AGES: Agentsheets Genetic Evolutionary Simulations

by

BRADEN SCOTT CRAIG

B.A. University of North Carolina-Chapel Hill, 1992

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirement for the degree of

Master of Science

Department of Computer Science

1997

This thesis entitled:

AGES: Agentsheets Genetic Evolutionary Simulations

written by Braden Scott Craig

has been approved for the Department of Computer Science

Prof. Clayton Lewis

Prof. Alexander Repenning

Date_____

The final copy of this thesis has been examined by the signators, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Craig, Braden S. (M.S. Computer Science)

AGES: Agentsheets Genetic Evolutionary Simulations

Thesis directed by Professor Clayton Lewis

Agentsheets Genetic Evolutionary Simulations (AGES) is a system developed with Agentsheets with VisualAgenTalk (AS-VAT) to promote the study of complex adaptive systems (cas) subject to evolution. AGES can be used to model a wide array of cas, from ecological systems such as a rain forest to economic systems such as the New York Stock Exchange. Being embedded in AS-VAT, AGES enables non-expert end-users of all ages to explore interesting and important concepts of cas through programming.

CONTENTS

TABLES

Table

FIGURES

Introduction

Agentsheets Genetic Evolutionary Simulations (AGES) is a system that has been developed within Agentsheets with VisualAgenTalk (AS-VAT) to enable a wide audience of non-expert end-users to study complex adaptive systems (cas) through programming. Murray Gell-Mann's broad definition of complex adaptive systems shows how many phenomena can be described in terms of cas:

> I favor a comprehensive point of view according to which the operation of CAS encompasses such diverse processes as the prebiotic chemical reactions that produced life on Earth, biological evolution itself, the functioning of individual organisms and ecological communities, the operation of biological subsystems such as mammalian immune systems or human brains, aspects of human cultural evolution, and adaptive functioning of computer hardware and software. [13]

With such a notion of cas in mind, it is not surprising that one would want to study them; cas are pervasive and can be found in many different domains of inquiry. As well, complex adaptive systems tend to be characterized in terms of the *interactions* between their parts as opposed to being defined in terms of properties that components exhibit when studied in isolation [18, 27, 53]. The end-users that AGES seeks to introduce to the study of cas might be middle-school students playing with a fish tank simulation, or they might be middle-aged professionals with interests in Artificial Life. In any case, such users are not experts in computer science, so they often lack the skills to program in traditional high-level programming languages such as C/C++ or Java.

This is fine if users only want to explore cas by playing with simulations that are pre-made for them. (Or maybe they just want to read about cas.) They might be able to modify parts of these programs within certain limits, but it is unlikely they will be able to change the behavior of their programs in ways not predefined by the original programmers. For example, in SimCity(tm) users are able to place houses, factories, parks, etc. in different configurations in order to observe the interactions between them. Cities will prosper, become more or less industrialized, etc., according to decisions a user makes. But no such user can change the way a house or a factory behaves. No users will be able to create a new kind of "houseboat" for instance. Such changes would be beyond the realm of allowable changes that were predefined by the original programmers of SimCity(tm). To make these changes users would need to construct new entities through programming.

The above considerations lead one to make comparisons between Instructionist learning on the one hand and Constructionist learning on the other [38, 46]. Instructionist techniques for learning about complex adaptive systems might involve reading books about them. To a certain extent, instructionist approaches to learning focus on a learner as a passive recipient of ready-made knowledge. Knowledge exists as predefined and communicable. A student's job is then to absorb such predefined knowledge. Contrastingly, constructionist approaches to learning focus on a learner's ubiquitous *nnterpretations* of any knowledge he/she learns. In this sense, a person's social, economic, and cultural histories affect

any knowledge a person learns. Each student must construct his/her own knowledge because each must interpret any incoming events in order to understand them and infuse them with meaning. Learning is always an active event of constructing knowledge as opposed to a passive event of receiving it.

With the above considerations in mind, reading might be seen as a fairly instructionist approach to learning when compared with computer programming. [1] A reader does not literally rewrite passages of a book in order to learn from it; he usually just reads what an author has already written in order to learn about a book's message(s). In this sense, a book *instructs* readers about knowledge it contains. Within limits, readers do not construct knowledge they obtain from a book; they *receive* such knowledge.

Between reading and computer programming lie games like SimCity(tm). In these games a learner *molds* his knowledge in order to learn. As described above, a limited modifying/molding capability is afforded players of such strategy games. Players learn by assembling predefined pieces into more complex aggregate constructs. The interactions between different assemblages of premade components will indeed give rise to widely varying system-wide properties. Still, a user can only assemble "worlds" with pre-made pieces; no new pieces can be generated/programmed by users.

Finally, at the Constructionist end of the spectrum one finds activities like "programming in traditional high-level programming languages such as C/C++ and Java." By definition, programming aims at constructing computatonal entities. In using languages like C/C++ and Java to explore complex adaptive systems an expert programmer can create arbitrarily complex interactive simulations. Within such sims a user might assemble predefined pieces in addition to creating new ones. However, most users are not "expert users" and are either unwilling or unable to create such complex constructs in traditional high-level languages. Though these languages are highly expressive, they are not very usable. There is a very steep learning curve for becoming a C++ programmer capable of creating simulations, or even agents within them, that are interesting. Again, this steep learning curve keeps most users from becoming programmers in traditional high-level languages. Such a deterrent, in turn, hinders any constructionist approach to studying cas through programming in such languages.

But why would users want to program in order to study cas? Is the study of complex adaptive systems a domain where constructionist learning techniques can be profitably applied? Given that interesting properties of cas are, by definition, dependent upon the interactions between parts that make

---

[1] This is description is not uncontroversial. Constructionists would argue that in reading (and in linguistic discourse in general) interpretation gives meaning to a text. In turn, such interpreations are inevitably constructed by readers as opposed to being received by them as ready-made knowledge chunks.

up the system, studying cas seems especially well-suited to constructionist approaches to learning utilizing programmable simulations. Computer simulations can serve to illustrate and make intuitive the interactions that define cas. Resnick and other researchers at MIT explicitly promote such approaches to learning [45, 46, 47, 51, 12]. Their ideas of Constructionism center around enabling learners to construct personally meaningful artifacts that exemplify ideas about domains they are interested in. Computer simulations are flexible enough in their scope to enhance learning about many disparate domains in novel and engaging ways [51] . More specifically, *agent-based* simulations of different domains pump intuitions in ways that raw numbers or even graphs just can't.

Relatedly, Csikszentimihalyi has written extensively about many aspects of human creativity [10]. He argues that the role of play is central to living and thinking creatively. It seems quite clear that students'/children's creativity can be enhanced if they are given more freedom to play with ideas they encounter in classroom settings. Computer simulation tools like AGES can aid in creating such freedom without totally abandoning structure in the classroom. Hopefully, through the use of tools like AGES and AS-VAT kids will be motivated to construct personally meaningful worlds for themselves that allow them to actively engage with ideas that were previously constrained to pencil and paper presentations.

At MIT Mitch Resnick has undertaken studies with high school students to explore ways in which simulations can aid kids in understanding complex self-organizing processes. (In the following simulation, no explicit adaptive mechanisms are programmed into the agents of the system. Nevertheless, complex self-organizing properties of the system emerge.) For example, in his *Turtles, Termites, and Traffic Jams* Resnick speaks of a termite simulation that he built with a high school student using StarLogo. The object of this simulation was to get logo termites to gather wood chips into piles. A couple of approaches emerged that were more or less complicated. The first approach was more simple. All the termites followed the following simple strategy:

- If you're not carrying anything and you bump into a wood chip, pick it up.
- If you are carrying a wood chip and you bump into another wood chip, put down the wood chip you are carrying. [46]

Both Mitch and Callie were skeptical about this strategy. (It had been suggested by someone else.) But this strategy performed fairly well. The number of piles decreased monotonically as the simulation ran. It worked because, as termites picked up wood chips and dispersed them into different piles, certain piles "became extinct". The former piles would find themselves holding no wood chips. Once this happened, no more wood chips could be dropped at that site. Over time, this led to a steadily decreasing number of wood chip piles.

Still, the piles seemed to decrease at a rather slow rate. To alleviate this problem, Mitch and Callie added a new rule to the strategy that "protected" larger piles. Using this augmented strategy, termites

could only take wood chips away from piles with nine or fewer chips. This indeed led to quicker convergence of wood chips into smaller piles, but ultimately, this strategy reached an unbridgeable limit. After all the wood chips were located in piles of ten or more chips, no more chips could be moved. Although, this strategy initially led to more rapid convergence, it could never lead to a state where all the chips were located in one pile. Both Mitch and Callie preferred the feel of the sim where termites acted according to the original simpler strategy. This strategy eventually leads to one pile containing all the wood chips even though it is less efficient as wood chips are first gathered. Perhaps more interestingly, as termites follow this rule piles change size and move around, and as Callie put it, "It [the system] feels more alive." (46: p.75-81).

Such an example shows how many simple and homogeneous agents interacting can bring about interesting and global behavior. Both Mitch and Callie were skeptical about whether their simple strategy would work. But by incorporating such local rules into a StarLogo simulation they were able to watch the effects of their strategy and to gain new insight into it. Only after watching the sim did it dawn on Resnick that being in a "trapped state", i.e. that of containing no wood chips, would have a grand effect on the formation of termite piles in general [46: p.79].

These types of interactions with computer simulations highlight the intuitive power that sims hold. It is difficult, and in some cases impossible, to think of what will happen as many agents act in parallel. But the world is a highly parallel place.[26, 46] It is full of "agents" acting simultaneously. Before the advent of computers, it was far too tedious to carry out by hand the huge numbers of calculations that describe local interactions leading to complex global behavior in situations modeled by today's agent-based simulations. With the advent of high speed computers we can today tackle problems, via simulation, that were deemed unapproachable a mere fifty years ago.

When Mitch and Callie programmed their termites in StarLogo, they didn't have to watch graphs to notice that "the number of piles decreased monotonically." They saw the number of piles steadily decrease as they watched their termites pick them up and move them from pile to pile. Such metaphor-builders as these simulations allow people to use their everyday-tacit knowledge in dealing with computers. Turkle explores such "soft" approaches to computers in her book "Life on the Screen." As Turkle points out, it is ironic that a machine that has been so maligned by many on the grounds that it is just a "number cruncher" has been such an effective tool for developing more *informal* approaches to math and science. [51]

I submit that computers can also be thought of as language-crunchers. Computers have been responsible for much insight into formal and informal languages since their inception and are indebted to such fields for their very existence. Today research abounds in the field of Computational Linguistics, Natural Language Processing, Formal Languages, etc. Iconic programming languages, such as VAT, might be viewed as informal languages made possible, and just as importantly, made

accessible, by new advances in computing technology. They push the boundaries of how we understand language. By focusing on behaviors of interacting agents, they provide a kind of animated theatre-canvas upon which interested end-users can create fantastic tales to be played out in full splendor [39, 42].

The problem this thesis addresses might then be stated as follows: Using programming, how can one best promote constructionist learning in studying cas? A computational tool for studying cas should allow users to program the behavior of agents interacting to form the cas to be simulated. Such flexibility allows users to create varying simulations to study different cas with only one tool. Still, in providing such programmability any computational tool/language will be faced with a trade-off between expressiveness and useability.

Traditional high-level languages like C/C++ or Java are very expressive, but they are not easy to use. Again, learning curves for such languages are steep enough to prevent the average user from programming in them. In contrast, languages/tools like AGES are less expressive but more easily used. These languages are often tailored to supporting specific domains [41, 43]. The domain of complex adaptive systems subject to evolution is the domain of AGES. AGES allows users to program the behavior of interacting cas agents through tactile programming in VisualAgenTalk [40]. The programming approach of VAT allows users to quickly program the behavior of computational agents without the need to focus on complicated syntax. As opposed to traditional high-level languages, the learning curve for VAT is fairly flat. Users can quickly learn and use VAT to create and program novel agents. Preliminary studies indicate that AGES' commands are also easily understood and used in programming VAT agents.

In addition to providing programmable agents, a domain oriented design environment for studying cas should also allow users to easily change the look of agents. This will allow users to create computational agents that resemble real-world agents they represent. Sharks that eat fish should look like sharks, and the fish they eat should look like fish. Grey and yellow blobs representing sharks and fish, respectively, aren't as convincing as life-like iconic agents. The ability to create iconic agents can be important in motivating and encouraging end-users to program in AGES.

AGES exists as a number of commands in Agentsheets that are aimed at supporting the programming of simulations for exploring complex adaptive systems. In order to understand AGES one must first understand a bit about Agentsheets with VisualAgenTalk.

AS-VAT: Agentsheets with VisualAgenTalk

AS-VAT is an visual programming language/environment that enables end-users to create Sim-City(tm) like simulations [40, 41, 43, 44]. Agents in AS-VAT can be created by end-users using built-in drawing palettes or by capturing images from the screen. Agents are programmed by dragging and

dropping conditions and actions in an agent's rule-editor. An agent's rule-editor contains lists of if/then rules that define an agent's behavior. More complicated programming constructs can also be achieved by embedding method/function calls within an agent's if/then rules. This message-passing capability allows a VAT developer to program arbitrarily complex computational structures to define the behavior of an agent.

To create a simulation a VAT developer places agents in a 2-D grid called a worksheet. Worksheets and AS-VAT in general are based on concepts found in spreadsheet programming [44, 35]. Cells in a worksheet can contain multiple agents. Agents interact in a worksheet by checking conditions, based on information about themselves or on information obtained from agents in other cells, and executing those actions whose conditions are met. The actions of only one rule per method will be executed for any agent, but since agents can call other methods in their rules, an agent might execute an arbitrary number of actions located in an arbitrary number of rules during any given time step.

Complex Adaptive Systems, Emergence, and Genetic Algorithms

Complex Adaptive Systems (cas)

AS-VAT can readily be used to model different complex adaptive systems. AS-VAT agents most often act according to rules based on local information. Such agent-based local rules often bring about complex nonlinear interactions between agents behaving within a worksheet over time. From this play of interacting agents, global behavioral patterns for the system as a whole often arise. These patterns are typical of *complex systems*.

Complexity also arises when the primitive components of a system can change or evolve over time. *Evolution* occurs in *populations* of agents that change over generations. Evolution is genotypic. *Learning* occurs in *individual* agents that adapt to their external environments within a given lifetime. Learning is phenotypic. Systems that exhibit evolution and/or learning might be called *complex adaptive systems* [18]. Complex adaptive systems are often best described in terms of nonlinear relations between their simpler lower-level components. Within AS-VAT, agents become the lower-level components of systems modeled in different simulations where agents interact on a grid-based worksheet.

Emergence

Complex processes often arise as a result of interactions between individuals and their environment (which may include other individuals). Sometimes it is difficult, if not impossible, to describe these processes with the vocabulary used to describe individual behavior. The need to create new categories of description in order to describe a process might indicate that the process described is an emergent

one. Steels promotes the use of such a “new vocabulary” criterion in recognizing emergent phenomena in the following passage:

> From the viewpoint of an observer, we call a sequence of events a behavior if a certain regularity becomes apparent. This regularity is expressed in certain observational categories [of agents], for example, speed, distance to walls, changes in energy level. A behavior is emergent if new categories are needed to describe this underlying regularity that are not needed to describe the behaviors (i.e., the regularities) generated by the underlying behavior systems on their own.... *Thus, the regularities observed in the collective behavior of many molecules requires new categories like temperature and pressure over and above those needed to describe the motion of individual molecules.* [49, emphasis added]

Underlying behavior systems such as molecules can generate regularities, i.e. global system-wide properties like temperature and pressure, that are not properties of these same underlying behavior systems when they are observed in isolation. No molecule has a temperature in isolation. Temperature is a property of groups of molecules that collide and release energy in doing so. A molecule in a vacuum will have no temperature. Temperature is a new category introduced to describe *interactions* between components of a system that are obviously not present when a single component acts in isolation. This “new vocabulary” criterion, bound up with notions of global vs. local behavior, is important for recognizing emergence.

Steels also focuses on a distinction between controlled and uncontrolled variables in order to define emergent properties. *Controlled* variables are those that can be manipulated directly by a system. For example, if an autonomous agent can directly manipulate the speeds at which its different motors operate, it has direct control over the variable(s) “motor speed. *Uncontrolled* variables cannot be directly manipulated within a system. Clark offers a good example of uncontrolled variables when he speaks of Hofstadter’s operating system that begins to “thrash around” once about thirty-five users are on-line:

> In such a case, Hofstadter notes, it would be a mistake to go to the systems programmer and ask to have the “thrashing number” increased to, say, sixty. The reason is that the number 35 is not determined by an inner variable upon which the programmer can directly act. Instead: “That number 35 emerges dynamically from a host of strategic decisions made by the designers of the operating system and the computer’s hardware and so on. It is not available for twiddling” (Hofstadter 1985:642) [9].

These examples illustrate that issues of collectivity are often intertwined with issues concerning controlled vs. uncontrolled variables. The uncontrolled “thrashing number” of Hofstadter’s operating system is *uncontrolled* precisely because it results from the interactions of *numerous* parts of the computer. Clark makes the notions of collective activity and control explicit when he defines “emergent phenomena” in the following way:

> Emergent phenomena, as I shall use the term are thus any phenomena whose roots involve uncontrolled variables and are thus the products of collective activity rather than of dedicated components or control systems [9].

If a system has direct control over a variable, no collective activity is needed to change the value of this variable. The variable is "twiddle-able". When a system has no direct control over a variable, adjusting its value in predictable ways involves understanding the interactions between many components of the system. In this sense, issues of control and collectivity are often intimately linked, and both are important in recognizing and defining emergent properties of a complex adaptive system.

