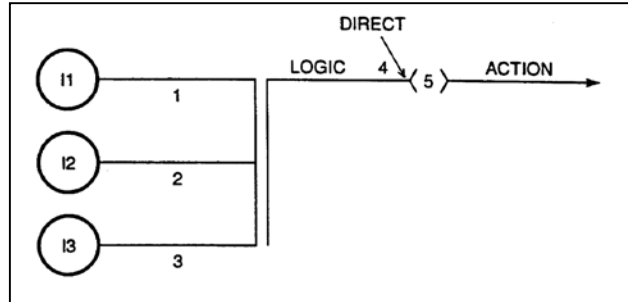


Figure 10: An Assemble Event

When a sub-process ends, assemble logic is tested. The path number associated with the 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 subprocess paths one and two and three must be completed before these processes are assembled. Another example logic is $((1 * 2) + 3)$ which means that subprocess paths one and two or three must complete before these processes assemble since the + is a logical OR.

When assemble event "<5>" occurs, the three subprocesses and associated process instances are destroyed and a single subprocess continues. In comparison, assemble logic that includes a logical OR is very difficult for Petri nets to model because some process instances must be found and destroyed. However, the OpEMCSS assemble block can model OR logic easily and automatically destroys the required sub-process process instances.

3 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.

The OpEMCSS blocks used in the Prisoner's Dilemma model are discussed in this section. A more complete description of all the OpEMCSS blocks is found in (Clymer 1999, 2000, 2001).

The Begin Event block (label in Figure 11 is "Begin Games") generates an initial process instance item and initializes its attributes to start a simulation run. In EXTEND attributes are of the form "Attribute Name = Numerical Value" are used to implement the OpEM language state variables discussed in the previous section.

The End Event block (label in Figure 11 is "End Games") 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 Split Action and Assemble Event blocks, working in pairs, allow sub-processes and 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.

The Reaction Time Event block (labels in Figure 11 are "Agent X Decides") has a Gamma distributed reaction time specified in the block dialog, and it is used to simulate a reaction state as shown in Figure 3. Event actions permitted using this block 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.

A Local Event Action block allows process instance attributes (OpEM state variables) of the process instance item passing through the block to be modified. If "Accumulate" is set true in the block dialog, the block accumulates process instance items (deletes them) passing through the block until the last one is received, which is sent to the output connector. Thus process instance items created by a generator process that complete their state-event sequence can depart rather than being stored in an Assemble Event block. This allows the model to run much faster if many processes are created.

