SANDIA REPORT
SAND2001-2097
Unlimited Release
Printed Date

Echo Implemented: A Model for Complex Adaptive Systems Computer Experimentation

David L. Harris

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND2001-2097
Unlimited Release
Printed July 2001

Echo Implemented: A Model for Complex Adaptive Systems Computer Experimentation

David L. Harris
Networked Systems Survivability & Assurance Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0449

Abstract

This paper provides an overview John Holland's Echo model, describes an implementation of the model, documents results from preliminary experiments using the model, and proposes further research in using Echo to study complex adaptive systems. Echo simulates the behavior of complex adaptive systems and can provide an experimental testbed for exploring theories of, and developing tools useful for analyzing these systems. Preliminary results indicate that the dynamic behavior of Echo can be used to generate interesting, time-series data that will be useful for evaluating the applicability of and developing tools, techniques, and possibly general theories, for the analysis of specific complex adaptive systems.

Echo Implemented: A Model for Complex Adaptive Systems Computer Experimentation

1. Introduction.

This paper provides an overview John Holland's Echo model [1], describes an implementation of the model, documents results from preliminary experiments using the model, and proposes further research in using Echo to study complex adaptive systems. Echo simulates the behavior of complex adaptive systems and can provide an experimental testbed for exploring theories of, and developing tools useful for analyzing these systems. Preliminary results indicate that the dynamic behavior of Echo can be used to generate interesting, time-series data that will be useful for evaluating the applicability of and developing tools, techniques, and possibly general theories, for the analysis of specific complex adaptive systems.

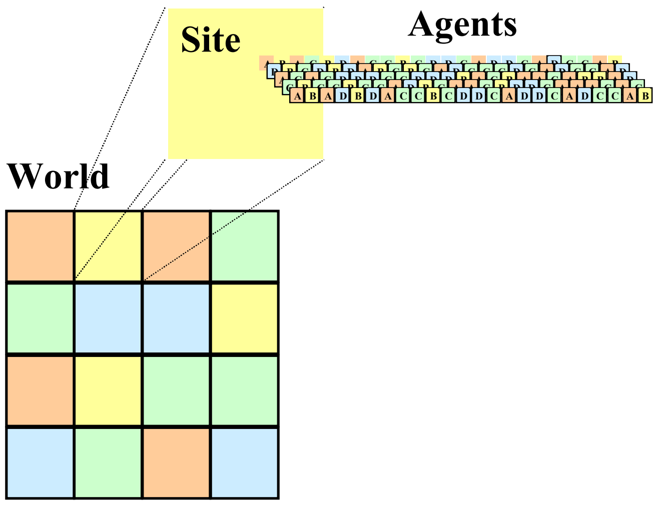
Section 2 of this paper provides a brief overview of Echo, followed in Section 3 by a more detailed description of the implementation. Section 4 includes a description of the model's execution life cycle and the parameters that are used to vary the model's environment and dynamics. Section 5 provides preliminary results related to initial experiments used to study the exchange of resources. Section 6 proposes additional research, experiments, and analysis to be carried out using the model. Section 7 is a summary including conclusions about the model and the initial experimental results. Appendix A contains a brief summary of the implementation and the model execution environment. Appendix B contains the Java source code of the model implementation.

2. The Echo Model.

In his book Hidden Order, How Adaptation Builds Complexity, John Holland [1] characterizes all complex adaptive systems as being comprised of seven fundamental characteristics. These seven characteristics consist of four properties: aggregation, nonlinearity, diversity, and flows and three mechanisms: tagging, internal models, and building blocks. Echo, as described in Hidden Order, is a model that simulates the behavior of complex adaptive systems. It consists of a basic model, along with a series of enhancements that are used to increase its ability to capture these seven fundamental characteristics, and can be used as an experimental testbed for exploring issues related to the general study of complex adaptive systems. It can be used to study the emergent behaviors of a population of agents over time; behaviors such as population growth dynamics and diversity, exchange patterns, and hierarchical aggregation. Echo provides an opportunity for study that can increase the understanding of such complex adaptive systems as economies, ecologies, organizations, communication networks, and immune systems. Various implementations of Echo have been developed and used to study complex adaptive systems. One has been used for the study of population diversity and dynamics by Forrest and Jones [2] and another for the study of food web complexity by Schmitz and Booth [3]. Santa Fe Institute has a web page devoted to the Echo model [4].

Echo is an agent-based, micro-simulation model in which "agents" interact within a "site" located in a "world". The world consists of a topology and a clock, which provides a spatial and temporal context for populations of agents that reside at specific sites. A site consists of a descriptor, made up of a sequence of resources, and a resource fountain. Each site's fountain provides a renewable source of resources to the agents and a local environment where agent encounters and interactions take place. Sites may have different descriptors and may differ in the types and amounts of resources that their fountains provide to their agents. An agent consists of a descriptor and a resource reservoir. Figure 1 is an illustration of an Echo world.

Figure 1. Echo world.



An agent interacts on a periodic basis with the site where it resides to absorb resources into its reservoir or to dissipate resources from its reservoir. Agents interact on a stochastic basis with each other to exchange resources between their reservoirs and to reproduce. In each case, an agent's or a site's descriptors are used to arbitrate an interaction and to determine its outcome, thus the descriptors provide distinguishing features for differentiating individual agents and the various sites. Although interactions between agents occur only between agents that occupy the same site, a capability allowing agents to migrate between sites could exist.

The ability to acquire resources from either the site or other agents allows an agent to reproduce. In general, agents unable to acquire resources become extinct and those agents adept at acquiring resources may reproduce. An agent is eligible to reproduce when it has acquired enough resources in its reservoir to duplicate its descriptor and may reproduce when it encounters a suitable mate. Eligible pairs of agents reproduce by copying their descriptors using resources from their reservoirs. Operators, which mimic the genetic crossover and mutation operators, are applied to the copies of these pairs of agent descriptors and the recombined and mutated pairs are added to the site's population. The crossover and mutation operators allow agents to adapt to their environments and for populations to evolve. These operators provide the mechanisms that the model uses to explore the space of possible descriptors, searching for those descriptors that are the most efficient at collecting resources. Since the environment is continuously changing, descriptors that may be efficient at one period of time may, as other agents evolve, lose their competitive advantage. As new agent types evolve, the patterns of exchange between agents also will evolve, creating a rich dynamic.

The emergent behavior of the world is due to the interactions of the individual agents and it is this emergent behavior that we are interested in studying. Several interesting aggregate measures include: population, reproduction and death rates, agent diversity, exchange amounts, exchange patterns, and etc. Each of these aggregates and the patterns in aggregate behaviors over time is caused by the individual interactions that the agents undertake within their environment; patterns that would not be discernable by investigating the individual agents in isolation.

3. An Echo Implementation.

This implementation of Echo is based on an object-oriented decomposition of the principal entities of the Echo model: resource, tag, condition, descriptor, reservoir, agent, fountain, site, and world, along with the periodic and stochastic processes for controlling agent activities and interactions.

The fundamental entity within the Echo model is the resource. There are multiple types of resources and a single character represents each resource. In order to represent tags and conditions, resources are combined by concatenating their character representations into sequences. Descriptors are represented by concatenating tags and conditions. The methods for employing the descriptors are described in detail below. Unlike descriptors, reservoirs and fountains are simply collections of resources with no particular relevance currently given to their ordering.

As stated earlier and illustrated in Figure 2, each site consists of a resource fountain and a descriptor. The resource fountain provides a source of resources to the agents. Each site descriptor, which is assigned when the site is created (and remains static), is comprised of two tags: an offense tag and a defense tag. At each time period, a site's fountain is replenished with a certain number and distribution of resource types. The site's descriptor and the number and mix of resources provided by the fountain allow for sites to create different environments for the agents that occupy the site.

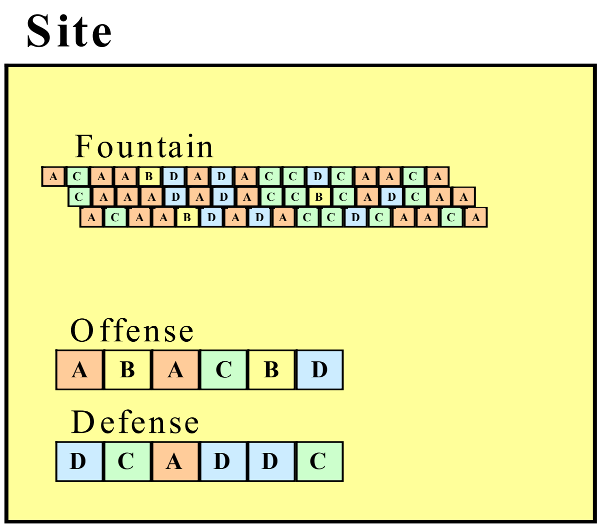
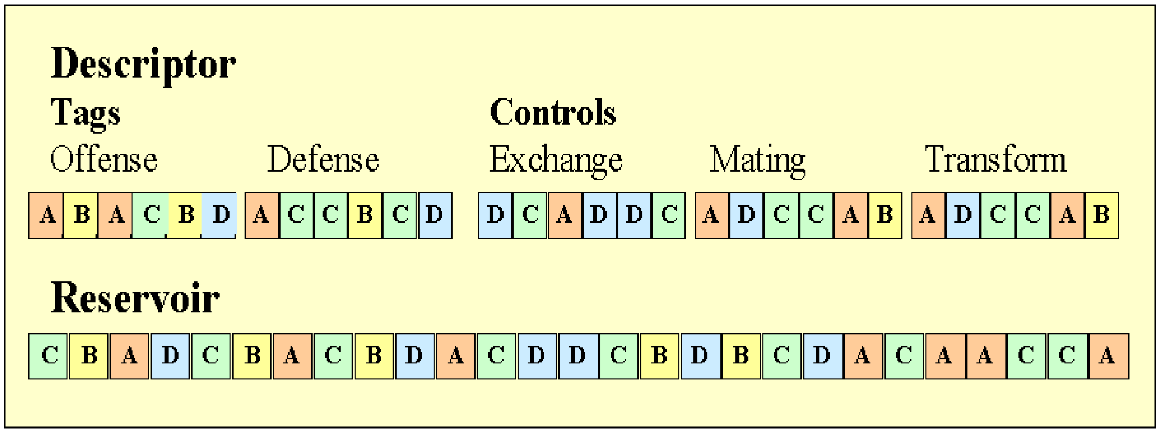


Figure 2. Echo site.

The spatial arrangement of the sites is provided by the world. In this implementation the sites simply occupy a 2-dimensional plane with adjacency defined in four directions (north, east, south, and west). The world can be either periodic (wraps around at the edges of the plane) or not. Agents may be given an opportunity to migrate to a different site within the world based on some migration policy (for example, the inability of an agent to acquire resources in its current environment may allow it to migrate to an adjacent site) but, this currently is not implemented. The world provides a simulated time clock for controlling the sites and each site manages the sequencing of its agent's life cycle. At each time step, the site schedules a specific sequence of activities; however, the individual agents only participate in these activities based on probabilities set in the model operating parameters and on the agents' current states (the contents of their tags, conditions, and reservoir). Although methods within the world and the site schedule the agent life cycles, neither completely determines the outcome of agent encounters and interactions. Agents are the dynamic elements in the current model and how they evolve determines the behavior of the model over time.

An agent has a resource reservoir and a descriptor, Figure 3. The reservoir will expand and shrink depending on exchange and reproduction interaction outcomes. The reservoir is used to store the resources that an agent has been able to acquire through its interactions with the site and with other agents. An agent may dissipate resources from its reservoir to other agents or to the site during exchange interactions. The agent also removes resources from its reservoir for reproduction. An agent may cease to exist if it is unable to maintain a minimal number of resources or if it is unable to reproduce frequently.

Figure 3. Echo agent.



An agent's descriptor consists of two types of elements: conditions and tags. Similar to the site, an agent contains an offense tag and a defense tag. It also contains three conditions: an exchange condition, a mating condition, and a transformation condition. The descriptors determine how agents interact with their site and with each other. Encounters are random meetings between pairs of agents, which are arbitrated using the conditions within their descriptors (precisely how this arbitration works is described below). Based on this arbitration, an interaction for exchange or reproduction may occur. The resources that are contained within the conditions and tags are used to determine the scores when evaluating the descriptor conditions. Scoring the encounters and interactions is done by summing the pair-wise scores, determined by lookup matrices, between the two participant's descriptor element resources. Tags are also used to determine the outcomes of the various interactions that do take place. For example, at each time-step an agent unconditionally interacts with its site to absorb and dissipate resources. The agent's offense and defense tags, in conjunction with its site's defense and offense tags, are used to determine how many resources an agent dissipates to and absorbs from its site during the site interaction (note that exchange interactions are not symmetric; agents may dissipate more resources than they acquire). The agent transform condition is used to allow an agent to convert resources within its reservoir from one type to another.

In addition to the unconditional exchange interaction with its site, an agent will conditionally interact with other agents it encounters to exchange resources and to reproduce. At each time step, random pairs of agents selected from the same site will encounter each other to exchange resources. When an agent encounters another agent, the agent's tags and conditions are scored and this score is used to arbitrate whether an interaction between the two will take place and also is used to determine the degree of interaction. To determine if a random exchange encounter will result in an interaction, the exchange condition of one agent is scored against the offense tag of the other agent, and vice versa:

1. If both scores exceed the exchange threshold, resources are exchanged in both directions.
2. If one score exceeds the exchange threshold, resources may be exchanged to the agent with the higher score.
3. Otherwise, no exchange interaction occurs.

At each time step, all of the agents in each site are evaluated to determine if they have enough resources in their reservoirs to reproduce their descriptors. Those that can reproduce are matched with the other eligible agents to determine whether a mating interaction can occur. The mating condition of one agent is scored against the offense tag of the other agent, and vice versa, to arbitrate the encounter:

1. If both scores exceed the mating threshold, two new agents are created and added to the population.
2. Otherwise no reproduction occurs.

Resources and their types are represented in the model using the characters $\{a, b, c, d\}$ and the conditions and tags are defined as sequences of six resources each. Although these sizes appear to provide adequate population diversity, they were chosen arbitrarily with no other specific requirements (more sophisticated Echo models allow these descriptor elements to vary in length).