Another important aspect of complex adaptive systems revolves around the nonlinear relations that often hold between their parts. Chris Langton speaks of non-linear systems as those in which the behavior of the whole is *more* than the sum of its parts. In contrast, the behavior of linear systems is best characterized as the sum of its parts. Langton expands on these notions when he writes:

> Linear systems are those which obey the *principle of superposition.* We can break up complicated linear systems into simpler constituent parts, and analyse these parts *independently*. Once we have reached an understanding of the parts in isolation, we can achieve a full understanding of the whole system by *composing* our understandings of the isolated parts. This is the key feature of linear systems: by studying the parts in isolation, we can learn everything we need to know about the complete system.
> This is not possible for non-linear systems, which do *not* obey the principle of superposition. Even if we could break such systems up into simpler constituent parts, and even if we could reach a complete understanding of the parts in isolation, we would not be able to compose our understandings of the individual parts into an understanding of the whole system. The key feature of non-linear systems is that their primary behaviours of interest are properties of the i*nteractions between parts*, rather than being properties of the parts themselves, and these interaction-based properties necessarily disappear when the parts are studied independently. [26 emphasis in original]

Langton emphasizes the need to analyze the behavior of parts of a system *in situ*. Similar views are espoused by Wimsatt in his compelling article on "Forms of Aggregativity:"

> We have seen a variety of ways in which the properties of the whole may be "more" than the "sum" of its parts if by this we mean that properties of the whole depend not only upon the presence of the parts but also upon how they are arranged and how they interact. Probably the majority of interesting properties of complex systems are of this sort [53].

In Wimsatt's terms, if a property is aggregative, it is not emergent; it is just the sum of its parts. Only non-aggregative properties can be emergent. Interestingly, Wimsatt also draws a connection between emergence and reductionism:

> It is worth noting that in the progress of a reductionistic research program earlier simpler models which tend to treat parts as isolated and as characterized in terms of context-independent monadic properties are replaced by later more complex models in which relational properties of the parts enter and their behavior is thus rendered increasingly context-dependent.... If this picture is correct then we have, if anything, the opposite of the picture painted by the positivists. Rather than emergence disappearing with the progress of reductionistic theories, we have it growing in demonstrated scope and importance as we move from the earliest and simplest models to more complex and realistic ones. Thus some holists, at least, can draw only comfort from the continued successes of reductionistic approaches [53].

It is beyond the scope of this thesis to dealve deeply into matters relating holism to reductionism. In any case, "reductionist" or not, simulated studies of cas will be most helpful in exploring those

relations between parts of a system that are context-dependent and nonlinear. From different perspectives, groups of these relations can be thought of as uncontrolled variables of a system. Such relations in a system will generate emergent properties. Emergent properties can appear in systems with non-adaptive components merely through their interactions. Mitch and Callies termites are such agents. They always follow the same two simple rules, yet interesting pile-building properties of the system emerge. Similarly, molecules are non-adaptive agents that interact to create emergent properties such as temperature and pressure.

But there are also many complex systems whose agents are adaptive. As previously mentioned, these agents will often be subject to evolution and/or learning. In studying complex adaptive systems subject to evolution *genetic algorithms (GA's),* loosely based on principles borrowed from biological genetics, will without a doubt play a role. Such algorithms provide computational simulations the ability to explore emergent properties exhibited by processes similar to biological evolution **.**

Genetic Algorithms

Genetic Algorithms (GA's) were first introduced by John Holland in his book *Adaptation in Natural and Artificial Systems* [20]. Since then, many variations of the GA have been implemented [24, 25, 22]. GA's are used to simulate evolution in computational systems. The most simple versions of the GA contain selection, crossover, and mutation operators described by Melanie Mitchell below:

> ***Selection***. This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce.
>
> ***Crossover*** This operator randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring. For example, the strings 10000100 and 11111111 could be crossed over after the third locus in each to produce the tow offspring 10011111 and 11100100. The crossover operator roughly mimics biological recombination between two single-chromosome (haploid) organisms.
>
> ***Mutation*** This operator randomly flips some of the bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small (e.g. 0.001). [33]

Standard GA's often use some form of Replacement as well. The replacement operator keeps the population size constant. In contrast, some GA's do not maintain a constant population size. The actual form and role of genomes in different implementations of the GA varies widely. (This will become evident as different systems using GA's are described in later sections.)

The notion of fitness is also integral to GA's. Fitness is a measure of "how well" and organism is surviving in a given environment. Some GA's make use of exogenous fitness functions [20]. *Exogenous fitness* functions are external to an agent and are explicitly defined. An agent's fitness is calculated according to this function. In modeling real-world cas it is often impossible to determine the form of an exogenous fitness function. In these situations fitness is said to be endogenous [19, 26, 4].

Endogenous fitness is internal to an agent and is defined only implicitly. Holland characterizes systems with endogenous fitness when he writes:

> Discovering lever points and other critical cas phenomena is particularly difficult because contexts and activities are continually changing as the agents adapt. It is rare that we can even determine the utility of a given activity. The utility of the various activities of a given agent depends too much on the changing context provided by other agents. In mimicry, symbiosis, and other properties, the welfare of one agent depends critically on the presence of other, different agents. Fitness (reward, payoff) is implicitly defined in such cases. We cannot assign a fixed fitness to a chromosome because that fitness, however defined, is context dependent and changing. So it is for all cas. Our first order of business, then, is to provide a class of models in which the welfare of an adaptive agent stems from its interactions rather than from some predetermined fitness function. [19]

In the next section I present the commands that embody AGES. AGES commands provide end-users the ability to easily program agent behaviors in Agentsheets such that interacting agents in a worksheet can easily be interpreted as forming complex adaptive systems in which intersting system-wide properties emerge. Crossover in AGES is geared toward operating in systems where fitness in endogenous. Such systems are more like many real-world cas whose shapes and forms are constantly changing.

AGES (Agentsheets Genetic Evolutionary Simulations)

The commands making up AGES are shown in Appendix I. A subset of these is described below. When reading about these commands, the reader should keep the following “notes” in mind:

1) A “4hood” prefix in a command’s title indicates that the command operates over a von Neumann neighborhood (4 neighbors: N, S, E, W). By default, relevant commands with no “4hood” prefix in the title operate over Moore neighborhoods (8 neighbors: NW, N, NE, E, SE, S, SW, W).

2) A “Des” prefix in an eating command’s title indicates that it executes destructively. Destructive eating commands erase “eaten” agents during execution. A “Ndes” prefix in an eating command’s title indicates that it executes non-destructively. Non-destructive do not erase “eaten” agents during execution.

3) Formula fields can take numerical values, attributes, and formulas involving valid combinations of such data-types as arguments. Appendix 2 shows a list of valid formula-types.

Genetic Actions

Figure 1 shows a version of the Crossover_w_Cradle command used to introduce a form of genetic algorithm into AGES.

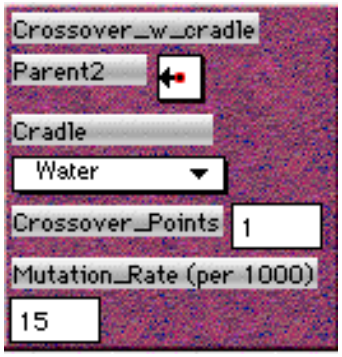


Figure 1: A version of Crossover with Mutation built-in.

The Crossover_w_Cradle command treats attribute values of AS-VAT agents as alleles on a haploid genome. It splices these genomes together as shown in Appendix 1b and as described by Mitchell above. It also allows users to define the direction in which to search for a parent. (See Appendix 1a for a "neighborhood" version of crossover.) Since this command looks to its neighbors to find a mate, it can be described as employing a spatially-constrained selection mechanism.

As well, being situated in AS-VAT, my crossover command asks users to choose a "Cradle" in which to place their children. If two agents of the specified cradle-type are found in the Moore neighborhood (8 neighbors) of a parent initiating crossover, two children are created and placed in their "cradles." If only one cradle-type agent is found in the vicinity one child is created. Else, no children are created. Finally, users choose the number of crossover points to use during the crossover operation and the likelihood (in 1000'ths) that point mutation will occur at each allele in the genome. (If a point mutation occurs 5 units are added or subtracted (random decision as to which) from the donated allele value at that locus.)

Figure 2 shows the set of "Remote_Child_1,2" commands to be used in conjunction with one of the crossover_w_cradle commands.

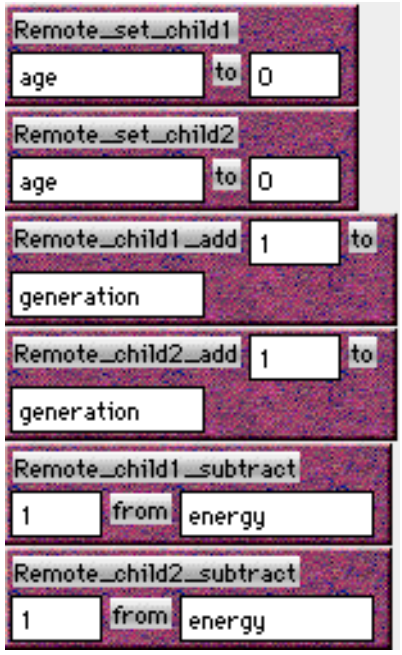


Figure 2: The "Remote_Child1,2" commands used to initialize variables of children created with the above crossover command.

These "Remote_Child1,2" commands allow an agent to set attributes of its children to user-specified attribute values and also to add and subtract from these attribute values. They are made to be used in conjunction with one of the crossover commands found in Appendix 1a. Any crossover command sets appropriate global variables that indicate the position, relative to the parent initiating crossover, of each child created. For this reason, "Remote_Child1,2" commands should be used directly after crossover is used, before any agents move or perform any other actions. If this convention is not adhered to, use of these commands will have mixed and unpredictable effects.

Figure 3 shows two versions of the Eat_Neighboring_Item command.

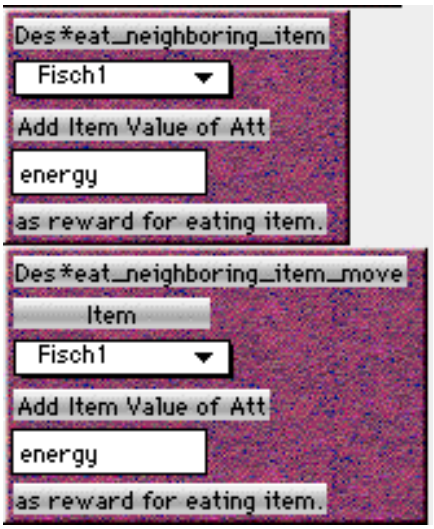


Figure 3: Eat_Neighboring_Item commands.

These commands tell an agent to "eat" a neighboring agent of a user-specified type <Fisch1> and to extract the value of a user-specified attribute of this prey/food <energy> as a reward for eating the item. Both commands shown execute over Moore neighborhoods.

The only difference between the Des*Eat_Neighboring_Item command and the Des*Eat_Neighboring_Item_Move command is fairly straightforward. Agents executing the latter command move to the cell previously inhabited by their just-eaten prey/food; agents executing the former command don't. Each of these commands eats the first prey/food agent that it finds.

Figure 4 shows the 4hood_Ndes*Eat_Max_Food_Move command.

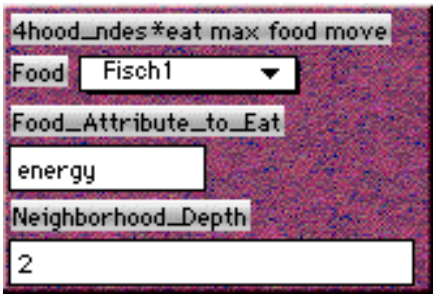


Figure 4: 4hood_Ndes*Eat_Max_Food_Move command.

This command tells an agent to eat a user-specified food-type <Fisch1> and to extract a user-specified attribute value <energy> from the food-type agent as a reward for eating the food. The food-type agent in the executing agent's von Neumann neighborhood of user-specified neighborhood_depth <2> with the highest food-attribute value is eaten. The depth of the neighborhood to be searched is specified by users in a formula-window.

Figure 5 shows the 4hood_Random_Move command.

Figure 5: 4hood_Random_ Move command.

This command tells an agent to move a user-specified number <vision> of cells away in a randomly chosen direction. The command executes over a von Neumann neighborhood. The step size of the command is entered into a formula field.

Figure 6 shows the Set_Rand_Between command.

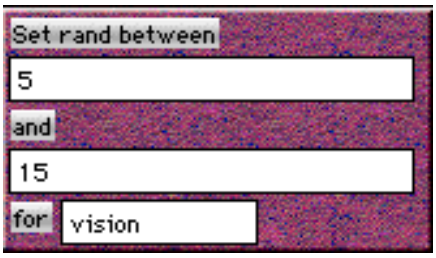


Figure 6: Set_Rand_Between command.

The Set_Rand_Between command allows users to initialize attributes <vision> to random values within a given range according to user-specified values <5, 15> entered into the command's formula fields. This command can be especially helpful in seeding a population with appropriate attribute values.

Figure 7 shows the Random_Placement command.

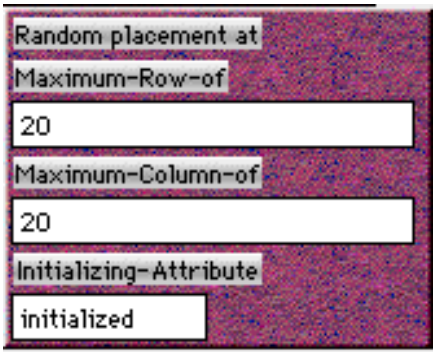


Figure 7: Random_Placement command.

The Random_Placement command is used to place copies of executing agents at random cells in the worksheet. Users specify a region of the worksheet where the agent is to be placed by entering maximum row <20> and maximum column <20> values for the newly replicated agent in the given formula windows. The agent is assumed to have a rule structure such as that shown in Appendix 3. With such a rule structure a user can specify an initializing attribute <initialized> in order to initialize

the agent during the time step immediately following its random placement. This command is most often used to maintain a stable population in sims where agents have finite lives.

Genetic Conditions

Figure 8 shows the 4hood*See_Food_in_Neighborhood command.

Figure 8: 4hood*See_Food_in_Neighborhood command.

The 4hood*See_Food_in_Neighborhood command looks for a user-specified food-type <Fisch1> agent in a title-specified neighborhood type of user-specified depth <vision>. This command is often used in conjunction with one of the eating action commands described above. It provides flexibility to users by allowing to them treat different agents as food and by allowing them to enter attribute-based formulas in order to specify neighborhood_depth. For example, this command could be used in implementing long-range vs. short range vision where short range vision searches over a Moore neighborhood of *normalvision* depth while long range vision searches over a von Neumann neighborhood of *longrangevision* depth.

Figure 9 shows the Do_Not_See command.

Figure 9: Do_Not_See command.

The Do_Not_See command is just the negation of the normal See command found in AS-VAT's default command palette. It is extremely useful in defining sims that will contain shared agents. For example, if one creates a sim with shooting space ships, one might want to determine an "immunity list" of agents not susceptible to laser fire. The Do_Not_See comand could be used to specify this list. Then, adding an agent to a given simulation might only require that a developer equip his/her new agents with a standard method defining what to do when one gets "hit." If sharing agents between sims is to be a goal, negation commands (and perhaps a built-in negation operator) will be a great additions to AS-VAT command palettes.

Figure 10 shows the Neighbor_of command.

Figure 10: Neighbor_of command.

The Neighbor_of command can be used in any situation where an agent wants to check a Moore neighborhood of depth 1 for the presence of comparator < ">" > a user-specified number <1> of a user-specified agent-type <Fisch1>. Such considerations are relevant to many kinds of actions in sims; e.g. pollution diffusion, mating, food-search, etc. In general, such considerations are integral to research involving cellular automata or similar systems.[2]

Figure 11 shows the 4hood*Test_Number_Neighbors_Having_Attribute command.

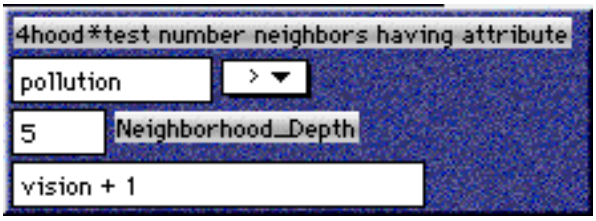


Figure 11: 4hood*Test_Number_Neighbors_Having_Attribute command.

This command can be used to test if comparator < ">" > a user-specified number of neighbors <5> in a title-specified neighborhood-type <4hood> of user-specified neighborhood_depth <vision + 1> of a given agent have a user-specified attribute <pollution>. (The above command checks its Moore neighborhood of depth "vision + 1" to see if greater than 5 agents in this region have pollution attributes.) This neighborhood-perusing command is also useful when programming cellular automata and like entities.

In my opinion, it is good to provide language pieces that address varying levels of discourse. For example, it is different to test if an agent "has a given attribute" as opposed to testing "the value of a given attribute" that an agent definitely possesses. Even if one can always find a way to pose one type of question in terms of the other, (which is doubtful, e.g. think of situations where attribute-less agents need to be initialized to contain certain attributes with certain initial values) the fact that these two approaches point to different ways of representing a given question lends credence to the idea that both forms of representation should be provided by a good language. i.e. It is often a good idea for a programming language to give users more than one way to implement their ideas. Providing commands that address different levels of discourse is one way to achieve this language-design goal.

---

[2] Cellular Automata have been extensively studied and have a vast literature associated with them. To begin see [52, 27, 54, 8].

Figure 12 shows the Remote_Test command.

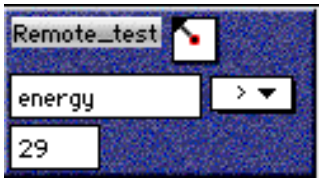


Figure 12: Remote_Test command.

This command tells an agent to check a user-specified attribute value <energy> of an agent at the cell in a user-specified direction 1 cell away to see if this attribute value is comparator < “>“ > a user-specified value <29>. This command can be used in *many* situations; e.g. an agent might check its northwest neighbors energy value to see if it is high enough for mating to ensue.

