

Simulation-Based Engineering of Complex Adaptive Systems Using a Classifier Block

John R. Clymer and David J. Cheng
California State University Fullerton
Jclymer@fullerton.edu Dcheng@fullerton.edu

Abstract

A Complex Adaptive System (CAS) is a network of communicating, intelligent agents where each agent adapts its behavior in order to collaborate with other agents to achieve overall system goals. Further, the overall system often exhibits emergent behavior that cannot be achieved by any proper subset of agents alone. A graphical simulation library called Operational Evaluation Modeling for Context-Sensitive Systems (OpEMCSS) has been developed to simulate complex systems, including CAS. This simulation library includes a Classifier Event Action block that is a forward chaining, expert system controller. The Classifier Event Action block can implement both crisp and fuzzy rules. 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. Out of the co-evolutionary interaction of these agents, emerges the cooperation that maximizes total system reward.

1. Introduction

A Complex Adaptive System (CAS) is a network of communicating, intelligent agents where each agent adapts its behavior in order to collaborate with other agents to achieve overall system goals. Further, the overall system often exhibits emergent behavior that cannot be achieved by any proper subset of agents alone. CAS includes satellite communications networks, vehicle traffic control networks, multi-national corporations, the global economic system, ecological systems, flexible manufacturing systems, and military command and control C4ISR networks.

A graphical simulation library called Operational Evaluation Modeling for Context-Sensitive Systems (OpEMCSS) has been developed to simulate CAS [8,9]. OpEMCSS works with EXTEND, a powerful yet relatively inexpensive simulation tool. OpEMCSS implements the Operational Evaluation Modeling (OpEM) graphical discrete event simulation (DES) language. The OpEM DES language, based on communicating parallel processes, includes primitives for

process flow and parallel process interactions such as resource contention, process synchronization, and process communication and adaptation [3,10, 13].

An agent is a CAS component capable of perceiving and acting on its own behalf, and it decides for itself what needs to be done to satisfy its own design objectives [2, 13, 14-15, 19]. Agent operation is modeled using the OpEM language as a collection of communicating process instances that perform all agent functions. Each agent decision-making function is implemented using a classifier system block [12-13].

John H. Holland of the University of Michigan and the Santa Fe Institute (SFI) [19] has promoted the idea of a classifier system for many years. Holland's classifier system receives input messages from the environment. Each of these messages is the output of a binary detector (0/1) that indicates if an environmental or system condition exists or not. The classifier system next applies condition-action rules of the form "IF (1#####) THEN (1000)" where (1#####) represents the state of all conditions and (1000) represents the state of all action messages. The "#" is a "don't care" which means, given condition set (1#####), if condition one was detected (has a value one) then the rule applies regardless any of the other conditions. However, a rule of the form "IF (10101) THEN (1000)" means that this rule applies only if all conditions have proper values. The first rule is much more general than the second rule because fewer conditions are required for it to apply.

Associated with each rule is a BID value that can be a function of all condition message confidences, overall rule confidence, and whether the rule is general or specific. When several rules apply in a particular condition context, the highest BID value is used to decide which rule actually fires (posts action messages).

Rules are adapted two ways: (1) when a payoff is received from the environment the chain of rules that fired leading to the payoff are appropriately rewarded or punished and (2) strong rules are recombined and mutated using genetic algorithms to produce new rules. Rules are rewarded by increasing their rule confidence values and punished by decreasing them. Thus, strong rules that are rewarded often have high rule confidence values and

weak rules that are punished often have low rule confidence values. Finally, through rule strength modification and generation of new rules, strong rules persist and weak rules are eliminated from the classifier system.

Westerdale [20] discusses the problem of rule strength modification. He compares the “Bucket Brigade Algorithm” used by Holland and others with another related approach called the “Profit Sharing Plan.” It is shown in the full paper of which [20] is a summary that the bucket brigade algorithm can diverge under some conditions; however, the profit sharing plan always converges if the learning rate is set small enough.

The OpEMCSS simulation library includes a Classifier Event Action block that is a forward chaining, expert system controller. Experiments using the OpEMCSS Classifier Event Action block to learn a sequence of rules that “unlocks” a Finite State Machine and receives a payoff when the final state is reached demonstrated that the profit sharing plan converges when the learning rate is small enough [13].

The OpEMCSS Classifier Event Action block operates conceptually similar to Holland’s classifier system, but there are some differences in implementation. First, the Classifier Event Action block receives input messages that are of the form “AttributeName=RealValue” which are used by EXTEND to describe process instance state variables [10-13]. Holland uses binary (0/1) values for condition messages. Second, the Classifier Event Action block uses rules, as discussed for the part production system in Clymer [12], of the form:

```
Rule1:IF
  W1=very_low,
THEN
  R1=two AND
  R2=zero,CF=100%
```

```
Rule2:IF
  W1=very_low AND
  W2=large,
THEN
  R1=two AND
  R2=zero,CF=50%
```

If an attribute name is not listed in a rule, it is the same as don’t care “#” in one of Holland’s rules. Therefore, rules that have few condition facts are general rules (Rule1), and rules that have many condition facts are specific rules (Rule2), similar to Holland’s rules. The method used to decide which rule fires is similar to Holland’s classifier system; however, rule strength modification is done using the profit sharing plan method described by Westerdale [20].

From rules 1 and 2 above, one can see that when a rule fires, the action messages must be transformed back to the form “AttributeName = RealValue,” which EXTEND requires, before being added to process instance item passing through the block, changing the value of a process state variable [3, 13].

Holland’s classifier system uses a genetic algorithm [15-16,19] to generate new rules based on (0/1/#). The Classifier Event Action block employs an evolutionary rule induction method that involves integer choices among a set of condition or action attribute values in order to generate new rules. The intent in any case is the same; to discover rules during system design that achieves CAS component self-organization and adaptation.