Agents, with some probability, may transform resources within its reservoir from one type to another. At each time step, the transform condition and the agent's reservoir are examined randomly to see if a transform will occur. The six resources within the transform condition encode two possible transformations. Each encoding uses three resources consisting of a source, cost, and destination resource. If the reservoir contains resources for both the source and the cost then the transformation from source to destination will take place with the loss of the cost resource. For example, if the transform descriptor and reservoir contain the following, then a 'c' is transformed to an 'a' at the expense of a 'b', the second transform will not take place since the 'b' was used in the first transformation:

transform descriptor: *cba*||*abd*
Before transform operator:
Reservoir: *ccbcaaa*
After transform operator:
Reservoir: a*ccaaa*

The condition and tag scores are determined using the two scoring matrices shown in Table 1. The condition score matrix is used when scoring encounters and the tag score matrix is used when scoring interactions. The rows in these matrices are indexed by the resource from the first agent and the columns by the resource from the second agent. The condition or tag score is computed by summing the individual scores of the six resources within the condition or tag. The condition score matrix allows agents to treat one resource, '*d*', as a "don't care" resource so that it will match any of the second agent's resources (i.e., receives the highest score). This changes the relative values of the resources and may provide a competitive edge to agents that have conditions that can 'match' tags more often. This is not true for the tag score matrix.

Table 1. Scoring Matrices.

Condition score matrix:

	a	***b***	***c***	***d***
a	2	1	0	1
b	1	2	1	0
c	0	1	2	1
d	2	2	2	2

Tag score matrix:

	a	***b***	***c***	***d***
a	2	0	-1	0
b	0	2	0	-1
c	-1	0	2	0
d	0	-1	0	2

As an example, suppose that a site has the following descriptor with an offense tag (OT) and a defense tag (DT) and an agent has the following descriptor with a defense tag and an offense tag. Using the tag score matrix to evaluate an interaction between this site and this agent would result in the agent dissipating three resources from its reservoir and absorbing one resource from the site's fountain:

Example exchange interaction scoring:

site1:	OT = *acbbda*, DT = *bcccab*	site offense score:	0 + 2 + 0 + (-1) + 0 + 2 = 3
agent1:	DT = *bccdaa*, OT = *ddcbda*	agent offense score:	(-1) + 0 + 2 + 0 + 0 + 0 = 1

A second example is an agent exchange encounter. At each time step, pairs of agents within a site are selected randomly for an exchange encounter. To arbitrate the type of encounter the exchange condition (EC) of agent1 is scored against the offense tag of agent2 using the condition score matrix, and vice versa. The scores are compared to the exchange interaction threshold (assume ten for this example). In this example, the exchange is unilateral from agent2 to agent1 (although agent2 has some, usually small, probability of avoiding the exchange). The number of resources transferred is determined in the same way that the transfer from the site fountain was determined, by scoring the agent2 offense tag and agent1 defense tag using the match tag score matrix.

Example agent exchange encounter scoring:

agent1: EC = *caadbc*, OT = *cbcabd*   agent1 condition score: 1 + 2 + 2 + 2 + 0 + 1 = 8 agent2: OT = *baacdd*, EC = c*dbbdd* agent2 condition score: 2 + 2 + 1 + 1 + 2 + 2 = 10

A third example is an agent mating encounter. At each time step, a random sample of agents that are eligible to reproduce (those that can duplicate their descriptor from their reservoir) are selected from the site. Each of these agents is compared with the other selected agents to determine if a mating interaction can take place. For example, suppose that agent4 and agent9 have the following mating conditions (MC) and offense tags and that the mating threshold were 8. In this case, since both scores exceed the threshold, two new agents would be created, each duplicated by removing resources from the reservoir of their parent agent (any resources remaining in the parent reservoir are split with the offspring agent).

Example agent mating encounter scoring:
agent4: MC = *acccda*, OT = *abbaba* agent4 condition score: 0 + 2 + 2 + 2 + 2 + 1 = 9
agent9: OT = *ccccbd*, MC = *adbbdd* agent9 condition score: 2 + 2 + 2 + 1 + 2 + 2 = 11

The descriptors of the new pair of agents, with some probability of crossover, would be recombined and, with some probability of mutation, would be mutated. Crossover works by selecting a location in the descriptor's resource sequence and recombining the ending resources from one descriptor with the beginning resources of the other. Mutation works by arbitrarily selecting a different resource to replace a current resource at random places within the descriptor's resource sequence.

Before crossover operator:
agent1: descriptor *cbcabd bcc||daa caadbc acccda*
agent2: descriptor *ccccbd bac||cab cdbbdd adbbdd*

After recombination applied at loci 9 of the new agent descriptors:
agent1: descriptor *cbcabd bcc||cab cdbbdd adbbdd*
agent2: descriptor *ccccbd bac||daa caadbc acccda*

The intent of the crossover operator is to allow the population to exploit the better agents; recombination assembles better descriptors from parts of descriptors that have been shown to be effective in acquiring resources. The intent of the mutation operator is to explore the space of possible agents. To continue the biological metaphor, each resource can be thought of as an allele, and the sequences of alleles that make up each tag or condition corresponds to a gene. The entire set of tags and conditions in the descriptor represents a complete chromosome. Several evolutionary computational techniques, such as evolutionary programs, genetic algorithms, and genetic programs have explicit fitness functions that are used to determine which chromosomes to select for future generations. However, within Echo the fitness function is implicit in an agent's ability to collect resources thus, enabling it to reproduce. The goal of each agent is to collect resources and use them to create new agents.

The preceding description did not establish the correspondence between the seven fundamental characteristics and model elements; this list, where applicable, establishes explicitly that correspondence.

1. *Aggregation*: This implementation of Echo does not include aggregation tags. Consequently, agents do not explicitly form multi-agent complex structures. This version of the model has been used primarily for investigating aggregate behaviors such as exchange totals and population dynamics, a form of aggregation that is clearly a different notion of aggregation.

However, the emergent properties of agent aggregations into tightly coupled social networks, as described by Wasserman and Faust in [5], or other implicit aggregate forms is hypothesized and needs to be studied further.

2. *Nonlinearity*: Nonlinearity in exchange and reproduction is the result of the encounters that take place. Agents are created and destroyed resulting in varying populations of each agent type. Since agents randomly are selected for encounters, the probability that an encounter between any two agent types will result in an interaction is a product of the populations of their respective types. This results in the nonlinear behavior of their interactions.
3. *Diversity*: The diversity of the population is created by the mutation and recombination of descriptor resources during reproduction interactions. These operators allow the population to effectively search for new, viable agent types.
4. *Flows*: The exchanges of resources between sites and their agents and between the agents are flows.
5. *Tags*: The offense and defense tags, which are contained within the site and agent descriptors, are observable by the other agents and provide a tagging mechanism for selective interactions.
6. *Internal models*: The conditions that an agent uses to arbitrate encounters act as its internal model. The conditions essentially allow agents to "choose" what other agent types, based on their exposed tags, it is willing or desires to interact with when it encounters another agent. Agents evolve conditions that result in more effective interactions.

4. Model Execution.

The model executes in two main phases: initialization and evolution. During initialization the world is constructed by creating each site and its initial population of agents. The site descriptor, initial fountain, and fountain replenishment are typically randomly chosen sets of resources, although sets with specific resource distributions could be selected to give sites unique qualities. The descriptors and reservoirs for the initial population of agents also are typically randomly chosen sets of resources. But, they also could be seeded with specific descriptors if particular initial populations were desired. This may be the case if particular population dynamics were desired (for example, a site could be initialized with agents to model a specific predator and prey dynamic). The evolutionary phase is where all the real action takes place. At each time step a site first replenishes its fountain and then guides its population of agents through their 'daily' life cycle:

1. *Replenish*: Replenish each site's fountain.
2. *Exchange*: Randomly select agents for exchange, arbitrate agent exchange encounters, and process any exchange interactions.
3. *Transform*: Allow an agent to transform resources that are contained in its reservoir from one type to another type.
4. *Mate*: Randomly select eligible agents for mating, arbitrate agent mating encounters, and process any mating interactions. This includes evolving new agents using the recombination and mutation operators.
5. *Absorb*: Each agent absorbs resources from the fountain based on its offense tag and the site's defense tag.
6. *Dissipate*: Each agent dissipates resources to the fountain based on its defense tag and the site's offense tag.
7. *Kill*: Randomly remove agents whose reservoirs are below the sustenance level or who have not been able to reproduce for their life span number of time steps.

Several operating parameters control the initialization and execution of the world, the site, and the agent daily life cycles. Table 2 contains a list of these parameters and their default values.

Table 2. Echo Default Operating Parameters:

Operating Parameter	***Value***
Random Seed	random
Time Steps	5,000
Number of Sites	1
Replenishment Quantity	150
Initial number of Agents	500
Initial Reservoir	10
Sustenance Level	15
Probability of Death	0.900
Life Span	750
Probability of Exchange	0.850
Exchange Threshold	10
Probability of Avoidance	0.100
Probability of Transform	0.025
Probability of Mating	0.750
Mating Threshold	7
Probability of Mutation	0.050
Probability of Crossover	0.700

The following list contains a brief description of each of the operating parameters used to adjust the execution of the model.

1. *Random seed*: The random number generator initial seed value. This can be set to repeat a particular Echo execution.
2. *Time steps*: The number of time steps to run each site's life cycle.
3. *Number of Sites*: The number of sites in the world.
4. *Replenishment quantity*: The number of resources added to the fountain at each time step.
5. *Descriptor Field Length:* The number of resources in each of the descriptor's tag and condition fields (for the sites and agents).
6. *Initial number of Agents*: The initial population of agents seeded in each site.
7. *Initial Reservoir*: The number of resources initially given to each of the agents.
8. *Sustenance level*: The minimal number of resources an agent needs to maintain in its reservoir in order to avoid being susceptible to elimination.
9. *Probability of death:* Probability that an agent will be eliminated if its reservoir does not exceed the sustenance level.
10. *Life Span:* The number of time steps an agent can exist without reproducing.
11. *Probability of exchange*: Probability of an agent being selected for an exchange encounter.
12. *Exchange threshold*: Minimal exchange encounter score necessary for an exchange interaction to occur.
13. *Probability of avoidance*: Probability that a low scoring agent can avoid an exchange interaction.
14. *Probability of Transform:* Probability that an agent will be selected to transform one of its stored resources.

15. *Probability of mating*: Probability that an eligible agent will be selected for a mating encounter.
16. *Mating threshold*: Minimal mating encounter score necessary for a mating interaction to occur.
17. *Probability of mutation*: Probability that the mutation operator will be applied to an allele in an agent descriptor.
18. *Probability of crossover*: Probability that the crossover operator will be applied to a pair of agent descriptors.

5. Resource Exchange Patterns, Preliminary Results.

This section presents preliminary results from executing the Echo model. The intent of showing these results is to illustrate that the Echo model is capable of generating complex time-series data. Admittedly, the analysis of this data has been at best, superficial. However, these results should provide ample evidence that Echo is indeed a valuable model for developing some of the tools and techniques, and possibly some theories, to further the study of complex adaptive systems in general.

The following results were obtained by post-processing exchange interaction data collected during initial experiments used to observe the emergent patterns in the exchange of resources. The model was executed using operating parameters from Table 2. Each exchange of resources between interacting agents was logged between time step 1,000 and time step 5,000. The initial 1,000 time steps were used to allow the model time to generate a population of agents with an evolved exchange pattern rather than using the pattern from the initial, random population. Each entry in the log represented the outcome of an exchange interaction between two agents (it does not include exchanges with the site) and contained the descriptors of both agents, the number of resources acquired by each agent, and the time step that the exchange occurred. To determine the set of agent types, the agent descriptors were sorted in ascending lexicographical order, duplicate descriptors were removed, and a sequence number was assigned to identify each of the unique descriptor strings. From this, an agent type is defined by the string representation of its descriptor and there may exist several agents of the same type. During the time period that exchanges were logged, 1,372 different agent types (unique agent descriptors) participated in 228,188 exchange interactions where a total of 623,459 resources were exchanged.

Figure 4 illustrates the quantity of resources acquired, by agent type, during agent exchange interactions during the period from time step 1,000 to time step 5,000. This contour plot was generated from data points (*x*, *y*, *z*), where *z* is the natural log of the number of resources acquired by the agents of type *x* from the agents of type *y*. The data points were aggregated from the exchange interaction log data by summing the individual agent acquisitions by agent types.

Figure 4. Resource acquisition contour plot.

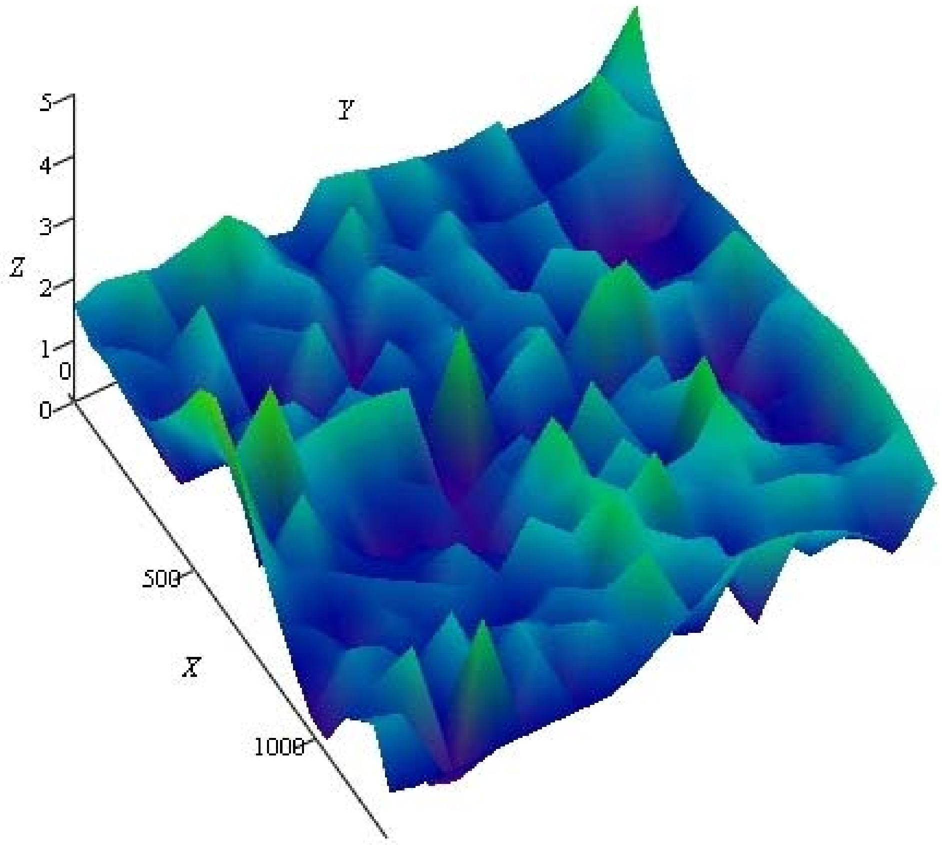
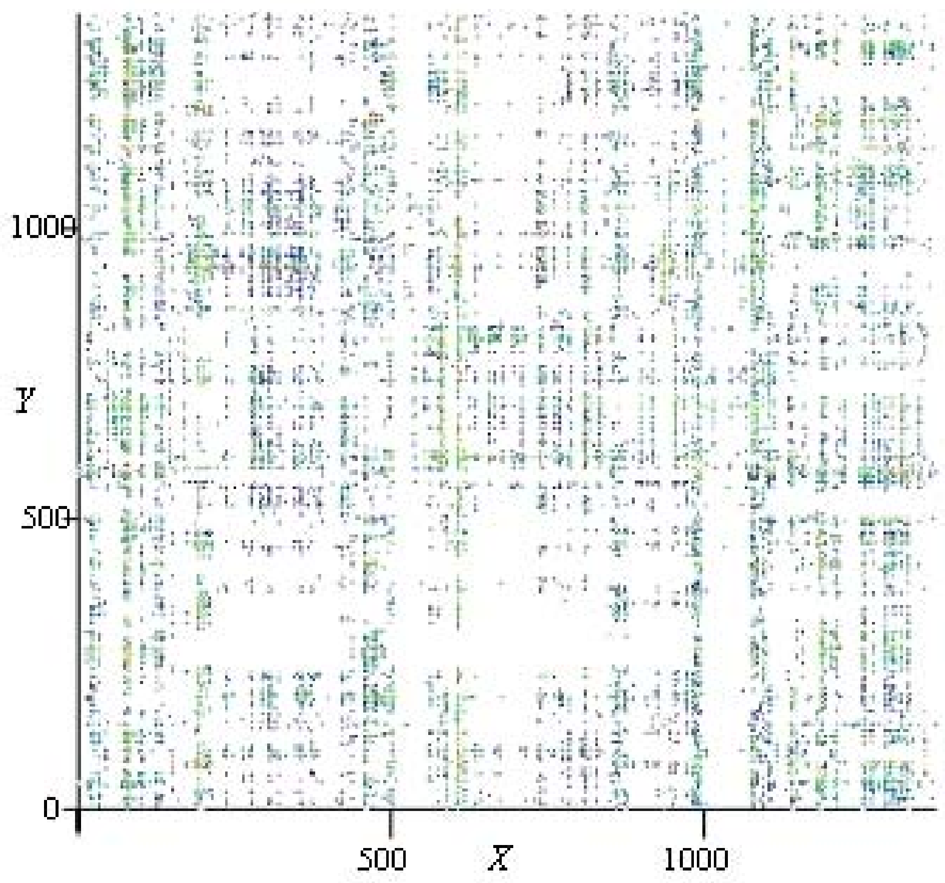