Figure 13 shows the Test_Absolute_Row_Val and Test_Absolute_Column_Val commands.

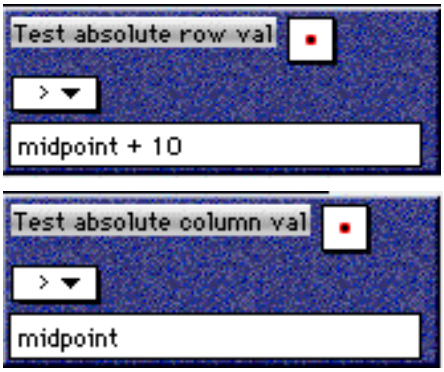


Figure 13: Test_Absolute_Row_Val and Test_Absolute_Column_Val commands.

These commands tell an agent to test itself (or any of its Moore neighbors at depth 1) to see if its row or column value (xposition and yposition in a worksheet) is comparator < “>“ > a user-specified formula value <e.g. midpoint + 10>. These commands can be used in situations where an agent requires information about its own position within a worksheet in order to execute a given rule. For example, I use these commands to help implement seasons in one of my Sugarscape sims. Winter and summer alternate between northern and southern sugar&spice agents (i.e. those located above and below the middle row of the worksheet, respectively) every seasonchange time steps.

Figure 14 shows the Test_String_Attribute command.

Figure 14: Test_String_Attribute command.

This command tells an agent to test if a user-specified attribute <season> is string-comparator <String-Equal> a user-specified string <winter>. This command should be useful in numerous situations. For example, I also use it in making sugar&spice agents change from season to season as described above.

AGES Simulations

In this section I will present the different simulations that I have so far generated with AGES. Hopefully, these simulations will give the reader a feel for the variety of cas that can be implemented in AGES via end-user programming in VAT. No programming in LISP is necessary to generate the following AGES sims.

Genetic Fishtank

By using combinations of the AGES commands shown in Appendix 1a coupled with the traditional AS-VAT commands I have been able to generate simulations of a number of different complex adaptive systems. My first simulation stems from the work of Gorman, Papp, and Pedritti [14]. GPP created a Genetic Fishtank Project in Agentsheets that promoted evolution of agents solely through the use of a mutation operator. More importantly, GPP introduced an interesting use of AS-VAT agent attribute values in their study. Designated attributes of AS-VAT agents are treated as genes. These genes take on integer values in the range [0..100] and represent independent probabilities that certain methods of an agent will be called at each time step. I use this mechanism to promote evolution in AGES. AGES also adds a host of other commands, including crossover, that were not available in the original Genetic Fishtank Project. (See Appendix 1a for a list of all AGES commands.)

In the Genetic Fishtank fish eat plants, and sharks eat fish to survive. Plants grow at a user-specified rate and have the ability to spread to different sites. The fish and the sharks in the tank reproduce and contain genes that are subject to simulated evolution. Figure 15 shows a list of a fish’s attributes/genes.

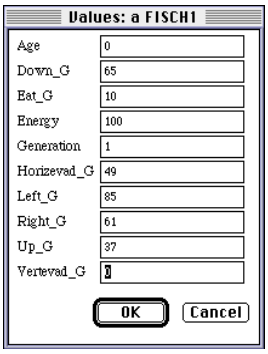


Figure 15: List of a Fish's attributes.

Each gene value above is indicated by a "_G" suffix. (Other non-gene attributes are also manipulated by the crossover command. This affects evolution by changing the schema boundaries of the genome [18, 20]. However, since no effort is being made to track and analyze schemas during evolution, variations such as these can be ignored. In fact, some algorithms explicitly shuffle gene orderings to minimize the effect of gene placement on evolution [4].) Figure 16 shows an example of how a gene value is used to determine the behavior of a fish. It shows how the gene "up_g" is related to its behavior.

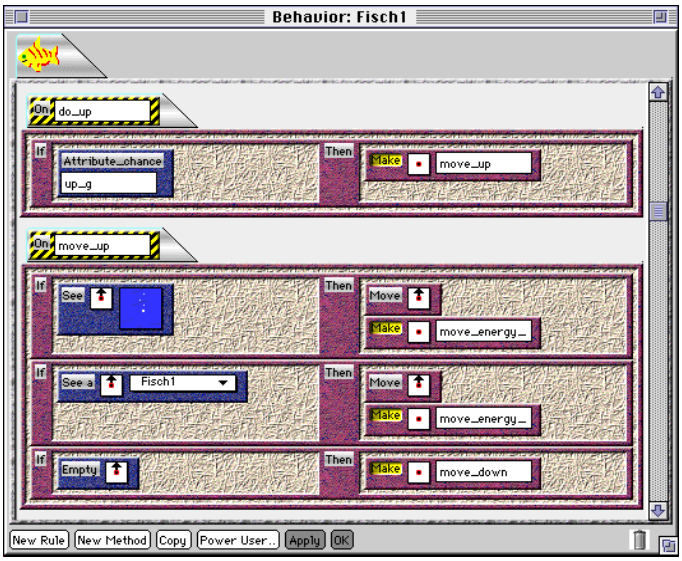


Figure 16: How gene values relate to method calls.

The above rule-editor shows how the value of a fish's up_g determines if its method move_up is called. A "roll of the dice" by the Attribute_Chance command generates a number in the range [0.. 100]. If this number is less than the attribute value accessed by the command, the agent's move_up method is called, else control is passed to a higher level. If move_up is called the fish moves up if it sees water or another fish above it. It then calls its own move_energy_cost method to "tax" itself for

moving. Such a move_energy_cost simulates metabolisms in real fish. If the cell above the fish is empty, i.e. the boundary of the worksheet in the Genetic Fishtank sim is one cell above, the fish moves down instead.

Each of the other methods of the fish is similarly "gated" by its attribute gene values. This structure, coupled with the crossover command that allows fish to reproduce and evolve, allows behaviors to emerge in fish that depend on evolution. Figure 17 shows how crossover is used to allow a fish to spawn and create evolved offspring.

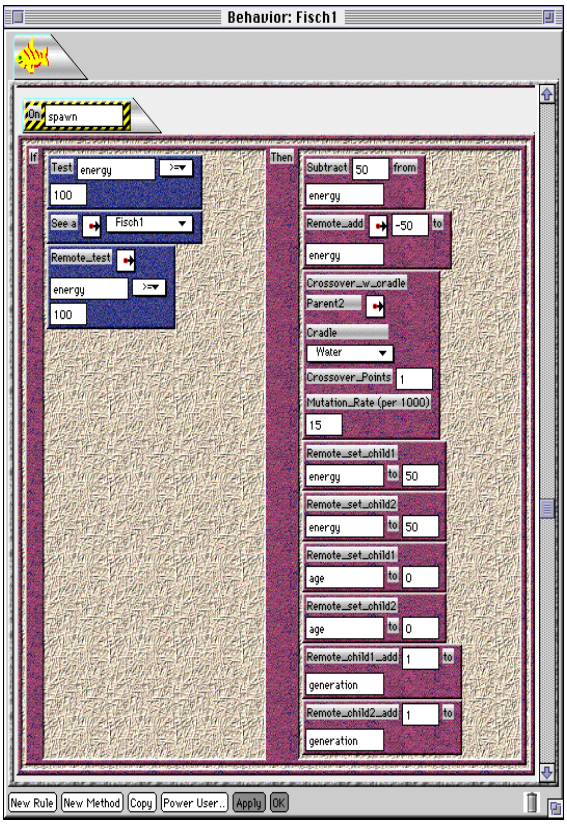


Figure 17: How a fish uses crossover to spawn.

In this case, no genes are involved in determining if a fish will spawn or not. (One could add such a relation if one so chose). When a fish's spawning method is called, which in these fish happens at every time step, it first checks to see if it has enough energy to spawn.[3] If the fish has enough energy, it then checks in a given direction to see if it is neighboring another fish. (The crossover direction and the

---

[3]Since fish start out with an energy supply of 50 units, each fish is required to have twice that amount to spawn. This allows each parent to donate 50 units of energy to a child. And since two offspring are produced, optimally, at each spawning, this leaves each parent and each of its offspring with energy values of 50 after crossover is performed.

See_a direction should be the same.) If the fish is next to another fish in the right direction, the executing fish then checks its neighbor's energy level to make sure it has enough energy to mate. If all these conditions are met, the fish executes its list of actions.

It first decrements its own and then its neighbor's energy values to account for the "donations" made by each to their offspring. Crossover is then performed with the appropriate parameters. In this case, two offspring are created and placed on top of neighboring water agents. Crossover generates the offspring's genomes using one crossover point and a probability of mutation at each locus of 15/1000. (See Appendix 1b for further information on the workings of the crossover operator.) The relevant variables of the children are then set using the Remote_Child commands. Similar VAT methods can be used to support simulated evolution in a wide variety of AS-VAT agents.

Appendix 4 shows the results of running the Genetic Fishtank simulation and allowing evolution to occur. All simulations shown started with the same seeds. (i.e. Each started with fish and sharks having the same initial attribute/gene values and located in the same initial positions.) It is easy to see that changing the position of plants in the tank leads to varying grouping behavior in the fish. When plants are placed at the bottom of the tank the gene values of surviving fish evolve to promote bottom feeding. When plants are placed at the top of the tank, the opposite occurs. Over time, appropriate sets of gene values evolve to allow fish to gain energy and survive.

Importantly, no methods exist in a fish that tell it to group. A fish's methods only tell it to move in a given direction, to avoid sharks vertically and/or horizontally, and to eat. Grouping behavior emerges as a result of evolution and interactions between fish and their environment (including other fish). Such emergent properties are indicative of complex adaptive systems. The Genetic Fish tank shows one way in which the commands provided by AGES can be used to model and explore emergent properties exhibited by complex adaptive systems.

Sugarscape

The new "bottom-up" approach to "constructing" understandings, interpretations, and models of interesting processes that we care about (e.g. economic processes, political processes, social processes, biological & ecological processes, and "categories-that-blend-the-above") has recently been championed by Axtell and Epstein in their book *Growing Artificial Societies*. In this work challenges are made to conventional ways of understanding economics. Traditional models tend to base their predictions and interpretations of economic systems on global considerations. For example, markets in traditional Walrasian economic systems work under the guide of an imaginary auctioneer that distributes information to all participants in the market. The effect this has on prices is to cause every merchant to sell comparable goods at an optimal global clearing price. Under such analysis, untended

markets tend toward a general equilibrium. Axtell and Epstein explain such a system in the following passage:

> The equilibrium concept used in general equilibrium theory is a deterministic one. That is, once the auctioneer announces the market-clearing price vector, all agents trade at exactly these prices. Each agent ends up with an allocation that cannot be improved upon. That is, a Pareto-optimal set of allocations obtains. Because these allocations are optimal, no further trading occurs, and the economy is said to be in equilibrium. Overall, equilibrium happens in a single trade step. [4]

Agent-based models like Sugarscape, on the other hand, involve agents with internal states (rules and attributes) that interact with a separate environment over time. Heuristically, one can think of the entire system in the following way:

> ...artificial society as a discrete dynamical system in which the vector **A** of all agent internal states and the vector of all environment external states **E** interacting as a high-dimensional dynamical system of the form:
>
> $$\mathbf{A}^{t+1} = \mathbf{f}(\mathbf{A}^t, \mathbf{E}^t)$$
>
> $$\mathbf{E}^{t+1} = \mathbf{g}(\mathbf{A}^t, \mathbf{E}^t)$$
>
> where the vector functions **f(*)** and **g(*)** map the space of all states at time t+1. [4].

Sugarscape models tend toward statistical equilibria as opposed to tending toward a static equilibrium [11]. This is partly due to the fact that the environment and the agents interacting within it are coupled in the above way. The effects of a changing environment on an agent, including a changing population of agents, tends to eliminate static niches. These interactions between agents and an external environment continually form and reform fitness landscapes such that agents tend not to get stuck in any one attractor (26). Statistical equilibria arise naturally within such coupled settings.Axtell & Epstein emphasize the importance of these considerations when they write:

> This brings us to the so-called First Welfare Theorem of neoclassical economics. The result is the foundation for economists' claims that markets allocate goods to their optimal social uses. The theorem states that Walrasian equilibria are Pareto-efficient. They are states in which no reallocation exists such that an agent can be made better off without making at least one other agent worse off. But in statistical equilibrium the First Welfare Theorem should be revised to say that a market equilibrium approximates but cannot achieve a Pareto-efficient allocation. How close a given market comes to Pareto-efficiency can be measured by the price dispersion in transactions. [Foley 1994: 343] It is exactly this price dispersion that we studied above and will investigate further below in the context of non-neoclassical agents. Thus, the philosophical underpinning for laissez-faire policies appears to be weak for markets that display statistical equilibrium. [2].

Axtell and Epstein go on to point out more dissimilarities between their Sugarscape models and classical economics models as Sugarscape agents are endowed with finite lives and changing preferences over time. Both extensions seem to make Sugarscape more like real-world agents and not less. Thus, it seems that Sugarscape models offer an alternative way to do economics, from the bottom-up, that captures many salient emergent properties of economic systems operating in far-from-

equilibrium conditions that cannot be described within traditional economic models. Such findings have profound implications for future policy decisions, and they emphasize the importance of studying cas through agent-based models.

Although I have not implemented all of the models described in *Growing Artificial Societies*, I have been able to successfully implement most of those models found in chapters 2 and 3 in AGES. Interesting aspects of these models include but are not limited to sexual reproduction with evolution, primitive tag-based cultural transmission, and combat among agents. They are the precursors to agents from chapter 4 that engage in trade and in so doing exhibit many of the interesting properties described above. It is hoped that AGES can be extended in the future to encompass all the types of agent transactions exemplified by Sugarscape models. (These include trade, hierarchical creditor-debtor relationships, and tag-based immunological adaptation.)

The above AGES simulations demonstrate the flexibility of AGES as a tool for modeling cas. AGES is capable of generating a wide variety of simulations with no recourse to programming in LISP. Many varied cas can be modeled simply by programming in VAT.

Empirical Work

This section will show how a small group of middle school students was also able to use commands from AGES to program creatures that behaved in ecologically-minded ways. Some creatures the kids programmed eat food and gain energy for doing so in addition to reproducing by using the crossover command. This preliminary study indicates that young children can use AGES as a way to explore ecological simulations involving ideas of metabolism and energy consumption, eating, and reproducing. More comprehensive studies will need to be carried out in the future to better assess the strengths and weaknesses of AGES as a pedagogical tool for studying cas.

My Experiences With the Science Discovery Kids

During a three day period I showed AS-VAT and AGES to five middle school students, ages 11-14. These kids were recruited from a summer program at CU called Science Discovery. All who participated did so voluntarily. The particular kids that I worked with were recruited from a “Math and Computers” class that met for one hour a day for four days. During this time the kids learned a bit about spreadsheets. Other than this, these kids claimed to have little or no “programming” experience. The following is an account of my three-days/three-hours with them.

Preface

I had mixed experience with these kids. A couple of them were mildly interested and just wanted to play around and have fun with the software. This was fine. I was hoping that they might come up with

something interesting. I was trusting that kids would learn more if they were allowed to playfully create worlds that were *personally meaningful* for them [46, 12].

All in all, this strategy seemed to work. I also had one student, Craig, who definitely preferred the "apprentice" approach learning [28]. As opposed to creating his own sims independently, Craig preferred work on projects together with me. As Mitch Resnick did with his study where he introduced StarLogo to high school kids, I openly helped students create programs in AS-VAT [46]. I saw myself as a collaborator as well as a mentor. Again, this attitude was especially prevalent in my dealings with Craig.

Day1

On the first day I introduced five interested students to ideas about Crossover and about Evolutionary Programming in general. I presented these ideas within the context of my Genetic Fishtank simulation.

I described the actions of the fish in the tank in terms of their gene values. I couched these relations in terms of "rolling the dice" to see which actions were prescribed by an agent's gene values at each time step. (I even used percentile dice to make these ideas more concrete.)

I told the kids that they might want to add new fish or just new creatures into this fish tank to begin. Or they might want to think about "geneticizing" a different "Space_Ships" sim where an Enterprise-like ship and a Romulan vessel could shoot at one another and move around the screen via user-chosen command keys. If they wanted, they could even start their own project. Actually, I was worried that they might not be able to even get to my "Genetic Commands" due to time constraints. Again, none of the five original participants claimed to have any programming experience. One student did say that he had "programmed some in VisualBasic with his Dad." Since I only had three days with these kids, it was doubtful that any "evolutionary" programming would emerge. Still, I hoped that the kids would be able to play with the simulations I had already created to help them gain an understanding of evolutionary and ecological processes .

On this first day, in explaining the above ideas I had students adjust parameters like the rate of growth of plants in the fish_tank, "move_costs" for fish and sharks, basal metabolism for fish and sharks, and relative numbers of fish and sharks in a given tank. My original sims already varied food placement and ratios of sharks to fish as well as varying absolute number differences in populations of fish and sharks in a tank.[4]They did well with adjusting these parameters and picked up on ecologically

---

[4]There is a big difference between a 7 to 1 ratio of sharks to fish exhibited by a tank with 7 fish and one shark as opposed to the same ratio of fish to sharks exhibited by a tank that contained 42 fish and 7 sharks. Initial population counts can be extremely crucial when working with genetic algorithms.

relevant considerations. For example, Phillip quickly noticed that putting too many sharks in the tank would result in all the fish getting eaten, which in turn would make all the sharks starve to death.

Day2

On the next day, I let the kids do what they wanted to do. I told them they could play with the Fish_Tank, play with geneticizing the “Space_Ships” sim, or they could start a sim of their own. Since I had not been able to go into detail about how AS-VAT worked, not many kids had the necessary knowledge to create a whole new simulation of their own.

