# Genetically Programmed Learning Classifier System Description and Results

Gregory A. Harrison
Lockheed Martin Corp.
12506 Lake Underhill Rd. MP-823
Orlando, FL 32825-5002
+1 407 306 6580
gregory.a.harrison@lmco.com

Eric W. Worden
Gestalt, LLC
3505 Lake Lynda Drive, Suite 115
Orlando, FL 32817
+1 407 581 6390
eworden@gestalt-llc.com

## ABSTRACT
An agent population can be evolved in a complex environment to perform various tasks and optimize its job performance using Learning Classifier System (LCS) technology. Due to the complexity and knowledge content of some real-world systems, having the ability to use genetic programming, GP, to represent the LCS rules provides a great benefit. Methods have been created to extend LCS theory into operation across the power-set of GP-enabled rule content. This system uses a full bucket-brigade system for GP-LCS learning. Using GP in the LCS system allows the functions and terminals of the actual problem environment to be used internally directly in the rule set, enabling more direct interpretation of the operation of the LCS system. The system was designed and built, and underwent independent testing at an advanced technology research laboratory. This paper describes the top-level operation of the system, and includes some of the results of the testing effort, and performance figures.

## Categories and Subject Descriptors
I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods – *representations*

I.2.6 [**Artificial Intelligence**]: Learning

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – *intelligent agents*

## General Terms
Algorithms, Design, Experimentation, and Theory.

## Keywords
Genetic Programming, Learning Classifier System, intelligent agent, bucket brigade, reinforcement learning, evolutionary computation, genetics-based machine learning (GBML), autonomous agent, agent learning, complex adaptive system.


*GECCO'07*, July 7–11, 2007, London, England, United Kingdom.


## 1. INTRODUCTION
The use of Genetic Programming (GP) in a Learning Classifier System (LCS) allows flexibility and automatic tailoring of the rule set to operate in a given environment by using the function calls available in that environment. There are added complexities involved in implementing some of the LCS theory in a GP-oriented system. These include the bidding, specificity, bucket brigade, and rule structure, for instance. A technique to accomplish the crowding algorithm in GP is also described here, which is also useful for GP processing outside of an LCS system. A general description will be supplied, and detailed to describe the elements of the technology.

Other LCS systems were researched before deciding on using GP in a full bucket-brigade implementation. These include LCS with GP as described in [1], as well as the classifier system works of [2], [3], [4], [5], [6], [7], [8] and many others. The GP-LCS system was constructed using a mobile agent-based architecture, to better enable it to operate in a Complex Adaptive System (CAS) type of environment [9]. This system was a multiyear development, and completed the initial phase of implementation and development by the year 2001. This system provides one more facet of LCS implementation techniques with GP rules.

It was originally designed as an intelligent agent learning and performance system to support worldwide maintenance of the Joint Strike Fighter (JSF) aircraft, then in design by the Lockheed Martin Corp. The constructed system was tested by the General Electric Global Research team, and by Lockheed Martin, for performance on sets of abbreviated jobs. These tasks included inventory purchase, adaptation to environmental changes, mobile agent functionality, and tasks exhibiting non-Markovian learning. This testing is described in the Results section. It has been shown to perform learning, in a non-Markovian sense, to optimize its results with respect to fitness and exogenous reinforcement, and to respond to changes in the environment, while creating rule chains for execution of problems in a multi-computer agent-based architecture. If new information becomes available, the system incorporates this information. If a performance dip occurs, the system will try other avenues of system operation to attempt to increase performance.

## 2. STRUCTURE OF THE SYSTEM
The operation of "rule-firing based upon message matching", and performance of the "bucket-brigade" method [10] of rule-chain learning, has proven to be a difficult conceptual task to re-create with genetic programming, because the genetic programs are of

varied shapes, and may individually consist of a single rooted program tree [11], and the implementation of messaging requires changes to adapt to the GP paradigm. The rule has thus been structured to implement this concept, with the system creating a segmented, multi-part rule. Just like the ternary LCS, the GP-LCS rule has an 'if', or antecedent, part and a 'then', or consequent, part. Messages are implemented using matching techniques, such as fuzzy message matching for numerical messages. At least one message must be won in order for the rule to be able to fire. The GP-LCS rules can require multiple message matches to fire, and may post multiple messages.

The specificity characterization of the binary LCS is also provided by the GP-LCS through structural and mechanical specificity. This is discussed in the section on the Fully Expanded Hyperdimensional Notation (FEHN) concept.

The overall learning system is shown in Figure 1, where the learning system is embedded in an intelligent agent infrastructure. There are environmental interfaces that allow the agent to encounter the external CAS, and exchange messages with the external message lists. Exogenous rewards from the environment enter through the environmental interfaces, and are provided to the individual rule that posted the messages on the external message lists that resulted in the reward. There is an internal message list also, and when rules are created, they are forced to reference at least one of the internal or the external message lists in order to fire, and the rules are also forced to post at least one message on either the internal and or the external message list.