The evolutionary rule induction method discussed in this paper is designed to operate in an environment where the correct decision for each decision situation is not known in advance. Further, each decision situation that requires a decision is also not known in advance. Thus, the classifier system block must first learn what decision situations require decisions and then it must discover the best action to take in each situation. Because of this, graded rule induction, where decision fitness guides the search, is used rather than supervisory rule induction, where the correct decisions are known in advance. Most work in rule induction and neural networks found in the literature has been based on supervisory learning [4-8,18], which provides some insight but little direct assistance in solving this problem. The work done on classifier systems [16,19] provides the direct assistance that is needed here because graded learning based on fitness is used. In order to understand the graded learning problem for agents operating in a co-evolutionary environment, the paper begins with a discussion of the concept of the “situational universe” followed by lessons learned from supervisory induction.

The paper continues with an overview of the evolutionary rule induction method used by the Classifier Event Action block. The concept of situational ambiguity is introduced as a means to effectively control the search for situation decision rules during graded learning. This feature allows the classifier block to learn rules while the set of situations present changes during execution of a plan or in a co-evolutionary environment such as the Prisoner’s Dilemma. Next, the features of this classifier block are described, and it is shown how to set up the block dialogs to model CAS. The Prisoner’s Dilemma model is described as an example of a CAS operating in an co-evolutionary environment. The paper concludes with a summary and discussion of future research.

2. Situational Universe for a Classifier System

The situational universe for a classifier system block is comprised of all possible combinations of the rule condition facts of the form "AttributeName = ValueName" defined in the Conditions dialog for the Classifier Event Action block. The situational universe defines all possible universe elements that can be recognized by the classifier block. A situation is a subset of the universe, consisting of one or more elements, that requires the same decision or action. Several rules, all specifying the same action, may be required to cover all elements of a situation. Clearly, how a system perceives its environment and how precisely it can distinguish what is happening there depends on the condition facts defined.

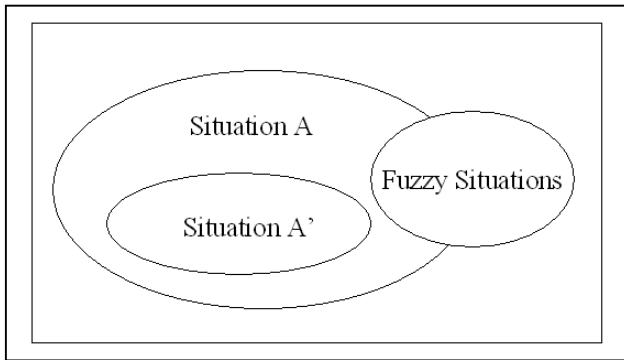


Figure 1. Example Situations

For example, consider situation A elements that surround situation A' elements as shown in figure 1. Situation A' elements require a different decision than situation A. Suppose some rules generated for situation A also cover elements of situation A'. Any such general rule that covers both situation A and A' elements will only be proper for elements of situation A. An improper action for elements A' of the universe will result in the rule being punished when applied to these elements. When the rule is made more specific such that the action is always proper and, thus, always rewarded, it achieves maximum strength. If situations A and A' are disjoint subsets of the universe, the classifier system will always be able to find rules that are proper 100% of the time for both situations.

A situation labeled fuzzy overlaps situation A as shown in figure 1. The universe elements shared by situation A and the fuzzy situation are associated with different actions. These overlapping situational elements are called ambiguous. Ambiguous situations will occur temporarily during rule induction when rules covering the same situation have different actions. Rule strength modification can usually determine which is the best action for a situation such that only the best rule persists. However, when the condition facts available are insufficient to decide what to do, ambiguous situations persist such that several fuzzy rules cover a situation and

have different actions. The fuzzy rule having the highest BID strength is selected to post its action.

In summary, the evolutionary rule induction problem has two parts for each situation: (1) to find the smallest and most general set of non-fuzzy rules that cover the situation and (2) to determine the required action (or actions) necessary to achieve the classifier system's goals. The search for these rules starts at the top of a rule network, as shown in figure 2, and proceeds down the network hierarchy until the least fuzzy rules are found. The search must be focused (called inductive bias) so that all combinations of condition facts and action facts (1680 potential rules or hypotheses for the part production system [12]) are not required to be considered in order to discover the best rules to cover all situations that require an action decision.

2.1 Lessons Learned from Previous Research

The type of expert system used in the OpEM Pascal Simulation Programming System was backward chaining. The Expert System Controller, as it was called, was applied in a series of experiments to study CAS: (1) a visual flying rules (VFR) air traffic control system where the aircraft are intelligent agents [5,8], (2) a railroad system where the trains are the intelligent agents [6], (3) a sonar array system where the sonar nodes are the intelligent agents [7], and (4) a vehicle traffic grid where the traffic light controllers are the intelligent agents [9].

In each of these experiments, the same lessons were learned. There was always a critical collection of rule condition facts that were shared among the agents in order for the overall system to achieve its goals. Further, finding the rules manually to control the system was usually very difficult and time consuming; especially for these highly context-sensitive systems where agents share feature facts in order to collaborate.

For example, the train system is very context-sensitive with the propensity to deadlock if trains are not moved in the right order. For the OpEM Pascal Train simulation, cases had to be decided manually in order to produce a case file of proper decisions to guide rule induction. A case is an element of the situational universe plus the correct decision fact. This proved to be very tedious and time consuming.

Recent experiments with an OpEMCSS train model, that uses the Classifier Event Action block, have shown that some rules are generated that do not allow the trains to move even when there is no conflict. These rules can also cause deadlocks. The problem here seems to be providing the proper rule reward when there is no alternative way to recognize situations where train waiting is proper and when it is not. The solution to this problem is probably to constrain the rules in clear-cut decision situations and allow the system to learn the best decision rules in the fuzzy situations (future research).

The sonar array system was too difficult for even an expert to find the rules; although, the expert did specify a **maximal** set of condition facts. An off-line induction program was created for the OpEM Pascal Simulation System that determined a minimal set of rules that covered all of the decision cases correctly, using only a small subset of the condition facts specified.