I then told the students that I had planned to work on “geneticizing” the Space_Ships sim myself and that anyone who wanted could join me. I used this as an opportunity to show them how to program in AS-VAT. We quickly programmed the “space” in the Space_Ships game to “grow some food.” Together we came up with the idea of making a game that was a bit of a mix between Pac Man and “shooting” games. In this game, the space_ships would move around and eat food while having to defend themselves from evil Red-Dwarves (another agent in the gallery). The Enterprise and Romulan ships were equipped with torpedoes and lasers, respectively. They could use these to shoot the Red Dwarves. I continually asked the kids what they thought we should do next, but for this session I was on the keyboard. Again, I was showing them how to use AS-VAT.

After we programmed space to grow some food, everyone went to their respective computers and resumed playing with AS-VAT. Craig took my place at the keyboard, and he and I remained to program more into the Space_Ships sim. We started to tackle the problem of getting the ship to eat food and to gain energy from doing so. Since the ship already had other methods to make it fire, navigate, etc., it seemed natural to create another method that would allow it to eat. In my Genetic Commands I had created commands to do just this sort of thing. I asked to Craig to pick out which condition he thought might work for our task. He picked out the command “See_Agent_in_4neighborhood” and set it with a depth of 1. He then perused the Genetic Action Palette and quickly found the Eating Commands. He picked the “Eat_Neighboring_Item” command and chose to extract the “energy” from the items he ate as a reward for eating each item. This combination of condition and action worked as expected. After making a new call to our eat method and applying the new rules, the Enterprise moved around the screen eating the newly-grown food and extracting energy for doing so. At this point, 5:30 rolled around, and the session ended. Craig’s work for Day2 is shown in Figure 18.

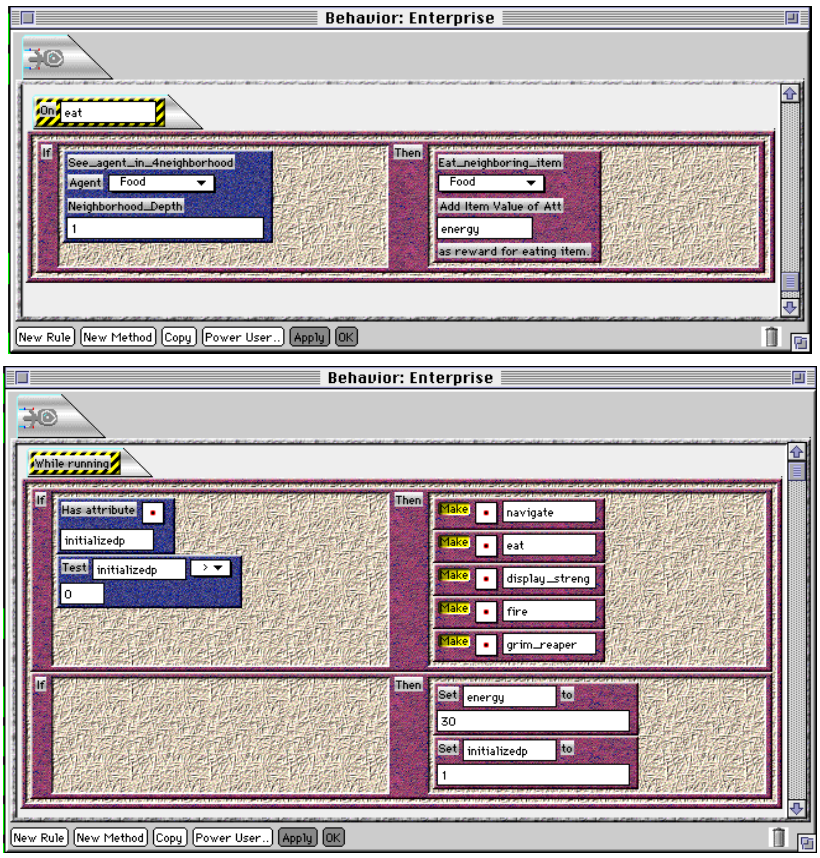


Figure 18: Craig created the eat method (top) and also made a call to this new method in the main loop of the Enterprise's program (bottom).

Another student, Mark, made a tank full of fish and plants of many varieties. After he left, I looked at his simulation that was still running. It contained a "school" of fish at the bottom of the tank even though there were no plants growing there. This was a classic case of overpopulation. I checked the attribute values for these fish, and not surprisingly, many had high values for their "Down_gene" and high "generation" values. It was obvious what had happened. Some initial fish had originally begun to congregate at the bottom of the tank due to their initial random gene values. Being near food at the bottom of the tank, these fish gained energy from eating and reproduced. These fish were bottom feeders as dictated by their genes, and so they tended to produce offspring that were bottom feeders as well. But after too much growth in the fish population due to abundant resources and overbreeding, the fish eventually ate all the plants. As I watched the simulation run on, the fish began to die off. Too bad Mark had to leave before his simulation dramatically played out its version of Malthusian overpopulation tendencies. (I showed it to him the next day, and he was pleased to see the results of letting the sim play out its theatrics.)

Day3

The next day, Craig and I continued programming the Space_Ships game. I had made some modifications overnight; I had created some Space_Crunchers to be new opponents for players of the game. The Space_Crunchers were swarming bugs that would move randomly if they were not next to a space ship. But once they began neighboring a space ship they stayed next to it and decreased the ship's energy until the ship moved, died, or until it shot the Space_Cruncher.[5] Craig quickly tweaked the parameters of the Space_Crunchers to increase their energy-draining capabilities. He wanted to make the game a little more difficult. The ultimate idea was to have these creatures eat food and reproduce. This would also help make them more worthy opponents for a master Space_Ships player.

On this day, Phillip was sitting near, and he also wanted to make his creatures reproduce. Phillip is quite imaginative and seemingly quite egalitarian if his programming ideas tell us anything about his personality. You see, in the fish tank, fish eat plants, and sharks eat fish, but nothing eats sharks. Phillip told me that he added his *telephones* into the tank to rectify this inequity. But he did so with a twist for the underdog; he programmed his telephones to eat sharks, but the fish could still eat the telephones, of course![6] Now he wanted to let his telephones mate and reproduce in the tank just like the fish and the sharks did.

I asked Craig and Phillip what they thought they needed to do to get their agents to reproduce, and they suggested we use Crossover. (I had been talking about Crossover since the first day they came in to play with the software.) At this point, something happened that I had not explicitly planned on. Phillip was using the Crossover operator basically as a mating operator without really thinking of it as an explicit genome splicer. Phillip wanted to make his telephones reproduce when they had no genes/attributes. It made perfect sense within the context of his simulation.

Through Phillip's actions I realized the crossover command could be used as a way to get kids thinking of reproducing creatures whose behaviors are not dependent on gene values as described above. It seems that this gentle introduction to crossover could also pave the way for understanding more complex genetic algorithm ideas later. Kids might be able to grasp the functionality of the

---

[5]I created Space_Crunchers in an attempt to make acreatur that would readily be understood as engagin in collective activity with other creatures of its same type. Kids would naturally want to make these creatures reproduce in order to observe such collective activity. This in turn would hopefully lead them to use my crossover command.

[6]Actually, his phones either erased themselves if they were to the left or to the right of a fish, or they were erased sharks if sharks were to the left or right of them. This is not "eating" in the same way that my "eating" commands define, but it sure looks the same!

crossover command one step at a time. First, it can be seen as a “mating” operator, and once kids get familiar with this idea, they can expand and explore ideas relating to gene values. Indeed, letting kids use the crossover command brought to the fore interesting and somewhat unexpected modes of use that involve no mention of genes at all. Craig was able to program his agents to reproduce, again, using one rule and a call to the method encapsulating it. Phillip never quite got around to calling his mate method, and when he dragged and dropped this method onto his telephones to see them reproduce it crashed the system. Again, I had not planned on such a use for crossover. This occurrence makes clear the extreme importance of the iterative approach to design. Phillip's and Craig's rules and calls are shown below.

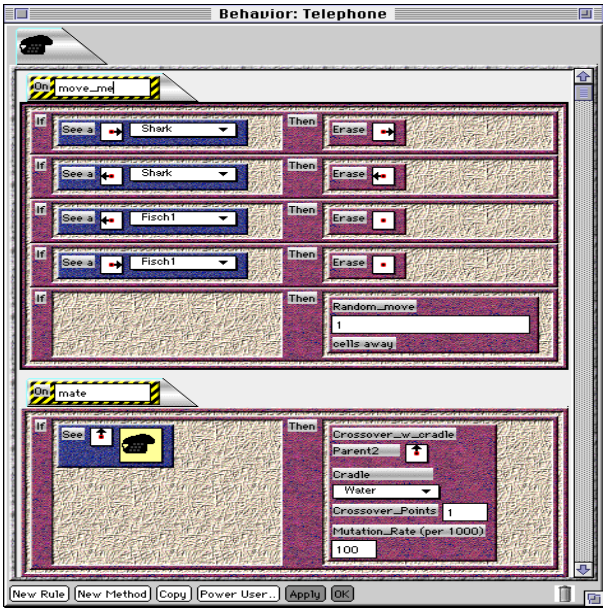


Figure 19: Rules for Phillip's egalitarian telephones.

The above rule editor shows the rules for Phillip’s telephones. He programmed the telephone himself during his third hour of using AS-VAT

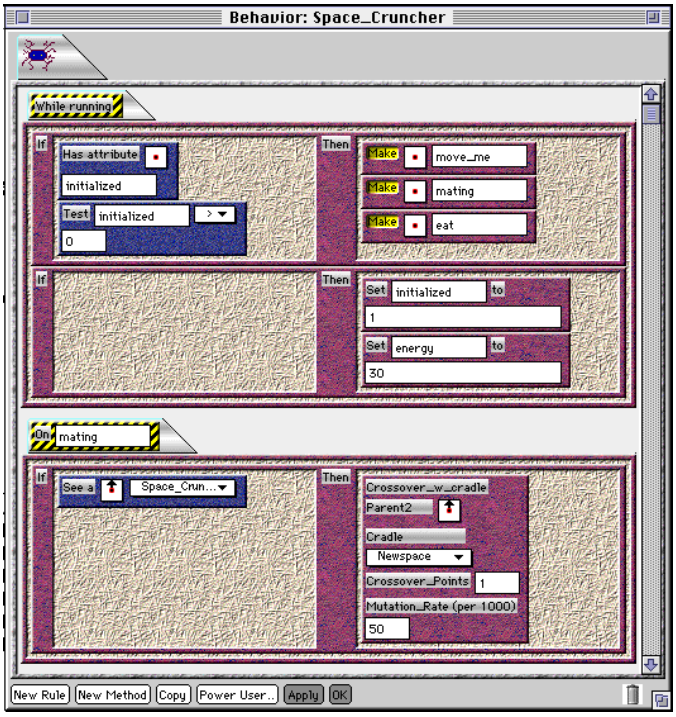


Figure 20: Rules for Craig's reproducing Space_Crunchers.

This rule-editor shows Craig's rule to make the Space_Crunchers reproduce (bottom), and it shows the call to his "mating" function in the Space_Cruncher's main loop (top).

Another interesting use of the genetic fish tank came from Mark. Again, on the previous day, Mark had produced overpopulation and starvation behavior that he left too early to see. On this day he had questions for me about how to program the plants not to grow past the rocks. I went over the structure of the program with him, and we ended up in the "sprout" routine which called "sprout_left" and "sprout_right." We looked at the "sprout_left" routine to see how the plants grew. After seeing a couple of rules that stopped the plants from growing in certain situations Mark decided that, to make the plants not grow if they saw a rock to their left, he should make them "do nothing" if they saw a rock to their left. He added this rule to the "sprout_left" method and a similar one to the "sprout_right" method and was noticeably pleased with the results.

He then proceeded to carry out an interesting experiment with the fish tank. In my original examples and in my demos I had always placed plants either at the bottom or at the top of the tank. This makes it easy to see that schooling behavior of the fish is affected by placement of the food in the tank. In different environments fish will school in different places, i.e. where the plants were. Putting the food at the top and the bottom of the tank also makes for an easy interpretation of how the gene values of genes such as the "up_gene" and the "down_gene" of the fish affect their behavior. High values for the "up_gene" in conjunction with plants at the top of the tank lead to schools of top-feeders. High values

for “down_genes” in conjunction with plants at the bottom of the tank lead to schools of bottom-feeders.

But Mark wasn’t satisfied with this. Instead, he created a tank with “ledges”, having rocks as “plant-ends” and plants growing between the rocks. He placed five ledges in the tank at various heights. Schooling behavior emerged around two of them toward the middle of the tank. Mark had used the environment as his programming tool, and he tested how the fish would react in an environment where the relationship between gene values and actions was not so direct. Clearly, the simulation stimulated him to think about the interactions of the genetically-affected fish as they interacted with their environment (including the other fish). In doing so, he even explored the emergent schooling property of the fish in a way that my original demos had not. A screen snap of his evolved fish tank with ledges is shown below.

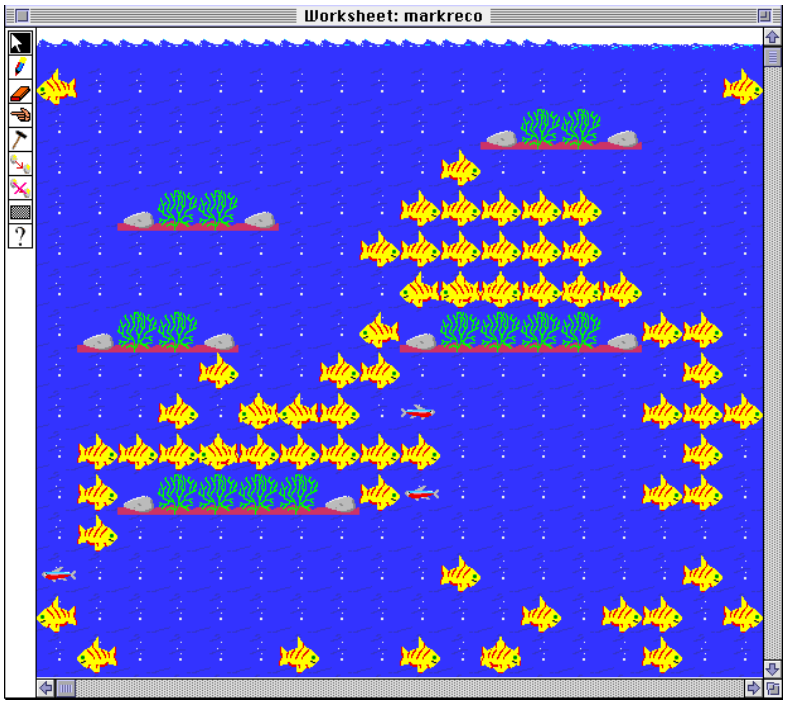


Figure 21: Mark's ledges.

The screen above shows Mark’s ledge filled tank with schools of fish around two of the ledges. Mark was exploring the relationships between gene values and behavior by changing the environment to see how the fish would react.

Analysis

In analyzing the effectiveness of AGES as a tool for enticing children to think about systems and the local interactions that generate many of their interesting properties, one should remember that the above children spent only three hours getting acquainted with and using AS-VAT and AGES. Overall, I consider these explorations with AGES to be a success. Although no kids actually grasped the

concept of evolving gene values enough to use them in programming evolutionary creatures, all kids seemed to understand and be interested in other important ecological concepts like basal_metabolism, movement_costs, and ideas concerning the relationship between rewards and penalties for actions. Craig was constantly calculating how many points should be gained or lost for certain actions. He wanted to make sure that the Space_Ships game was not too hard and not too easy, but just right. This led to him think about tradeoffs between different parts of the system. He said things like the following: How many points should the Space_Crunchies suck out of a ship? How much energy should we gain from eating a piece of food? “I’ll let the Space_Crunchies take 5 points away since we get ten points for eating a piece of food.”

Phillip was obviously concerned with “cycles”. His addition of *telephones* into the fish_tank was a curious choice, but time did not permit me to gracefully ask him about this. Telephones may have just been something that was fairly easy to draw in 3D. Phillip's addition of telephones to the tank instead of fish emphasizes the importance of letting kids playfully interact with concepts in creating artifacts that are *personally meaningful* to them. I would never have thought to teach Phillip to add telephones to the tank, so he definitely generated a novel and personally meaningful way to interact with the ideas I was presenting. And if Phil can abstract away from the situation, enough to add his somewhat egalitarian, underdog-supporting, shark-eating, fish-leary telephones to the fish tank, who knows what other kinds of abstractions he may be capable of carrying out *intuitively* when he has a say about the context within which they are introduced-to/generated-by him.

In addition, Phillip showed me a new way to think about my own crossover command, as a simple mating operator. Anyone who had listened to me talk about crossover would have thought I was well aware of this possibility, but I really wasn’t. After seeing Phillip’s approach to using crossover I will most certainly try to present the crossover command differently the next time I introduce it to kids.

Finally, Mark’s explorations were extremely enlightening. He was a little older than Craig and Phillip, and perhaps this had something to do with his grasp of the genetic concepts of the fish_tank. Again, although Mark didn’t use AGES commands to program, per se, he did use the environment as a programming tool in order to explore the relationships between the gene values of the fish and their behavior in the tank.

Comparisons With Other Alife Systems

In the previous section preliminary studies with middle school students indicated that AGES is easily learned and can be used to introduce concepts dealing with cas to grade school children. Introducing cas to such young audiences is not the goal of the systems to be described in the following section. Still, the following systems represent some of today's most interesting simulation-based approaches to studying cas. In this section, four systems for studying cas will be described and compared to AGES.

An overview of these comparisons will then be presented in the form of a table, and brief mention of directions for future work on AGES will be made.