Figure 5 displays the incidence relation using the same exchange interaction log data as Figure 4. Two agent types are incident if an agent, a_x, of type *x* acquired resources from an agent, a_y, of type *y,* i.e., $a_x \mathrm{R} a_y$ iff a_x acquired resources from a_y. The *z* axis, directly out from the page, can be ignored in this figure. Each data point (*x*, *y*) represents the incidence of agent types during the period from time step 1,000 to time step 5,000. Closer examination of the data showed that there were only 21,000 agent acquisition incidences out of a possible $1{,}882{,}384 = 1{,}372^2$ pairs.

Figure 5. Incidence relation plot.



Both Figures 4 and 5 illustrate resource acquisition patterns that suggest agents are selective in their interactions, as shown by the peaks and valleys in the contour plot and by the sparseness of the incidence relation plot. As shown in Figure 4, the total number of resources exchanged varies by greater than 2 orders of magnitude and, as shown in Figure 5, the incidence of resource acquisition occurred infrequently between agent types. These facts indicate that significant exchange patterns between agent types have been established. Even though this was to be expected, since the agent's descriptor's are used to arbitrate encounters, that the Echo model was used to generate this data set is an indication that it also could be used as a model for generating data sets useful in other experiments, experiments that could be used for developing tools to explore further these model features and also the features of other complex adaptive systems.

Figure 6 shows resource acquisitions at 1,000 time step intervals (again, starting at time step 1,000). Each of these four contour plots are similar to those of Figure 4 (although not labeled, the axis are as in Figure 4). As can be seen, the acquisition patterns change to produce different contours. For example, high peaks in one time period are gone in another. This could be a result of the expiration of some agent types or the population growth of alternative exchange candidates.

Figure 6. Agent resource acquisition contour plots.

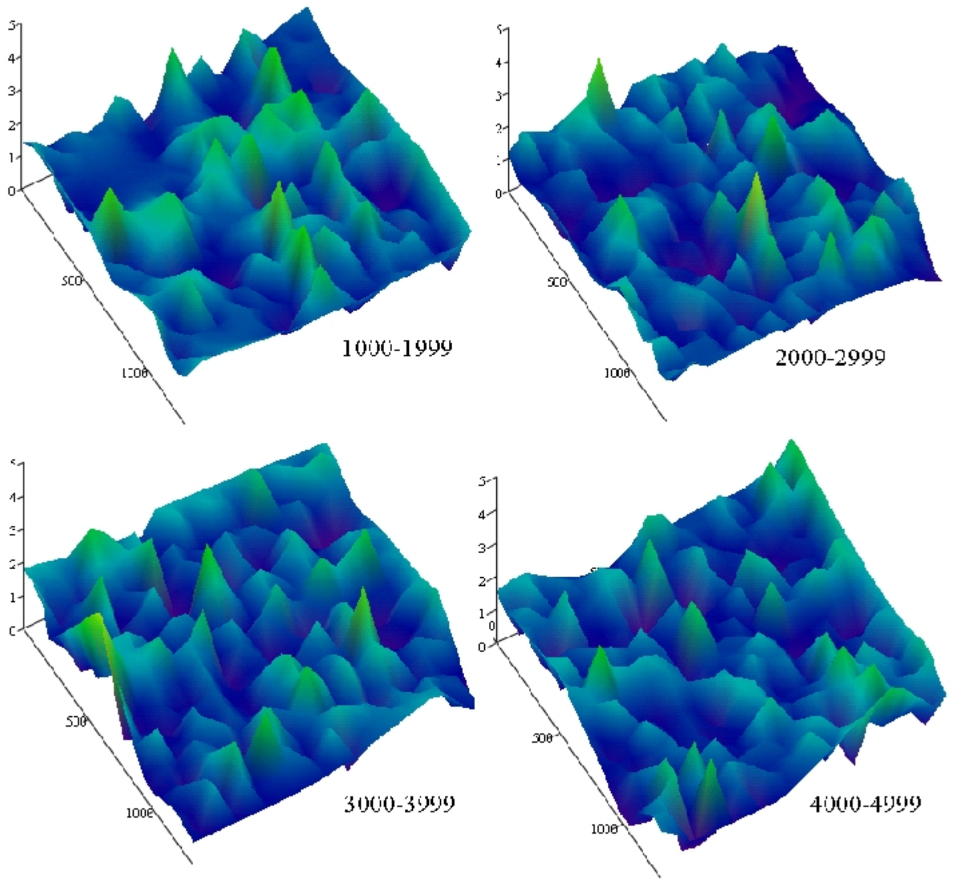


Figure 7 illustrates the resource acquisition relationship for paths of length up to 9. For example, if agent types $a_0....a_n$ exchanged resources, then an exchange path of length n, n = 1..9, exists. This plot was created from the incidence data of Figure 5. First, the Boolean incidence relation matrix B, where B_{xy} is true if $a_x \mathrm{R} a_y$ and false otherwise, was computed. Then, the matrix product of B^9 was generated. In this plot, the diagonal shown in red indicates a cycle with a fixed length of 9 (or 1 or 3, the factors of 9). The vertical gaps indicate that those agent types did not acquire resources from any other agent types; they dissipated resources (they may have acquired resources from the fountain, or had short lives).

Figure 7. Agent resource acquisition paths of length 9 or less.

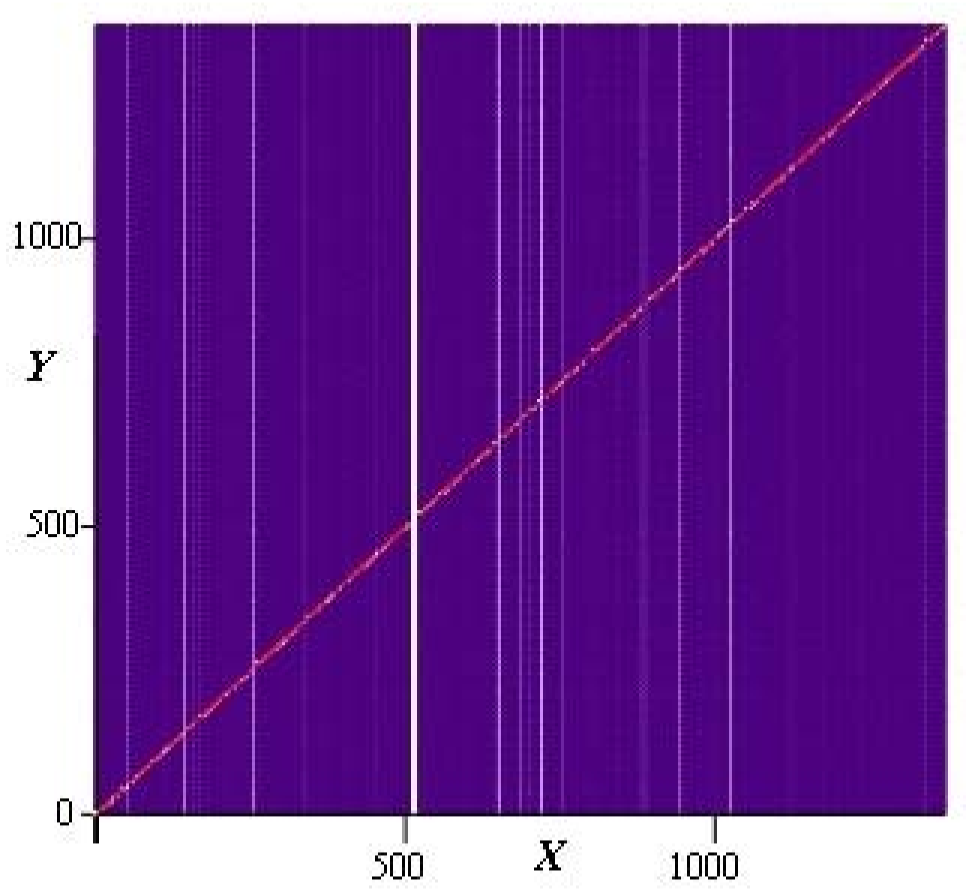
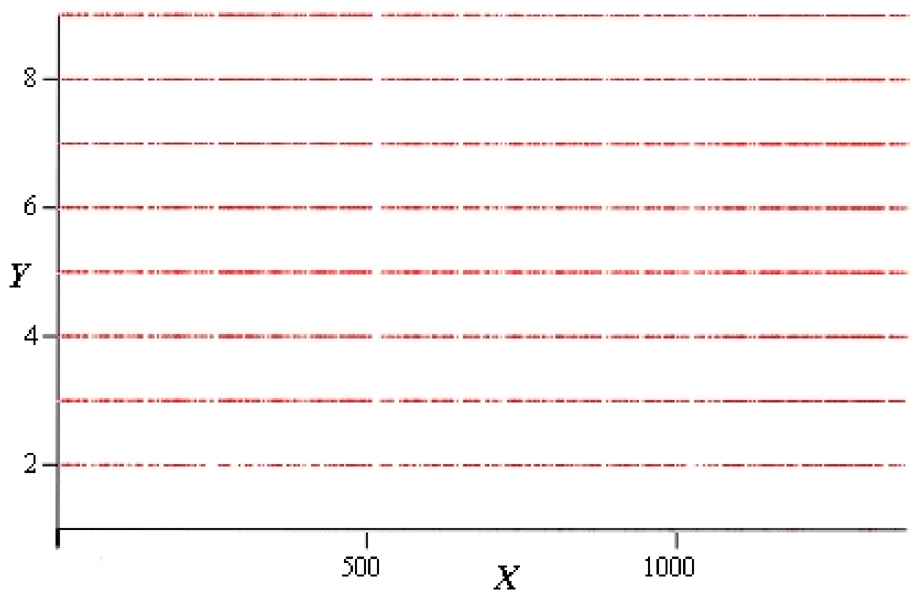


Figure 8 shows cycles of resource acquisitions of lengths 1..*n* for *n* ∡ 9. For example if agent types $a_1....a_n,a_1$ have exchanged resources then an exchange cycle of length *n* exists. This was computed by collecting the diagonals from each of the B^i, for $i = 1..9$, generated while computing the paths for the previous Figure. In this plot, the *x* axis is the agent type and the *y* axis is the length of the cycle, *n*, that exists from an agent of type *x* back to itself.

Figure 8. Agent resource acquisition cycles.



Figures 7 and 8 indicate that the Echo model can produce complex exchange interaction patterns. The existence of the lengthy paths and the cycles of interactions indicate that relatively complicated exchanges of resources are taking place. This is significant since it indicates that Echo could be used for computer simulation experiments related to traffic analysis. For example, communication traffic analysis has been used as an effective method for inferring human behaviors and Echo could be used to generate test data for research related to traffic pattern analysis.

6. Further Research.

Considerably more work is needed to validate the premise that Echo can be used as an experimental testbed for a particular complex adaptive system. Although the previous figures illustrate interesting behaviors, further analysis is warranted to better understand their details and to establish the correspondence between Echo behaviors and the behaviors of a particular complex adaptive system. This is envisioned to be a research activity consisting of two parallel components: 1) to establish the model's correspondence to a particular target complex adaptive system and 2) to further develop analytic tools and techniques to further understand the behaviors of the Echo model and the target system. The goal of the first activity of this research is to demonstrate the correspondence between the exchange dynamics of populations of agents within the model and those from other complex adaptive systems. This correspondence will be established by illustrating that the dynamic behavior of time series data from populations of Echo agents is similar to the behavior of the complex adaptive system under study. Once this correspondence is established, Echo computer simulations could be used as an environment to investigate the behavior of the target system. For example, Echo could be used to study the effects of introducing individual agents into an established site to determine if there are any predictable outcomes. The assumption being that the same type of perturbation in the target system would elicit a similar response.

Although resource transformation is implemented within this version of the model, no analysis has been done on its impact to agent behaviors. Resource transformation would be an attractive addition to the model analysis since the model would be capable potentially of simulating production specialization and provide an opportunity to simulate more complex economic behaviors. In addition to the previously mentioned research activities, features such as agent migration and agent aggregation could also be added to the model (each has been suggested in Hidden Order). These features would also extend the models ability to simulate complex adaptive system behaviors.

7. Conclusions.

I believe that the data presented in the earlier figures clearly illustrate the richness of the dynamics being created by interactions between the agents in this basic Echo model and that further study is both warranted and necessary to produce more formal descriptions of these dynamics. If the correspondence between the Echo model and additional complex adaptive systems could be established it would provide a basis for applying the same analytic techniques to those target systems and better establish Echo's use as a tool for furthering the study of complex adaptive systems behaviors in general.