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

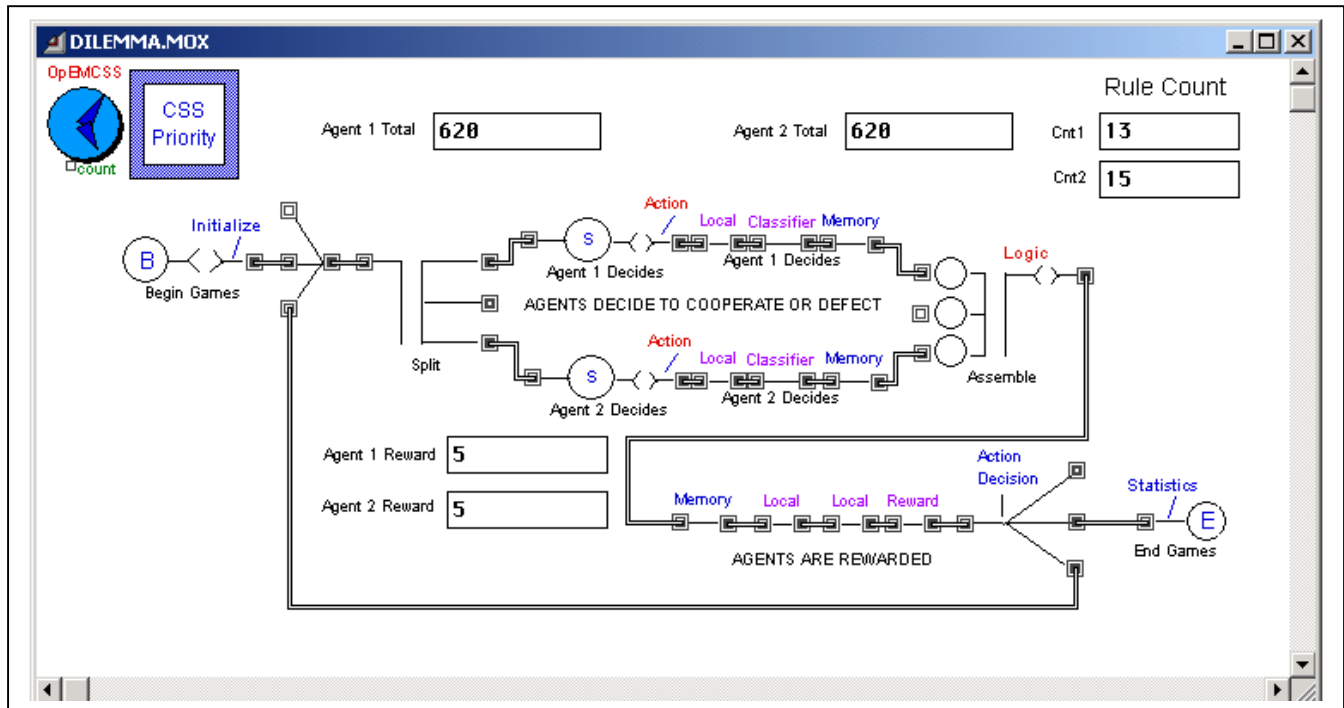


Figure 11: The Prisoner's Dilemma Model

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 strictly a function of rule strength. Probability of rule selection is used mainly for rule learning, but the maximum BID can be used both during learning and once all rules have been determined.

Rule induction begins with a subset of the most general rules as the initial knowledge base such that each possible value for the rule condition fact is covered. The classifier block uses these rules to make decisions and the reward block evaluates the quality of each decision, sending a payoff to the classifier block. The classifier uses the payoff to reward or punish the rules. The result is that rule strength increases for good rules and decreases for bad rules.

The rule induction algorithm randomly selects rules for modification, from the current set of rules that cover decision situation S, based on situational ambiguity. Situational ambiguity is high when all eligible rules specifying alternative decisions for a situation S compete and becomes low when one rule dominates. Rule induction for situation S stops when one rule dominates, allowing rule learning to focus on other situations that still have high ambiguity.

The induction operators that can be applied are “change a fact value” in the selected rule, “add a new fact” to the rule, and “delete an old fact” from the rule. These operators are applied to either the condition or action of a rule based on a probability. Another probability is used to decide whether to change a rule fact or add / delete a fact. Given add / delete is selected, a probability function is used to select either add or delete. Initially, this probability function is zero and then increases exponentially, as rules are modified, until a maximum value of 0.9 is reached. The result is that rule induction initially focuses on the most general rules at the top of rule hypothesis network but eventually generates more specific rules further down the rule network.

In summary, the search for the best rules is guided by situational ambiguity and proceeds from the top to the bottom of the rule hypothesis network. The result is that a default hierarchy of rules that can make all decisions correctly is found.

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.

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.

An important feature of the OpEMCSS 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 important when modeling systems where the number of an object is variable in simulated time and changes as the model executes.

4 PRISONER'S DILEMMA

The Prisoner's Dilemma model (Axelrod 1984) is shown in figure 11, and it is comprised of the OpEMCSS library blocks discussed in the previous section. How the blocks work together to simulate a system is discussed in this section.

The ExecutiveS block, in the upper left-hand corner, controls the sequence of events in the model, and updates agent position if motion blocks are included. Next to the ExecutiveS block is a Context-Sensitive Priority block that can update the priority of each process instance currently in the model at every discrete time. This block also works with the ExecutiveS block to produce the state-trace when the model is in trace mode.