Recent experiments using an OpEMCSS SonarNet model, that uses the Classifier Event Action block, produced a similar set of rules that decides 100% of surface ships and 100% of submarines correctly. However, the sonar-array learning problem is much easier than the train system problem. The train system problem requires a graded learning method because the correct answer is not known whereas the sonar array can use supervisory learning where the correct answer is available to reward each decision (we know the correct classification in advance).

One thing in common with all of these experiments, regardless of the learning method used, is that if a **maximal** set of available condition facts is defined for the induction process, a critical subset of required condition facts emerges, becoming conditions in the induced rules. The system design strategy, implied here, is for agents to share a maximal set of situational feature facts in the beginning in order to determine the subset of facts required for the rules.

2.2 OpEM Supervisory Induction Algorithm

A supervisory induction algorithm was used in each of the OpEM Pascal Simulation experiments, discussed above, to generate a set of system decision-making rules. This induction program was designed to generate rules off-line for the backward chaining expert system controller. Unfortunately, the algorithm applied is not appropriate for generating rules on-line for a forward chaining classifier system that uses graded learning. However, this induction program has a very effective way to focus the search for rules that may provide some useful insight into the on-line, graded learning induction problem.

The OpEM off-line induction program receives as input a case file generated by an OpEM discrete event simulation program. Each case consists of a single decision action fact plus a set of all condition facts available. A case is comprised of a single element of the situational universe that is contained in an overall situation, combined with the required decision action. Cases are searched for associations between condition facts and alternate decision action facts. The algorithm counts the number of times each condition fact is associated with each decision action fact and the number of times it is not associated with other alternate decision action facts. Condition facts having relatively higher

counts are better classifiers because the rules formed using them tend to be more general and less fuzzy.

For each decision action fact a set of simple hypotheses, where each hypothesis is a potential rule, is formed. Each hypothesis relates an action fact to one of the better condition facts. If a hypothesis is consistent, then it covers only positive cases and no negative cases. A positive case belongs to the situation for which we are trying to find rules. A negative case belongs to any other situation having a different decision action. The best consistent hypotheses, the ones that cover the most positive cases as compared with other consistent hypotheses, are accepted as rules.

If not all cases are covered by the accepted rules, combining the better condition facts forms other more specific hypotheses. Each new hypothesis is tested to determine if it is an acceptable rule. This top-down combinatorial search for rules continues from general to specific until all cases are covered or until all condition facts have been considered in various combinations.

If some cases remain uncovered but all condition facts have been considered, ambiguous situation elements exist that can-not be crisply classified given the available condition facts. These remaining cases are covered by selecting a hypothesis as a rule, which best covers these ambiguous elements, from all the hypotheses rejected previously.

The important point here is that the off-line search for the best rules is effectively focused through the use of the best classifier facts in each rule condition, resulting in the most general and least fuzzy rules being found to cover all decision situations. The off-line induction program uses supervisory learning where the correct decision is known for each case, as discussed above. Because the correct decision is known for each case in the training set, it is possible to analyze these cases and determine the best classifier facts. Using the best classifiers first in the search for rules, results in a very effective inductive bias. The Classifier Event Action block uses graded learning where the affect of using hypotheses to make system decisions are evaluated over time, and feedback regarding the effectiveness of each hypothesis is often delayed. The classifier block tends to generate top level, single condition hypotheses first that may provide information regarding the best classifier condition facts. If bad rules are eliminated quickly, only good rules with the best classifier conditions are left to build on.

2.3 Theory of Inductive Learning

The situational universe is defined as the set U of all possible combinations of available condition facts. A situation S is a subset of U , a collection of related elements from this universe. Each element in set S is associated with the same correct decision (one or more required actions). A single element of the universe combined with the correct decision is called a case, as

discussed above. Given a randomly selected element u from U , the problem is to decide if u is contained in S . A set of rules must be found to decide if an element u is contained in S or not, as discussed above.

For off-line rule induction, training set $Q = P + N$ is a randomly selected subset of positive cases P from S of the situation to be decided plus other negative cases N contained in U and not contained in S . Off-line induction requires that the correct decision be known in order to discover the best rules. On-line induction does not have this information provided, and it must infer the best decision through rule strength modification provided through graded learning. In other words, a set of eligible rules competes to determine what is the best decision for each situation that requires a decision. Thus, the off-line induction algorithm, discussed here, is an example of supervised learning where the correct answer is provided for each case, and on-line induction is an example of graded learning where the result of using a decision is evaluated and a grade is provided for each case either now or later. Either way, there is a correct decision action associated with each situation.

Hypothesis space H is the set of all possible hypotheses (expressed as rules) defined on U and the set of all possible associated decision actions. A hypothesis is consistent with an unambiguous situation S if it covers cases in P and does not cover any cases in N . In off-line induction we have no trouble determining if a hypothesis is consistent since we know the correct decision; however, in on-line induction consistency is determined gradually over time as the hypothesis is evaluated each time it is used. As discussed above, if a situation is ambiguous, then there are no consistent hypotheses.

Version space is the subset of all hypotheses contained in H that are consistent with situation S . Maximally specific hypotheses HS are consistent and there is no hypothesis more specific and consistent. The dual set HG contains hypotheses that are consistent and there is no hypothesis more general and consistent. Version space is defined by subsets HS and HG . If a subsequent positive case is added to training set Q , hypotheses in subset HS may become more general. If a subsequent negative case is added to training set Q , hypotheses in subset HG may become more specific. As cases are added to Q , version space is reduced until either empty or until only hypotheses defining S remain ($HS=HG$). If version space is empty, situation S is not definable by H or is ambiguous. Practical algorithms to reduce version space only work where all cases in situation S are covered by a single conjunctive (AND operator) rule, however. Usually, situations require two or more conjunctive rules to cover all cases contained in subset S of universe U . The choice of rules allows for disjuncts (OR operator) in situation S

Inductive bias is the means by which one consistent hypothesis is chosen over another. Bias results if the shortest or the simplest hypotheses are preferred (Occam's razor). Both the off-line and on-line induction algorithms discussed here use the Occam's razor bias to focus the search to find the simplest rules. As discussed above, off-line induction also uses the best classifier facts first to focus the search. Another bias is to focus rule search toward each situation one at a time. Off-line induction does this by considering only cases for a given decision, and OpEMCSS on-line induction does this by only considering the modification of rules eligible for situation S in order to find better rules. If only rules having the best classifiers persist for situation S then the best overall rules are generated. Average situational ambiguity shuts off the search for rules for each situation when the best rules are evident, as discussed later in the paper.