Bibliography

1. Holland, John, Hidden Order, How Adaptation Builds Complexity. Addison Wesley, 1995. ISBN 0-201-44230-2.

2. Forrest, Stephanie and Terry Jones, "Modeling Complex Adaptive Systems with Echo". http://www.csu.edu.au/ci/vol02/forrest/FORREST.html

3. Schmitz, Oswald J. and Ginger Booth, "Modeling food web complexity: the consequence of individual-based spatially explicit behavioral ecology on trophic interactions". http://peaplant.biology.yale.edu:8001/papers/gecko.html

4. Santa Fe Institute, "Echo". http://www.santafe.edu/projects/echo/echo.html

5. Wasserman, Stanley and Katherine Faust, Social Network Analysis: Methods and Applications. Cambridge University Press, 1994. ISBN 0-521-38707-8

6. Arnold, Ken, James Gosling, and David Holmes, The Java™ Programming Language, Third Edition. Addison Wesley, June 2000. ISBN 0-201-70433-1.

7. Tenenbaum, Aaron M. and Moshe J. Augenstein, Data Structures Using Pascal. Prentice Hall, 1981. ISBN0-13-196501-8.

Appendix A.

A Note About the Implementation

The model has been implemented in Java using the JDK 1.3 (downloaded from the Sun Microsystems Java web site). The standard Java pseudorandom number generator has been used. It generates uniformly-distributed floating-point, pseudorandom numbers, *r*, where $0.0 \measuredangle r \| 1.0$, using a linear, congruential generator [6]. Graph matrix algorithms are from Tenenbaum[7]. The post-processing of the log files was done using additional Java programs

The model has been executed on an Intel 667 MHz Pentium III computer with 512 MBytes of RAM and other similar systems. The model has been executed on both the Microsoft Windows NT and Windows 2000 operating systems.

Mathcad 8 from MathSoft, Inc. was used to create the plots and graphs from post-processed log files.