The first block in the model is the Begin Event block that creates a process instance item, a record that is passed from block to block to model process flow. The Begin Event block initializes some state variables, called attributes in EXTEND, that are of the form "AttributeName = NumericalValue." Attributes "Agent1" and "Agent2" are initialized to one, and they are used to define the current play of each agent (1-cooperate, 2-defect). Attributes "Past1" and "Past2" are initialized to one, and they are used to define the past play of each agent (1-cooperate, 2-defect). Initialized to zero are two counters, "Agt1Totals" and "Agt2Totals," that are used to accumulate the total payoff of each agent in the game. The attribute "CountGames" is initialized to zero, and the attribute "MaxGames" is initialized to the maximum number of games for a simulation run.

The Begin Event block passes the process instance item to an Event Occurrence(3) block that increments the attribute "CountGames" and that provides for multiple paths into the Split Action block that follows.

The Split Action block receives a single item and splits the sequential process into two concurrent process instances where each process instance represents an

intelligent agent playing the game. The top and bottom sub-process diagrams shown in Figure 11 consist of a Reaction Time Event block followed by a Local Event Action block, a Classifier Event Action block, and a Memory Event Action block. The Reaction Time Event block represents a period of time spent in a discrete state. The following Local Event Action block sets “PastX” to remember the agent’s previous play. The initial rules used by agent1 are as follows:

```
Rule1:IF
  Agent2=Cooperate,
THEN
  Agent1=Defect,CF=50%
```

```
Rule2:IF
  Agent2=Defect,
THEN
  Agent1=Cooperate,CF=50%
```

The rules used by agent2 are similar. Attributes “Past1” and “Past2” are also allowed as conditions in the rules. The Classifier Event Action blocks in both sub-processes have been set up for rule learning discussed below. The parallel sub-processes both connect to an Assemble Event block that produces a single process instance as its output. The Memory Event Action blocks, that are placed before and after the assemble, store and retrieve attributes in a global memory named “Dilemma.”

Following the Memory Event Action block that retrieves attributes for both agents, two Local Event Action blocks use the rules of the prisoner’s dilemma game to determine each agents reward based on current play and then accumulates the total reward for each agent. The Reward Event Action block computes the payoff for each of the Classifier Event Action blocks

The classifier blocks always converge to the following rules for agent 1. Agent 2 rules are similar.

```
Rule1:IF
  Agent2=Defect,
THEN
  Agent1=Cooperate, CF=53.877%
```

```
Rule2:IF
  Agent1=Defect,
THEN
  Agent1=Defect, CF=25.877%
```

```
Rule3:IF
  Agent1=Defect,
THEN
  Agent1=Cooperate, CF=50.455%
```

```
Rule4:IF
  Agent1=Cooperate AND
```

```
  Agent2=Cooperate,
THEN
  Agent1=Cooperate, CF=100%
```

Therefore, the agents start out playing the “zero sum game” strategy that is win-lose, defined by the initial rules, and end up playing a win-win rule strategy that maximizes their total reward. However, it is interesting that if one agent is locked into the “zero sum game” strategy of win-lose through its reward payoff equation and the other agent’s strategy remains win-win, the heterogeneous agents can only adapt to a win-win rule strategy if the rule-learning rate is balanced. If both agents play the “zero sum game” strategy of win-lose due to their reward payoff equations, the game converges to lose-lose.

If you watch the model operate, you will observe that before the rules converge to complete cooperation (rule 4 above), the agents play a “tit-for-tat” strategy (Axelrod 1984). The rules are: 1) if you cooperate then I cooperate and 2) if you defect then I punish you and defect, cooperating with you at a later time. The “tit for tat” strategy is implemented by rules one through three above.

5 SUMMARY AND FUTURE RESEARCH

OpEMCSS model development in support of systems design and evaluation of complex systems usually applies the following steps:

1. Define the system to be evaluated and describe system operational scenarios. This step produces system scenario diagrams that describe the interactions between the system and its environments.
2. Define missions of the system, mission objectives, and measures of effectiveness for each mission. This step defines system effectiveness and performance parameters to be estimated by the model and specifies the model scoreboard.
3. Define objectives for study of the system. What questions a model must answer dictates model details.
4. List functional activities or tasks that when performed will achieve each mission objective. Identify hard and soft constraints on how tasks can be performed.
5. Subdivide functional activities into periods of time represented by process instance states where: (a) an activity is being performed by a resource and (b) an activity is waiting for a logical condition to be satisfied.
6. Group purely sequential states, representing activities, into the same processes and states that can be performed concurrently into separate processes. Always allow for maximum

permissible parallelism: activity states are sequential only if they must be done that way.