The rules are contained in the Population, and are processed using a Michigan-approach system, but involving Genetic Programming (GP) instead of Genetic Algorithms (GA). A full bucket-brigade fitness passing mechanism is used. The rules have associated wallets, as represented by the $ sign next to each rule in the population. Tags are also used, to provide the ability to develop multiple task knowledge programs within a single population. A generation gap is used to control the amount of the population to be replaced.

The population contains a Crib, where newly constructed rules (individuals, children) are placed before incorporating them into the population, after being clone checked to prevent duplicate rules from entering the population. Crowding is performed to help maintain diversity. Children are compared to a set of the most similar adults, and the least fit adult is replaced with the new child. Migrants and mutants are handled in a similar fashion.

Implementing the crowding algorithm with a set of rules that are created using genetic programming requires new techniques to judge the similarity of one rule to another. In this regard, the rules were considered to exist in a hyperspace of all possible rules, with given rule gene graphs viewed as hypersurfaces within the constrained hyperspace of all possible rules. Important alleles were given more influence in the crowding comparison.

Rules that immigrated from other demes are stored in the Immigration Depot until they can be integrated into the population. This allows agents to learn and pass their intelligence on to other agents also tasked with certain problems.

There is a Resource Reservoir that contains the internal fitness function that provides endogenous rewards for the agent in the learning of its job. One fitness function exists for each job to be learned. The Resource Reservoir also contains the raw genetic material consisting of functions and terminals that may be combined into chromosomes, in definition. A function can accept other functions or terminals as arguments, but a terminal takes no arguments. The system uses strong typing, so that only certain functions or terminals may be used as input to a given function. The genetic material is composed of information pertinent to the environment, and new information can be sent to the Resource Reservoir for future incorporation into new rules. New information is also integrated through covering operations, where rules are created to match information on the message boards if the message has not yet been matched with the given rule set.

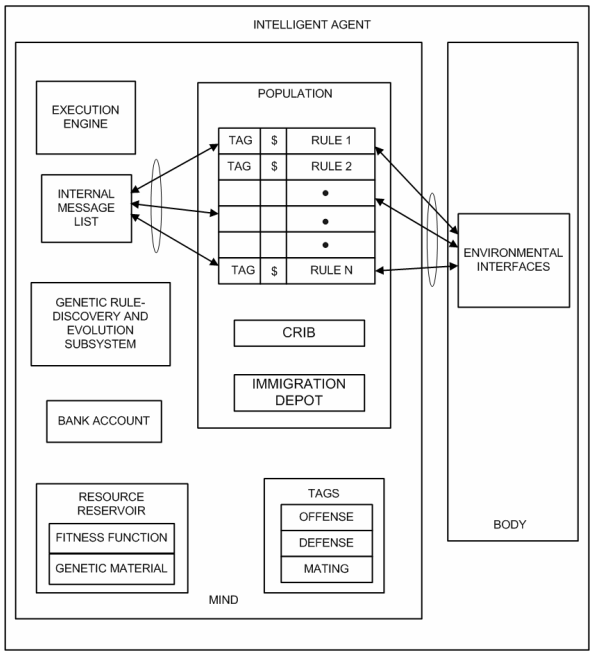


**Figure 1. GP-LCS intelligent agent learning system.**

Money is used for many purposes in the system. It is used to buy information from the message boards, it is increased when payments are received from other individuals for messages they purchased, it serves as a measure of the relative usefulness of this individual to the society, and it is increased or decreased when rewards or punishments are received during the process of learning a job. The system also maintains a bank account, shown in Figure 1, as an overall measure of the agent's strength.

A Bank Account is allotted to the agent as a whole, to help determine best agents to mate in a pseudo-Pittsburgh-type of manner, in an overall multi-agent society. The system uses differential-fitness to let the learning system detect improvements and declines in fitness over a given run at accomplishing a job. Differential-fitness starts at $0.00 at the beginning of a job, and is adjusted by adding all the endogenous and exogenous fitness and rewards, and subtracting the payments and taxes, that accrue during the running of a given job. Differential fitness thus serves as a way to grade the agent with respect to job performance, and without dependence on the total amount of cash that the agent has.

The execution engine controls the operation of the entire agent learning system, and controls when and how the genetic system operates on the rule set. The genetic rule-discovery and evolution subsystem operates on the population after a number of epochs of rule set testing. It selects rules for mating using a form of roulette-wheel selection for the parents, although the diversity is influenced through sigma-truncation [12]. The oscillating sigma-truncation algorithm used here allows the choice of parents to come from mainly the more fit individuals or from a broader set of the population, by adjusting derived fitness values of the individuals based upon the standard deviation of fitness values in the mating set.