Appendix B.
Java Source Code Listing

```
/**
 * Title:       World
 * Description:  An Echo world.
 * Copyright:    Copyright (c) 2001
 * Company:
 * @author David L. Harris
 * @version 1.0
 */

import java.util.*;

public class World {
   private String      name  = "Dave's World";
   private Site[][]    sites = new Site[XSITES][YSITES];

   // Define some general runtime parameters.
   public static final int    RUNTIME     = 6000;    // 6000 Number of timesteps.
   public static final int    NUMWORLDS   = 1;       // Number of worlds.
   public static final int    XSITES      = 1;       // World X dimension.
   public static final int    YSITES      = 1;       // World Y dimension.
   public static final int    AGENTS      = 1000;    // 1000 Initial population.
   public static final int    RESLENGTH   = 20;      // 15 Initial reservoir length
   public static final int    PCI         = 1;       // Resource absorption rate.
   public static final int    MTR         = 1;       // Resource dissipation rate.
   public static final int    POVERTY     = 15;      // Poverty level
   public static final double PDEATH      = 0.90;    // 0.90 Probability of death from lack of resources
   public static final int    LIFESPAN    = 750;     // 750 Oldest age an agent can live without reproducing
   public static final int    GDP         = 150;     // 150 PCI * AGENTS;  Fountain refill amount
   public static final int    OTAGLEN     = 6;       // Offense tag length
   public static final int    DTAGLEN     = 6;       // Defense tag length
   public static final int    CFLDLEN     = 6;       // Control field length
   public static final int    ETAGLEN     = 6;       // Exchange condition length
   public static final int    MTAGLEN     = 6;       // Mating condition length
   public static final int    TTAGLEN     = 6;       // Transform condition length
   public static final double PEXCHANGE   = 0.85;    // 0.85 Probability of an exchange encounter
   public static final int    EXCHANGES   = 10;      // 10 Condition score for an exchange interaction
   public static final double PAVOID      = 0.10;    // 0.10 Probability of avoiding exchange encounter
   public static final double PTRANSFORM  = 0.025;   // 1.00 Probability of transforming resources
   public static final double PMATING     = 0.75;    // 0.75 Probability of a mating encounter
   public static final int    MATINGS     = 7;       // 7 Condition score for a mating interaction
   public static final double PMUTATION   = 0.03;    // 0.03 Probability of a chromosome mutation during mating
   public static final double PCROSSOVER  = 0.70;    // 0.70 Probability of crossover during mating

   public static long   seed;
   public static int    runTime     = RUNTIME;
   public static int    numWorlds   = NUMWORLDS;
   public static int    xSites      = XSITES;
   public static int    ySites      = YSITES;
   public static int    agents      = AGENTS;
   public static int    resLength   = RESLENGTH;
   public static int    pci         = PCI;
   public static int    mtr         = MTR;
   public static int    poverty     = POVERTY;
   public static double pDeath      = PDEATH;
   public static int    lifeSpan    = LIFESPAN;
   public static int    gdp         = GDP;
   public static int    oTagLen     = OTAGLEN;
   public static int    dTagLen     = DTAGLEN;
   public static int    cFldLen     = CFLDLEN;
   public static int    eTagLen     = ETAGLEN;
   public static int    mTagLen     = MTAGLEN;
   public static int    tTagLen     = TTAGLEN;
   public static double pExchange   = PEXCHANGE;
   public static int    exchangeS   = EXCHANGES;
   public static double pAvoid      = PAVOID;
   public static double pTransform  = PTRANSFORM;
   public static double pMating     = PMATING;
   public static int    matingS     = MATINGS;
   public static double pMutation   = PMUTATION;
   public static double pCrossover  = PCROSSOVER;

   public static Random random = new Random(0);

   public static Log    agent;
   public static Log    aggregates;
   public static Log    census;
   public static Log    chromosome;
   public static Log    exchange;
   public static Log    incidence;
   public static Log    tableau;
   public static Log    unique;       //  Unique agent types
   public static Log    vitals;       //  Vital statistics for each individual agent
   public static final boolean agentLog      = false;
   public static final boolean aggregatesLog = true;
   public static final boolean censusLog     = false;
   public static final boolean chromosomeLog = false;
   public static final boolean exchangeLog   = false;
   public static final boolean incidenceLog  = true;
   public static final boolean uniqueLog     = false;
   public static final boolean vitalsLog     = true;

   //  A world is comprised of a set of sites where each site is occupied
   //    by a population of agents.
   //  The sites within a world are spatially related to each other
   //    in some way.
   //  Agents may migrate from one site to another related site, thus
   //    providing one mechanism for allowing for variation in a world.
   //  A world's evolution is based on the adaptations of the agents
   //    to their sites, to other agents within their site, and to
   //    the migration of agents between sites.

   public static void main(String[] args) {
       String LogPrefix = "C:/Logs/Echo/";

       //  Create the logs.
       tableau = new Log(LogPrefix + "tableau", true);
       tableau.logIt("Openned log files: \r\n");
       if (agentLog) {
           agent  = new Log(LogPrefix + "agent",  true);
           tableau.logIt(agent.getFileName() + "\r\n");
       }  //  if
       if (aggregatesLog) {
           aggregates = new Log(LogPrefix + "aggregates", true);
           tableau.logIt(aggregates.getFileName() + "\r\n");
       }  //  if
```

```
if (censusLog) {
   census = new Log(LogPrefix + "census", true);
   tableau.logIt(census.getFileName() + "\r\n");
}  //  if
if (chromosomeLog) {
   chromosome = new Log(LogPrefix + "chromosome", true);
   tableau.logIt(chromosome.getFileName() + "\r\n");
}  //  if
if (exchangeLog) {
   exchange = new Log(LogPrefix + "exchange", true);
   tableau.logIt(exchange.getFileName() + "\r\n");
}  //  if
if (incidenceLog) {
   incidence = new Log(LogPrefix + "incidence", true);
   tableau.logIt(incidence.getFileName() + "\r\n");
}  //  if
if (uniqueLog) {
   unique = new Log(LogPrefix + "unique", true);
   tableau.logIt(unique.getFileName() + "\r\n");
}  //  if
if (vitalsLog) {
   vitals = new Log(LogPrefix + "vitals", true);
   tableau.logIt(vitals.getFileName() + "\r\n");
}  //  if

Date start = new Date();
//seed   = 8998867453600502784L;
seed   = (long) (Math.random() * Long.MAX_VALUE);
random = new Random(seed);
System.out.println("Random seed = " + seed);
tableau.logIt("\r\nStart time " + start
          );
log(tableau);

//  Simulate the worlds.
Tag siteOTag = new Tag(OTAGLEN);
Tag siteDTag = new Tag(DTAGLEN);
for (int i = 1; i <= numWorlds; i++) {
   start = new Date();
   tableau.logIt("\r\nStart of run  " + i
              + "  Time "         + start
              );

   //  Create a new world and run the simulation.
   World world = new World();
   world.sites[0][0].setOffenseTag(siteOTag);
   world.sites[0][0].setDefenseTag(siteDTag);
   try {
      world.simulate();
   } catch (Exception e) {
      System.out.println("End of run, exception caught.");
   }  //  try

   //  Ouput some useful information.
   Date stop = new Date();
   double etSeconds = (double) (stop.getTime() - start.getTime()) / 1000.0;
   double etMinutes = etSeconds / 60.0;
   double etHours   = etMinutes / 60.0;
   System.out.println("End of run " + i
                  + "  Seconds = " + etSeconds
```

```
                   + "  Minutes = " + etMinutes
                   + "  Hours = "   + etHours
                   );
         System.out.println("Total agents created = " + Agent.getAgentNextId()
                   );
         Reservoir.printTransforms();
         Reservoir.logTransforms(tableau);
         Resource.print();
         Resource.log(tableau);

         tableau.logIt("\r\nTotal agents created       " + Agent.getAgentNextId()
                 + "\r\n\r\nEnd time              " + stop
                 + "\r\nExecution time, seconds  " + etSeconds
                 + "\r\nExecution time, minutes  " + etMinutes
                 + "\r\nExecution time, hours    " + etHours
                 + "\r\n"
                 );

         logAggregates(world, i, aggregates);
         Agent.setAgentNextId(0);
         Reservoir.resetAggregates();
         tableau.flush();
      }  //  for i

      //  Close out the logs.
      if (agentLog)      agent.close();
      if (aggregatesLog) aggregates.close();
      if (censusLog)     census.close();
      if (chromosomeLog) chromosome.close();
      if (exchangeLog)   exchange.close();
      if (incidenceLog)  incidence.close();
      if (uniqueLog)     unique.close();
      if (vitalsLog)     vitals.close();
      tableau.close();

      System.out.println("World done.");

   }  //  main

   public World() {
      //  Create each site.
      for (int i = 0; i < sites.length; i++) {
         for (int j = 0; j < sites[i].length; j++) {
            String name = "Site" + i  + j;
            sites[i][j] = new Site(name, exchange, incidence, chromosome, vitals);
         }  //  for j
      }  //  for i

   }  //  World

   public World(boolean specified) {
      String   oTag   = "cccccc";
      String   dTag   = "cccccc";
      String[] agents = {"aaaaaaccbaaaccccccaaaaaa",
                      "ccccccbbaaabcccaaacccccc"};

      String[] agents2 = {"aaaaaacccccccaaaaaaaaaaaa",
                       "ccccccaaaaaaaaaaaacccccc"};
      //  Create each site.
      for (int i = 0; i < sites.length; i++) {
        for (int j = 0; j < sites[i].length; j++) {
            String name = "Site" + i  + j;
            sites[i][j] = new Site(name, oTag, dTag, agents, exchange, incidence, chromosome, vitals);
        }  //  for j
    }  //  for i

}  //  World

//  Simulate the evolution of the world by the progression
//    of time as a sequence of discrete time steps.  Stop
//    if the world dies or runs out of time.
public void simulate() {
    boolean deadWorld;
    for (int i = 1; i < runTime; i++) {
        //  Execute the next timestep.
        deadWorld = timeStep(i);
        if (deadWorld) {
            System.out.println("World has died.");
            break;
        }  //  if deadWorld
        System.gc();
    }  //  for i

}  //  simulate

/**
*    This is where all the action starts.
*/
public boolean timeStep(int time) {
    boolean deadWorld = true;

    //  Evolve each living site.
    for (int i = 0; i < sites.length; i++) {
        for (int j = 0; j < sites[i].length; j++) {
            //  See if the site is alive.
            if (sites[i][j].alive()) {
                if (time == 1000) {
                    sites[i][j].resetAggregates();
                }  //  if time
                deadWorld = false;
                sites[i][j].evolve(time);
                if (!sites[i][j].alive()) {
                    tableau.logIt("Site " + sites[i][j].name
                                + " died at " + time
                               );
                }  //  if !alive

                //  Output some useful statistics.
                log(sites[i][j], time, true);
                if ((time < 25) || (time % 25 == 0) || (time == runTime -1)) {
                     sites[i][j].print(false, time);
                     //sites[i][j].resetStats();
                }  //  if time
            }  //  if alive
        }  //  for j
    }  //  for i

    return deadWorld;

}  //  timeStep
```

```
public static void adjacent() {
   //  Determine which sites are adjacent to each other.
}  //  adjacent

public static void logAggregates(World world, int i, Log log) {
   if (log == null) return;
   for (int x = 0; x < world.sites.length; x++) {
      for (int y = 0; y < world.sites[x].length; y++) {
         world.sites[x][y].logVitals(vitals);
         log.logIt("World" + i
                   + "  "  + world.sites[x][y].name
                   + "  "  + world.sites[x][y].getTime()
                   + "  "  + world.sites[x][y].getNumBirths()
         );
         world.sites[x][y].logAgentTypes(unique);
         if (unique != null) unique.flush();
         world.sites[x][y].logAggregates(log);
         Reservoir.logAggregates(log);
         if (log != null ) aggregates.logIt("\r\n");
      }  //  for y
   }  //  for x
   log.flush();
}  //  logAggregates

//  Log some useful statistics.
private void log(Site site, int time, boolean full) {

   if (time == 1 && tableau != null) {
      tableau.logIt(  "\r\nSite                    " + site.getName()
                 + "\r\nOffense Tag             " + site.getOffenseTag()
                 + "\r\nDefense Tag             " + site.getDefenseTag()
                 + "\r\n\r\n"
                 );
   }  //  if time
   if (agentLog) {
      if (time % 25 == 0) {   // || full) {
         site.logAgents(agent);
      }  //  if
      agent.flush();
   }  //  if agentLog
   if (censusLog) {
      site.log(census, time);
      census.flush();
   }  //  if censusLog

}  //  log

//  Log system run default parameters.
public static void log(Log log) {
   if (log == null) return;
   log.logIt(  "\r\nRandom seed             " + seed
          + "\r\nRun time - time steps    " + runTime
          + "\r\nWorlds                   " + numWorlds
          + "\r\nX sites                  " + xSites
          + "\r\nY sites                  " + ySites
          + "\r\nAgents                   " + agents
          + "\r\nInitial reservoir length " + resLength
          + "\r\nPCI                      " + pci
          + "\r\nMTR                      " + mtr
```

```
                + "\r\nPoverty level            " + poverty
                + "\r\nProbability of death     " + pDeath
                + "\r\nLife span                " + lifeSpan
                + "\r\nGDP                      " + gdp
                + "\r\nOffense tag length       " + oTagLen
                + "\r\nDefense tag length       " + dTagLen
                + "\r\nControl field length     " + cFldLen
                + "\r\nExchange tag length      " + eTagLen
                + "\r\nMating tag length        " + mTagLen
                + "\r\nTransform tag length     " + tTagLen
                + "\r\nProbability of Exchange  " + pExchange
                + "\r\nMinimum exchange score   " + exchangeS
                + "\r\nProbability of avoidance " + pAvoid
                + "\r\nProbability of transform " + pTransform
                + "\r\nProbability of mating    " + pMating
                + "\r\nMinimum mating score     " + matingS
                + "\r\nProbability of mutation  " + pMutation
                + "\r\nProbability of crossover " + pCrossover
                + "\r\n\r\n"
              );
        log.flush();
    }  //  log

    public void print(boolean full, int time) {
        System.out.println("World: " + name);
        if (full) {
            for (int i = 0; i < sites.length; i++) {
                for (int j = 0; j < sites[0].length; j++) {
                    sites[i][j].print(true, time);
                }  //  for j
            }  //  for i
        }  //  if full
    }  //  print

}  //  class World


/**
 * Title:        Site
 * Description:  An Echo site.
 * Copyright:    Copyright (c) 2001
 * Company:
 * @author David L. Harris
 * @version 1.0
 *
 * November 16, 2000  Added the shuffle method to randomly permute the order in
 *                    which the agents absorb resources from the fountain.
 */
import java.util.*;

class Site {
    Reservoir fountain;
    Tag       offenseTag;
    Tag       defenseTag;
    Agent[]   agents;

    int       time = 0;
    String    name;
    Log       eLog = null;        //  Exchange log.
    Log       iLog = null;        //  Incidence log.
Log       cLog = null;         //  Chromosome log.
Log       vLog = null;         //  Vitals log.
String[]  chromosomes;
String[]  species;
Map       agentTypes      = new TreeMap();
int       totalPopulation = 0;
int       numBirths       = 0;
int       numDeaths       = 0;
int       numExEncs       = 0;
int       numExchanges    = 0;
int       amtExchanged    = 0;
int       amtAbsorbed     = 0;
int       amtDissipated   = 0;
int       numMatings      = 0;
int       numReservoir    = 0;
int       maxReservoir    = 0;

//  Define some runtime parameters.
private static final int    agentLen       = World.agents;
private static final int    PCI            = World.pci;
private static final int    MTR            = World.mtr;
private static final int    poverty        = World.poverty;
private static final double pDeath         = World.pDeath;
private static final int    lifeSpan       = World.lifeSpan;
private static final int    GDP            = World.gdp;
private static final int    oTagLen        = World.oTagLen;
private static final int    dTagLen        = World.dTagLen;
private static final double pExchangeEnc   = World.pExchange;
private static final int    exchangeEncS   = World.exchangeS;
private static final double pAvoidExchange = World.pAvoid;
private static final double pTransform     = World.pTransform;
private static final double pMatingEnc     = World.pMating;
private static final int    matingEncS     = World.matingS;

//  Sites provide the agents with a fountain of resources to renew their
//    resources and an environment within which they can interact.
//  The site has an offense and a defense tag that agents match against
//    when dissipating or absorbing resources to or from the fountain.
//  The evolution of a site is dependent on the adaptation of the agents
//    to the site and to other agents within the site.
//  The site also controls the sequencing of operations that determines
//    the evolutionary behavior of the agents at each time step.
//  This sequence includes agent encounters for exchanging resources and
//    for mating to produce offspring.

//  Create a new site with random agents and random offense and defense tags.
Site(String name, Log eLog, Log iLog, Log cLog, Log vLog) {
   this.name = name;
   this.eLog = eLog;
   this.iLog = iLog;
   this.cLog = cLog;
   this.vLog = vLog;
   fountain  = new Reservoir(0);
   agents    = new Agent[agentLen];
   //  Create each agent.
   for (int i = 0; i < agents.length; i++) {
      agents[i] = new Agent(name + " " + i, time);
   }  //  for i
   offenseTag = new Tag(oTagLen);
   defenseTag = new Tag(dTagLen);
```

```
    resetAggregates();
    stats();

  }  //  Site

  //  Create a new site with specified offense tag, defense tag, and agents.
  Site(String name, String oTag, String dTag, String[] c, Log eLog, Log iLog, Log cLog, Log vLog) {
    this.name = name;
    this.eLog = eLog;
    this.iLog = iLog;
    this.cLog = cLog;
    this.vLog = vLog;
    fountain  = new Reservoir(0);
    agents    = new Agent[agentLen];
    //  Create each agent.
    for (int i = 0; i < agents.length; i++) {
      int j = i % c.length;
      agents[i] = new Agent(name + " " + i, time, c[j]);
    }  //  for i
    offenseTag = new Tag(oTag);
    defenseTag = new Tag(dTag);
    resetAggregates();
    stats();

  }  //  Site

  //  Evolve the site:
  //    Replenish the site fountain.
  //    Exchange resources between agents.
  //    Transform resources.
  //    Mate agents together.
  //    Exchange resources with the fountain.
  //    Allow agents to migrate between sites.  (not yet)
  //    Kill off the wimpy or impotent agents.
  //    Collect some statistics.
  public void evolve(int time) {
    this.time = time;
    replenish();
    exchange();
    transform();
    mate();
    absorb();
    dissipate();
    migrate();
    kill();
    stats();
  }  //  evolve

  //  Renew the site fountain.
  public void replenish() {
    //fountain = new Reservoir(GDP);
    fountain.acquireFrom(new Reservoir(GDP), GDP);
  }  //  replenish

  //  Exchange resources between agents.
  public void exchange() {
    Agent     a1;
    Agent     a2;
    Reservoir r1;
    Reservoir r2;
```

```
boolean[] picked;

int    id1;
Tag    oTag1;
Tag    dTag1;
Tag    eTag1;
String as1;
int    eCond1  = 0;
int    eScore1 = 0;
int    exc1    = 0;

int    id2;
Tag    oTag2;
Tag    dTag2;
Tag    eTag2;
String as2;
int    eCond2  = 0;
int    eScore2 = 0;
int    exc2    = 0;


if (pExchangeEnc == 0.0) return;
//  Randomly pick a set of agents for an exchange encounter.
picked = pick(pExchangeEnc, agents.length);

//  Match exchange conditions and exchange resources depending on the scores.
//  Match tags and exchange resources depending on the scores.
for (int i = 0; i < agents.length; i++) {
   if (picked[i]) {
      //  Get the first agent.
      a1      = agents[i];
      a1.incNumExchangeEncounters(1);
      numExEncs++;
      id1     = a1.getId();
      oTag1   = a1.getOffenseTag();
      dTag1   = a1.getDefenseTag();
      eTag1   = a1.getExchangeCond();
      as1     = oTag1.toString() + dTag1.toString() + eTag1.toString();
      eCond1  = 0;
      eScore1 = 0;
      exc1    = 0;
      //  Randomly pick a different second agent for the encounter.
      int secondPick = World.random.nextInt(agents.length);
      if (secondPick == i) {
         continue;
      }  //  if
      a2      = agents[secondPick];
      a2.incNumExchangeEncounters(1);
      numExEncs++;
      id2     = a2.getId();
      oTag2   = a2.getOffenseTag();
      dTag2   = a2.getDefenseTag();
      eTag2   = a2.getExchangeCond();
      as2     = oTag2.toString() + dTag2.toString() + eTag2.toString();
      eCond2  = 0;
      eScore2 = 0;
      exc2    = 0;

      //  See if the exchange conditions match.
      //    Match a1's offense tag with a2's exchange condition.
```

```
        eCond1 = a1.matchMyExchange(a2);
        a1.setExchangeCondScore(eCond1);
        eCond2 = a2.matchMyExchange(a1);
        a2.setExchangeCondScore(eCond2);
        r1 = a1.getReservoir();
        r2 = a2.getReservoir();

        //  See who exchanges resources with whom.
        //    If both conditions are satiisfied, then exchange resources both ways.
        //    If neither condition is satisfied, then no resources are transferred.
        //    If only one condition is, then the other agent can avoid an exchange.
        double a1Avoid = World.random.nextDouble();
        double a2Avoid = World.random.nextDouble();
        if (eCond1 > exchangeEncS &&
            (eCond2 > exchangeEncS || a2Avoid < pAvoidExchange)
           ) {
            //  Exchange condition met:
            //    If a1's offensive tag matches a2's defensive tag then
            //      a1 may acquire resources from a2.
            eScore1 = a1.matchMyOff(a2);
            a1.setEncounterOffScore(eScore1);
            if (eScore1 > 0) {
                exc1 = r1.acquireFrom(r2, eScore1);
                amtExchanged += exc1;
                a1.incAmtGained(exc1);
                a2.incAmtLost(exc1);
            }  //  if
            numExchanges++;
            a1.incNumExchanges(1);
            a2.incNumExchanges(1);
        }  //  if eCond1
        if (eCond2 > exchangeEncS &&
            (eCond1 > exchangeEncS || a1Avoid < pAvoidExchange)
           ) {
            //  Exchange condition met:
            //    Now, match a2's offense tag with a1's defense tag,
            //      a2 may acquire resources from a1.
            eScore2 = a2.matchMyOff(a1);
            a2.setEncounterDefScore(eScore2);
            if (eScore2 > 0) {
                exc2 = r2.acquireFrom(r1, eScore2);
                amtExchanged += exc2;
                a2.incAmtGained(exc2);
                a1.incAmtLost(exc2);
            }  //  if