ECHO

Echo class models were first introduced by John Holland [19]. These models are based on tag-mediated interactions between agents. Echo class models have the following components:

1) A Performance System: A performance system is composed of a set of detectors, a set of If/Then rules, and a set of effectors. They specify the agents'capabilities at a given point in time.
2) A way of adapting or evolving over time: Competition between agents, with local payments, allows agents to evolve over time and increase endogenous fitness measures.
3) A method of rule discovery: Rule Discovery allows agents to generate new rules for its performance system using previously tested rules in its performance system as building blocks. [18]

Much like in AS-VAT, agents in Echo class models inhabit individual sites where resources “grow” at different rates. Numerous agents can inhabit one sight. The agents at a site are arranged in a 1-D array in random order to simulate spatial proximity at each site. Agents gain resources from the site and from one another through trade and interchanges with the environment. In some echo class models a tax is levied against each agent at every time step. (This tax can be interpreted as biological metabolism in ecological sims.) Agents also trade and interact with one another according to their proximity in the array. Agents can move within this array. As well, if an agent gains no resources from a site during a given number of time steps, it can move to another randomly neighboring site to seek resources there. Agents live finite lives and try to gain resources in order to reproduce. Agents reproduce in accordance with their fitness, i.e. the amount of energy they have gained from their interactions with the environment. Those agents that have more energy are more likely to reproduce. [21]

Echo agents also have the ability to create aggregate-agents. This is accomplished by enlarging boundaries of agents in ways that preserve quick interactions within an aggregate while maintaining slow interactions with external agents outside the aggregate. For example, the reserves of all agents in an aggregate are pooled and are immediately available for use by all internal subagents. As well, damage to an aggregate agent affects all internal agents immediately. Contrastingly, the effects of an agent’s interactions with an external environment, including other agents, must percolate from agent to agent and from site to site at each time step. Effects are not directly felt by the environment “as a whole.”

The ability to aggregate, coupled with the ability to engage in message-passing, allows agents to create hierarchical programs during simulated evolution.[7] Such mechanisms exhibit powerful rule-discovery capabilities which enable echo class models to explore infinitely-dimensioned solution spaces through simulated evolution.

Though echo class models provide a wide variety of interactions among agents, such models do not aim at providing end-users a way to develop particular simulations, specifically tailored to their own interests, without resorting to programming in C or C++. Although many professional researchers are dedicated enough to endure this, many who would otherwise be greatly interested in cas, but who are not interested in programming, would probably not use Echo models to create simulations. In this situation, the ideas of cas, so important to us all, are left to be explored only by a programming-literate elite.

AGES, on the other hand, aims to spread concepts of cas to a much wider audience. With this in mind, AGES can be seen as a limited but powerful Artificial Life construction-kit tailored toward bringing concepts of self-organization and evolutionary adaptivity to a wide range of audiences.

Sugarscape

AGES' relationship with Sugarscape models has been described in a previous section, but there are still a few comparisons worth mentioning. AGES and Sugarscape take almost the same approach to modeling cas, and both even run on a Mac. Agents in both act on a 2-D grid according to simple rules. Both employ endogenous fitness functions, and both use traditional crossover operators to simulate mating. Sugarscape even employs the idea of a "cradle" described in the AGES section above. (Axtell & Epstein don't actually talk about "cradles," but they use the concept in their mating rule. [2: p. 56])

Unlike AGES, Sugarscape has much in the way of analytic capabilities. In this sense it is better suited for serious researchers studying cas than is AGES. However, Sugarscape does not provide nearly as much end-user modifiability as does AGES. To program novel simulations in Sugarscape requires object oriented programming in C++. In this sense AGES is better suited to bringing the study of cas to a wide range of audiences than are Sugarscape models. New simulations in AGES can be programmed using only VisualAgenTalk. Both models have their strengths and their weaknesses, but it is the overwhelming similarity between the two approaches that enabled me to replicate at least some of the findings of Sugarscape models in AGES.

---

[7]Echo agents can execute an action that is just passing a message to another agent or to itself indicating that some action should be executed. This ability is exhibited by AS-VAT as method calls.

Swarm

The Swarm simulation system is yet another creation emanating from researchers at the Santa Fe Institute. Information about and beta versions of Swarm can be obtained at the projects web pages: http://www.santafe.edu/projects/swarm/. Swarm is a toolkit for building multi-agent simulations to model complex adaptive systems. Like AGES, it attempts to provide a modeling framework within which interested researchers from varying disciplines can create computer simulations of cas. Swarm is based on object oriented technology and is written in Objective C. It can be run on Unix machines running X windows. Some of its interface components are programmed in Tcl/Tk which is in turn dependent on X windows.

In Swarm, numerous agents make up a swarm. In fact, any agent can be composed of a group of agents. This is reminiscent of Echo's aggregate agents. Each agent is an instantiation of a class and thus has its own private state variable values while sharing its behavior/methods with other members of its class. Swarm agents often interact within an environment. An environment is defined to be just another agent. In principle, this allows Swarm to provide many different types of environments within which other agents may interact.[8]

Swarm provides such agents as the above-mentioned **space** in the form of libraries. The **swarmobject** library contains the core classes from which all other agents in Swarm models inherit. The **activity** library contains Swarm's scheduling data structures and execution support. Interestingly, "probe" facilities have been implemented for Swarm classes that allow an object's state to be read or set and its methods to be called in a generic fashion without the need to generate extra code. The **simtools** library contains monitoring classes that can noiselessly "probe" Swarm simulations for data. Different classes in this library also provide graphs and summaries of statistical data. **ga** and **neuro** libraries that provide various genetic algorithm and neural network capabilities for Swarm agents.

Swarm is intended to be "an efficient, reliable, reusable software apparatus for experimentation" [32]. The goal of Swarm is to provide researchers with a common modeling kit that gives them the basic classes from which to develop a wide variety of cas. Swarm can be used to model different cas in different fields of study, from chemistry to political science.

Swarm seems well-suited to its task, provided that researchers become familiar with concepts surrounding object-oriented inheritance hierarchies. Such understanding is crucial if one is to create a simulation using agents that are similar-but-different from those provided in Swarm's libraries.

---

[8]At present, only a 2-D grid agent has been implemented. There are future plans to implement spaces with continuous dynamics defined by differential equations as well as spaces with three dimensions, non-discrete coordinates, and arbitrary graph structures.

Extensions such as these are made by specializing classes provided by Swarm's libraries. Programming in Objective C is required to make them. Programming in Tcl/Tk is required to make interface changes and extensions to Swarm.

Because Swarm is aimed at such an expert audience, namely scientists and other researcher of cas, expecting users to perform such programming may not be asking too much. Indeed, a tool such as Swarm greatly simplifies any programming effort that such a researcher would have to make if creating a simulation from scratch. As well, Swarm provides a universally-available free software package for creating cas simulations. If researchers wish to accurately replicate one another's results in order to build on them, such tools are a necessity.

Nevertheless, the aim of AGES is to bring the study of cas to a wider audience than just professional researchers. Although AGES in its present form does not provide all the capabilities represented in Swarm, it does provide all end-users, not just research scientists, with an agent-based simulation generator that supports evolutionary programming. End-users, even kids, have consistently been able to program AS-VAT simulations that exhibit interesting and complex properties. AGES gains such end-user modifiability for free by being embedded in AS-VAT. As well, preliminary studies with middle school students indicate that the commands specific to AGES are fairly intuitive. While AGES is neither as powerful nor as complete as Swarm, it is much easier to use.

Genesys/Tracker

The Genesys/Tracker system was developed for the Connection Machine by a group of distinguished researchers at UCLA [22]. The GT system evolved from an earlier, more limited Alife system called RAM [47]. RAM represented each organism as a parameterized LISP function and a sequence of parameter values that acted the organism's genome.[9] The genome was then subject to evolution according to a version of the genetic algorithm. Numerous studies were and are undertaken using RAM [50]. However, the authors wished to extend the ideas in RAM to include open-ended evolution. In other words, they wanted to add Holland's third capability of Rule Discovery to their system in order to allow the very form of the functions executed by their simulated creatures to evolve, as opposed to

---

[9]This is similar to the way AGES is structured. However, instead of being parameters to a function that describes the behavior of an agent, the genomes of AGES agents are made up of gene values that represent independent probabilities that "a given method will get called," "a given action or set of actions will be executed," etc. Which of these latter descriptions applies depends on the end-user-generated VAT rules used to specify the behavior of a given agent. Of course, simulations employing more RAM-like structures could also be implemented in AGES.

only evolving parameters to a user-defined, unchanging behavior function. GT represents the results of this extension.

GT provides users a way to simulate evolution through the use of a fairly standard genetic algorithm using a bit-string representation of the genome, executing random selection, and applying crossover and point-mutation operations to mating pairs. The system also employs two different representations of phenotypes; they are represented both as Finite State Automata (FSA's) and as Artificial Neural Nets (ANN's). The researchers developing GT chose these two forms of representation for phenotypes in order to assure that results obtained from simulation experiments would not be best explained as artefacts of any particular phenotypic representation used. It was thought that FSA's and ANN's are sufficiently dissimilar representations to ensure that results obtained using both would not be best construed as such representational artifacts. [23]

GT researchers are most concerned with exploring evolutionary systems "from scratch." They wish to explore open-ended evolution where little if any information about fitness functions is built-in to representations of the system. They are also concerned with the biological verisimilitude of their evolving creatures.

The Genesys/Tracker system shows how explicit phenotypic representations can be used to create more biologically realistic simulated creatures. Future work with GT might incorporate learning into the ANN ants to further pursue such realism. As well, different tasks might be attempted that are more amenable to spatially-constrained mating schemes as opposed to the random mating scheme currently used.

The GT system is more robust and flexible than AGES, especially with respect to its ability to engage in rule discovery and its ability to employ FSA's and ANN's as phenotypes. Still, GT does not allow users to easily create new simulations. Users must program in C++ to create new simulations in GT. In contrast, users can create many new and different simulations in AGES using only VAT. Again, this allows cas to be explored both by non-programmer experts in various fields and by non-expert end-users with interests in cas.

Comparison Overview & Future Work

Table 1 provides an overview of the above comparisons between AGES and other Alife systems used to study cas.

Table 1: System Comparison Chart

	Open-Ended Evolution (Rule-Discovery)	End-User Modifiable	User-Defined "looks" for Agents	Analytic Capabilities	ANN Phenotype Available	Grid-Based

Echo	X			X		X
Sugarscape				X		X
Swarm	X			X	X	X
Genesys/T	X			X	X	X
AGES		X	X			X

As Table 1 shows, AGES is the only system surveyed that allows users to program agents in an end-user programming language. Although AGES is limited in other capabilities, this property alone makes it a good candidate system for introducing ideas about complex adaptive systems to non-experts. AGES is also the only system surveyed that allows users to easily create iconic depictions for agents composing a cas. It is claimed that this capability makes simulations more intuitive and engaging for end-users. Agents can take on lifelike shapes and forms that remind users of real-world cas.

Nevertheless, AGES can be extended in ways that, like GT, emphasize more biological realism and an agent's ability to engage in open-ended evolution. Artificial Neural Network commands would be a welcome addition to AGES. First steps toward introducing them could allow evolution to "train the weights" as GT does, via crossover and mutation. More advanced versions might introduce different training algorithms including Backpropagation (not biologically realistic in its own right) and/or Reinforcement Learning Algorithms [1]. Adding in the ability to engage in rule discovery might require a more extensive reworking of the framework of AGES. In addition, analytic capabilities could be added to AGES in order to make it a more serious university-level research tool for studying cas. Such extensions are left as future work.

Interested readers might want to explore two other systems that are relevant to the topics discussed in this thesis. StarLogo is a system implemented by Mitch Resnick at MIT to introduce high school students (and perhaps younger ones) to decentralized parallel processes [46]. For those interested in agents exhibiting biological realism, Yeager's Polyworld is an interesting system to explore [55]. Like GT, Polyworld also combines ANN's and genetic algorithms, but it does so in a 3-D non-grid-based world.

AS-VAT Programming Maxims

The following maxims apply at least to programming in AS-VAT in particular, but they are perhaps more general than this. Similar maxims and programming guidelines can be found in works by Parnas [36,37] and Brooks [6, 7]. In addition, though time constraints have not allowed me to reimplement my original commands according to the maxims described below, and thus, these newly proposed

commands have not been tested, I am still fairly certain that changes made to the below commands according to the prescribed maxims will only be helpful for future users of AGES. It is hoped that the following maxims might be seen as a stand-alone contribution from this thesis that is not directly tied to AGES. Again, the following maxims might be seen as general programming guidelines with special relevance for VAT programmers.

When designing AS-VAT commands a number of issues arise as to how one should present a given command. Deep questions about how a user can best understand a command in different contexts become crucial. The interface of a command can closely mimic its underlying processes or not. There is hardly a limit to the amount of dissociation in structure to be found between the process-as-described and the process-as-implemented. This still does not imply that relations between such ways of understanding a process are inescapably opaque. Concrete design examples are probably most helpful in clearing this opacity.

Maxim**: Task-Based Design**

"Keep your audience in mind at all times. Design for specific explicitly defined tasks to be carried out by members of an explicit target audience."

This all-purpose design maxim is that espoused by many software engineers who take the iterative approach to design. [15, 16, 29, 6]. Such an iterative approach advises designers to design tools around the tasks that actual users of the system will use it to perform. This requires that such users and their tasks be explicitly defined and considered in defining various aspects of the system.

In designing AS-VAT one must keep in mind the age group and levels of expertise of end-users who will use newly-created commands. As well, AS-VAT command designers have much to say about the level of flexibility a user has in specifying parameters for different commands. More or less information hiding, e.g., will be desired for different audiences, e.g. middle school students as opposed to Professors holding PhD's in various fields.

Even so, it will always be impossible, even for the experts themselves, to predict how a tool will fail to meet certain task-dependent desiderata. Thus, it will always be necessary to produce a number of versions of a piece of software to be tested/used by users. Feedback from such testing should then be incorporated into a round of implementation improvements which then lead to a new version of the software to be tested.... Such an iterative process is the most effective way to arrive at pieces of software that are tailored to the unforeseeable needs of actual users of a system.

Maxim**: Forms vs. Formulas**

"Form-based approaches aren't always more simple than algebraic ones."

It is not uncommon to present users with forms in which parameters to a given function are to be specified. Such form-based presentations of parametric functions hide complexities of the functions being used. Forms can often simplify the process of specifying parameters. However, it is also possible for such information hiding to obscure important relations between parameters that are explicitly represented in formula-based representations of such functions.

Figure 22 shows and example command whose form-based interface seems more confusing than a corresponding formulaic version.

Figure 22: Form-based Pollution command

This Eat_Max_Food_Move_w_Pollution command implements a pollution formation rule that is described as follows by Axtell&Epstein:

> Pollution formation rule P_{ab}: When sugar quantity *s* is gathered from the sugarscape, an amount of production pollution is generated as quantity *a* * *s*. When sugar amount m is consumed (metabolized) consumption pollution is generated according to *b* * *m*. The total pollution on a site at time t ,p^t, is the sum of the pollution present at the previous time, plus the pollution resulting from production and consumption activities, that is,
>
> $p^t = p^{t-1} + (a * s) + (b * m)$. [2, p47]

The command can be presented to users in a form-based way as in the above command. The mapping from the above equation to the command is as follows:

p^{t-1} = value of Pollution_Attribute
s = value of Food_Attribute_to_Eat
m = value of Metabolism_Attribute
a = value of Food_Pollution_Parameter
b = value of Metabolism_Pollution_Parameter

The user-specified neighborhood_depth and title-specified neighborhood type descriptors determine an agent's range for eating food. Often times the neighborhood_depth is set to the value of an agent's vision attribute as in the above example.

It is hardly clear that separating the variables of the command into separate editing windows is helpful in understanding what this command does. With such an interface to the command the relationships between parameters described by the above equation are lost. This can sometimes be more confusing than viewing the actual equation. The command below shows the same command with the above equation entered into a formula box:

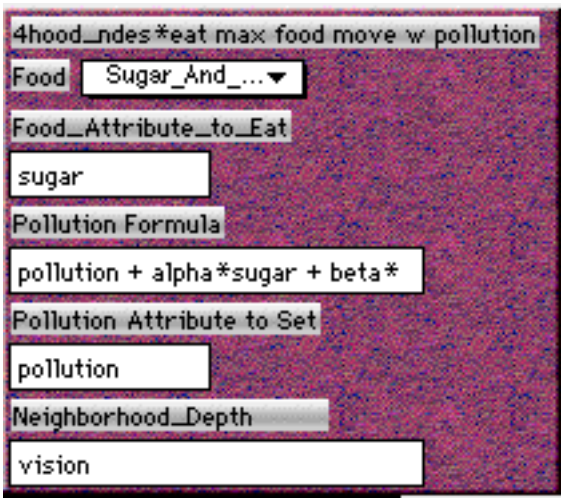


Figure 23: Formula-based 4hood_ndes*eat_max_food_move_w_pollution command.

In the new command the old "Metabolism Attribute," "Food_Pollution_Parameter," and "Metabolism_Pollution_Parameter" fields are packed into the current "Pollution Formula" field. If the equation could be viewed in its entirety, this command would show a mixed approach to using such mathematical notations. Instead of using just letters to denote variables, actual words are used. This makes algebraic formulations more readable and intuitive to end-users/VAT-programmers. In addition, such a formulaic interface to expressing functional relations between agents makes explicit the relations between parameters that are left implicit in form-based descriptions. Such capabilities of formula-types in AS-VAT provide a direct bridge to applying well-known and even yet-to-be-discovered mathematical descriptions of common real-world agent behaviors to the definition of simulated agent behaviors in direct and intuitive ways.

Maxim**: Explicit Assumptions**

"Make all relevant assumptions of a command explicit parameters to it in some fashion."

Although it is usually wise to hide irrelevant details from users [36, 37] it is detrimental to hide relevant details of functions from them. If this occurs relevant aspects of the interface of a function that are hidden from the user can lead to unexplained or confusing behavior. The 4hood*Random_Move

command of Figure 24 suffers from the malady of hiding relevant aspects of its functioning from users/VAT programmers.

Figure 24: 4Hood*Random_Move command.