## 2.1 Population

A GP-LCS agent contains a population of rules. These rules operate through fitness sharing to create a set of rules that cooperate to accomplish a task. The environment supplies rewards or punishments for various agent actions, and these payments or deductions get sent directly to the particular rule in the particular agent that submitted the message to the external message list, or that performed an operation.

## 2.2 The Rule Structure

There are four gene graphs in the rule, as shown in Figure 2, comprising the internal and external antecedents and consequents. The system grows the antecedent and consequent gene graphs from different sets of genetic material and potential messages that operate in the appropriate internal or external environment.

There must be at least one internal or external message in both the antecedent and the consequent. These messages are chosen from the most aged genetic material in the resource reservoir to help clear messages from the message lists and to encompass the genetic material related to the problem space.

To save processing power, and to make sure that the whole antecedent is considered as one, the internal antecedent (IA) is checked first to see if it will be able to execute at all. The IA is tested to determine: if it is executed, will it return a Boolean TRUE as a result. It is tested as if it had obtained all available messages that it matched, even though it has not yet bid on them.

If the IA can return a valid Boolean TRUE, then the external antecedent (EA) is tested to see if any of its messages match something on the external message board. If either the IA or EA fails the test, then the rule is not executed, saving bids that might have been placed in vain. This rule will lose money on each epoch of LCS through a life tax mechanism. A rule possessing an IA and EA that both pass the tests is considered a runnable rule.

The next step is to bid on all internal messages that are matched on the internal message list by the IAs of all the runnable rules. A bid tax is paid for every bid placed. Each bid includes a small amount of zero-mean Gaussian noise to help break ties during the auction.

After the bids are placed for all the rules, the internal auction house closes the auction. The auction compares all the bids, if two or more rules bid on the same messages, then the rule that had submitted the highest bid wins the message. The winning rules are notified, and they pay their bids to the internal clearinghouse. The clearinghouse subsequently distributes the payment to the rule that had posted the winning messages. The IAs are then run, to obtain the results, yielding Boolean TRUE or FALSE indications for the IA section of each runnable rule.

The EA messages are bid on for all the runnable rules that have a passing IA, also paying a bid tax for each bid. When the EA message auctions close, the rules are notified as to their success, and they pay for their winning messages. Then the EA is run, using the results of the auction. This is the final gate. If a rule makes it through the running of the EA, then it will then execute its consequent.

Both the internal and the external consequents now execute, as indicated in Figure 3. If during their execution, they encounter any messages in their chromosome, then these messages are posted on the appropriate message board. As the agent executes the eligible rules, the reinforcement controller is continually looking for reasons to reward or punish the agent. If, during the execution of a rule, a reward given to the agent, then the reward is tagged with the identification of the rule that was executing. This rule will be given the reward during the rewards state.

The rewards state occurs after all rules have executed. This is when distribution of all reinforcement rewards and punishments, and auction payoffs occurs. Each rule that has a monetary adjustment coming gets the adjustment made to its wallet. Life taxes are also taken at this time. The life tax is a certain percentage of the amount of money a rule has in its wallet.

Each rule gets taxed just for existing. This tax is called lifeTax, or $T_L(t)$, and has a tax rate, $K_L$, with respect to the strength of the rule, as follows

$$T_L(t) = K_L \cdot U(x,t) \tag{1}$$

where $U(x,t)$ is the fitness of rule $x$ at the start of epoch $t$.

The lifeTax rate is set based upon the free-fall half life of the rule's strength, given decreases only attributable to life taxation.

From Richards, 1998 [13], the lifeTax will be set to:

$$K_L = 1 - \left(1/2\right)^{\frac{1}{T_{GP}}}$$

where

$T_{GP}$ = number of LCS epochs between rule-discovery using the genetic programming system.

Rules that continually are taxed, but never receive any money will eventually get replaced during genetics processing. After the rewards have been distributed, the LCS epoch is complete. The age of the rules are all increased by one. When the agent has processed the entire LCS epoch, if more epochs are specified for the current job, the agent will start at the first step of the LCS epoch processing again.

If there were messages on the message boards that were not purchased, the system puts these messages into the respective message depots. As the agent learns, it first pulls any available genetic material from the message depots to create its new rules before creating any random data. Thus, it learns to respond to novel situations occurring in the environment and to link up rules through messages to form a chain. If there were no runnable rules, then a special cover message operation will be performed

where a new rule will be created immediately to allow the system to recognize the current environment.

