

Social Theory Building with the ETHOS Multi-Agent Simulation Framework: A Preliminary Evaluation

Jorge Simão¹ and Luís Moniz Pereira²

Centro de Inteligência Artificial – CENTRIA

Faculdade de Ciências e Tecnologia – Universidade Nova de Lisboa

2829 - 516 Caparica, Portugal

E-mail:¹ jsimao@di.fct.unl.pt; ² imp@di.fct.unl.pt

August 11, 2003

Keywords Social Theory, Human social behavior, agent-based modelling, MAS simulation framework

Abstract

We present a framework for a Multi-Agent System (MAS) devised to support the modeling and simulation of agent-based models of human social behavior and culture change. The framework takes a step further than current generic MAS by providing a set of features that help the modeler to shape the model structure. We describe the framework main abstractions, and make a preliminary evaluation of their usefulness and generality by looking at how several examples and models previously published from the literature have been re-implemented in it. We compare the framework with others and make the tentative conclusion the framework provides features that simplify the model design process for a wide range of models of social behavior, beyond what current MAS accomplish.

1 Introduction

Human social behavior and culture change are amongst the most complex phenomena studied by science. This results from the large number of interacting entities within a society, the different neuropsychological mechanisms underlying human social interaction, and the multiple channels of information inheritance, amongst other factors. All of these contribute to create non-trivial dynamics in inter-personal relationships, social structure, and collective beliefs and values, which require careful analysis. This has challenged many researchers to propose unified theoretical framework, towards a social science based on a naturalistic interpretation of human behavior and culture (Durham, 1990; Boyd & Richerson, 1985; Laland, Odling-Smee, & Feldman, 2000; Barkow, Cosmides, & Tooby, 1992; Sperber, 1996; Deacon, 1998; Donald, 1993; Mithen, 1996).

In spite of these efforts, there is an understanding that one still lacks a common theoretical language in which to naturally express a large range of models of social phenomena (Gilbert, 2000; Doran, 2000). The lack of shared language frustrates possible advances in scientific research insofar as it makes harder to compare conflicting models and model predictions, and puts barriers to the communication between overlapping research communities. Computational tools are a valuable avenue in this regard. Because computational tools require all abstraction to be formally specified (at least up to the programming language level), they need to make explicit what its ontological commitments are. By doing this, they establish the set of building blocks upon which models can be built. Thus providing a common set of abstraction that modeler can use and the community can share.

Nonwithstanding this targeted prospect, computational tools employed to support the modelling and simulation of systems with many interacting elements often lack features that permit capturing human specifics in a simple manner. For example, frameworks like SWARM, REPAST, and ASCAPE (Minar, Burkhart, Langton, & Askenazi, 1996; Collier, 2002; Brookings, 2000), while largely convergent on the set of tools provided, do not provide any specific flexible mechanisms to simplify the modelling of human social learning (e.g. observational learning (Bandura, 1985; Boyd & Richerson, 1985)), social relationships’ dynamics (Nowak & Vallacher, 1998), dissemination of values (Durham, 1990), emotional contagion (Doran, 2000), or non trivial behavioral control (Bryson, 2000a). This has the effect of making many interesting models hard to program, thus losing out on some of the advantages of using computational models as a complement to analytic mathematical ones. Namely, fast prototyping and analysis, and relaxation of assumptions made mostly for mathematical tractability (e.g. focusing on equilibrium states, and the use of infinite populations sizes).

To address these unique characteristics, we have been developing a new conceptual framework and an object-oriented JAVA implementation of a Multi-Agent System (MAS). This framework, ETHOS, takes the traditional approach of most other MAS a step further. While traditional frameworks provide computational abstractions that allow for the modeler to ignore implementation detail of features such as data visualization and event scheduling, ETHOS provides features that help to shape model structure. These include the provision of flexible behavior selection mechanisms, social influence through participation in shared activities, and management of agents’ social relationships. ETHOS also ensures a simple and yet flexible mechanism for event scheduling based on hierarchies of population objects. All these features, taken together, permit the social science researchers to start with model design employing higher-level building-blocks than otherwise possible with current MAS. This simplifies their work and speeds up the design development cycle. Moreover, students of social theory can be more easily inducted into their modelling activity by relying on abstractions made available to them from the start — an ideal consensual in the research community.