7. Develop time-lines (see Figure 2) of concurrent states that describe system operation and assist visualization of system operation. Show process states on the time-line and indicate when one process sends a message to another process or process synchronization occurs.
8. Develop a directed graph model diagram on the computer screen. Start with placement and connection of the OpEMCSS blocks then add state variables, using block dialogs, to model interactions and physical details. State sequences are visualized from the time-line then generalized to place and connect the OpEMCSS blocks to model all cases.
9. Generate event-state traces or time-lines to verify and validate the model and modify the directed graph model diagram until satisfied.
10. Operate simulation program to gain an understanding of the problem space and to evaluate alternative solution concepts and system designs.

In multi-agent systems, the fundamental design problem is always for each agent to discover what critical set of features is required in order to collaborate and for each agent to learn a set of rules that use these features to make the proper decision in each situation. The goal is that the performance for the whole system is optimized.

Figure 12 shows a model of a single agent. In multi-agent system, agents communicate and share knowledge in

order to achieve overall system objectives. Agents have two kinds inputs and two kinds of outputs. One kind of input comes from sensors that produce a continuous stream of raw data. Raw data must be transformed into a set of features that describe everything the agent perceives that might be useful to make decisions. Features measure aspects of the raw data that may relate to various possible goals the system may have. However, for a specific goal only a subset of features are useful in making a decision. The feature extraction block transforms a very large set of generally useful features into a small set of specific features that support a minimal set of decision-making rules. Another kind of input is messages from other agents. Messages and decision features are both used to make decisions. Agent outputs are messages sent (self messages change agent state) and actions that change the environment. Environmental changes are often perceived by agent sensors and can also be a form of input message.

The evaluator block produces learning feedback for the classifier block and the feature extraction block. Local and global MOEs and MOPs are used to determine the learning feedback for the classifier and the feature extraction blocks.

The feature extraction block uses its learning feedback to guide an evolutionary search for the best transformation from a large set of general features to a much smaller set of specific features. The goal here is to minimize decision ambiguity, which results when decision features are not capable of an unambiguous decision, and to minimize the total number of rules required.

The classifier system block is capable of two kinds of rule learning. In Stimulus-Response (SR) learning, the