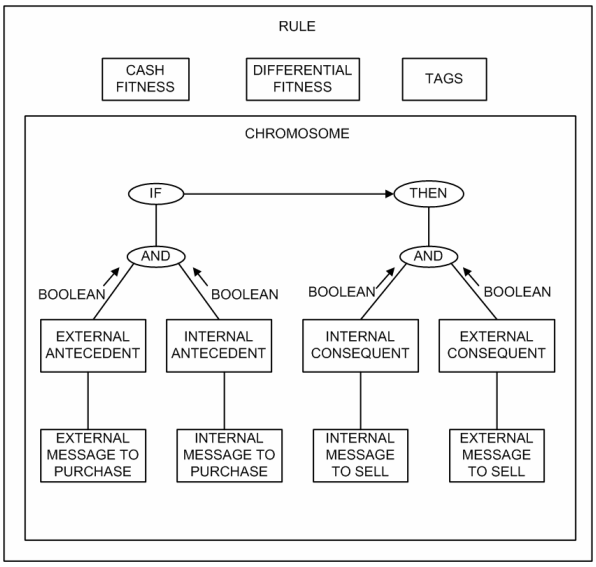


**Figure 2. The rule structure enforces the antecedent/consequent structure.**

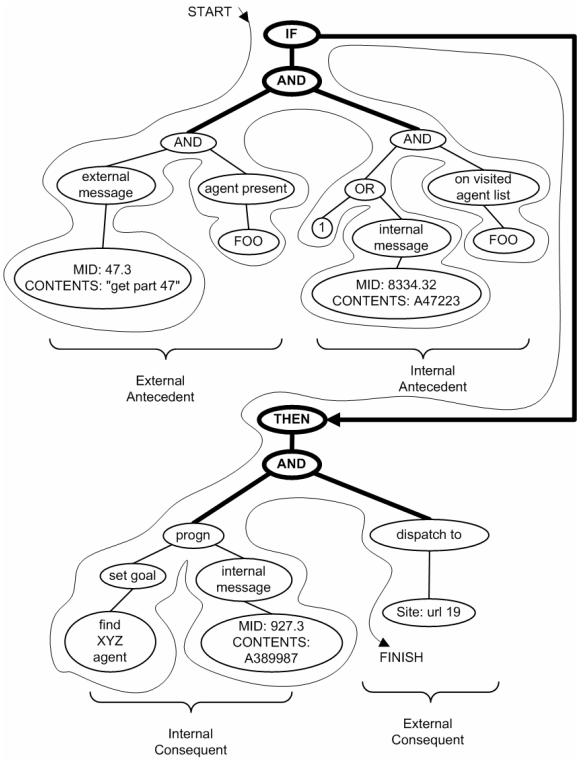


**Figure 3. Execution of a GP-LCS rule begins with the antecedents.**

## 2.3 Message List

The message list, shown in Figure 4, is the primary internal and external communications unit for the agent system. The message list comprises a dynamic group of messages, each of which has a Message Allele. These messages are placed onto the list by an agent or rule, and remain on the list until they are purchased, or for a certain duration, otherwise.

The message contains the identification of the Originator Agent and Rule, so that the agent that put the message on the rule list can get paid when the message is purchased. The Rule ID is also supplied so that the correct rule of the originating agent can receive the payment. The message includes a Born-On Date for use in age determination and operation with algorithms to process old un-bought messages.

The message list contains an Auctioneer that controls all purchases of messages from a message list. As rules bid on the message, the Auctioneer accumulates bids for each message. The auction is a sealed-bid English auction, where the highest bidder wins, and no competitor sees the bid of any other competitor. When the auction closes, the Auctioneer informs the winning rule of the winning agent that it won the message. The message list Clearinghouse collects the money from the winning rule, and sends it to the agent rule that originally put the message on the message list.

## 2.4 Message Allele

Similar to genes, each message contains an allele that takes on some number of values [14]. The Message Allele expresses the substance of the message, which is composed of the Contents and the Message ID. The Message ID is in the form of a number and identifies the contents of the message. The contents of the message are in the form of a String type. These could include "Find Part A009976-A11" or "Sell 1000 Shares of XYZ".

Message alleles are implemented using fuzzy logic techniques. They have a center numerical value and a range band of acceptable values.