Hypothesis error is the probability of drawing a random element u from U where the hypotheses and correct decision disagree. If version space could be reduced until exhausted, either empty or containing only hypotheses defining S , the hypothesis error would go to zero. Exhausting version space may require infinite examples in set Q and set HG may expand exponentially as Q increases, making it computationally infeasible [18]. However, it is possible to draw enough examples "to probably almost exhaust version space." An inductive bias is used to minimize hypothesis error such that hypotheses and situation differ only by a small set of cases that rarely occur given the real world probability of occurrence of instances [18]. The idea of using the best classifiers first appears a good bias to probably almost exhaust version space.

Situations can overlap where an element u of universe U has several different correct decisions associated with it. Such an element is called ambiguous, as discussed above. Evaluating the strength of evidence for each alternative decision and selecting that decision having highest strength can resolve the ambiguity that results from such elements. If all rules eligible for situation S have about the same strength, the ambiguity associated with selecting a decision is high. If one of the eligible rules has a much higher strength than all the other eligible rules, the ambiguity is low. We use a such an average situational ambiguity measure based on these ideas to control the search for new rules in the Classifier Event Action block. A future modification of the Classifier Event Action block is to compute the amount of ambiguity in situation S by comparing the BID values for all alternative decision actions and adding an attribute to a process instance state variables.

2.4 Off-line Versus On-line Induction

In off-line induction, a set of cases is analyzed to find the minimum set of rules that covers all cases where the correct decision is known for each case in advance. In on-line induction, elements u of universe U are encountered by the induction process one at a time, such as in the reduction of version space discussed above. Methods have been devised to incrementally generate rules on-line that reduce hypothesis error as the cases are received, but these methods do not work well when the cases are fuzzy [18].

The Classifier Event Action block, discussed next, generates new rules by going top-down from very general to more specific hypotheses as shown in figure 2, and it makes use of all of the lessons learned using the off-line induction theory [4-8, 12, 18] and Holland's classifier system [19] experience, discussed above. In particular, the concept of situational ambiguity is very useful when focusing the rule search for situation S in order to terminate search when the best rules have been found; further, exploring single condition rules first seems to provide information about the best classifier facts needed to build the best rules.

3. Evolutionary Rule Induction Overview

Figure 2 shows a hierarchical rule network for the part production system discussed in Clymer [12-13]. There are a total of 1680 possible rules using various combinations of condition and action facts.

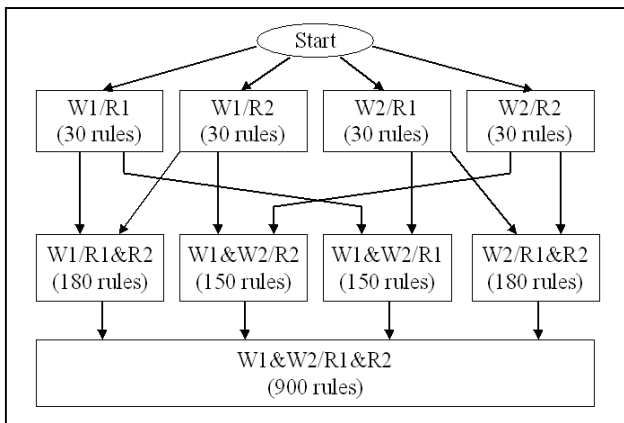


Figure 2. Hierarchical rule network.

For example, the following rule is one of the 30 possible rules included in the W1/R1 box.

```

Rule3:IF
  W1 = very_low,
THEN
  R1 = zero, CF=100%
    
```

Rule induction begins with a subset of these 30 rules as the initial knowledge base such that each possible value

for condition fact W1 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 selects a set of rules for modification, as discussed above, based on average situational ambiguity for each rule, which is high when all eligible rules for a situation S compete and becomes low when one rule dominates. 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 the rule network, shown in figure 2, 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 network. The result is that a minimal set of the most general rules that can make all decisions correctly is found. Because of this inductive bias implemented in the evolutionary rule induction algorithm, only a small fraction of the 1680 rules are examined.

4. Classifier Event Action Block

The 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 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. If several different actions are implied by the condition attributes (i.e., several rules are eligible to fire for a situation S), the best action is selected based on either the maximum BID value or a probability. The BID is a function of rule strength (confidence factor CF), 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.

Process attributes of the form "AttributeName = RealValue" must be transformed into a symbolic form "AttributeName = ValueName" suitable for the inference algorithm. To do this, a range of numerical attribute values is associated with each symbolic value name. A LegalConditionVals command having the following format must be placed at the beginning of the rule file to accomplish the required transformation:

```
LegalConditionVals(AttributeName)=
  ValueName1(Low1:High1),...,
  ValueNameN(LowN:HighN)
```

The Low and High values define the range of numerical values allowed for each symbolic value name. The result of each transformation is a set of one or more facts of the form "AttributeName=ValueName" that can be used by the inference algorithm.