This command seems ok (except for the title) at first glance. It should make an agent move to a randomly chosen cell one cell away in that agent's von Neumann neighborhood. But there is an implicit variable hidden in this command. When I made it, I assumed that a worksheet in which agents would interact would be tiled with other agents to serve as an environment. Water agents serve this purpose in the Genetic_Fish worksheets; NewSpace agents serve this purpose in my Space_Ships worksheets; etc. However, if a user wishes agents to move within blank worksheets, those with no active agents (from the gallery) serving as external environments, this command won't work.

To remedy this situation, all parameters implicit in this commands functioning should be made explicit. But once we make explicit the notion that an agent should randomly move only on top of another type of agent, specifying different step sizes for the agent becomes problematic. It is both computationally and conceptually intensive to think of an agent moving only on a path tiled with a given agent-type at each step. For these reasons it is probably best to amend this command by limiting it to a step size of 1 and by providing either a depiction-specification or a class-specification of the agents that can be "stepped on." Such modifications would result in the following two commands that determine "paths" of movement according to depictions or to class-types:

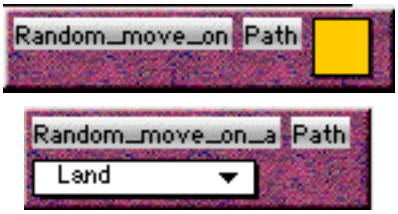


Figure 25: Revised Random_Move_On commands with no relevant assumptions left implicit.

Determining which assumptions of a command are relevant and worthy of being included as user-specified parameters is a somewhat artistic design decision at this point. Increased experience at designing AS-VAT language pieces inevitably leads to increased awareness of the "relevance" of given assumptions.

Maxim**: Specialization and Redundancy**

"There is always a trade-off between creating a number of specialized, finely-tuned yet redundant commands and the cognitive overhead incurred by having to use a greater number of commands."

In learning different constructs of a programming language, users/programmers will often find the functionalities of different language pieces to be redundant. At least, there will usually be more than one way to program a given function within a language. For example, there is often a choice between implementing an iterative loop as opposed to a recursive function to program a counter. The functionality of iterative loops and recursive functions is fairly redundant. It is good for a programmer to be familiar with both approaches. Still, it might be best to teach beginning programmers one approach at a time in order to minimize confusion. Exposing them to too many ideas at once can be overwhelming. Figure 26 shows a set of AGES commands whose functionality is fairly redundant.

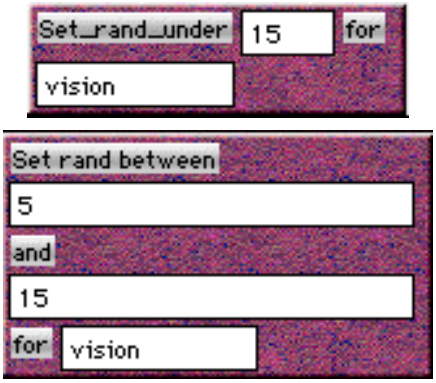


Figure 26: Set_Rand_Under and Set_Rand_Between commands

Both of the above commands can be used to perform the action indicated by the Set_Rand_Under command. (Just use “Set_Rand_Between 0 and 15 for vision.”) But only the Set_Rand_Between command can be used to perform its task. Set_Rand_Between subsumes the functionality of the Set_Rand_Under command. Any task that one can perform using Set_Rand_Under, one can also perform using Set_Rand_Between, but not vice versa.

Often, this redundancy is not to be avoided. Users may prefer to use the two-parameter Set_Rand_Under command as opposed to the three-parameter Set_Rand_Between, especially if they encounter many situations in which the former will work well. But one must always keep in mind that too many commands in a palette can be overwhelming. Depending on the audience at which a set of commands is aimed, this maxim becomes more or less important. (e.g. If the intended audience needs to be “coaxed” into “programming,” too large a number of commands in a palette can seem daunting. If the intended audience is a group of programming experts, these considerations become less relevant.)

Maxim**: Parameters**

“Don’t wrap too many command decisions into a name. Make parameters of them.”

Sometimes a function can become too specialized, working in only a small number of situations. When this happens it is usually a good idea to make the function more flexible by making parameters

of values in the function that were previously treated as constants or programmer-defined variables. This fix will result in more general commands/functions that empower knowledgeable programmers.

Figure 27 shows a version of the 4hood_Ndes*Eat_Max_Food_Move command with both the Neighborhood-type (4hood) and the choice of non-destructive or destructive eating built-in to the title.

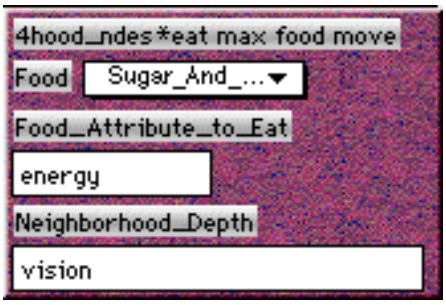


Figure 27: 4Hood_Ndes*Eat_Max_Food_Move command.

"4hood" at the beginning of the command indicates that it operates over a von Neumann neighborhood. "ndes" following this indicates that the agent executing this command eats "nondestructively." Such wrapping of design decisions into names takes flexibility away from the user if other specialized commands representing alternative choices are not available. Within reason, it is good to include such considerations as command parameters. An example of such a "fix" of the problematic components of the above command is shown below:

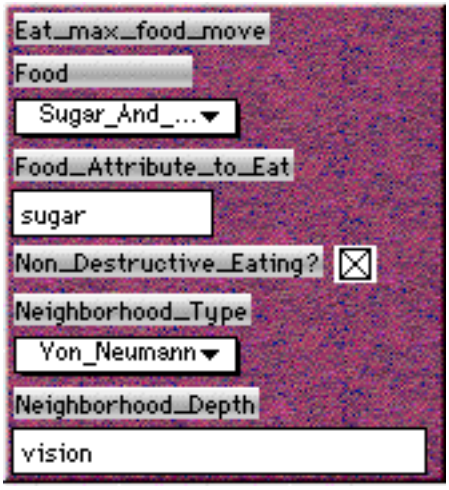


Figure 28: Parameterized version of the Eat_Max_Food_Move command.

This new version of the Eat_Max_Food_Move command is more general. It allows users to choose the neighborhood-type within which the command will search for food (e.g. Von_Neumann (4hood) as opposed to Moore (8hood)) It also lets the user specify whether the eating takes place "destructively" or "non-destructively." Such flexibility gives users more choices and alleviates the need to create numerous specialized version of the same command.

However, such power is passed on to users at the cost of introducing a more complex command. The new command has more parameters than the original. Still, if these parameters are relevant to the problem at hand, understanding their meaning will be necessary in order to choose "which specialized command to use when" in the same way that such understanding will be necessary in making choices for parameter value specifications. My preferences in such cases, all other considerations being equal, is to opt for creating one flexible, user-empowering command as opposed to creating many more specialized ones.

Maxim: Overly General Commands

"Sometimes overly general commands and/or unintuitive initial settings can lead to confusion instead of user-empowering clarity."

This maxim is almost the opposite of the preceding one. It warns about making commands/functions that are overly-general and confusing. Parnas argues that only relevant interfaces of functions should be made available to programmers, nothing more [37]. Deciding how to best present a function will always be somewhat ill-defined and dependent on each particular programming situation. Figure 29 shows examples of the Test_Absolute_Column_Val command that can be confusing to VAT programmers.

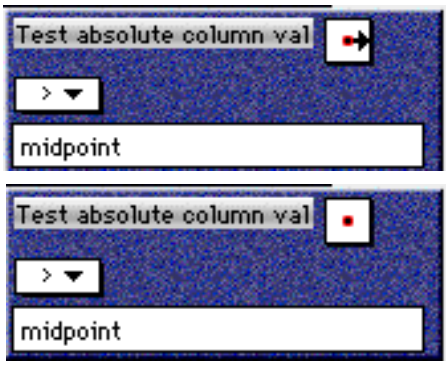


Figure 29: two examples of the Test_Absolute_Column_Val command.

Seeing the first version of this command might be confusing to some people. The direction arrow is probably not very intuitive; it might lead one to think that the neighbor being specified has some special sort of "column" property, etc. However, after seeing the bottom version of the command, the command's use might become more clear. This portrayal reads "Test to see if the absolute column val of myself is greater than midpoint." In such case, it might become more clear that "column val" refers to the xposition of an agent. (Perhaps a simple renaming of this command along these lines is in order.)

Nevertheless, the command exhibits the property that an overly-general command, coupled with an unintuitive set of initial variables, can lead to confusing interpretations of the command's function. In

such cases it is a good idea to limit the degrees of freedom of the command, i.e. remove parameters (such as the direction parameters above). One can also specify more intuitively appealing initial variable values for the command; e.g. initialize the direction parameter to refer to oneself (bottom). Learning to adequately manage such design decisions comes from experience and at this point remains a fairly artistic endeavor.

Maxim: Consistency

"Consistency with naming conventions observed by other previously-defined commands should be strived for."

It is a commonly held belief that consistently applied naming schemes can help users/programmers better understand their own and others' code. Some advocate the use of non-mnemonic variable and function names in order to force users to read code carefully in order to understand it [37]. Others advocate mnemonic variable-naming schemes to make code more readable. In either case, the importance of naming consistency is stressed. Figure 30 shows a naming convention adopted for the See and See_a VAT condition commands.

Figure 30: See and See_a commands.

These commands observe the naming convention that commands involving "depiction-types" (above) do not contain indefinite articles while those commands involving "vat-class-name-types" (below) do. If such a naming convention is to be strived for we should also try to apply it to analogous uses of depiction-types and class-types in other commands. The "Next to" and "Neighbor_of" commands provide us this opportunity.

Figure 31: Next_to and Neighbor_of commands.

Unfortunately, this naming convention doesn't easily apply here. Again, such naming conventions should be "strived for"; these maxims are not written in stone.

Maxim**: Use Formula Fields**

“Always use formula-types as parameters when the situation permits.”

Within AGES much power and flexibility in exploring cas arises when attributes/genes are subject to evolution. Using formula fields to designate parameter values in commands makes it easier to incorporate evolving variable values into complex attribute relations within and between agents. Figure 32 shows a version of the Neighbor_of command that uses a simple number-type field as a parameter.

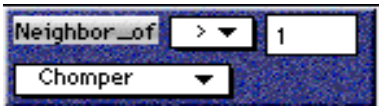


Figure 32: Neighbor_of command.

If this command used a formula-type parameter instead of a number-type parameter <1>, attribute values and formulas manipulating such values, in addition to simple numerical values, could be parameters to the command. The formula-type parameter subsumes the current number-type parameter. The altered Neighbor_of command is shown below:

Figure 33: Neighbor_of command with formula field.

In this case, the above command checks to see if greater than a specified threshold number of Chompers neighbors the executing agent in its Moore neighborhood. One can make great use of formula fields when programming sims for AGES. Again, this is because any value that can be expressed as an attribute is then subject to evolution according to crossover and mutation. In the above case, the attribute "chomper_tolerance" could be subject to evolution and could be used to explore poupulation dynamics among heterogeneous agents. An evolutionary exploration into segreagation ideas posed by Schelling could be carried out with the help of the above command. Formula windows allow users to use attribute values in specifying mathematical relations within and between agents. As attribute values evolve, so do the mathematical relations within which they are embedded.

Hopefully , the above maxims will prove useful to those designing new AS-VAT commands and perhaps to software designers in general. Again, the preceding maxims can guide design, but they need not fully determine it. As always, much interpretation and innovation is left in the hands of the designer.

Conclusions

AGES has already been used to model various complex adaptive systems. The genetic_fishtank project points the way toward using cas in modeling ecological systems. That AGES has been used to implement various cas as described in chapters 2 and 3 of Axtell and Epstein's *Growing Artificial Societies* indicates that AGES is versatile enough to model systems exhibiting cultural norms and combat as well. Such demonstrations indicate the flexibility of AGES and AS-VAT in modelling complex adaptive systems. No programming in LISP was necessary to implement these different cas. All were implemented using VAT coupled with the new commands provided in AGES. Given these findings, it is likely that experts from different fields who study cas could quickly learn to use AGES to model cas of particular interest to them. Systems that allow users to program cas without starting from scratch are needed as the study of cas continues to grow.

Unlike other systems, however, AGES has been demonstrated to be accessible even to children. This points the way toward introducing studies of complex adaptive systems to grade school children. Such an introduction would inevitably change the way kids understand science and the world in general. By introducing children to cas through AGES we give them the chance to question centralized approaches to explanation at an early age. In questioning these approaches they will learn more about both centralized and decentralized thinking. Relevant applications and strengths and weaknesses of both types of explanation can be made evident by creating simulations in AGES.

It is also hoped that using AGES can break down the barriers between work and play. As these barriers erode kids sill only be more excited and motivated to learn. As educators, we should do everything possible to direct most kids' frenetic energy toward playful yet educational activities. AGES represents an attempt to do just this.

AGES' import, however, is not limited to teaching children. Many adults, including professional researchers, might prefer programming in VAT to programming in lower-level languages like C and C++. AGES allows even sophisticated users to pursue their interests in cas. Still, more sophisticated analytic capabilities will need to be added to AGES before it can be used as a serious university-level research tool for studying cas.

Finally, I believe that the field of complex adaptive systems is one of the most important fields that can be studied. It is crucial to the well-being of humanity that characteristics of cas be explored and understood. The workings of global and local markets, of political and social institutions, and of the brain itself are but a few of the important processes addressed by studies of cas. Such studies of cas are today promoting levels of interdisciplinary scholarship that have not been seen since the Renaissance. Hopefully, such scholarship will shed new light on old questions. I agree with John Holland in being

optimistic about the import of trying to uncover the common secrets influencing all complex adaptive systems:

> It is an endeavor that can hardly fail. At worst it will disclose new sights and perspectives. At best it will reveal the general principles that we seek [18].

AGES is a tool aimed at actively sharing such insights about cas with as wide an audience as possible.

REFERENCES

[1] Ackley, D., Littman M. (1994) A Case for Lamarckian Evolution. C.G. Langton (ed.) *Artificial Life III*. Addison-Wesley, 3-10.

[2] Axtell, Robert. & Epstein, J. M. (1996) *Growing Artificial Societies: Social Science from the Bottom Up*. Brookings Institution Press, MIT Press.

[3] Baecker, R.M.; Grudin, J.; Buxton, W.A.S.; Greenberg S.; (1995) Design and Evaluation. Intro to Chapter 2 in *Human-Computer Interaction: Toward the Year 2000*. Morgan-Kaufmann, 73-91.

[4] Bartlett, Geoff. (1995) Genie: A First GA. in Lance Chambers (ed.) *Practical Handbook of Genetic Algorithms: Applications, Volume 1*. CRC Press, Boca Raton, 31-56.

[5] Boden, Margaret. (1996) Autonomy and Artificiality. in Margaret Boden (ed.) *The Philosophy of Artificial Life*. Oxford University Press, 95-108.

[6] Brooks, F.P. (1975) *The Mythical Man-Month.* Addison-Wesley.

[7] Brooks, F.P. (1987) No Silver Bullet. *Computer*. v. 20, #4, 10-19.

[8] Conway, J.H. (1982) What is Life? In *Winning Ways for Your Mathematical Plays*, edited by E. Berlekamp, J. H. Conway and R. Guy, Vol. 2, chap. 25. New York: Academic Press.

[9] Clark, Andy. (1996) Happy Couplings: Emergence and Explanatory Interlock. in Margaret Boden (ed.) *The Philosophy of Artificial Life*. Oxford University Press, 262-281.

[10] Csikszentmihalyi, M. (1996) *Creativity.* HarperCollins.

[11] Foley, D.K. (1994) A Statistical Equilibrium Theory of Markets. *Journal of Economic Theory.* 62: 321-45.

[12] Gargarian, G. (1996) The Art of Design. in Y. Kafai & M. Resnick (eds.) *Constructionism in Practice*. Lawrence Erlbaum, 125-159.

[13] Gell-Mann, M. (1994) Complex Adaptive Systems. in George Cowan, David Pines, David Meltzer (eds.) *Complexity: Metaphors, Models, and Reality.* Addison-Wesley, 17-28.

[14] Gorman, A., Papp, R., Pedretti, J. Genetic Fishtank Project. Unpublished paper. Fall '96. AI-CSCI 5582 University of Colorado-Boulder

[15] Gould, J. (1995) How to Design Usable Systems in R.M. Baecker, J. Grudin, W.A.S. Buxton, S. Greenberg (eds.) *Human Computer Interaction: Toward the Year 2000*. Morgan-Kaufmann, 93-121.

[16] Gould, J., Lewis, C. (1985) Designing for Usability: Key Principles and What Designers Think. *Communications of the ACM* 28(3), 300-311.

[17] Hofstadter, D. (1985) *Metamagical Themas: Questing for the Essence of Mind and Pattern.* Hammondsworth: Penguin.

[18] Holland, John H. (1995) *Hidden Order: How Adaptation Builds Complexity*. Addison-Wesley.

[19] Holland, John H., (1994) Echoing Emergence: Objectives, Rough Definitions, and Speculations for ECHO-Class Models. in George A. Cowan, David Pines, David Meltzer (eds.) *Complexity: Metaphors, Models, and Reality*. Addison-Wesley, 309-333.

[20] Holland, John H. (1992) *Adaptation in Natural and Artificial Systems*. University of Michigan Press. Second edition: MIT Press.

[21] Hraber, P.T., Jones, T., Forrest, S. (1996) The Ecology of Echo. to appear in *Artificial Life*. Addison-Wesley.

[22] Janikow, C. Z., Michalewica, Z. 1991. An experimental comparison of binary and floating point representations in genetic algorithms. R.K. Belew and L.B.Booker (eds.) *Proceedings of the Fourth International Congress on Genetic Algorithms*, Morgan Kaufman.

[23] Jefferson, D. et.al. (1991) Evolution as a Theme in Artificial Life: The Genesys/Tracker System. C. Langton, C. Taylor, J.D. Farmer, S. Rasmussen (eds.) *Artificial Life II.* Addison-Wesley, 549-578.