            numExchanges++;
            a2.incNumExchanges(1);
            a1.incNumExchanges(1);
        }  //  if eCond2
        if (exc1 > 0 || exc2 > 0) {
            logExchange(eLog, time,
                        id1, oTag1, dTag1, eTag1, eCond1, eScore1, exc1,
                        id2, oTag2, dTag2, eTag2, eCond2, eScore2, exc2
                       );
            logIncidence(iLog, time, as1, id1, exc1, as2, id2, exc2);
        }  //  if
      }  //  if picked
    }  //  for i

  }  //  exchange
```

```
// Allow each agent to transform its resources.
public void transform() {

   if (pTransform == 0.0) return;
   // Calculate resource transformations for all agents.
   for (int i = 0; i < agents.length; i++) {
      agents[i].transform(pTransform);
   } // for i

} // transform

// Mate agents together.
public void mate() {
   boolean[] picked;
   Vector    births = new Vector();
   int       a1CondS;
   int       a2CondS;
   Agent     a1;
   Agent     a2;
   Agent     b1;
   Agent     b2;

   // Randomly pick agents for a mating encounter.
   picked = pick(pMatingEnc, agents.length);

   // Toss out the infertile ones.
   for (int i = 0; i < picked.length; i++) {
      if (picked[i]) {
         picked[i] = agents[i].canMate();
      } // if
   } // for i

   // Breed the fertile agents.
   int i = 0;
   int j = 0;
   while (i < agents.length) {
      if (picked[i]) {
         // Only one chance to find a mate
         picked[i] = false;
         j = i + 1;
         // Look at all the fertile ones for a mate
         while (j < agents.length) {
            if (picked[j]) {
               a1 = agents[i];
               a1.incNumMatingEncounters(1);
               a2 = agents[j];
               a2.incNumMatingEncounters(1);
               a1CondS = a1.matchMyMating(a2);
               a1.setMatingCondScore(a1CondS);
               a2CondS = a2.matchMyMating(a1);
               a2.setMatingCondScore(a2CondS);
               if (a1CondS > matingEncS &&
                  a2CondS > matingEncS
                 ){
                  picked[j] = false; // only mate once
                  b1 = a1.replicate(time);
                  a1.setLastMating(time);
                  a1.incNumSiblings(1);
                  b2 = a2.replicate(time);
```

```
                    a2.setLastMating(time);
                    a2.incNumSiblings(1);
                    Agent.mate(b1, b2);
                    b1.logC(cLog, time);
                    b2.logC(cLog, time);
                    births.addElement(b1);
                    births.addElement(b2);
                    mapAgent(b1);
                    mapAgent(b2);
                    numBirths += 2;
                    numMatings++;
                    break;
                  }
                } // if j picked
                j++;
              } // while j
            } // if i picked
            i++;
          } // while

          // Put the births into the site.
          if (births.size() > 0) {
            Agent[] newAgents = new Agent[agents.length + births.size()];
            for (int k = 0; k < agents.length; k++) {
              newAgents[k] = agents[k];
            } // for k
            for (int k = 0; k < births.size(); k++) {
              newAgents[agents.length + k] = (Agent) births.elementAt(k);
            } // for k
            agents = newAgents;
          } // if

        } // mate

        // Allow the agents to acquire resources from the site fountain.
        public void absorb() {
          Reservoir agentRes;
          Tag       agentTag;
          int       score = 0;
          int       amt   = 0;
          int[]     orders;

          // Allow each agent, in a random order, to absorb resources from the site.
          orders = shuffle(agents.length);
          for (int i =0; i < agents.length; i++) {
            int a = orders[i];
            agentTag = agents[a].getOffenseTag();
            agentRes = agents[a].getReservoir();
            // If the agent's offense tag matches the site's defense
            //   tag then the agent will acquire resources from the
            //   fountain (the score plus the per capita income).
            score = agentTag.matchTag(defenseTag);
            agents[a].setSiteOffScore(score);
            if (score > 0) {
              amt = agentRes.acquireFrom(fountain, score + PCI);
              amtAbsorbed += amt;
              agents[a].incAmtAbsorbed(amt);
            } // if
          } // for i
```

```
}  //  absorb

//  Allow the agents to return resources to the site fountain.
public void dissipate() {
    Reservoir agentRes;
    Tag       agentTag;
    int       score = 0;
    int       amt   = 0;

    //  Allow each agent to return resources to the site.
    for (int i =0; i < agents.length; i++) {
        agentTag = agents[i].getDefenseTag();
        agentRes = agents[i].getReservoir();
        //  If the site's offense tag matches the agent's defense
        //    tag then the agent will dissipate resources to the
        //    fountain (the score plus the marginal tax rate).
        score = offenseTag.matchTag(agentTag);
        agents[i].setSiteDefScore(score);
        if (score > 0) {
            amt = fountain.acquireFrom(agentRes, score + MTR);
            amtDissipated += amt;
            agents[i].incAmtDissipated(amt);
        }  //  if
    }  //  for i

}  //  dissipate

//  Allow the agents to migrate to a different site.
public void migrate() {
}  //  migrate

//  Kill off agents.
public void kill() {
    Agent     agent;
    Reservoir reservoir;
    int       alive  = agents.length;
    Agent[]   survivors;

    //  Kill off agents we don't like with pDeath probability
    //    (the wimpy ones with resources below the poverty level or
    //    if you don't mate, you die).
    for (int i =0; i < agents.length; i++) {
        agent     = agents[i];
        reservoir = agent.getReservoir();
        if ((reservoir.length() < poverty && World.random.nextDouble() < pDeath)
          || (agent.getAge(time) > lifeSpan && (time - agent.getLastMating()) > lifeSpan)) {
            agent.setDoD(time);
            agent.logVitals(vLog, time);
            agents[i] = null;
            alive--;
            numDeaths++;
        }  //  if
    }  //  for i

    survivors = new Agent[alive];
    int j = 0;
    for (int i =0; i < agents.length; i++) {
        if (agents[i] != null) {
             survivors[j++] = agents[i];
        }
    }  //  for i
    agents = survivors;

}  //  kill

void stats() {
    chromosomes      = Agent.getChromosomes(agents);
    species          = Agent.getSpecies(agents);
    totalPopulation += getPopulation();

    numReservoir = 0;
    maxReservoir = agents[0].getReservoirLen();
    for (int i = 0; i < agents.length; i++) {
        int len = agents[i].getReservoirLen();
        numReservoir += len;
        if (len > maxReservoir) maxReservoir = len;
    }  //  for i

}  //  stats

//  Reset the site's statistics.
public void resetAggregates() {
    totalPopulation = 0;
    numBirths       = 0;
    numDeaths       = 0;
    numExEncs       = 0;
    numExchanges    = 0;
    amtExchanged    = 0;
    amtAbsorbed     = 0;
    amtDissipated   = 0;
    numMatings      = 0;
    numReservoir    = 0;
    maxReservoir    = 0;
    agentTypes = new TreeMap();
    mapAgents(agents);
}  //  resetAggregates

public int[] shuffle(int len) {
    List     orderList = new ArrayList();
    Iterator iterator;
    int[]    shuffled = new int[len];

    for (int i = 0; i < len; i++) {
        Integer intObj = new Integer(i);
        orderList.add(intObj);
    }  //  for i
    Collections.shuffle(orderList);
    iterator = orderList.iterator();
    int i = 0;
    while (iterator.hasNext()) {
        int order = ((Integer)iterator.next()).intValue();
        shuffled[i++] = order;
    }  //  while

    return shuffled;

}  //  shuffle

//  Chose a random collection of agents.
boolean[] pick(double probability, int len) {
    boolean[] picked = new boolean[len];
    boolean   preset = false;
    double    pEnc   = probability;

    //  If the probablility of an encounter is greater than 0.5 then it
    //    is easier to find those that do not have an encounter.
    if (probability > 0.5) {
       preset = true;
       pEnc   = 1.0 - probability;
    }  //  if
    for (int i = 0; i < picked.length; i++) {
       picked[i] = preset;
    }  //  for i

    //  Randomly pick agents that may (may not) have an encounter.
    int num = (int) (picked.length * pEnc);
    for (int i = 0; i < num; i++) {
       do {
          //  Keep checking random numbers until an unpicked agent is selected.
          int pick = World.random.nextInt(picked.length);
          if (picked[pick] == preset) {
             picked[pick] = !preset;
             break;
          }  //  if
       }  while (true);
    }  //  for i

    return picked;

 }  //  pick

 private void mapAgent(Agent agent) {
    String type = agent.getChromosome().toString();

    //  See if the chromosome is in the map.
    Integer count = (Integer) agentTypes.get(type);
    if (count == null) {
       agentTypes.put(type, new Integer(1));
    } else {
       agentTypes.put(type, new Integer(count.intValue() + 1));
    }  //  if

 }  //  mapAgent

 private void mapAgents(Agent[] agents) {
    //  Map each agent.
    for (int i = 0; i < agents.length; i++) {
       mapAgent(agents[i]);
    }  //  for i

 }  //  mapAgents

 public boolean alive() {
    return (agents.length > 1 ||
            numBirths != 0 ||
            numDeaths != 0
           );
 }  //  alive

 public int getTime() {
    return time;
 }  //  getTime

 public String getName() {
    return name;
 }  //  getName

 public int getPopulation() {
    return agents.length;
 }  //  getPopulation

 public int getTotalPopulation() {
    return totalPopulation;
 }  //  getTotalPopulation

 public int getNumBirths() {
    return numBirths;
 }  //  getNumBirths

 public int getNumDeaths() {
    return numDeaths;
 }  //  getNumDeaths

 public int getNumExEncs() {
    return numExEncs;
 }  //  getNumExEncs

 public int getNumExchanges() {
    return numExchanges;
 }  //  getNumExchanges

 public int getAmtExchanged() {
    return amtExchanged;
 }  //  getAmtExchanged

 public int getAmtAbsorbed() {
    return amtAbsorbed;
 }  //  getAmtAbsorbed

 public int getAmtDissipated() {
    return amtDissipated;
 }  //  getAmtDissipated

 public int getNumMatings() {
    return numMatings;
 }  //  getNumMatings

 public Tag getOffenseTag() {
    return offenseTag;
 }  //  getOffenseTag

 public void setOffenseTag(Tag oTag) {
    this.offenseTag = oTag;
 }  //  setOffenseTag

 public Tag getDefenseTag() {
    return defenseTag;
 }  //  getDefenseTag

 public void setDefenseTag(Tag dTag) {
    this.defenseTag = dTag;
}  //  setDefenseTag

public int getMaxReservoir() {
    return maxReservoir;
}  //  getMaxReservoir

public int getNumReservoir() {
    return numReservoir;
}  //  getNumReservoir

public int getNumAgentTypes() {
    return agentTypes.size();
}  //  getNumAgentTypes

public void logAgents(Log log) {
    if (log == null) return;
    for (int i = 0; i < agents.length; i++) {
        agents[i].logC(log, time);
    }  //  for i
}  //  logAgents

public void logVitals(Log log) {
    if (log == null) return;
    for (int i = 0; i < agents.length; i++) {
        agents[i].logVitals(log, time);
    }  //  for i
}  //  logVitals

public void logUnique(Log log) {
    if (log == null) return;

}  //  logUnique

public void logIncidence(Log log,    int time,
                         String as1, int id1, int exc1,
                         String as2, int id2, int exc2
                        ) {
    if (log == null) return;
    if (time < 1000) return;
    log.logIt(Pad.pad(time, 6)
            + " " + as1
            + " " + Pad.pad(id1, 6)
            + " " + Pad.pad(exc1, 2)
            + " " + as2
            + " " + Pad.pad(id2, 6)
            + " " + Pad.pad(exc2, 2)
            + "\r\n"
           );
}  //  logIncidence

public void logExchange(Log log, int time,
                        int id1, Tag oTag1, Tag dTag1, Tag eTag1, int eCond1, int eScore1, int exc1,
                        int id2, Tag oTag2, Tag dTag2, Tag eTag2, int eCond2, int eScore2, int exc2
                       ) {
    if (log == null) return;
    log.logIt(name
            + " " + Pad.pad(time, 6)
            + " " + Pad.pad(id1, 6)
            + " " + oTag1.toString()
            + " " + dTag1.toString()
            + " " + eTag1.toString()
            + " " + Pad.pad(eCond1, 2)
            + " " + Pad.pad(eScore1, 2)
            + " " + Pad.pad(exc1, 2)
            + " " + Pad.pad(id2, 6)
            + " " + oTag2.toString()
            + " " + dTag2.toString()
            + " " + eTag2.toString()
            + " " + Pad.pad(eCond2, 2)
            + " " + Pad.pad(eScore2, 2)
            + " " + Pad.pad(exc2, 2)
            + "\r\n"
           );
}  //  logExchange

public void logAgentTypes(Log log) {
   int index = 0;

   if (log == null) return;
   for (Iterator it = agentTypes.entrySet().iterator(); it.hasNext(); ) {
      Map.Entry e = (Map.Entry) it.next();
      StringBuffer field = new StringBuffer((String)e.getKey());
      log.logIt("  " + index++
              + "  " + e.getKey()
              + "  " + e.getValue()
              + "\r\n"
             );
   }  //  for it

}  //  logAgentTypes

public void logAggregates(Log log) {
   if (log == null) return;
   log.logIt("  " + getTotalPopulation()
            + "  " + getNumBirths()
            + "  " + getNumDeaths()
            + "  " + getNumAgentTypes()
            + "  " + getNumExEncs()
            + "  " + getNumExchanges()
            + "  " + getAmtExchanged()
            + "  " + getAmtAbsorbed()
            + "  " + getAmtDissipated()
           );
}  //  logAggregates

public void log(Log log, int time) {
   if (log == null) return;
   log.logIt(name
            + " " + time
            + " " + offenseTag.toString()
            + " " + defenseTag.toString()
            + " " + getTotalPopulation()
            + " " + getPopulation()
            + " " + getNumBirths()
            + " " + getNumDeaths()
            + " " + getNumAgentTypes()
            + " " + getNumExEncs()
            + " " + getNumExchanges()
            + " " + getAmtExchanged()
            + " " + getAmtAbsorbed()
            + " " + getAmtDissipated()
            + " " + getNumMatings()
            + " " + getNumReservoir()
            + " " + getMaxReservoir()
            + "\r\n"
           );
  }  //  log

  public void print(boolean full, int time) {

    System.out.println("Time " + time
                  + " "          + name
                  + "  Pop "     + getPopulation()
                  + "  B "       + getNumBirths()
                  + "  D "       + getNumDeaths()
                  + "  Types "   + getNumAgentTypes()
                  + "  ExEncs "  + getNumExEncs()
                  + "  Exches "  + getNumExchanges()
                  + "  Exched "  + getAmtExchanged()
                  + "  Abs "     + getAmtAbsorbed()
                  + "  Diss "    + getAmtDissipated()
                  + "  Matings " + getNumMatings()
                  + "  Resrv "   + getNumReservoir()
                  + "  Max R "   + getMaxReservoir()
                 );

  }  //  print

  public class AgentType implements Comparable {
    int    count;
    String type;

    public AgentType(int count, String type) {
      this.count = count;
      this.type  = type;
    }  //  AgentType

    public int compareTo(Object o) {
      AgentType a = (AgentType) o;

      return type.compareTo(a.type);

    }  //  compareTo

    public void print() {
      System.out.println("count: " + count + " type: " + type
                      );
    }  //  print

  }  //  class AgentType

}  //  class Site


/**
 * Title:        Agent
 * Description:  An Echo agent.
 * Copyright:    Copyright (c) 2001
 * Company:
```

```
 * @author David L. Harris
 * @version 1.0
 */

class Agent {
   private int       id;
   private Chromosome chromosome;
   private Reservoir  reservoir;
   private int       DoB;
   private int       DoD;
   private int       parent1;
   private int       parent2;
   private int       numMatingEncounters;
   private int       lastMating;
   private int       numSiblings;
   private int       numTransforms;
   private int       numExchangeEncounters;
   private int       numExchanges;
   private int       amtGained;
   private int       amtLost;
   private int       amtDissipated;
   private int       amtAbsorbed;
   private int       siteOffScore;
   private int       siteDefScore;
   private int       encounterOffScore;
   private int       encounterDefScore;
   private int       exchangeCondScore;
   private int       matingCondScore;
   String           name;

   private static int    nextId     = 0;

   private static int    resLength  = World.resLength;
   private static double pMutation  = World.pMutation;
   private static double pCrossover = World.pCrossover;

   //  An agents survival and reproduction is based solely on its
   //    ability to collect resources.
   //  Agents transfer resources during exchange encounters with other
   //    agents and with the site fountain.
   //  During agent exchange encounters offense and defense tags are
   //    match scored to determine the outcomes.
   //  Agents absorb resources from the fountain based on their ability
   //    to adapt to the sites provisional biases.
   //  They dissipate resources to the fountain based on the site's
   //    ability to extract resources from the agent.

   //  Create a new Agent with a random chromosome and reservoir.
   Agent(String name, int DoB) {
      this.name  = name;
      this.DoB   = DoB;
      chromosome = new Chromosome();
      reservoir  = new Reservoir(resLength);
      id         = getNewId();
   }  //  Agent

   //  Create a new agent that has chromosome c and a random reservoir.
   Agent(String name, int DoB, String c) {
      this.name  = name;
      this.DoB   = DoB;
```

```
    chromosome = new Chromosome(c);
    reservoir  = new Reservoir(resLength);
    id         = getNewId();
} // Agent

// Create a new agent that has chromosome c and reservoir r.
Agent(String name, int DoB, String c, String r) {
    this.name  = name;
    this.DoB   = DoB;
    chromosome = new Chromosome(c);
    reservoir  = new Reservoir(r);
    id         = getNewId();
} // Agent

// Determine if the agent has enough resources in its reservoir
//   to replicate itself.
public boolean canMate() {
    boolean fertile = true;
    boolean found   = false;

    // See if my reservoir contains another me in it.
    char[] c  = chromosome.toString().toCharArray();
    char[] r  = reservoir.toString().toCharArray();
    int    cl = c.length;
    int    rl = r.length;

    // Check the agent's reservoir.
    if (cl > rl) {
        fertile = false;
    } else {
        // See if each resource in the chomosome has a
        //   match in the reservoir.
        for (int i = 0; i < cl; i++) {
            found = false;
            for (int j = 0; j < rl; j++) {
                if (c[i] == r[j]) {
                    r[j] = ' ';
                    found = true;
                    break;
                } // if
            } // for j
            if (!found) {
                fertile = false;
                break;
            } // if !found
        } // for i
    } // if

    return fertile;

} // canMate

// Apply crossover and mutation operators to two mating agents.
public static void mate(Agent a1, Agent a2) {

    // Crossover the contensts of two agent's chromosomes.
    if (World.random.nextDouble() < pCrossover) {
        Chromosome c1 = a1.getChromosome();
        Chromosome c2 = a2.getChromosome();
        Chromosome.crossover(c1, c2);
        a1.setChromosome(c1);
        a2.setChromosome(c2);
    }  //  if

    //  Assign parent Ids.
    a1.parent2 = a2.parent1;
    a2.parent2 = a1.parent1;

    //  Mutate resources within the agents chromosomes.
    if (World.random.nextDouble() < pMutation) {
        a1.chromosome.mutate();
    }  //  if
    if (World.random.nextDouble() < pMutation) {
        a2.chromosome.mutate();
    }  //  if

}  //  mate

//  Transform two resources from the reservoir into other resources.
public void transform(double pTransform) {
    char f;
    char t;
    char c;
    Tag  tag = chromosome.getTransformCond();

    if (World.random.nextDouble() <= pTransform) {
        f = tag.charAt(0);
        t = tag.charAt(1);
        c = tag.charAt(2);
        if (reservoir.transform(f, t, c)) {
            incNumTransforms(1);
        }  //  if transformed
    }  //  if
    if (World.random.nextDouble() <= pTransform) {
        f = tag.charAt(3);
        t = tag.charAt(4);
        c = tag.charAt(5);
        if (reservoir.transform(f, t, c)) {
            incNumTransforms(1);
        }  //  if transformed
    }  //  if

}  //  transform

//  Create a new agent by removing resources from the reservoir
//    and splitting any of the remaining resources.
public Agent replicate(int time) {
    Agent birth = null;

    char[] c  = chromosome.toString().toCharArray();
    char[] r  = reservoir.toString().toCharArray();
    int    cl = c.length;
    int    rl = r.length;
    char[] leftOvers;

    //  Remove resources from the reservoir to duplicate the chromosome.
    for (int i = 0; i < cl; i++) {
        for (int j = 0; j < rl; j++) {
            if (c[i] == r[j]) {
                r[j] = ' ';
                break;
            }  //  if
        }  //  for j
    }  //  for i

    //  Split the remaining resources.
    leftOvers = new char[rl - cl];
    int j = 0;
    for (int i = 0; i < rl; i++) {
        if (r[i] != ' ') {
            leftOvers[j++] = r[i];
        }  //  if
    }  //  for i
    String rstr = new String(leftOvers);
    int h = j / 2;
    String ores = new String(rstr.substring(0, h));
    String nres = new String(rstr.substring(h, j));
    reservoir   = new Reservoir(ores);

    //  Create the new agent.
    String cstr   = new String(c);
    String name   = this.name + "b";
    birth         = new Agent(name, time, cstr, nres);
    birth.parent1 = id;

    return birth;

}  //  replicate

//  Match my offense tag with agent a1's defense tag.
public int matchMyOff(Agent a1) {
    Tag oTag  = chromosome.getOffenseTag();
    Tag dTag  = a1.chromosome.getDefenseTag();
    int score = oTag.matchTag(dTag);

    return score;

}  //  matchMyOff

//  Match my exchange condition with agent a1's offense tag.
public int matchMyExchange(Agent a1) {
    Tag eCond = chromosome.getExchangeCond();
    Tag oTag  = a1.chromosome.getOffenseTag();
    int score = eCond.matchCondition(oTag);

    return score;

}  //  matchMyExchange

//  Match my mating condition with agent a1's offense tag.
public int matchMyMating(Agent a1) {
    Tag mCond = chromosome.getMatingCond();
    Tag oTag  = a1.chromosome.getOffenseTag();
    int score = mCond.matchCondition(oTag);

    return score;

}  //  matchMyMating

public static synchronized int getNewId() {
    return nextId++;
} //  getNewId

public static synchronized int getAgentNextId() {
    return nextId;
} //  getAgentNextId

public static synchronized void setAgentNextId(int num) {
    nextId = num;
} //  setAgentNextId

public int getDoB() {
    return DoB;
} //  getDoB

public int getDoD() {
    return DoD;
} //  getDoD

public void setDoD(int dead) {
    DoD = dead;
} //  setDoD

public int getAge(int time) {
    int age;

    if (DoD != 0) {
        return DoD - DoB;
    } else {
        return time - DoB;
    } //  if

} //  getAge

public int getNumMatingEncounters() {
    return numMatingEncounters;
} //  getNumMatingEncounters

public void incNumMatingEncounters(int inc) {
    numMatingEncounters += inc;