Further, a LegalActionVals command having the following format must be placed at the beginning of the rule file to accomplish the required reverse transformation when a rule fires:

```
LegalActionVals(AttributeName)=
  ValueName1(Low1:High1),...,
  ValueNameN(LowN:HighN)
```

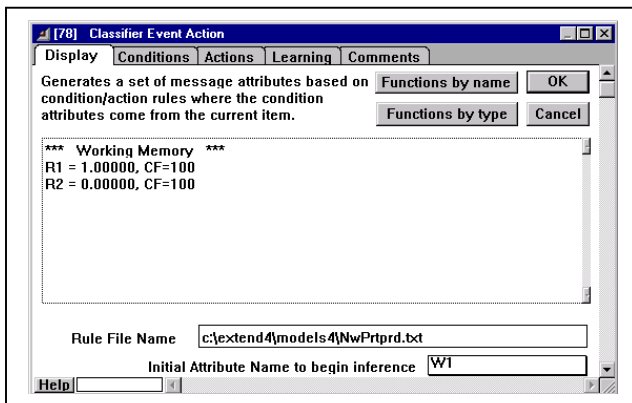


Figure3. Display dialog.

The Low and High values define the range of numerical values allowed for each symbolic value name. For crisp rules a single value is specified; however, for fuzzy rules (not discussed in this paper) a range of values must be specified. The result of each transformation is one or more process attributes of the form

"AttributeName = RealValue" added to the process instance item passing through the block.

An example rule is as follows:

```
RuleName:IF
  ConditionName1 = ValueName1 OR
  ConditionName1 = ValueName2 AND
  ConditionName2 = ValueName3,
THEN
  ActionName1=ValueName4 AND
  ActionName2=ValueName5, CF=100.0%
```

The condition of each rule must be in product of sums format "((A or B) and (C or D))". For crisp logic rules, there is no problem having disjunctions "(A or B)" in the condition; however, for fuzzy rules disjunctions may not work correctly. The problem is that in a forward chaining inference process, the first true condition fact may cause a disjunctive rule to fire. The condition support for the rule may be greater if we wait for all condition facts to be considered, but there is no way to know when to wait.

If a rule name includes "constraint," the rule always wins the competition. Further, the CF is always 100 and is never changed by learning. There are several uses for "constraint" rules. First, constraint-rules impose constraints on actions that must not be changed by learning in order to place limits on system behavior. Second, in learning situations where some contexts are far more frequent than others, constraint-rules allow rules to be learned in stages. Rules generated for contexts already covered by constraint rules are quickly eliminated so that rule search can find rules for rare contexts.

4.1. Dialog Choices

4.1.1. RuleFileName. Defines the rule file path and name. "C:\Extend5\Models5\NwPrtrpd.txt" is an example, as shown in figure 3.

4.1.2. InitialFact. The forward chaining inference engine requires an attribute name to begin the reasoning process. Select an attribute name that is always found in the process instance items passing through this block and in the condition of at least one rule.

4.13. Condition Names. Up to 16 condition attribute names to be used in rules can be specified in the condition dialog.

4.14. Action Names. Up to 16 action attribute names to be used in rules can be specified in the action dialog.

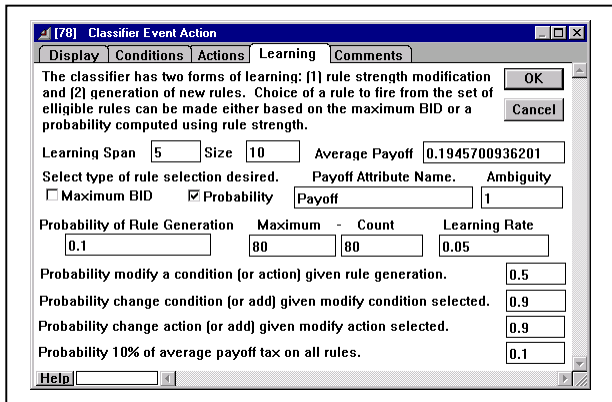


Figure 4. Learning dialog.

4.1.5. LearningSpan. If LearningSpan is greater than zero, a Classifier Event Action block can learn through rule strength modification and generation of new rules. A classifier block receives a reward payoff message from a Reward Event Action block, as shown in figure 5. Such a message contains the payoff attribute name, a process identifier, and the payoff value. If the payoff name and process identifier in a message are proper for a classifier block, a rule sequence is rewarded using profit sharing with penalty. A linear decay is used to reward rules in the rule sequence in memory. Profit sharing decay mitigates learning noise problems that cause good rules to become extinct through bad luck [20]. Another way to reduce learning noise is to make the LearningSpan as small as possible and still achieve the delayed payoff required to reward a responsible rule.

4.1.6. MemorySize. The MemorySize should be set to the learning span times the maximum number of concurrent process instances expected. Because a process identifier is part of the reward message, many concurrent process instances can be rewarded simultaneously. Each process instance has an independent rule sequence in the memory of length LearningSpan. Indeed, this feature reduces all of the learning noise that would occur if all process instances were combined into one rule sequence.

4.1.7. MaximumBID. If the MaximumBID check box is checked, rules are selected on the basis of maximum BID strength. If MaximumBID is checked, Probability can-not be checked.

4.1.8. Probability. If the Probability check box is checked, rules are selected based on a probability computed using rule strength and specificity. If Probability is checked, MaximumBID can-not be checked.

4.1.9. PayoffAttributeName. The Reward Event Action block broadcasts a reward payoff message to all classifier blocks in a model. The message contains a

payoff attribute name, which is compared against the PayoffAttributeName specified in the learning dialog. If the names compare, then the classifier block can accept the message.

4.1.10. AmbiguityThreshold. If all rules eligible for situation S have about the same strength, the ambiguity is high. If one of the eligible rules has a much higher strength than all the other eligible rules, then the ambiguity is low. Situational ambiguity for a rule is averaged over all situations where the rule is eligible and rule modification occurs. The probability of selecting a rule for modification decreases as average situational ambiguity decreases. The ambiguity threshold specifies the number of rule modifications before calculation of average situational ambiguity begins. The ambiguity threshold can have a value from 1 to 1000.

4.1.11. PrbRuleGen. This probability parameter is used to decide if rule generation will occur when a payoff message is received, accepted, and the fired rule for the situation is rewarded. Rules are selected for modification uniformly based on average situational ambiguity. Rule generation proceeds from general to more specific rules, as discussed above.