This article presents the ETHOS framework, and is organized as follows: In section 2, we describe ETHOS’s main abstractions in terms of the class structure implemented in the JAVA language¹. Next, in section 3, we present several application case studies, intended to test the usefulness and generality of the framework’s features. In section 4, we compare ETHOS’s features with those afforded by other MAS. We describe ongoing and future work and finally, in section 5, produce our conclusions.

2 Framework Overview

The ETHOS framework offers as basic building blocks the kind of entities the informed modeler is likely to consider when thinking intuitively about human social behavior and culture. This includes objects describing the structure and topology of physical spaces, physical entities placed in this space (such as resources and agents with varying attributes and genetic makeups), distinct kinds of social relationships amongst agents, behavior selection mechanisms, mechanisms for the social influencing of agents’ mental states, defined contexts of individual action and social interaction, beside others. In the sequel we describe all such abstractions in greater detail. In figure 1, we depict the key abstractions of ETHOS’s and how they relate to each other. The text labels in figure 1, and the highlighted words in the presentation below correspond to object classes in the framework. The relationships between the main classes of ETHOS’s meta-model in terms of inheritance, aggregation, and acquaintance is shown in fig. 2. Due to the wide variety and expressiveness of Ethos code abstractions, specific models may of course use only a subset of all those made available.

¹Elsewhere we described the features of an earlier version of ETHOS (Simão & Pereira, 2003). The specification in this paper should be taken to supersede the earlier version.