} //  incNumMatingEncounters

public int getLastMating() {
    return lastMating;
} //  getLastMating

public void setLastMating(int time) {
    lastMating = time;
} //  setLastMating

public int getNumSiblings() {
    return numSiblings;
} //  getNumSiblings

public void incNumSiblings(int inc) {
    numSiblings += inc;
} //  incNumSiblings

public int getNumTransforms() {
    return numTransforms;
```

```
}  //  getNumTransforms

public void incNumTransforms(int inc) {
   numTransforms += inc;
}  //  incNumTransforms

public int getNumExchangeEncounters() {
   return numExchangeEncounters;
}  //  getNumExchangeEncounters

public void incNumExchangeEncounters(int inc) {
   numExchangeEncounters += inc;
}  //  incNumExchangeEncounters

public int getNumExchanges() {
   return numExchanges;
}  //  getNumExchanges

public void incNumExchanges(int inc) {
   numExchanges += inc;
}  //  incNumExchanges

public int getAmtGained() {
   return amtGained;
}  //  getAmtGained

public void incAmtGained(int inc) {
   amtGained += inc;
}  //  incAmtGained

public int getAmtLost() {
   return amtLost;
}  //  getAmtLost

public void incAmtLost(int inc) {
   amtLost += inc;
}  //  incAmtLost

public int getAmtDissipated() {
   return amtDissipated;
}  //  getAmtDissipated

public void incAmtDissipated(int inc) {
   amtDissipated += inc;
}  //  incAmtDissipated

public int getAmtAbsorbed() {
   return amtAbsorbed;
}  //  getAmtAbsorbed

public void incAmtAbsorbed(int inc) {
   amtAbsorbed += inc;
}  //  incAmtAbsorbed

public static String[] getChromosomes(Agent[] agents) {
   String[] str = new String[agents.length];

   for (int i = 0; i < agents.length; i++) {
      str[i] = agents[i].chromosome.toString();
   }  //  for i
```

```
    return str;

  }  //  getChromosomes

  //  A "biological" species is determined by the offense tag and the
  //    mating condition.
  public static String[] getSpecies(Agent[] agents) {
    String[] str = new String[agents.length];

    for (int i = 0; i < agents.length; i++) {
      str[i] = agents[i].chromosome.getOffenseTag().toString()
          + agents[i].chromosome.getMatingCond().toString();
    }  //  for i

    return str;

  }  //  getSpecies

  public int getId() {
    return id;
  }  //  getId

  public Chromosome getChromosome() {
    return chromosome;
  }  //  getChromosome

  public void setChromosome(Chromosome c) {
    chromosome = c;
  }  //  setChromosome

  public Tag getOffenseTag() {
    return chromosome.getOffenseTag();
  }  //  getOffenseTag

  public Tag getDefenseTag() {
    return chromosome.getDefenseTag();
  }  //  getDefenseTag

  public Tag getControlField() {
    return chromosome.getControlField();
  }  //  getControlField

  public Tag getExchangeCond() {
    return chromosome.getExchangeCond();
  }  //  getExchangeCond

  public Tag getMatingCond() {
    return chromosome.getMatingCond();
  }  //  getMatingCond

  public Tag getTransformCond() {
    return chromosome.getTransformCond();
  }  //  getTransformCond

  public int getSiteOffScore() {
    return siteOffScore;
  }  //  getSiteOffScore

  public Reservoir getReservoir() {
    return reservoir;
} // getReservoir

public int getReservoirLen() {
    return reservoir.length();
} // getReservoirLen

public void setReservoir(Reservoir r) {
    reservoir = r;
} // setReservoir

public void setSiteOffScore(int s) {
    siteOffScore = s;
} // setSiteOffScore

public int getSiteDefScore() {
    return siteDefScore;
} // getSiteDefScore

public void setSiteDefScore(int s) {
    siteDefScore = s;
} // setSiteDefScore

public int getEncounterOffScore() {
    return encounterOffScore;
} // getEncounterOffScore

public void setEncounterOffScore(int s) {
    encounterOffScore = s;
} // setEncounterOffScore

public int getEncounterDefScore() {
    return encounterDefScore;
} // getEncounterDefScore

public void setEncounterDefScore(int s) {
    encounterDefScore = s;
} // setEncounterDefScore

public int getExchangeCondScore() {
    return exchangeCondScore;
} // getExchangeCondScore

public void setExchangeCondScore(int s) {
    exchangeCondScore = s;
} // setExchangeCondScore

public int getMatingCondScore() {
    return matingCondScore;
} // getMatingCondScore

public void setMatingCondScore(int s) {
    matingCondScore = s;
} // setMatingCondScore

public void logC(Log log, int time) {
    if (log == null) return;
    log.logIt(time + " ");
    chromosome.log(log);
} // logC
```

```
  public void logVitals(Log log, int time) {
    if (log == null) return;
    log.logIt(Pad.pad(time, 3)
          + " " + Pad.pad(id, 4)
          + " " + Pad.pad(parent1, 4)
          + " " + Pad.pad(parent2, 4)
          + " " + chromosome.toString()
          + " " + Pad.pad(DoB, 3)
          + " " + Pad.pad(DoD, 3)
          + " " + Pad.pad(getAge(time), 4)
          + " m " + Pad.pad(numMatingEncounters, 3)
          + " " + Pad.pad(lastMating, 4)
          + " " + Pad.pad(numSiblings, 3)
          + " t " + Pad.pad(numTransforms, 3)
          + " e " + Pad.pad(numExchanges, 3)
          + " " + Pad.pad(amtGained, 3)
          + " " + Pad.pad(amtLost, 3)
          + " f " + Pad.pad(amtAbsorbed, 4)
          + " " + Pad.pad(amtDissipated, 4)
          + " " + Pad.pad(siteOffScore, 3)
          + " " + Pad.pad(siteDefScore, 3)
          + " " + Pad.pad(reservoir.length(), 3)
          + " " + name
          + "\r\n"
          );
  }  //  logVitals

  public void log(Log log, int time) {
    if (log == null) return;
    logVitals(log, time);
    chromosome.logC(log);
    reservoir.log(log);
    log.logIt("\r\n");
  }  //  log

  public void print() {
    System.out.println("Agent name " + name);
    chromosome.print();
    reservoir.print();
  }  //  print

}  //  class Agent


class Reservoir {
  private StringBuffer   resources;
  private static int     numCalls     = 0;
  private static int     numTrans     = 0;
  private static int     numSameCost  = 0;
  private static int     numSameFrom  = 0;
  private static int     numFoundFrom = 0;
  private static int     numFoundCost = 0;
  private static int[][] transforms   = new int[Resource.RESOURCES.length][Resource.RESOURCES.length];

  //  Create a new random Reservoir of length l.
  Reservoir(int l) {
    resources = new StringBuffer(l);
    for (int i = 0; i < l; i++) {
      resources.append(Resource.getResource());
    }  //  for i
}  //  Reservoir

//  Create a new Reservoir containing str.
Reservoir(String str) {
    resources = new StringBuffer(str);
}  //  Reservoir

//  Transform one resource into another and delete the cost resource.
public boolean transform (char from, char to, char cost) {
    boolean fromFound    = false;
    int     fromLocation = 0;
    boolean transformed  = false;

    numCalls++;
    if (to == cost) {
        numSameCost++;
    }  //  if

    if (from == to) {
        numSameFrom++;
        return false;
    }  //  if

    //  see if from is in the reservoir.
    for (int i = 0; i < resources.length(); i++) {
        if (resources.charAt(i) == from) {
            numFoundFrom++;
            fromFound = true;
            fromLocation = i;
            break;
        }
    }  //  for i

    if (fromFound) {
        //  see if the cost is in the reservoir.
        for (int i = 0; i < resources.length(); i++) {
            if ((resources.charAt(i) == cost) && (i != fromLocation)) {
                numFoundCost++;
                int f = Resource.getInt(from);
                int t = Resource.getInt(to);
                transforms[f][t]++;
                numTrans++;
                resources.setCharAt(fromLocation, to);
                resources.deleteCharAt(i);
                transformed = true;
                break;
            }
        }  //  for i
    }  //  if

    return transformed;

}  //  transform

//  Acquire n resources from Reservoir r1 and append
//    them to this reservoir.
public int acquireFrom(Reservoir r1, int n) {
    String str = new String(r1.resources);
    int    len = str.length();
    int    amt = 0;

    if (n >= len) {
       resources.append(str);                  //  get it all
       amt = len;
       r1.resources = new StringBuffer();      //  empty r1's
    } else {
       resources.append(str.substring(0, n));  // get the first n reaources
       amt = n;
       r1.resources = new StringBuffer(str.substring(n,len));
    }  //  if

    return amt;

 }  //  acquireFrom

 public static void resetAggregates() {
    numCalls     = 0;
    numTrans     = 0;
    numSameCost  = 0;
    numSameFrom  = 0;
    numFoundFrom = 0;
    numFoundCost = 0;
    clearTransforms();
 }  //  resetAggregates

 public static void clearTransforms() {
    for (int i = 0; i < transforms.length; i++) {
       for (int j = 0; j < transforms[i].length; j++) {
          transforms[i][j] = 0;
       }  //  for j
    }  //  for i
 }  //  clearTransforms

 public int length() {
    return resources.length();
 }  //  length

 public static int getNumTrans() {
    return numTrans;
 }  //  getNumTrans

 public static void setNumTrans(int n) {
    numTrans = n;
 }  //  setNumTrans

 public static int[][] getTransforms() {
    return transforms;
 }  //  getTransforms

 public String toString() {
    return resources.toString();
 }  //  toString

 public void log(Log log) {
    log.logIt(toString() + " ");
 }  //  log

 public static void logTransforms(Log log) {
    int   count  = 0;
    int[] counts = new int[transforms.length];

    log.logIt("Resource transformations.\r\n");
    log.logIt("Transform calls: " + numCalls + "\r\n");
    log.logIt("To resource same as cost resource: " + numSameCost + "\r\n");
    log.logIt("To resource same as from resource: " + numSameFrom + "\r\n");
    log.logIt("Found the from resource: " + numFoundFrom + "\r\n");
    log.logIt("Found the cost resource: " + numFoundCost + "\r\n");
    for (int i = 0; i < transforms.length; i++) {
        log.logIt("From " + Resource.getResource(i));
        for (int j = 0; j < transforms[i].length; j++) {
            log.logIt(Pad.pad(transforms[i][j], 9)
                      + "  " + Resource.getResource(j)
                      + "  "
                     );
            count     += transforms[i][j];
            counts[j] += transforms[i][j];
        }  //  for j
        log.logIt("\r\n");
    }  //  for i
    log.logIt("Total to       ");
    for (int i = 0; i < counts.length; i++) {
        log.logIt(Pad.pad(counts[i], 9)
                  + "  " + Resource.getResource(i)
                  + "  "
                 );
    }  //  for i
    log.logIt("\r\n");
    log.logIt("Total transforms: " + count + " \r\n");

}  //  logTransforms

public static void logAggregates(Log log) {
    int   count  = 0;
    int[] counts = new int[transforms.length];

    if (log == null) return;

    //  Aggregate the resource totals.
    for (int i = 0; i < transforms.length; i++) {
        for (int j = 0; j < transforms[i].length; j++) {
            count     += transforms[i][j];
            counts[j] += transforms[i][j];
        }  //  for j
    }  //  for i

    log.logIt("  "   + numCalls
            + "  " + count
            + "  " + numSameCost
            + "  " + numSameFrom
            + "  " + numFoundFrom
            + "  " + numFoundCost
           );
    for (int i = 0; i < counts.length; i++) {
        log.logIt("  " + counts[i]
                 );
    }  //  for i

}  //  logAggregates
```

```
    public static void printTransforms() {
        int   count  = 0;
        int[] counts = new int[transforms.length];

        System.out.println("Resource transformations.");
        System.out.println("Transfrom calls: " + numCalls);
        System.out.println("To resource same as cost resource " + numSameCost);
        System.out.println("To resource same as from resource " + numSameFrom);
        System.out.println("Found the from resource " + numFoundFrom);
        System.out.println("Found the cost resource " + numFoundCost);
        for (int i = 0; i < transforms.length; i++) {
            System.out.print("From " + Resource.getResource(i));
            for (int j = 0; j < transforms[i].length; j++) {
                System.out.print(Pad.pad(transforms[i][j], 9)
                                 + "  " + Resource.getResource(j)
                                 + "  "
                                );
                count     += transforms[i][j];
                counts[j] += transforms[i][j];
            }  //  for j
            System.out.println();
        }  //  for i
        System.out.println();
        System.out.print("To  ");
        for (int i = 0; i < counts.length; i++) {
            System.out.print(Pad.pad(counts[i], 9)
                             + "  " + Resource.getResource(i)
                             + "  "
                            );
        }  //  for i
        System.out.println();
        System.out.println("Total transforms: " + count);

    }  //  printTransforms

    public void print() {
        System.out.println("Reservoir: " + toString());
    }  //  print

}  //  class Reservoir


/**
 * Title:        Chromosome
 * Description:  An Echo chromosome.
 * Copyright:    Copyright (c) 2001
 * Company:
 * @author David L. Harris
 * @version 1.0
 */

class Chromosome {
    private StringBuffer chromosome;

    private static final int oTagLen   = World.oTagLen;
    private static final int dTagLen   = World.dTagLen;
    private static final int cFldLen   = World.cFldLen;
    private static final int eCondLen  = World.eTagLen;
    private static final int mCondLen  = World.mTagLen;
    private static final int tCondLen  = World.tTagLen;
```

```
//  Define the choromosome string format.
private static final int oTagStrt  = 0;
private static final int oTagEnd   = oTagStrt + oTagLen;
private static final int dTagStrt  = oTagEnd;
private static final int dTagEnd   = dTagStrt + dTagLen;
private static final int cFldStrt  = dTagEnd;
private static final int cFldEnd   = cFldStrt + cFldLen;
private static final int eCondStrt = cFldEnd;
private static final int eCondEnd  = eCondStrt + eCondLen;
private static final int mCondStrt = eCondEnd;
private static final int mCondEnd  = mCondStrt + mCondLen;
private static final int tCondStrt = mCondEnd;
private static final int tCondEnd  = tCondStrt + tCondLen;
private static final int chromLen  = oTagLen + dTagLen + eCondLen + mCondLen + tCondLen + cFldLen;

public  static final int OFFENSE_TAG    = 0;
public  static final int DEFENSE_TAG    = 1;
public  static final int CONTROL_FLD    = 2;
public  static final int EXCHANGE_COND  = 3;
public  static final int MATING_COND    = 4;
public  static final int TRANSFORM_COND = 5;
public  static final int NUMBER_FIELDS  = 6;
public  static final String[] FIELD_NAMES = {"Offense Tag    ",
                                             "Defense Tag    ",
                                             "Control Field  ",
                                             "Exchange Cond  ",
                                             "Mating Cond    ",
                                             "Transform Cond "
                                            };

//  Create a new random chromosome.
Chromosome() {
   chromosome = new StringBuffer(chromLen);

   Tag tag = new Tag(oTagLen);
   chromosome = chromosome.append(tag.toString());
   tag = new Tag(dTagLen);
   chromosome = chromosome.append(tag.toString());
   tag = new Tag(cFldLen);
   chromosome = chromosome.append(tag.toString());
   //chromosome = chromosome.append("cccccc");
   tag = new Tag(eCondLen);
   chromosome = chromosome.append(tag.toString());
   tag = new Tag(mCondLen);
   chromosome = chromosome.append(tag.toString());
   tag = new Tag(tCondLen);
   chromosome = chromosome.append(tag.toString());

}  //  Chromosome

//  Create a new chromosome containing str.
Chromosome(String str) {
   chromosome = new StringBuffer(str);
}  //  Chromosome

//  Mutate a single resource.
public void mutate() {
   int  locus = World.random.nextInt(chromosome.length());
   char c     = Resource.mutate(chromosome.charAt(locus));
```

```
    chromosome.setCharAt(locus, c);
  }  //  mutate

  //  Single point crossover of two chromosomes.
  public static void crossover(Chromosome c1, Chromosome c2) {
    String       str1  = new String(c1.chromosome);
    String       str2  = new String(c2.chromosome);
    int          l1    = str1.length();
    int          l2    = str2.length();
    int          locus = World.random.nextInt(l1);
    StringBuffer s1    = new StringBuffer();
    StringBuffer s2    = new StringBuffer();

    s1.append(str1.substring(0, locus));
    s2.append(str2.substring(0, locus));
    s1.append(str2.substring(locus, l2));
    s2.append(str1.substring(locus, l1));

    c1.chromosome = s1;
    c2.chromosome = s2;

  }  //  crossover

  public Tag getOffenseTag() {
    String cstr = new String(chromosome.toString());
    Tag    oTag = new Tag(cstr.substring(oTagStrt, oTagEnd));
    return oTag;
  }  //  getOffenseTag

  public Tag getDefenseTag() {
    String cstr = new String(chromosome.toString());
    Tag    dTag = new Tag(cstr.substring(dTagStrt, dTagEnd));
    return dTag;
  }  //  getDefenseTag

  public Tag getControlField() {
    String cstr = new String(chromosome.toString());
    Tag    cFld = new Tag(cstr.substring(cFldStrt, cFldEnd));
    return cFld;
  }  //  getControlField

  public Tag getExchangeCond() {
    String cstr  = new String(chromosome.toString());
    Tag    eCond = new Tag(cstr.substring(eCondStrt, eCondEnd));
    return eCond;
  }  //  getExchangeCond

  public Tag getMatingCond() {
    String cstr  = new String(chromosome.toString());
    Tag    mCond = new Tag(cstr.substring(mCondStrt, mCondEnd));
    return mCond;
  }  //  getMatingCond

  public Tag getTransformCond() {
    String cstr  = new String(chromosome.toString());
    Tag    tCond = new Tag(cstr.substring(tCondStrt, tCondEnd));
    return tCond;
  }  //  getTransformCond

  public String toString() {
        return chromosome.toString();
    }  //  toString

    public void logC(Log log) {
        if (log == null) return;
        log.logIt(getOffenseTag()
                + " " + getDefenseTag()
                + " " + getControlField()
                + " " + getExchangeCond()
                + " " + getMatingCond()
                + " " + getTransformCond()
                );
    }  //  logC

    public void log(Log log) {
        if (log == null) return;
        log.logIt(chromosome + "\r\n");
    }  //  log

    public void print() {
        System.out.println("Chromosome: " + toString());
        System.out.println("Offense tag "        + getOffenseTag().toString()
                    + "  Defense tag "    + getDefenseTag().toString()
                    + "  Control fld "    + getControlField().toString()
                    + "  Exchange cond "  + getExchangeCond().toString()
                    + "  Mating cond "    + getMatingCond().toString()
                    + "  Transform cond " + getTransformCond().toString()
                    );
    }  //  print

}  //  class Chromosome



class Tag {
    StringBuffer tag;
    int          value;

    //  Create a new tag from the StringBuffer b.
    Tag(StringBuffer b) {
        tag = b;
    }  //  Tag

    //  Create a new tag from the String s.
    Tag(String s) {
        tag = new StringBuffer(s);
    }  //  Tag

    //  Create a new random Tag with initial length l.
    Tag(int l) {
        tag = new StringBuffer(l);
        for (int i = 0; i < l; i++) {
            tag.append(Resource.getResource());
        }  //  for i
    }  //  Tag

    public char charAt(int i) {
        return tag.charAt(i);
    }  //  charAt
    //  Match this Tag with Tag t2 and return the score.
    public int matchTag(Tag t2) {
        int score = 0;

        int t1l    = tag.length();
        int t2l    = t2.length();
        int length = t1l;
        int diff   = t1l - t2l;

        if (t2l < t1l) length = t2l;
        for (int i = 0; i < length; i++) {
            score += Resource.matchTag(tag.charAt(i), t2.tag.charAt(i));
        }  //  for i
        score -= Math.abs(diff) * Resource.EXTRA;

        return score;

    }  //  matchTag

    //  Match this condition Tag with Tag t2 and return the score.
    public int matchCondition(Tag t2) {
        int score  = 0;

        int t1l    = tag.length();
        int t2l    = t2.length();
        int length = t1l;

        if (t2l < t1l) length = t2l;
        for (int i = 0; i < length; i++) {
            score += Resource.matchCondition(tag.charAt(i), t2.tag.charAt(i));
        }  //  for i

        return score;

    }  //  matchCondition

    public int length() {
        return tag.length();
    }  //  length

    public String toString() {
        return tag.toString();
    }  //  toString

    public int toValue() {
        return 0;
    }  //  toValue

    public void print() {
        System.out.println("Tag: " + toString());
    }  //  print

}  //  class Tag


class Resource {
    static final char[]  RESOURCES = {'a', 'b', 'c', 'd'};
    static final int     FIRST     = Character.getNumericValue(RESOURCES[0]);
    static final int[][] TAGSCORES  = {{ 2,  0, -1,  0},    //  matches tag scores
                                       { 0,  2,  0, -1},
                              {-1,  0,  2,  0},
                              { 0, -1,  0,  2}
                             };
static final int[][] CONDSCORES = {{ 2,  1,  0,  1},    //  matches condition scores
                              { 1,  2,  1,  0},
                              { 0,  1,  2,  1},
                              { 2,  2,  2,  2}
                             };
static final int     EXTRA      = -1;
static final int[]   counts     = {0, 0, 0, 0};
static final int[]   mutations  = {0, 0, 0, 0};

//  Return the i'th resource.
public static char getResource(int i) {
   return (RESOURCES[i]);
}  //  getResource

//  Return a random resource.
public static char getResource() {
   int i = World.random.nextInt(RESOURCES.length);
   counts[i]++;
   return (getResource(i));
}  //  getResource

//  Return a resource index integer value.
public static int getInt(char r) {
   return (Character.getNumericValue(r) - FIRST);
}  //  getInt

//  Mutate resource r.
public static char mutate(char r) {
   int i  = World.random.nextInt(RESOURCES.length - 1);
   int m = (getInt(r) + 1 + i) % RESOURCES.length;
   mutations[m]++;
   return getResource(m);
}  //  mutate

//  Tag score resource r1 against resource r2.
public static int matchTag(char r1, char r2) {
   int i1 = Character.getNumericValue(r1) - FIRST;
   int i2 = Character.getNumericValue(r2) - FIRST;

   return (TAGSCORES[i1][i2]);

}  //  matchTag

//  Condition score resource r1 against resource r2.
public static int matchCondition(char r1, char r2) {
   int i1 = Character.getNumericValue(r1) - FIRST;
   int i2 = Character.getNumericValue(r2) - FIRST;

   return (CONDSCORES[i1][i2]);

}  //  matchCondition

public static void log(Log log) {
   log.logIt("Resource counts:  ");
   for(int i = 0; i < counts.length; i++) {
      log.logIt(" " + counts[i]);
   }  //  for i
```

```
        log.logIt("\r\n");
        log.logIt("Mutation counts:  ");
        for(int i = 0; i < mutations.length; i++) {
            log.logIt(" " + mutations[i]);
        }  //  for i
        log.logIt("\r\n");
    }  //  log

    public static void print() {
        System.out.print("Resource counts:  ");
        for(int i = 0; i < counts.length; i++) {
            System.out.print(" " + counts[i]);
        }  //  for i
        System.out.println(" ");
        System.out.print("Mutation counts:  ");
        for(int i = 0; i < mutations.length; i++) {
            System.out.print(" " + mutations[i]);
        }  //  for i
        System.out.println(" ");
    }  //  print

}  //  class Resource
```