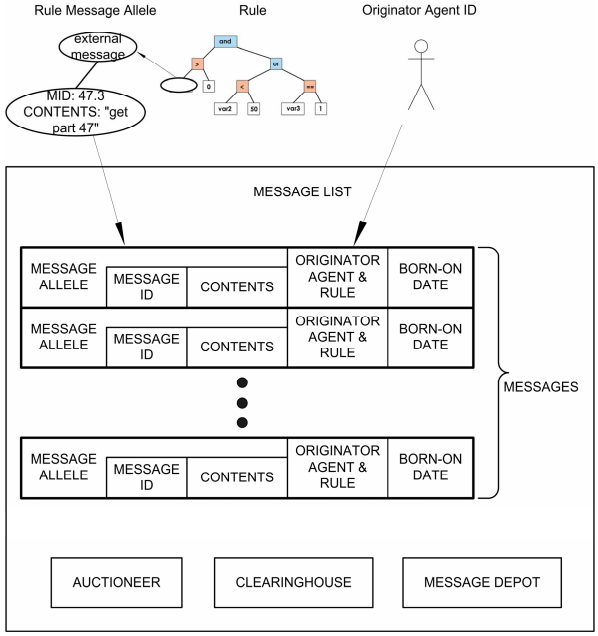


**Figure 4. Messages are placed on message list by a rule.**

The message primitive that is present in a rule can be called the message detector, because it is used to detect if a matching message is available on the associated message board, internal or external. The internal message list uses primarily numerical messages, and the message detectors can examine a number range. If the message number on the list falls within the range of the message detector on the rule, then the rule matches the message and will bid on it. The closer the message is to the middle of the of the message detector range the more that the rules will bid on the message. The external message list uses textual messages, where either a direct match or a theoretical proximity to the message is used for matching the message.

## 2.5 Resource Reservoir

The agent maintains a resource reservoir which holds essential data items and may be dynamic to some extent. The fitness function(s) provide the agent with a reference as to how well it is performing the various jobs that it is either learning or executing. The fitness function is used to help decide between different courses of action, because actions will be selected that yield a higher amount of fitness, expressed as monetary rewards given to individual rules when they have succeeded in accomplishing milestones that are monitored by the fitness function. Each job has its own fitness function.

The resource reservoir also contains raw genetic material for use in constructing new individuals. The raw genetic material consists of functions and terminals that may be combined into chromosomes. This system uses strong typing, so that only certain functions or terminals may be used as input to a given function. The automatic program generator initially makes random chromosomes. New raw genetic material may take the form of new terminals, such as "Generator 8" or "Inventory Site B". Other raw genetic material may take the form of functions, such as "SetGoal( string )", "increaseSpeed(Generator)", or "Dispatch( site )".As the system becomes aware of new functions and terminals, they become added to the resource reservoir for use in automatically creating and testing new programs. The function "SetGoal( string )" accepts a string as an input argument, and returns a certain data-type answer, such as a Boolean (true/false) value, when complete. The "Dispatch( site )" function accepts a site argument, which can be a terminal of type site, or a function that returns a site, and when the agent has dispatched to the site, it may return a Boolean TRUE or FALSE to indicate the success or completion of the operation.

## 2.6 Tags

Tags are used to help to identify the agent as being suited for particular tasks. The offense tags indicate what the agent is good at; the defense tags serve to protect the rule by indicating what strengths it has in certain areas, such as with respect to a certain job; and the mating tags segregate the agents when selective mating is performed. Tags are updated as the agent learns different jobs.

## 2.7 Environmental Interfaces

In the field of intelligent agents, and especially in the field of genetic algorithms, anthropomorphic terminology is used frequently, such that the agents and their components are described in terms of human or animal characteristics. For descriptive purposes, the intelligent agent is split into a body and a mind, as seen in Figure 1 [15]. The mind is the essential intelligence that allows the agent to learn, retain information, and determine actions. The body is the container for the mind, and provides the capability to execute commands that the mind has issued, and to provide information from the outside world into the mind.

The outside-world information is obtained through the environmental interfaces, consisting of components such as external message board links, mobility controls and reward acceptors. During high-speed training, the mind will leave the agent body, and will be linked with a different body in the simulator. The simulator body supplies the mind with the same inputs and outputs as the real-world body, but executes in the simulator, for increased speed and repetitive training. Using interface methods, the mind is connected to the correct body for either simulated or actual system usage.

## 2.8 Execution Engine

The agent executes its rules using a state-machine type of method, where the agent performs certain manipulations on the population in a given order. Many of these operations could be performed in parallel, but serial execution was used initially in this system. At the top-level view of the algorithm the agent views the environment, and looks at what it is currently doing, then takes any actions that it believes are appropriate, and makes changes to the environment and its internal state. All the while, the agent is expending energy in the form of fitness money, which will hopefully be balanced by the rewards it receives for doing the correct things at the correct times.

There are two modes of operation, that of learning, and that of executing the rule set. During the execution phase the agent is generally operating in the actual environment, with learning potentially disabled. During learning, the agent is generally operating in the simulator and performs frequent genetic processing and LCS strength updates. These phases can overlap.

# 3. BUCKET BRIGADE OPERATION

Through the passing of money through the system, the system is capable of rewarding rules that help achieve a good, profitable solution. Through multiple iterations, rules that receive payoffs from the environment wind up passing the money further up the chain, because they can bid more money on messages. The messages thus wind up forming a chain of rule executions that get reinforced due to their proper behavior. The amount that a rule bids on a message is directly proportional to the amount of money that a rule has. It is also proportional to how closely the message on the list is matched by the message in the rule.

Message alleles, representing a genetic primitive, are enforced to be used as part of the antecedent and the consequent program trees. When a rule fires, it places its messages on the message board. Then, when the rules are checking the message board for a message that matches the message allele in their antecedents, they use a fuzzy matching range to determine if the center value of the message on the message board is contained within their range. If so, then the message allele matches the message on the message board to a certain degree, and may bid on it.