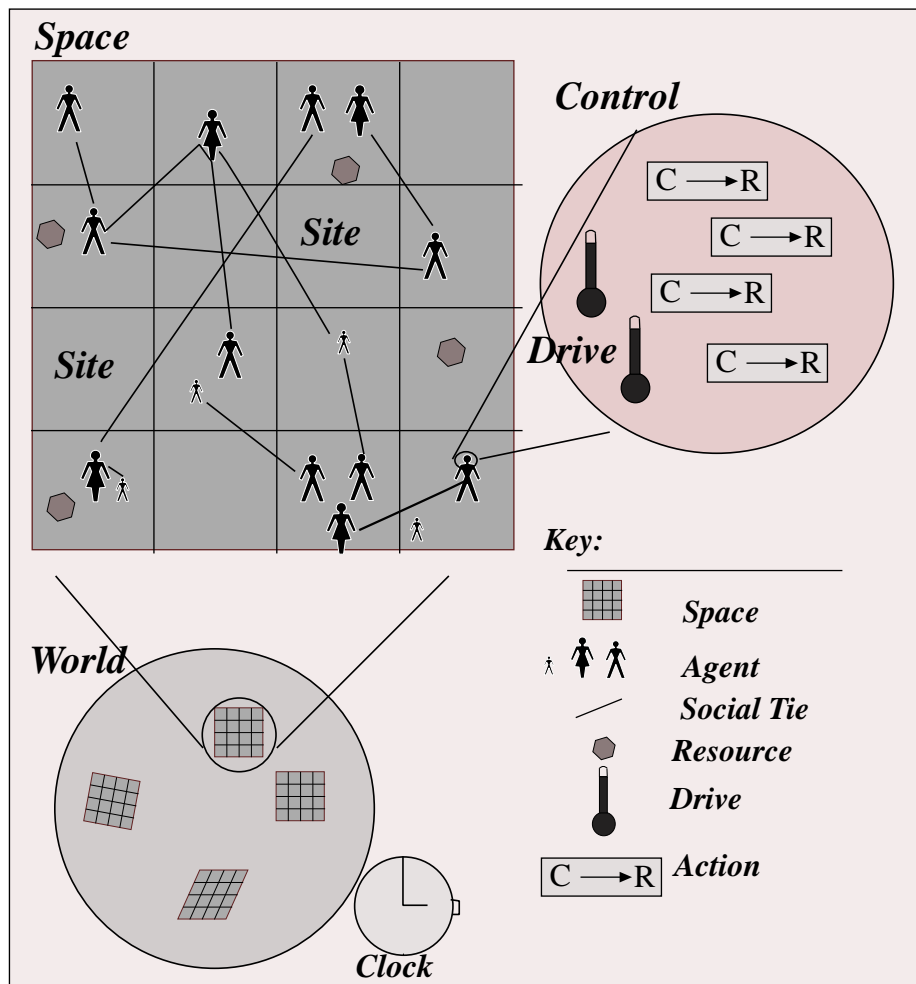


Figure 1: ETHOS's Framework Main Abstractions

2.1 Modelling the Physical and Social Environment

The top level abstraction of ETHOS is a World object containing a set of one or more physical spaces (Space object), whose events are timed by a common clock. Each such Space object consists of a topological arrangement of Site objects, each of which is a place-holder for a set of physical bodies (Body objects). A Space object defines its own geometry for the arrangement of Sites, and provides generic services to navigate between Sites. For example, finding the list of neighboring sites according to some neighborhood type (e.g. Moore or von Neumann), at a specified metric distance (e.g. Euclidean or Block), and contingent in the underlying Space topology (e.g. 2D torus or frame). Subclasses of Space currently implemented are: GridSpace that implements a 2D grid world, ListSpace for a one-dimensional Space object (possibly) with a dynamic number of sites, and VoidSpace which* has a single site where Body objects are positioned (see below).

The Body objects living in sites may be agents (Agent object), or other physical entities such as growing/consumable resources (Resource objects). Agents are usually made to move from site to site during a simulation, while other physical Bodies usually have a fixed site location. In addition to having a spatial location by virtue of being contained in a Site, Body objects also have (optionally) a position inside the site they currently are stationed in. Body objects also contain generic attributes, such as age, that are used in a large number of modelling scenarios. In general, a Body object provides the basic specification under which subclasses such as Agent objects can be defined. Agent objects themselves proffer additional commonly used attributes, such as sex and a one-dimensional quality label.

In addition to physical (site) neighbor relationships, agents may establish social relationships with other agents. Specifically, each agent maintains a list of social networks (**SocialNet** objects), each intended to correspond to a different relationship type (e.g. parent-offspring, acquaintance, sexual, etc.). Each specific agent-to-agent relationship is coded as a **Tie** object, that holds information about a relationship, such as the agents involved, its intensity, and its age. Parent-offspring relationships are created and managed automatically by **Ethos** during agent creation, while other types of relationship are defined and managed by the simulation model.

Different criteria can be set forth to specify which agents are added/removed from a particular social network of some agent. Agent selection in relationship management, or object selection in general, is facilitated by using **Selector** objects. **Selector** objects can be employed, with modeler specified criteria (implementations of interface **Criteria**), to implement a selection of some type of a subset of objects from a larger set (e.g. roulette-wheel, or tournament). Selection modes can be non-competitive, where objects are selected by a local criteria, or competitive, where agents are selected by rank.

Agents can have finite life-span, and may be dynamically created and eliminated during a simulation run. Agents have a genetic makeup (**Genome** object) which is inherited from one or two parents. Each agent's genome contains a list of genes, whose number, data type, and initial value distributions (when not inherited) are selected by the modeler. The details of the genetic system, such as type of crossover and mutation intensities can be selected from the ones available or defined by the model designer (e.g. the top level **Genome** class implements a multi-point crossover, while the subclass **Genome1PCX** implements one-point crossover (Goldberg, 1989)). The interpretation of gene's values is left to the modeler, but it is conceivable that in the future we will include genes interpreted by **ETHOS**'s runtime system (e.g. properties that modulate agents' behavior). This is complementary to agents' offspring being able to inherit behavior control information from their parents.

Agents live usually only in one space throughout their life-span, although they can migrate on demand between different spaces in the world. If an agent migrates to a different space in the world, all its current relationships are deleted. This simplifies the possible distribution of a simulation by putting different **Space** objects running in different address spaces and different machines. Because of this, the scheduling order of events between different **Space** of a **World** is undefined. Because most models use only one **Space** object we defer discussion of distribution issues .

2.2 Event Management and Population Structures

ETHOS uses a simple yet flexible discrete time step scheme to trigger events. **Population** objects are used to aggregate agents and other bodies into collective units, whose event dispatching is coordinated. Each **Population** object relays control to each member **Body** object — by calling some well defined method — according to a set scheduling policy. The policy specifies several things: whether the iterative sequence of members at each time step should be made random or fixed; the number of phases a simulation step has; and whether dispatching of events is asynchronous or synchronous when more than one phase is involved.