[24] Koza, J. (1994) Genetic Programming II. MIT Press.

[25] Koza, J. (1992) Genetic Programming. MIT Press.

[26] Langton, Christopher G. (1996) Artificial Life. in Margaret Boden (ed.) *The Philosophy of Artificial Life*. Oxford University Press, 39-94.

[27] Langton, C.G. (1986) Studying Artificial Life with Cellular Automata. *Physica D* **22**: 120-149.

[28] Lave, J., Wenger, E. (1991) *Situated Learning.* Cambridge.

[29] Lewis, C., Rieman, J. (1993) *Task-Centered User Interface Design*, self-published over the Internet.

[30] Lindgren, K. (1990) Evolutionary Phenomena in Simple Dynamics. in Christopher Langton, Charles Taylor, J.Doyne Farmer, Steen Rasmussen (eds.) Artificial Life II. Addison-Wesley, 1990, 295-312.

[31] Luce, R.D., Raiffa, H. (1957) *Games and Decisions*. New York: Wiley.

[32] Minar, N., Burkhart, R., Langton, C., Askenzai, M. (1996) The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations. http://www.santafe.edu/projects/swarm/

[33] Mitchell, Melanie. (1996) *An Introduction to Genetic Algorithms*. MIT Press.

[34] Molander, Per. (1985) The Optimal Level of Generosity in a Selfish, Uncertain Environment. Journal of Conflict Resolution. Vol. 29, No. 4, 611-618.

[35] Nardi, B. (1993) *A Small Matter of Programming*. MIT Press.

[36] Parnas, D. L. (1972) On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*. v. 15, #12, 1053-1058.

[37] Parnas, D.L. (1972) A Technique for Software Module Specifications with Examples. *Communications of the ACM*. v. 15, #5, 330-336.

[38] Papert, S. (1991) Situating Constructionism. I. Harel & S. Papert (eds) *Constructionism*. Ablex, 1-11.

[39] Repenning, A., Ambach, J. (1996) Participatory Theater: Interacting with Autonomous Tools for Creative Applications. *Journal of Knowledge Based Systems.*

[40] Repenning, A., Ambach, J. (1996) Visual AgenTalk: Anatomy of a Low Threshold, High Ceiling End User Programming Environment. Department of Computer Science, University of Colorado Tech Report # CU-CS-802-96, January.

[41] Repenning, A. and T. Sumner (1995) Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *Computer*, Vol. 28, pp. 17-25.

[42] Repenning, A. and T. Sumner (1994) Programming as Problem Solving: A Participatory Theater Approach. *Workshop on Advanced Visual Interfaces* 1994, Bari, Italy, pp. 182-191.

[43] Repenning, A. (1993) Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments. University of Colorado at Boulder, Ph.D. dissertation, Dept. of Computer Science, 171 Pages.

[44] Repenning, A. (1993) Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments. INTERCHI '93, Conference on Human Factors in Computing Systems, Amsterdam, NL, 1993, pp. 142-143.

[45] Resnick, M., Martin, F. Children and Artificial Life. I. Harel, S. Papert (eds.) *Constructionism.* Ablex, 379-390.

[46] Resnick, M. (1994) *Turtles, Termites, and Traffic Jams*. MIT Press.

[47] Resnick, M. (1996) New Paradigms for Computing, New Paradigms for Thinking. Y. Kafai & M. Resnick (eds.) *Constructionism in Practice.* Lawrence Erlbaum, 255-267.

[48] Smith, John Maynard. “Molecules are not Enough,” in *Did Darwin Get It Right? Essays on Games, Sex, and Evolution*.. Chapman & Hall, 1989.

[49] Steels, Luc. “The Artificial Life Roots of Artificial Intelligence,” in C. Langton (ed.) *Artificial Life IV*. Addison-Wesley, 1994, 75-110.

[50] Taylor, C.E., Jefferson, D.R., Turner, S., Goldman, S. (1989) RAM: Artificial Life for the Exploration of Complex Biological Systems. C. Langton (ed.) *Artificial Life.* Addison-Wesley.

[51] Turkle, S. (1995) *Life on the Screen.* Simon & Schuster.

[52] von Neumann, J. *Theory of Self Reproducing Automata*, edited and completed by A. W. Burks. University of Illinois Press, 1966.

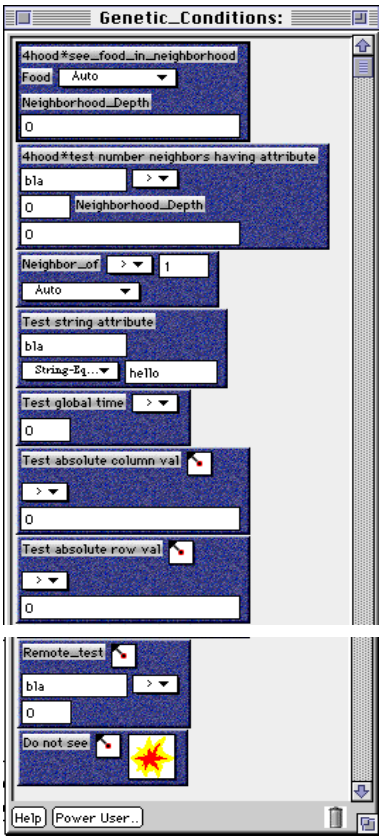
[53] Wimsatt, William C. “Forms of Aggregativity,” in A. Donagan, A.N. Perovich, Jr., M.V. Wedin (eds.) *Human Nature and Natural Knowledge*. Kluwer Academic Publishers, 1986, 259-291.

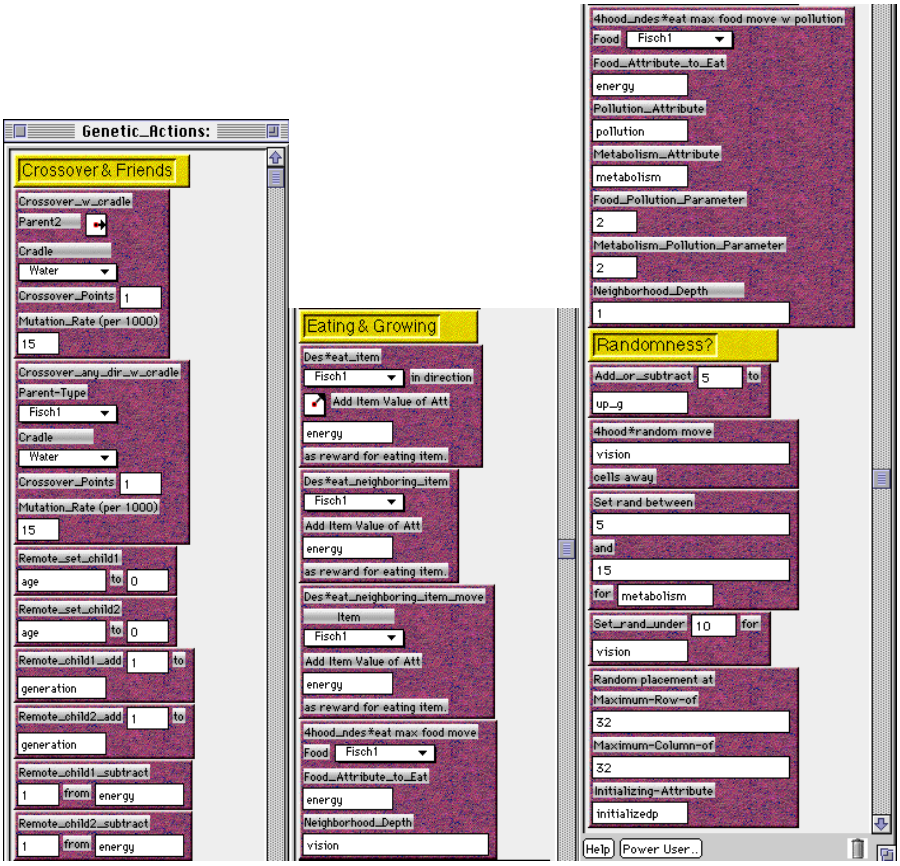
[54] Wolfram, S. “Universality and Complexity in Cellular Automata.” *Physica D* **10** (1984): 1-35.

[55] Yaeger, L. Computational Genetics, Physiology, Metabolism, Neural Systems, Learning, Vision, and Behavior or PolyWorld: Life in a New Context. C. Langton (ed.) *Artificial Life III.* Addison-Wesley, 263-298.

APPENDIX 1a

In viewing these commands one might notice the lack of symmetry between certain ones. For example, one might notice that although “4hood” versions of certain commands exist, “8hood” versions might not. When I originally created these commands, I thought such a naming scheme might be acceptable. Since then, as indicated in my “AS-VAT Programming Maxims” section, I have adopted a new and more flexible approach to defining aspects of such commands. (I am mainly referring to the **Maxim**: **Parameters** “Don’t wrap too many command decisions into a name. Make parameters of them.”) For this reason, I have not created “symmetric” commands in many cases. Because my maxims erupted at a fairly late date, I have also been unable to create new versions of these commands in accordance with the above maxim before the printing of this thesis. Though such changes will be complete at least immediately following, and perhaps before, my defense, they are technically left as future work.




Genetic_Actions:
Crossover & Friends
Crossover_w_cradle
Parent2
Cradle
Water
Crossover_Points
1
Mutation_Rate (per 1000)
15
Crossover_any_dir_w_cradle
Parent-Type
Fisch1
Cradle
Water
Crossover_Points
1
Mutation_Rate (per 1000)
15
Remote_set_child1
age
to
0
Remote_set_child2
age
to
0
Remote_child1_add
1
to
generation
Remote_child2_add
1
to
generation
Remote_child1_subtract
1
from
energy
Remote_child2_subtract
1
from
energy
Eating & Growing
Des*eat_item
Fisch1
in direction
Add Item Value of Att
energy
as reward for eating item.
Des*eat_neighboring_item
Fisch1
Add Item Value of Att
energy
as reward for eating item.
Des*eat_neighboring_item_move
Item
Fisch1
Add Item Value of Att
energy
as reward for eating item.
4hood_ndes*eat max food move
Food
Fisch1
Food_Attribute_to_Eat
energy
Neighborhood_Depth
vision
4hood_ndes*eat max food move w pollution
Food
Fisch1
Food_Attribute_to_Eat
energy
Pollution_Attribute
pollution
Metabolism_Attribute
metabolism
Food_Pollution_Parameter
2
Metabolism_Pollution_Parameter
2
Neighborhood_Depth
1
Randomness?
Add_or_subtract
5
to
up_g
4hood*random move
vision
cells away
Set rand between
5
and
15
for
metabolism
Set_rand_under
10
for
vision
Random placement at
Maximum-Row-of
32
Maximum-Column-of
32
Initializing-Attribute
initializedp
Help
Power User..

APPENDIX 1b

CROSSOVER OPERATOR

Here, the Crossover_Point is 3. Genes 1-3 are donated by one parent, and genes 4-7 are donated by the other parent.

Parent1

Eat_G	Up_G	Down_G	Left_G	Right_G	Horizevad_G	Vertevad_G
59	34	77	49	89	53	79

X

Parent2 X

Eat_G	Up_G	Down_G	Left_G	Right_G	Horizevad_G	Vertevad_G
71	95	44	19	64	53	29

from Parent1 from Parent2

Child1 <------- X ------->

Eat_G	Up_G	Down_G	Left_G	Right_G	Horizevad_G	Vertevad_G
59	34	77	19	64	53	29

from Parent2 from Parent1

Child2 <-------X------->

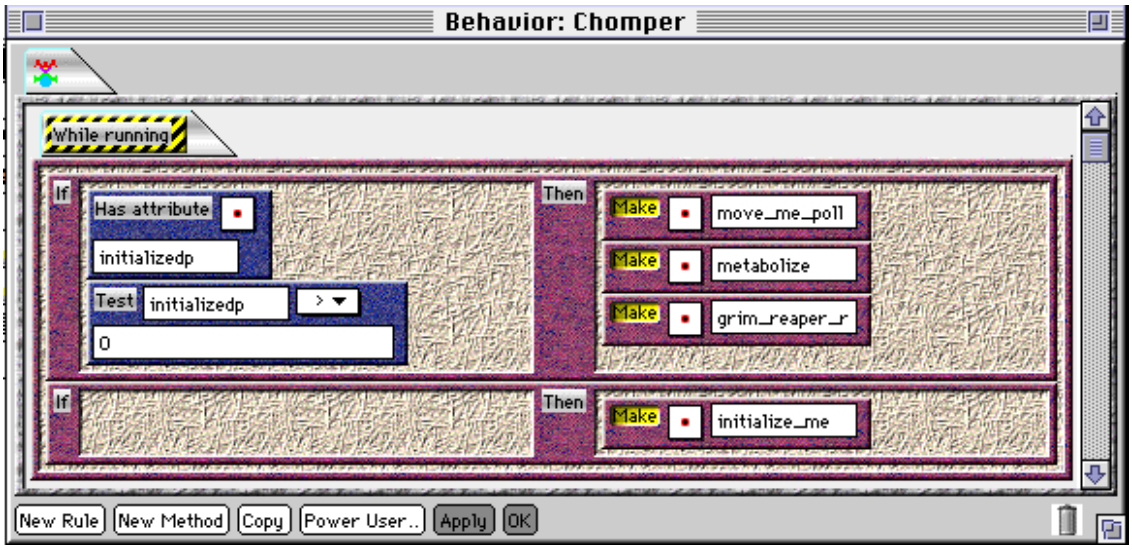
Eat_G	Up_G	Down_G	Left_G	Right_G	Horizevad_G	Vertevad_G
71	95	44	49	89	53	79

APPENDIX 2

The leftmost column shows the form to be entered into formula-boxes in AS-VAT.
The middle column names such formula types.
The rightmost column shows the equivalent infix LISP calls for the formula.

```
;;; Operators:
;;; NOTE: == is equality, = is assignment (C-style).
;;;
;;;  \     quoting character: x\-y --> x-y
;;;  !     lisp escape !(foo bar) --> (foo bar)
;;;  ;     comment
;;;  x = y    assignment      (setf x y)
;;;  x += y    increment      (incf x y)
;;;  x -= y    decrement      (decf x y)
;;;  x *= y    multiply and store    (setf x (* x y))
;;;  x /= y    divide and store    (setf x (/ x y))
;;;  x|y     bitwise logical inclusive or (logior x y)
;;;  x^y     bitwise logical exclusive or (logxor x y)
;;;  x&y     bitwise logical and   (logand x y)
;;;  x<<y    left shift      (ash x y)
;;;  x>>y    right shift     (ash x (- y))
;;;  ~x     ones complement (unary)  (lognot x)
;;;  x and y    conjunction     (and x y)
;;;  x && y    conjunction     (and x y)
;;;  x or y    disjunction     (or x y)
;;;  x || y    disjunction     (or x y)
;;;  not x    negation      (not x)
;;;  x^^y    exponentiation     (expt x y)
;;;  x,y     sequence      (progn x y)
;;;  (x,y)    sequence      (progn x y)
;;;       also parenthesis (x+y)/z --> (/ (+ x y) z)
;;;  f(x,y)    functions      (f x y)
;;;  a[i,j]    array reference    (aref a i j)
;;;  x+y x*y    arithmetic      (+ x y) (* x y)
;;;  x-y x/y    arithmetic      (- x y) (/ x y)
;;;  -y     value negation     (- y)
;;;  x % y    remainder      (mod x y)
;;;  x<y x>y    inequalities     (< x y) (> x y)
;;;  x <= y x >= y  inequalities     (<= x y) (>= x y)
;;;  x == y    equality      (= x y)
;;;  x != y    equality      (not (= x y))
;;;  if p then q   conditional     (when p q)
;;;  if p then q else r conditional     (if p q r)
```