Similar to the binary LCS, if no rule matches the messages on the message board, new rules are created at random, but provided with an antecedent message-allele that will match with a message on the message list; this is termed 'cover-detection'. The creation of messages is also supported, if all the messages have run out on the message list. With the use of the GP messages, more efficient information can be transported around the system. The meaning of messages expressed in the textual messages on the external message board can reduce message uncertainty entropy through the increased expressiveness and standardized agreements based upon the use of an ontological namespace, referencing an ontological definition or task ontology.

Jobs to be performed can be given to the GP-LCS as part of an external message, and the GP-LCS learns to recognize the message, and to perform tasks to satisfy requirements of the job.

# 4. FULLY EXPANDED HYPERDIMENSIONAL NOTATION

To obtain similar capabilities with a GP-LCS as with a Genetic Algorithm-based LCS it became necessary to be able to analyze the GP rules to provide data for algorithms such as bidding, specificity, and crowding. Due to the free-form rules that are obtainable with GP, a technique using a Fully-Expanded Hyperdimensional Notation array (FEHN array) was used.

The schema representations of Poli and Langdon [16] and Justinian Rosca [17] use variable size schematas to hold the program information. This causes a "competition of hyperspaces" as the structure of the program changes to adapt to the problem. To allow an efficient and direct computer implementation of the representation of program schemas, the concept presented here uses a fully-expanded hyperspace representation. Hyperspace representations can change, hyperplanar schemas can evolve, all without changing the base representation of the individuals.

Consider the gene graph of Figure 5. It shows a complete hyperdimensional expansion of all nodes and links possible for a genetic program that is made up of functions having a maximum fan-in of 2, and it has a maximum graph depth of 4. The maximums are used, although not all nodes have this maximum fan in, nor do all graphs have the max depth. The structure of the gene graph of Figure 5 allows the node names to be organized into the structure of Table 1, where the contents represent functions or terminals. Thus, all gene graphs complying with the fan in and depth specifications can be represented and compared in one common foundation.

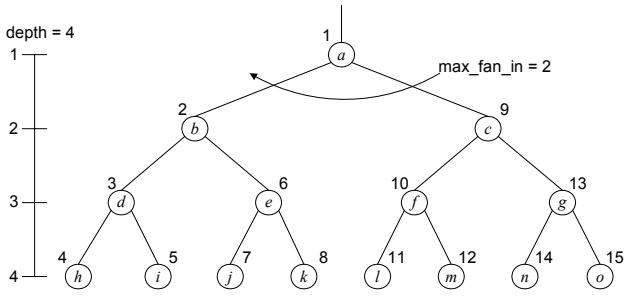


**Figure 5. Fully-dimensioned gene graph.**

**Table 1. FEHN array representation of fully-expanded gene graph for fan-in of 2 and depth 4.**

| Gene number | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Contents | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |

## 4.1 Crowding

The comparison distance between two gene graphs is used in the calculation of the crowding distance. New children are integrated into the population such that they replace adults of a similar crowd, where the adult to be replaced is the one with the lowest fitness in the group.

A measure of closeness is required. In a manner very similar to that used by Holland [18] for tag comparison in Echo CAS agents, and also in Holland [10] for schema processing, the genetically created programs are compared to each other and a measure of closeness is calculated.

The FEHN array structure allows all comparisons to be performed with a linear array. The match score between two gene graphs is the sum of the match scores of each of the positions in their associated FEHN array. These match scores may be as shown in Table 2.

**Table 2. FEHN array comparison values.**

| COMPARISON | MATCH SCORE |
|---|---|
| exact match of two, non-empty items at the same location in the FEHN array | +2 |
| mismatch of two, non- empty items at the same location in the FEHN array | -2 |
| mismatch between one non- empty item and one empty item at the same location in the FEHN array | -1 |
| if both items are empty at the same location in the FEHN array | 0 |

Thus, having a means to compare two gene graphs, a crowding technique may be implemented. A set of random individuals may be compared to a new child for similitude, replacing the most similar. Or the entire set of adult individuals may be FEHN checked, to determine the similarity of each to the new child. Then, the least fit of the most similar individuals may then be chosen for replacement.

## 4.2 Specificity

The GP-LCS rule structural specificity is determined from the number of nodes of the antecedent gene-graph containing information, not including the primitives {and, or, not, progn}, which do not lend much to the determination of how specific a gene graph is. The number of information nodes is divided by the maximum number of nodes possible. A higher the ratio indicates a higher specificity. Also, the mechanical specificity is determined by the width of the fuzzy-message detection range. The smaller the range, then the higher the specificity.