As discussed above, an average situational ambiguity measure is computed, based on all eligible rules for a situation, and used to control the search for rules. Using the ambiguity cutoff, the search for rules is focused on situations where the ambiguity is high. This strategy for selecting rules allows a sequence of planning rules to be discovered rather than just stimulus-response rules, as discussed above.

4.1.12. MaxRuleCount. The MaxRuleCount parameter controls the maximum number of rules generated in a run. Once all rules have been generated, rule strength modification eliminates weak rules, retaining a minimal set of the most general rules that can make all decisions correctly.

4.1.13. LearningRate. LearningRate, sigma, controls the amount each rule is rewarded and penalized. If the LearningRate is too high, good rules can be lost due to bad luck. If the LearningRate is too low, learning is slow. Further, LearningRate must be reduced as LearningSpan is increased to mitigate the affects of learning noise. Finding the correct LearningRate (and LearningSpan) is a trial and error process, but it is critical that the correct LearningRate and Learning Span be determined. A similar problem occurs in neural network learning in that the learning rate and number of hidden layer nodes must be just right.

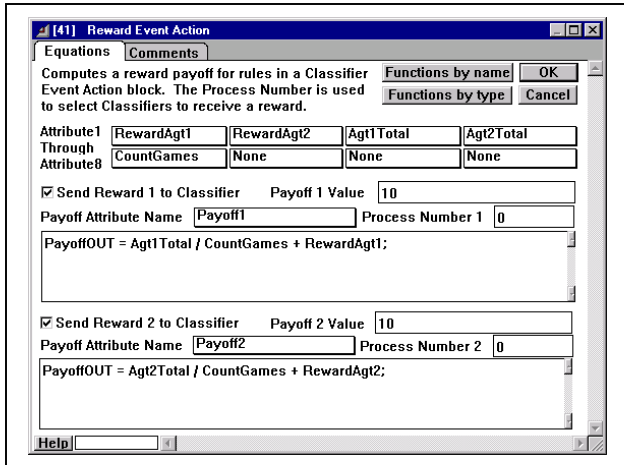


Figure 5. Reward Event Action block dialog.

4.1.14. **PrbModCondition.** Given that rule generation has been selected, this probability parameter is used to decide if a condition or an action will be modified.

4.1.15. **PrbChgCondition.** Given that “modify a condition” has been selected, this probability parameter is used to decide if a condition fact value name will be changed or if a condition fact will be added or deleted. Given that add/delete has been selected, a probability function is used such that initially the probability of

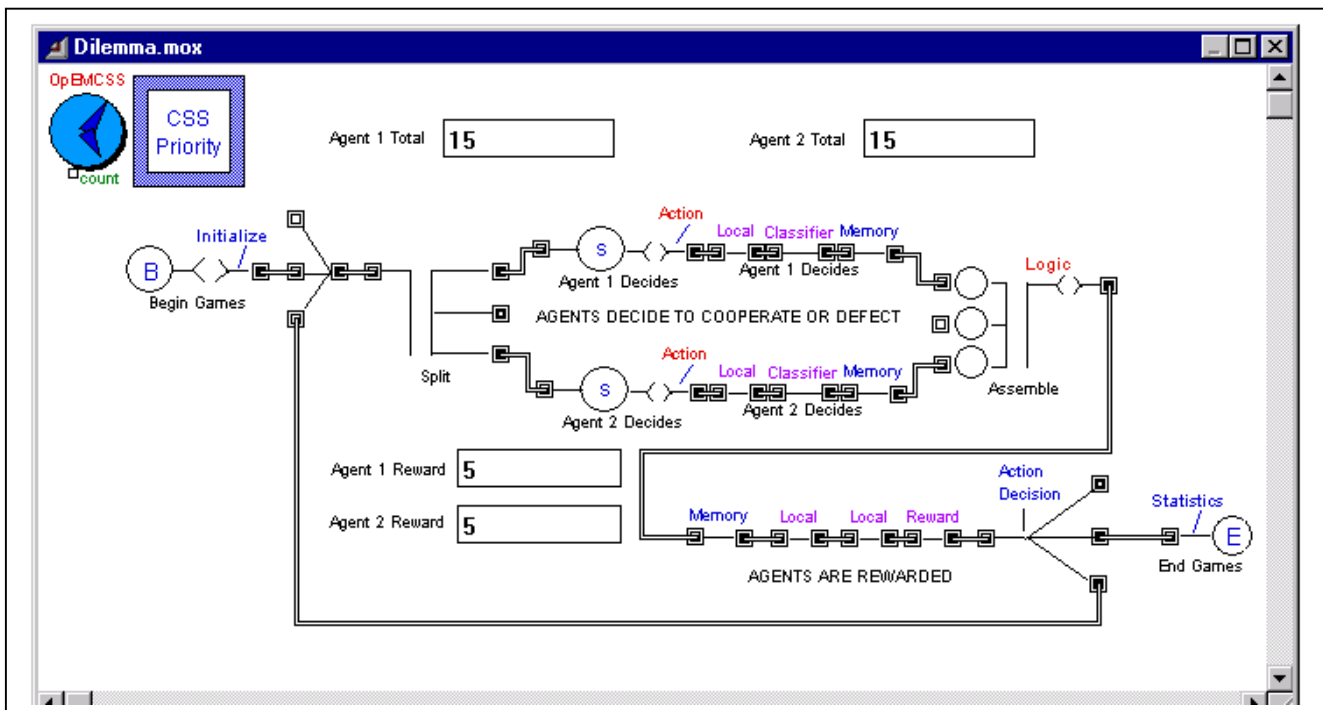
the top of the rule network, shown in figure 2, to be more fully explored and evaluated. The selection of which condition fact to change is uniformly distributed.

4.1.16. **PrbChgAction.** Given that “modify an action” has been selected, this probability parameter is used to decide if an action fact value name will be changed or if an action fact will be added or deleted. Given that add/delete has been selected, a probability function is used such that initially the probability of adding is zero, but it increases exponentially to a value of 0.9 as rules are generated. The selection of which action fact to change is uniformly distributed.