Population objects are also used to code population level operations on bodies. **Population** subclasses refine on the base scheduling policy and services provided. **AgentPopulation** is a subclass specific for **Agent** member objects. In addition to the inherited scheduling policies, **AgentPopulation** can be set to allow agent invocation to continue in the same simulation time step until all member **Agents** have run out of free time. This works in conjunction with the notification of time usage as agents perform actions and use up time.

Population objects are made to subclass **Body**. This allows for population objects to be arbitrarily composed in tree- or graph-like structures ². Event dispatching occurs in "depth-first order, has control passed to a **Population** always dispatches to member elements, and is not aware of super-ordinate **Population** other than the parent. By default, a sub-population inherits the scheduling policy of its parent **Population**.

²This corresponds to the implementation of a *Composite* object design pattern (Gamma, Helm, Johnson, & Vlissides, 1995).

Each Space object has an associated top-level population that is automatically created in the space initialization. This top population is used to add other Population (or Body) thus structuring the event scheduling order. Usually, the scheduling is static and implicit in the Population structure created. However, dynamics schemes are also possible by modification of Population member during a simulation. Scheduling of events at an arbitrary number of time steps can also be incorporated into ETHOS in the future if it proves useful (similar to that found in other MAS, such as in SWARM and Repast).

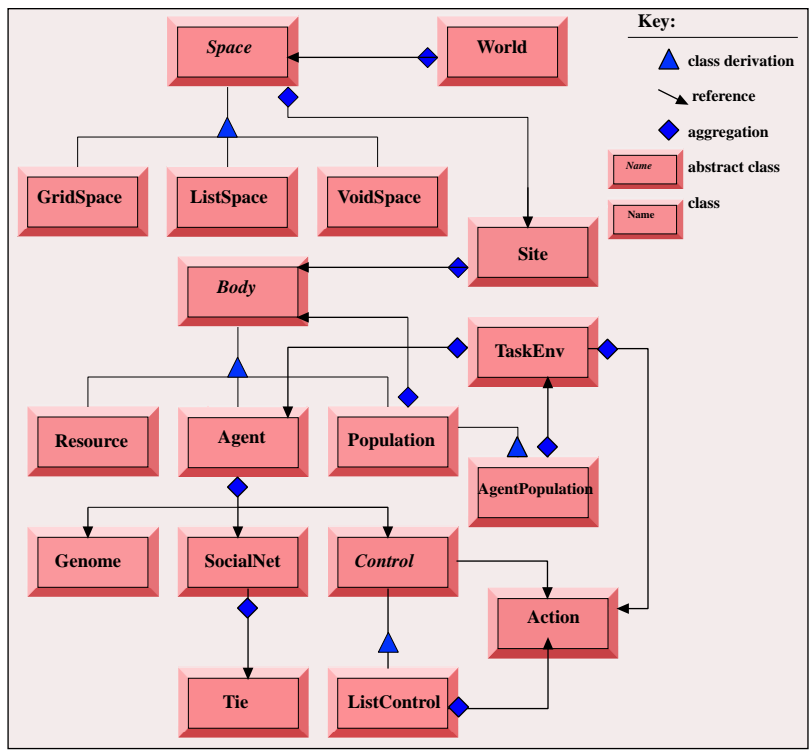


Figure 2: ETHOS's Class Hierarchy

2.3 Agents Behavior Control and Task-Environments

Agents can select what action to perform within a simulation time step using a Control object. A Control object maintains all the information that pertains to the Agent's mental state. It is used to decide what action to perform when faced with a context for action, and it's updated based on the outcome of actions. Action contexts are abstracted using TaskEnv objects. They correspond to opportunities in physical or social contexts for individual and collective action (Reed, 1996). Each TaskEnv has associated an arbitrary (user defined) context identifier that empowers the agent to discriminate between different TaskEnv. During each simulation time step a TaskEnv holds a set of Agent objects, possibly with a role identifier, whose result of interaction is computed. Thus, collective action is represented by a TaskEnv that contains more than one Agent. Sub-classes of TaskEnv can expand the basic services to match specific scenarios. Including: cost-benefit analysis, resource transfer, observation of others attributes and modification of self attributes, etc.