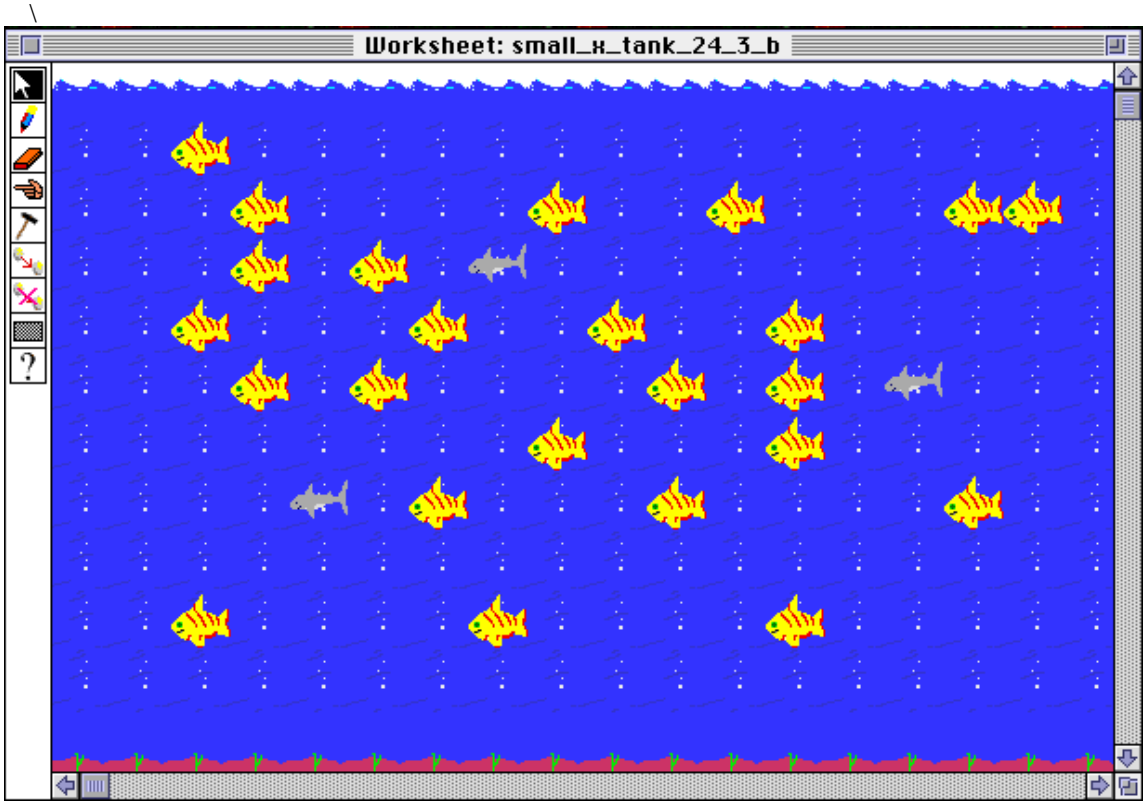
APPENDIX 3



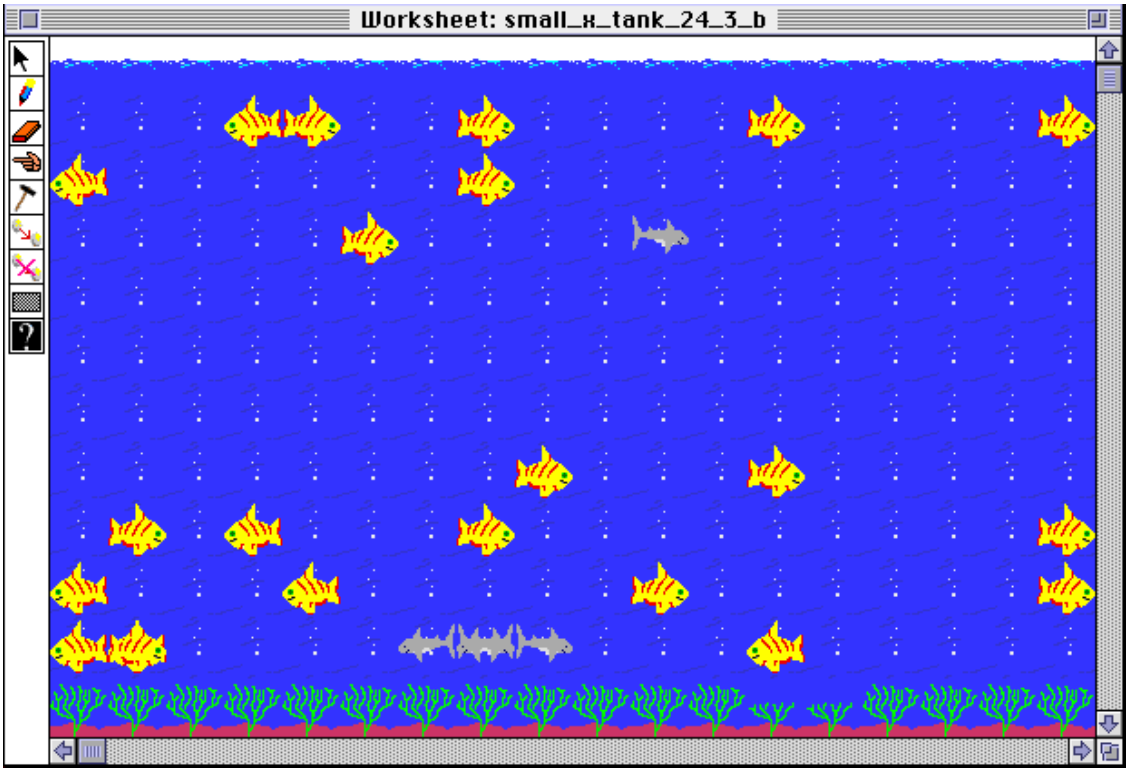
It is assumed that the initialize_me method will appropriately set all relevant attributes for its executing agent. Assuming that each agent's rule-editor is structured somewhat like the above rule-editor is the "least possible evil" when dealing with initializing newly-formed agents. This assumption allows users to determine which variables/attributes are relevant to their given simulation. (The Has_attribute condition is useful for initializing newly-created agents that might have no attributes. In such case one cannot initialize on the basis of a value of a given variable. The Test attribute condition is useful when reinitializing agents that already own at least the relevant initializing variable.) If this assumption is not made, in order to initialize newly-created agents, even more assumptions would have to be made about which variable/attributes are relevant for initialization purposes in a wide variety of sims. Assumptions such as these are obviously intolerable if a user is to retain much autonomy in programming the behavior of new AS-VAT agents.

APPENDIX 4

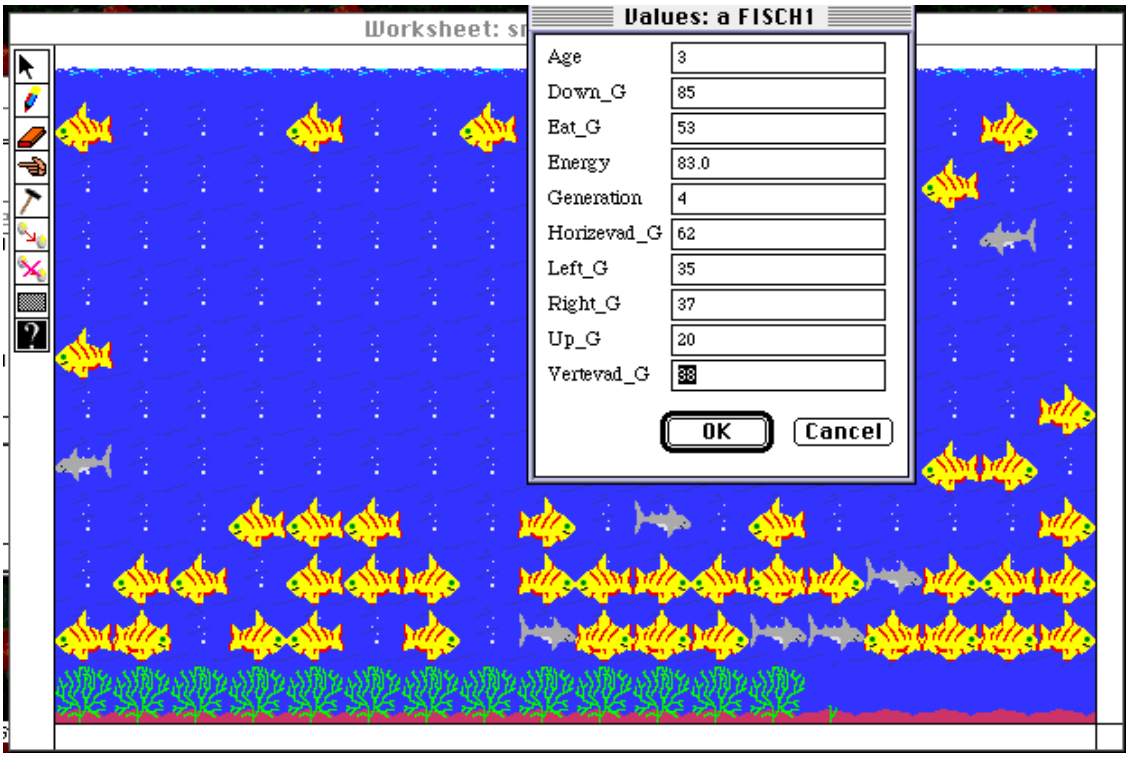
\



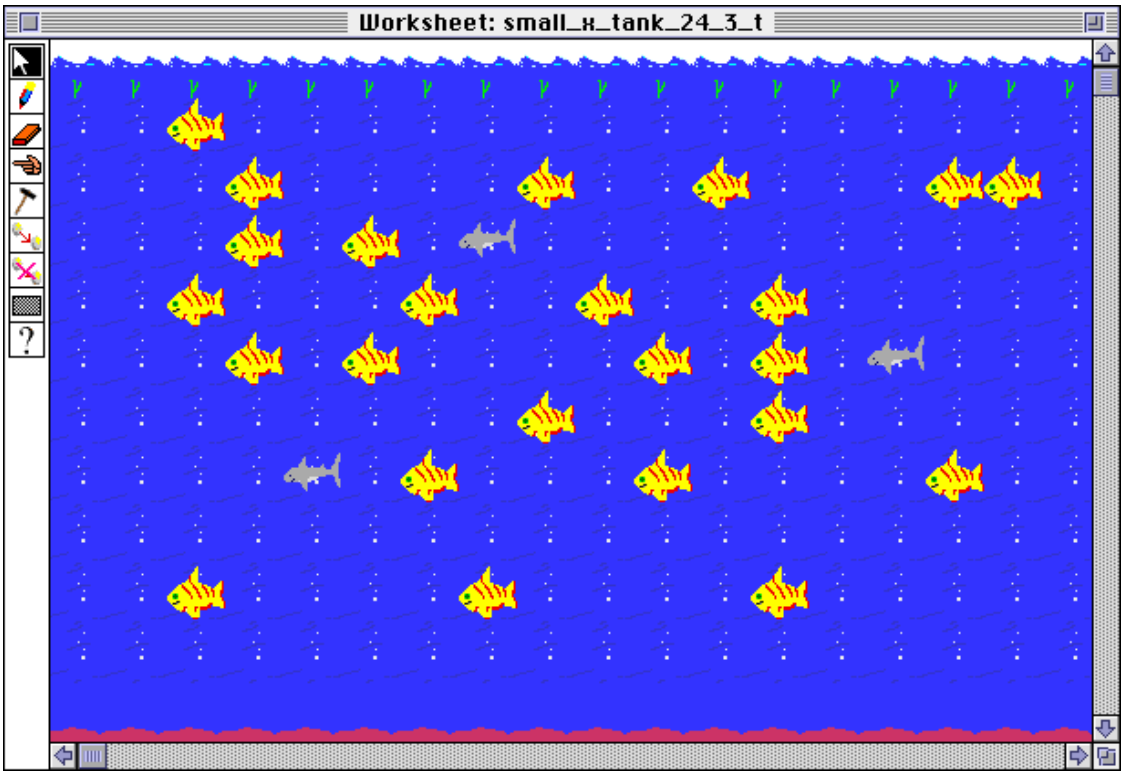
The above worksheet shows a fish tank with sharks, fish, and plants. Notice that the plants are located at the bottom of the tank. All fish and sharks start with randomized initial values.



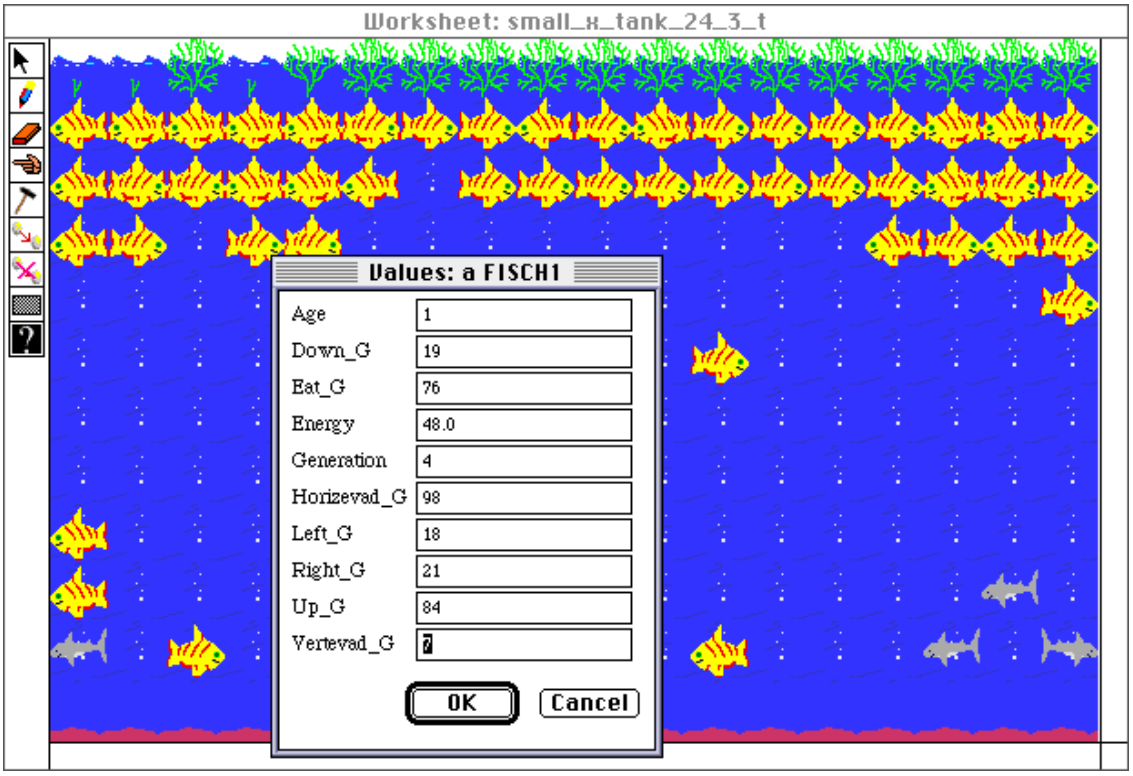
This screen snap shows the original tank after 19 time steps. Plants are growing well, and some fish have started to congregate at the bottom left corner of the screen. Still, no real groups have emerged. A few sharks are reproducing at the bottom center of the screen as well.



The above screen shows the original tank after about 60 time steps. Fish are definitely grouping at the bottom of the tank, where the food/plants are. The attribute window displayed is from a 4th generation fish at the bottom of the tank. Its Down_G value (85) and its Up_G value (30) have evolved to produce a tendency for this fish to swim down. In this environment such actions are advantageous as this fish will tend to stay near the food at the bottom of the tank. Groups of like fish emerge. They too have evolved to become bottom feeders.



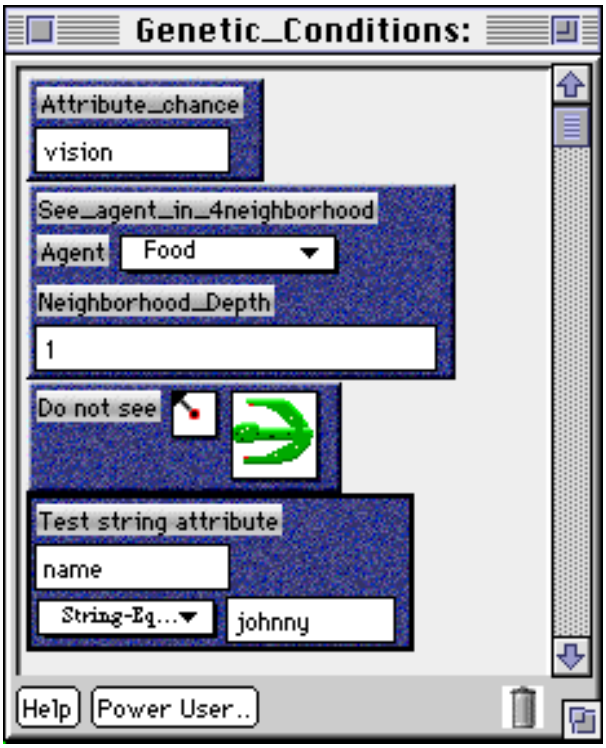
This tank is full of the same fish, sharks, and plants as the original tank above. These fish have the same initial attribute/gene values and the same initial positions as the agents in the original tank. The only difference between the two tanks is the location of the plants. Whereas plants are located at the bottom of the original tank, plants are located at the top of this tank.



This screen snap shows the previous tank after 60 time steps. Like the fish in the tank with bottom plants, these fish also congregate around the plants. But now the plants are located at the top of the tank. The same fish that evolved into a group of bottom feeders in the original tank evolve into a group of top feeders in this tank. Quick perusal of the genes of a 4th generation fish at the top of the tank show evolved gene values that make this fish tend to swim up toward the food at the top of the tank. Groups of fish emerge as surviving fish share this evolutionarily fit tendency.

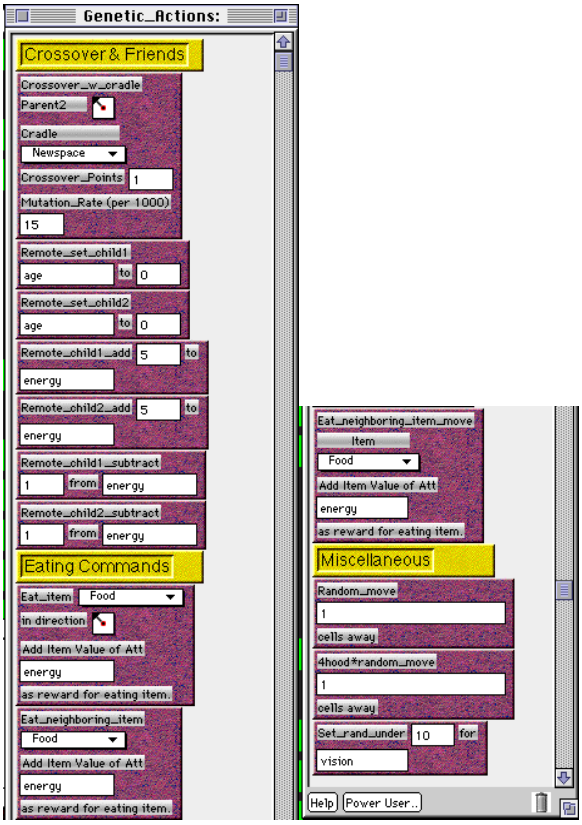
APPENDIX 5

KID_PACK: GENETIC CONDITIONS



This is the limited list of conditions I gave to the kids. I have also included the Attribute_chance command in this palette. This command was created by GPP for the original Genetic Fish Tank project.

KID_PACK: GENETIC ACTIONS



Above are the action commands I gave to the kids. I gave few commands with neighborhood-type tags in the title (e.g. 4hood), and I gave no commands with destructive/nondestructive tags built in to the title. Such conventions are somewhat confusing and ill-thought-out and will be changed in the future. All eating commands were destructive by default.