## 4.3 Bidding

The total bid provided by rule n at epoch $t$, $Bid_n(t)$, is based on

$$Bid_n(t) = K_R \cdot \{K_B + K_S \cdot S_R\} \cdot U(x,t)$$

where

$K_R$ = bid ratio constant, for example 0.1 portion of money

$K_B$ = base level of bid, constant

$K_S$ = specificity based portion of bid, constant

$S_R$ = specificity of rule, based on rule analysis

$U(x,t)$= strength of rule $x$ at the start of epoch $t$

# 5. RESULTS

The parameters used in the testing by the General Electric Global Research laboratory included: agentQuantity, classifierQuantity, initialRuleFitness, controls for the SigmaTruncationOscillation, migrantProportion, agentBirthRate, bidTaxLevel, specificityLevel (Ks), dontCareRatio, pMutation, generationGap, ADF0-4 max depths and create methods, crowdingFraction, and crowdingAlgorithm selection. The dontCareRatio works with the fuzzy rule matching, where 0 means an exact match, and 1 means that virtually any rule can match the message(s). Emphasis in this testing was based on characterizing the bucket-brigade market-based learning aspects.

The test cases included ordered traversal of different remote computer sites, as a path planning exercise, including cases where the agent was expected to visit a certain site more than once to test non-Markovian performance. An inventory scenario was also tested, involving agent purchase of parts. This scenario was expanded in subsequent testing to include multiple parts stores, and changing costs and availability, forcing on-line adaptation, and this was successfully demonstrated.

An example of the results for the testing of the specificity value, $K_S$, from Equation 3, is shown in Table 3. Along with the testing results of the other jobs, this shows that lower specificity resulted in improved learning performance; further testing will help generate a better curve for $K_S$. The Time value is in milliseconds. Cycles are epochs of rule list processing, and generation is the number of generations of evolution processing needed to correctly learn the job. Further training enforces parsimony, and increases the job performance. In many cases, as shown by the standard deviations, the learning performance can be greatly improved, or diminished, depending on the other parameters. For instance, with 8 agents (8 separate rule sets evolving and sharing migrants), with 100 classifiers each, it took only 8 seconds, 2 generations and 8 epochs, but this was probably based on a lucky first creation of the rule trees. It had generationGap = 0.24, max depth of rules of 2 or 3 which perhaps could have been deeper for better results but that helped enforce rule chaining, bid tax level 0.0004, crowding fraction 0.2, initial rule fitness/cash 50000, pMutation 0.4 which was sort of high, and specificity level 0.7, which is not what the overall results show as the best specificity, but it worked well in this case.

**Table 3. Testing of GP-LCS on a non-Markovian problem.**

| | Ks = 0.6 | Ks = 0.7 |
|---|---|---|
| Number of Successful Runs | 26 | 138 |
| Number of Runs | 26 | 145 |
| Average of "Time" | 5,457,331.38 | 6,478,668.91 |
| StdDev of "Time" | 3,897,044.68 | 5,449,800.56 |
| Average of cycle Counter | 712.69 | 916.04 |
| StdDev of cycle Counter | 659.37 | 920.83 |
| Average of generation | 177.96 | 224.51 |
| StdDev of generation | 164.86 | 216.42 |

Another test, to determine the appropriate number of classifiers to use for these sample problems is given in Table 1. This shows that the number of classifiers is relatively important in the speed of learning, although, it will learn satisfactorily without having an optimum number of classifiers. Having too many or too few classifiers is seen to impair learning performance. The learning was cut off after a certain number of generations, resulting in unsuccessful learning in some cases. If a more compact rule set is desired, then it will take more learning, if the particular job spectrum requires more classifiers for an optimum learning speed.

**Table 4. Testing the effect of different number of classifiers.**

| Number of Classifiers | 25 | 50 | 75 | 100 |
|---|---|---|---|---|
| Total Number of Successful Runs | 45 | 161 | 33 | 153 |
| Total Number of Runs | 45 | 176 | 33 | 165 |
| Total Average of "Time" | 6.03e6 | 7.52e6 | 44.84e | 5.62e6 |
| Total StdDev of "Time" | 3.84e6 | 7.18e6 | 3.79e6 | 5.20e6 |
| Total Average of cycle Counter | 1,274. | 1,301. | 898.06 | 937.60 |
| Total StdDev of cycle Counter | 865.10 | 1,414. | 809.69 | 1,138.3 |
| Total Average of generation | 284.04 | 288.81 | 204.42 | 210.80 |
| Total StdDev of generation | 185.85 | 320.58 | 180.72 | 289.79 |

Other results show that having more agents results in faster learning, but this can be offset in processing time, if a single agent learning site is used. Increasing the dontCare ratio to provide more generalization also speeds the learning process, similarly to increased performance resulting from less specificity.