Distribution:

Dr. Raymond F. Bernstein
New Mexico State University
Physical Sciences Laboratoy
Las Cruses, NM 88003-0001

Dr. Donald Birx
New Mexico State University
Physical Sciences Laboratoy
Las Cruses, NM 88003-0001

Dr. Robert Burch
Department of Philosophy
Texas A&M University
College Station, TX 77843-4237

Dr. Michael Coombs
New Mexico State University
Physical Sciences Laboratoy
Las Cruses, NM 88003-0001

Army Research Lab
AMSRL-SL-EA
Attn: Richard Flores
White Sands Missile Range, NM 88002

Dr. Stephanie Forrest
University of New Mexico
Computer Science Department
Farris Engineering Building
Albuquerque, NM 87131-1386

Dr. Mai Gehrke
New Mexico State University
Department of Mathematical Sciences
P.O. Box 30001
Department 3MB
Las Cruses, NM 88003-8001

Army Research Lab
AMSRL-SL-EA
Attn: Dan Landin
White Sands Missile Range, NM 88002

Army Research Lab
AMSRL-SL-EA
Attn: Dr. Kent Morrison
White Sands Missile Range, NM 88002

Dr. Edward Nozawa
Lockheed Martin Aeronautics Company
86 South Cobb Drive
Marietta, GA 30063-0249

Army Research Lab
AMSRL-SL-EA
Attn: Tom Reader
White Sands Missile Range, NM 88002

10	MS0188	D. L. Chavez, 1030
1	MS1140	S. G. Varnado, 6500
1	MS0451	R. E. Trellue, 6501
1	MS1140	L. J. Ellis, 6502
1	MS0451	J. E. Nelson, 6515
1	MS0449	R. L. Hutchinson, 6516
1	MS0449	P. L. Campbell, 6516
15	MS0449	D. L. Harris, 6516
1	MS0455	R. S. Tamashiro, 6517
1	MS0455	M. E. Senglaub, 6517
1	MS1138	B. N. Malm, 6531
1	MS0318	G. S. Davidson, 9212
1	MS0318	M. B. E. Boslough, 9212
1	MS1109	R. J. Pryor, 9212
10	MS0806	L. B. Dean, 9336
1	MS1221	D. M. Rondeau, 15003
1	MS1006	P. Garcia, 15202
1	MS1004	R. W. Harrigan, 15221
1	MS1125	A. B. Maish, 15252
1	MS1125	A. K. Miller, 15252
1	MS1221	R. D. Skocypec, 15310
1	MS1188	J. S. Wagner, 15311
1	MS0859	T. K. Stalker, 15351
1	MS0839	G. Yonas, 16000
1	MS0839	E. M. Raybourn, 16000
1	MS9018	Central Technical Files, 8945-1
2	MS0899	Technical Library, 9616
1	MS0612	Review & Approval Desk, 9612, for DOE/OSTI
1	MS0161	Patent & Licensing Office, 11500