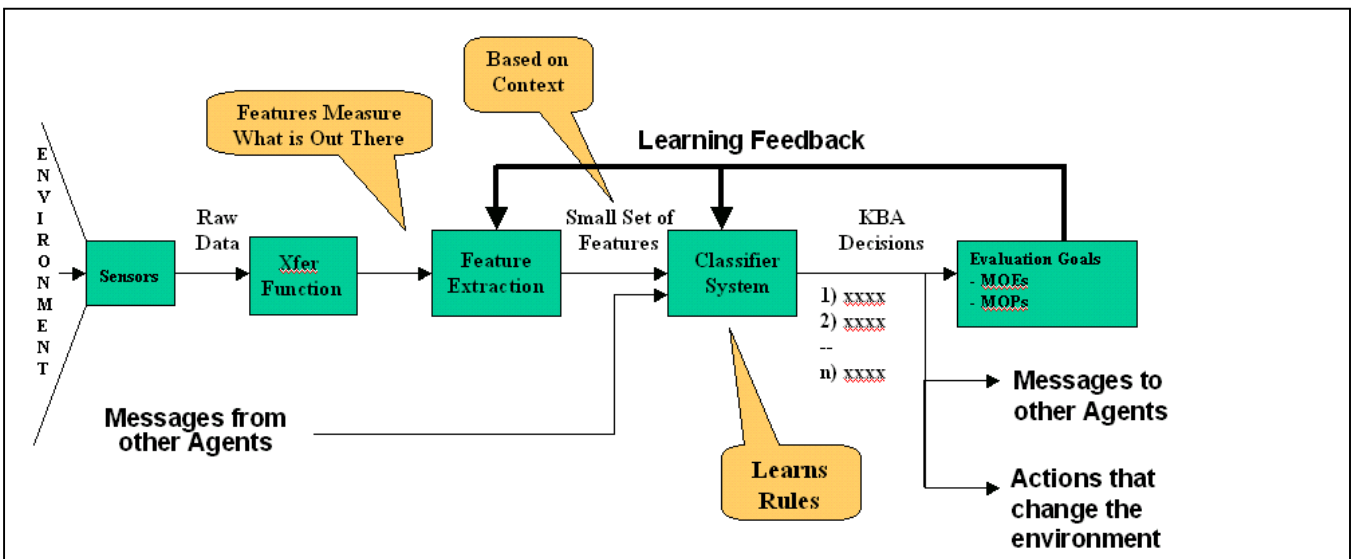


Figure 12: Single Agent Model

classifier system learns the best decision for each situation presented by the environment. If the decision is correct, a rule is rewarded. If a decision is not correct, a rule is punished. Other eligible rules not selected to decide receive a penalty. This approach results in a default hierarchy (a collection of general to specific rules that specify the correct action). A default hierarchy can be effective even when all decision feature facts are not known. The classifier may still be able to decide correctly.

The second type of learning is called Reinforcement Learning (RL). In RL the classifier system learns a set of rules that maximizes the total payoff from a sequence of decisions; indeed, RL can be shown to be equivalent to dynamic programming. The OpEMCSS classifier system block uses profit sharing with penalty to reward rules. Each time a rule receives a reward or punishment, it is shared with all previous rules in the decision sequence. However, rules that initially received a negative reward (punishment) do not share later in profit sharing.

Current research is to model a network of communicating agents, as shown in Figure 12, performing feature extraction and both SR and RL learning. For example, consider a manufacturing process where there are N tasks and M workstations to perform each task. Certain quality features are measured before the product leaves each workstation. An intelligent agent evaluates these features and decides pass, fail, or rework. After the last production task has been completed, additional tests are performed to determine overall quality and production yield. The production decision sequence is rewarded or punished based on overall quality and yield using profit sharing. Using computer simulation, can the individual agents learn a set of rules that will optimize overall quality and production yield?

REFERENCES

- Axelrod, R.M., 1984. *The Evolution of Cooperation*, New York, NY: Basic Books, Inc.
- 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., P.D. Corey, and J. Gardner, 1992. Discrete Event Fuzzy Airport Control, *IEEE Transactions on Systems, Man, and Cybernetics*, 22 (2): 343-351.
- Clymer, J. R., D. J. Cheng, and D. Hernandez, 1992. Induction of Decision Making Rules for Context Sensitive Systems, *Simulation*, San Diego, CA: The Society of Computer Simulation International, 59 (3): 198-206.
- Clymer, J. R., P. D. Corey, and H. Bandukwala, 1994. Induction of Classification Rules From Noisy Sonar Features, *Simulation*, San Diego, CA: The Society of Computer Simulation International, 62 (4): 256-267.
- Clymer, J. R., 1995. Induction of Fuzzy Rules for Air Traffic Control, *IN Proceedings-1995 IEEE International Conference on Systems, Man, and Cybernetics*, Vancouver, British Columbia, Canada, October 22-25, 1995, pages 1495-1502.
- 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.
- Clymer, J.R., 2000. Optimizing Production Work Flow Using OpEMCSS, IN Proceedings of the 2000 Winter Simulation Conference, ed. J.A. Joines, R.R. Barton, K. Kang, and P.A. Fishwick, 1305-1314, Piscataway, New Jersey: Institute of Electrical and Electronic Engineers.
- Clymer, J.R., 2001. *Simulation-Based Engineering Of Complex Systems*, Placentia, CA: John R. Clymer & Associates.
- Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, 1991. *Object-oriented Modeling and Design*, Englewood Cliffs, NJ: Prentice-Hall Inc.

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. He received his Ph.D. from Arizona State University in 1971. 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). His 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 (ARCSS) at CSUF. He is a member of IEEE, SCS, and INCOSE. His Email and website addresses are jclymer@fullerton.edu and <http://www.ecs.fullerton.edu/~jclymer>.