## 6. SUMMARY

In this paper, we described a Genetically Programmed Learning Classifier System for Complex Adaptive System Processing. We provided a method for GP "bucket-brigade" rule-chain learning by a type of fuzzy message matching and by implementing a specificity technique for GP-LCS. This specificity technique is characterized by the FEHN array technique which provides a GP gene graph comparison technique useful in the implementation of a crowding algorithm.

An LCS has many parameters, for administration, the market system, the chromosomes, and the genetics. In a large, complex system, testing of the effects of these parameters can greatly improve the learning and operating performance of the system. There is a careful balance of parameters required for optimal learning, and it leads to the concept of meta-LCS control for optimizing the learning parameters with respect to a given problem domain.

This GP-LCS implementation has been proven to work, and tests for scalability should be performed. It creates rules in an environment, and these rules consist of sets of programs automatically written in a high-level language of functions and terminals. It provides the ability to realize a large-scale Complex Adaptive System in a multi-computer environment. We believe that an expanded system, with attendant simulation systems for its training, and an infrastructure that interacts through the external message list could provide autonomic learning, processing and optimization for many enterprise-level tasks, much in keeping with John Holland's Echo and Complex Adaptive System environments.

## 7. ACKNOWLEDGEMENTS

This work was supported by Internal Research and Development at Lockheed Martin Simulation, Training & Support. We wish to acknowledge Robert K. Hollister, Mike Bodkin, and GE Global Research for their contributions to this work.

## 8. REFERENCES

[1] Ahluwalia, M. & Bull, L. A Genetic Programming-based Classifier System. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference.* Morgan Kaufmann, 1999, 11-18.

[2] Stolzmann, W.. Anticipatory classifier systems. *Proceedings of the Third Annual Genetic Programming Conference, July 22-25, 1998*. (University of Wisconsin, Madison, WI). Morgan Kaufmann Publishers, San Francisco, CA, 1998, 658-664.

[3] Wilson, S.W.. Generalization in the XCS Classifier System. *Proceedings of the Third Annual Genetic Programming Conference.* (University of Wisconsin, Madison, WI, July 22-25, 1998). Morgan Kaufmann Publishers, San Francisco, CA, 1998, 665-674.

[4] Bay, J.S. Learning Classifier Systems for Single and Multiple Robots in Unstructured Environments. Web page, Jan. 5. 1999. Internet. http://armyant.ee.vt.edu/

[5] Bonarini, A., Bonacina, C., and Matteucci, M. Fuzzy and crisp representations of real-values input for learning classifier systems. *Proceedings of the Genetic and Evolutionary Computation Conference, 1999, LCS workshop*. (Orlando, FL, 1999).

[6] Booker, L.B. Do we really need to estimate rule utilities in classifier systems? *Proceedings of the Genetic and Evolutionary Computation Conference, 1999, LCS workshop*. (Orlando, FL, 1999).

[7] Dorigo, M. Message-based bucket brigade: an algorithm for the apportionment of credit problem, *Proceedings of European Working Session on Learning '91*. (Porto, Portugal), also Web page, Internet http://iridia0.ulb.ac.be/~mdorigo/publications.html

[8] Smith, R.E., Dike, B.A., et al. The fighter aircraft LCS: a case of different LCS goals and techniques. *Proceedings of the Genetic and Evolutionary Computation Conference, 1999, LCS workshop*. (Orlando, FL, 1999).

[9] Holland, J.R. *Emergence: From Chaos to Order*. Addison-Wesley Publishing Company, Reading, MA, 1998.

[10] Holland, J.R. *Adaptation in Natural and Artificial Systems*. The MIT Press. Cambridge, MA, 1975.

[11] Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, 1992.

[12] de la Maza, M. Sigma Truncation in The Boltzmann selection procedure. *Practical Handbook of Genetic Algorithms, New Frontiers, Vol. II*. Chapman & Hall/CRC, Boca Raton, FL, 1995, 111-138.

[13] Richards, R. Zeroth-Order Shape Optimization Utilizing a Learning Classifier System. Web page, viewed December 3, 1998. Internet http://www.stanford.edu/~buc/SPHINcsX/book.html

[14] Goldberg, D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Reading, MA, 1989.

[15] Muller, J.P. *The Design of Intelligent Agents: A Layered Approach*. Springer-Verlag, Berlin Heidelberg, Germany, 1996.

[16] Poli, R. and Langdon, W.B. A new schema theory for genetic programming with one-point crossover and point mutation. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*. (Stanford University, July 13-16, 1997). Morgan Kaufmann, San Francisco, CA, 1997, 35-43.

[17] Rosca, J.P. Analysis of complexity drift in genetic programming. *Genetic Programming 1997: Proceedings of the Second Annual Conference*. (Stanford University, July 13-16, 1997). Morgan Kaufmann, San Francisco, CA, 1997, 286-294.

[18] Holland, J.R. *Hidden Order: How Adaptation Builds Complexity*. Addison-Wesley Publishing Company, Reading, MA, 1995.