Agents' individual actions are obtained by making a call to the Control of the Agent. This delivers an Action object coding an appropriate response. Once all agents have decided on their actions, they are evaluated by the TaskEnv and the payoffs are notified to participating Agent. This prompts a call to the Control objects for state update. The state update is usually contingent on each individual receiving a payoff (rewards - punishments), but social learning is also supported by looking at information of related other agents participating in the TaskEnv (see below). By convention, the role of AgentPopulation subclasses is to define and create TaskEnv and assign them to member agents. The workings of a Control are made transparent to

other objects, by having a Agent providing the commonly used operations and deferring the execution of this to the Control ³.

Control class is abstract, and is used indirectly as the modeler instantiates one of its subclasses. At the present time we support a ListControl subclass that maintains a list of actions of bounded size. When a ListControl is requested to select an action for a TaskEnv, it tries to match one of the Action objects stored with the TaskEnv. By default, Action subclasses match to a TaskEnv if the (perceptual) context identifier of each is the same, but modeler specified subclasses can override this. If a match in ListControl is found, there is a non-zero probability the response of the action be randomly mutated. This implements a simple exploration mechanism, similar to evolutionary strategies (Back, January 1996; Beyer, 2001). If no match is found, a new action is created derived from an existing one. If the maximum size of the action list is reached, one is selected for removal. Actions whose execution lead to smaller payoffs for the agent are more likely to be removed. In addition to ListControl, we are also planning to provide a neural network based controller that learns by association and reinforcement in the form of a Control subclass.

Control objects also provide several methods to mimic different types of social influence. The method updateByPriming(.) is intended to model the simple types of social influence (Heyes & Bennett G. Galef, 1996). The method updateByObservation(.) is intended to model learning by using others' payoff to update an agent's own control (Bandura, 1977, 1985). In ListControl this corresponds to finding a stored action that matches that performed by another agent and changing its valuation. Finally, updateByImitation(.) is used to model social learning which abstracts how behavior responses are passed from agent to agent (Boyd & Richerson, 1985). The specifics of the semantics of each of these methods is to be defined by subclasses of Control. They are defined for the purpose of structuring the task of modelling social learning.

2.4 Other Features

Similar to most other MAS frameworks for agent-based modelling, ETHOS provides miscellaneous features in a "ready to use" Graphic User Interface (GUI). These include: the control of a simulation execution, the visualization of the simulation state, the gathering and exporting of statistics using several types of data objects such as bidders and time-series, dynamic parameter setting, amongst other. In figure 3 we present a screen shot of the GUI.

In the lower left hand side, a table of parameters is shown. Parameters can be grouped into logical or conceptually related sets and are displayed according to this grouping. In the lower right hand side, two viewers are showed, one representing a data plot and another a graphical view of a GridSpace object. In this default GUI object, viewers are all showed as internal frames of a desktop area.

At the top of the figure 3, are the controls for setting up, starting, stopping, and stepping a simulation. These are very similar in functionality to those available on the GUI of other MAS frameworks, like REPAST and ASCAPE. The rightmost button allows data objects to be exported to data files, so they can be plotted and analyzed with more specialized tools. Below the control buttons, on the right, a set of progress bars indicate the percentage of the specified number steps, and runs a simulation has advanced. And to the left of these, a slide bar allows the refresh time of viewers to be set dynamically. This is the time interval between which viewers are updated to reflect their underlying observed object⁴. Also in this area, through a spinner, a time of delay can be imposed upon the simulation progress to perform slow motion execution of the simulation.

In the bottom part of figure 3 the parameter setting panels are show on the right, and several viewer objects are shown on the left. Since these features do not differ significantly from the ones available in other MAS for agent-based modelling, we do not discuss them in greater detail here.

³This corresponds to the implementation of a *Facade* object design pattern (Gamma et al., 1995).

⁴A variation of the *viewer-observer* object design pattern is used here (Gamma et al., 1995).