4.1.17. **PrbRuleBaseTax.** If a rule is generated that matches no contexts, the rule is never eligible and thus never fires. There is no way to eliminate such a rule through rule strength modification discussed above. Therefore, a tax on each such rule of ten percent of average payoff is levied randomly based on the PrbRuleBaseTax. As the eligibility of a rule increases, the probability of applying a tax to the rule decreases linearly. The value of this probability must be selected so that good rules, that are delayed in being eligible, are not eliminated along with the useless ones.

5. Reward Event Action Block

The reward event action block is used to compute a Classifier Event Action block reward payoff value based



adding is zero, but it increases exponentially to a value of _____ on an equation that can be a function of up to eight

Figure 6. Prisoner's dilemma process.

The payoff attribute name specified in the dialog is sent in the payoff message. Classifier blocks with the message attribute name equal to "PayoffAttributeName," specified in the Classifier block Learning Dialog, accept the message.

The process number specified in the dialog and duplicate process numbers for the process item passing through the block are sent in the payoff message. If the process number received in the message is zero or duplicate process numbers received are all zero, all process instances are rewarded. Otherwise, only process instances having the proper process identification are rewarded.

6. Prisoner's Dilemma Model

The Prisoner's Dilemma model [1] is shown in figure 6, and it is comprised of OpEMCSS library blocks. 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, and initializes some state variables, called attributes in EXTEND, that are of the form "AttributeName = ReallValue." 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). Also 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 model is executed for multiple runs in order to compute Measures of Effectiveness (MOE) statistics.

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 6 consist of a Reaction Time Event block followed by 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 and also sets "PastX" equal to "AgentX," for sub-process X, to remember the agent's previous play.

The Classifier Event Action block contains a forward chaining, inference algorithm 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. 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." The Memory Event Action block and the Message Event Action block allow process instances to communicate either through global memory or message passing; respectively. A third option in OpEMCSS is to use a set of blocks to send and receive message items in a hierarchical functional flow model.

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, dialog shown in figure 5, computes the payoff for each of the Classifier Event Action blocks.

The classifier blocks always converge to the following rules:

```
Agent1:
```

```
Rule1:IF
```

```
Agent1=Cooperate AND
Agent2=Cooperate AND
Past1=Cooperate,
THEN
Agent2=Cooperate, CF=100%
```

Agent2:

```
Rule1:IF
Agent2=Cooperate AND
Past2=Cooperate AND
Agent1=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, the agents play a “tit-for-tat” strategy [1]. 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.

7. Summary

During the last 350 years, scientists have strived to discover simple laws and fixed equations to predict various outcomes in natural systems. Examples include celestial mechanics to predict the position of the planets in the sky throughout the year, thermodynamics to predict the properties of gases in a closed volume, and relativity theory to establish the relationship between energy and matter. The result of such discoveries has been the industrial revolution that has provided us with abundant material goods, electric power generation that has energized our work and our leisure, advances in medical practices that has freed us from illness, and electronic devices that has allowed us to automate mental work and communicate with other people on a worldwide scale.

All of these developments were driven by a set of fixed laws that lead to the development of mechanistic systems where each system component does not change. The same input to a mechanistic system component always produces the same output. It was said during this time that nature

could be understood as a clockwork universe where everything was thought of as a mechanism or machine. In order to simplify calculations, natural systems were assumed to be linear such that the Linear Superposition Rule held. If X is the input, Y is the output, and $Y = F(X)$, then input $X1+X2$ implies output $Y1+Y2$. Thus, given any input to a mechanistic system the output is predictable. Based on the mechanistic system view, it was possible to reduce a system to its constituent components and understand the operation of each component in order to understand the whole system. This way of thinking about systems was called the reductionist-mechanistic paradigm.

During the 20th century the mechanistic system view began to be questioned for systems that did not fit the above assumptions. A Complex Adaptive System (CAS) is a network of agents where each agent is intelligent and adapts its behavior in response to messages received from other agents or perceived changes in environment [2, 13-15, 19]. Such an agent is no longer a mechanism and the same input does not necessarily produce the same output. Agent interactions create causal paths through space and time that transform agent behavior and produce emergent system behaviors. Emergent behaviors are a property of the whole system and not a property of any individual agent or proper subset of agents [13].

Because of the causal paths, CAS operation can be non-linear and often appears chaotic. In non-linear systems, small changes in initial conditions may result in large changes in agent and system outputs, making prediction of the exact system state trajectory impossible. However, it may be possible to predict emergent system properties based on the form of agent decision rules. The idea is that if certain agent rules are always associated with a particular emergent behavior, then maybe we can predict that the emergent behavior will occur if these rules are present. This is the inductive approach to developing scientific laws of complexity rather than the deductive approach that mathematicians use.

Therefore, the reductionist-mechanistic paradigm must be modified to deal with complexity. Systems design and analysis of CAS must be performed using a simulation of the whole system, even as the system is reduced to a component hierarchy. The only way to provide valid inputs to an agent is to let the system generate them during system operation; further, the simulation must include the environment and the operation of the system in each operational situation or context in order to validate that system requirements have been met [13].

Other properties of CAS are: (1) a random component must be added to the search for better decision making rules in order to cover more of the solution space, (2) a fitness MOE criterion must always be present that identifies best behavior for each agent in order to guide the search for better rules, and (3) CAS are always

balanced between order and disorder, operating on the edge of chaos such as shown in the vehicle traffic grid simulation discussed in this paper. Further, experience has shown that simple agent rules can lead to complex overall system behavior. Agents in the CAS network self synchronize (interactively reach a consensus as to the best system operation) with each other to produce emergent system level properties and an overall system level identity that cannot be produced by any agent alone.

Multi-agent system behavior, especially CAS behavior, can be complex and difficult to understand. Further, it is currently unknown if the emergent behavior of a complex adaptive system can be predicted from its system description. However, the underlying graphical modeling language (OpEM) that is used to describe system operation, the OpEMCSS library blocks that implement this language, and the system design and development methodology that is used to design and build models are simple. The ease of use of the OpEM methodology has been demonstrated through the many example models discussed in Clymer [13]. Further, these building blocks have been validated by 50 years of systems research and practice by engineers, scientists, and business people.

During the last few years, simulation-based engineering has become an essential tool for the design and evaluation of complex systems. Such systems include air and ground transportation networks, military C4ISR multi-agent systems, and complex, international business organizations. These systems are difficult to understand because each agent (subsystem) in the network adapts its behavior in response to knowledge received from other agents. Agents adapting in collaboration with other agents can lead to emergent behaviors that are much more complex than any agent could achieve alone. Such Complex Adaptive Systems (CAS) have the potential of much greater capability and effectiveness than traditional systems if designed properly using the OpEMCSS graphical simulation library discussed in this paper.

The vast majority of people possess, as common sense, the fundamental ability to understand such complex systems. Each one of us formulate goals for ourselves all the time to solve problems confronting us. Once we state our goal, we visualize a set of tasks or steps that takes us from our current situation to one satisfying our goal. If we can expect help from others to execute these tasks, we organize the tasks into a sequential and concurrent arrangement, more commonly called a plan. The execution of our plan by a group of people hopefully leads to goal satisfaction. The OpEM system design, analysis, and evaluation methodology has been built on these natural thought process in order to understand complex systems through simulation studies.

The revolution in complex adaptive systems is here, but it is just beginning. CAS are not well understood as of yet; although, systems engineers have been cognizant of

some of the attributes of complex systems for a long time. What we need is a simulation tool that is easy to use by the majority of CAS researchers and the broad spectrum of other interested people, all of whom have diverse backgrounds. The tool must describe complex system principles and facilitate understanding of intelligent, multi-agent systems and CAS. We also need a tool that can be easily extended as more is learned about the nature and phenomena of complex systems. I believe that OpEMCSS can be that tool. You can download everything you need from my website <http://www.ecs.fullerton.edu/~jclymer/> in order to evaluate OpEMCSS and decide for yourself.

8. References

- [1] Axelrod, R.M., *The Evolution of Cooperation*, New York, NY: Basic Books, Inc., 1984.
- [2] Axelrod, R.M., *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*, Princeton; Princeton University Press, 1997.
- [3] Clymer, J. R., *Systems Analysis Using Simulation and Markov Models*, Englewood Cliffs, NJ: Prentice-Hall Inc, 1990.
- [4] Clymer, J. R., "System Design Using OpEM Inductive/Adaptive Expert System Controller", IN *IASTED International Journal of Modeling & Simulation*, Volume 10, Number 4, 1990, pages 129-136.
- [5] Clymer, J. R., Corey, P. D., and J. Gardner, "Discrete Event Fuzzy Airport Control", IN *IEEE Transactions on Systems, Man, and Cybernetics*, Volume 22, Number 2, March-April 1992, pages 343-351.
- [6] Clymer, J. R., D. J. Cheng, and D. Hernandez, "Induction of Decision Making Rules for Context Sensitive Systems", IN *Simulation*, San Diego, CA: The Society of Computer Simulation International, September 1992 issue, Volume 59, Number 3, pages 198-206.
- [7] Clymer, J. R., P. D. Corey, and H. Bandukwala, "Induction of Classification Rules From Noisy Sonar Features", IN *Simulation*, San Diego, CA: The Society of Computer Simulation International, April 1994 issue, Volume 62, Number 4, pages 256-267.
- [8] Clymer, J. R., "Induction of Fuzzy Rules for Air Traffic Control", IN *Proceedings-1995 IEEE International Conference on Systems, Man, and Cybernetics*, Vancouver, British Columbia, Canada, October 22-25, 1995, pages 1495-1502.
- [9] Clymer, J. R., "Expansionist/Context-Sensitive Methodology: Engineering of Complex Adaptive Systems", IN *IEEE Transactions on Aerospace and Electronic Systems*, April 1997 issue, Volume 33, Number 2, pages 686-695.
- [10] Clymer, J.R., "Simulation-Based Engineering of Complex Adaptive Systems," In *Simulation*, Journal of

- the Society of Computer Simulation, San Diego, CA, Volume 72, Number 4, April 1999 issue, pages 250-260.
- [11] Clymer, J.R., "Optimization of Simulated Systems Effectiveness using Evolutionary Algorithms," IN *Simulation*, Journal of the Society of Computer Simulation, San Diego, CA, Volume 73, Number 6, December 1999, pages 334-340.
- [12] Clymer, J. R. and D.J. Cheng, "Simulation-Based Engineering of Complex Adaptive Systems Using a Classifier Block," IN *The 34th Annual Simulation Symposium*, Seattle, WA, April 22-26, 2001, pages 243-250.
- [13] Clymer, J.R., *Simulation-Based Engineering of Complex Systems*, Placentia, CA: John R. Clymer & Associates, 2001.
- [14] Epstein, J. M., Agent-Based Computational Models and Generative Social Science, *Complexity*, John Wiley & Sons, Inc., Volume 4, Number 5, pages 41-60, 1999.
- [15] Epstein, J. M., *Growing Artificial Societies: Social Science From the Bottom Up*, Cambridge, MA: MIT press, 1996.
- [16] Fogel, D. B., *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, The IEEE Press, Piscataway, NJ, 1995.
- [17] Goldberg, D.E., *Genetic Algorithms in Search, Optimization, and Machine learning*, Addison Wesley, 1989.
- [18] Haussler, D., "Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework," IN *Artificial Intelligence*, 36, 1988, pages 177-222.
- [19] Holland, J.H., *Hidden Order*, Addison-Wesley, 1995.
- [20] Westerdale, T.H., "An Approach to Credit Assignment in Classifier Systems," IN *Complexity*, Journal of the Santa Fe Institute, Volume 4, Issue 2, 1998, pages 49-52.