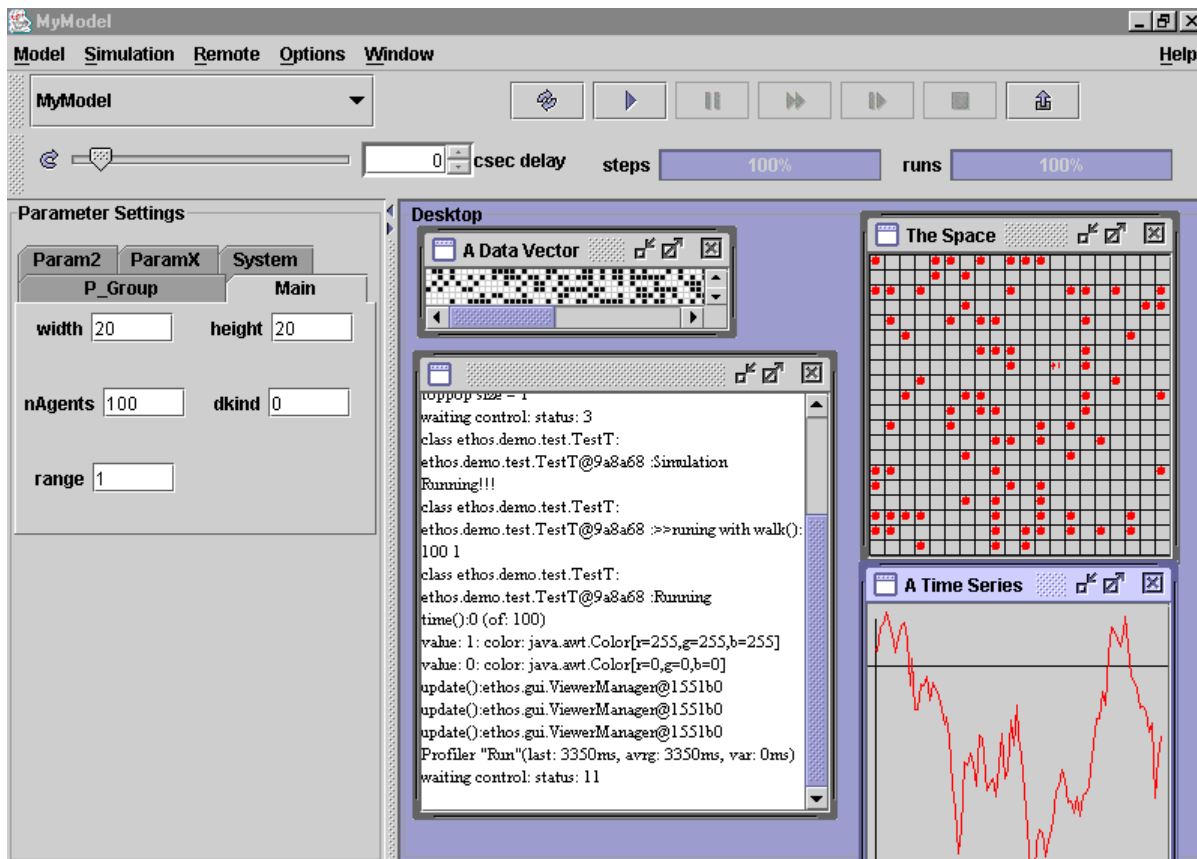


Figure 3: ETHOS's GUI look-and-feel.

3 Demo Application Case Studies

To test the usefulness and generality of our framework, we have re-implemented several agent-based models from the literature with some form of non-trivial social interaction. Of particular interest are models of gene-culture dual inheritance, and inter-personal social dynamics, since they constitute the main application target of ETHOS. We have also implemented new models inspired by the theoretical leverage provided by the ETHOS meta-model. In the following sections, we describe how we implemented two such models, summarizing first the structure of the models, followed by a description of how the abstractions of our MAS framework were employed. However, to increase the reader's acquaintance with ETHOS, we first present two toy models with code snippets that exemplify how the overall architecture is used. Important to have in mind in what follows, is that our goal in presenting the different models is not to make a scientific evaluation or replication of them. Rather, the purpose it to see how the different models map on ethos basic building blocks.

3.1 A Toy World

In the code below, a model top class derives from a World object and overrides a few methods to code model specific functionality. A typical situation is for a model to define a few model parameters that can be manipulated by the GUI user. In the constructor of the top class, we show how this is achieved in Ethos idiom. A regular expression is used to "catch public member fields in specified objects by calling Param.createPars(.). These are them grouped to appear together at the GUI.

```
import ethos.model.*; ...
```

```

public class Toy1 extends World {
    //model parameters
    public int width = 20;
    public int height = 20;
    public int nAgents = 100;

    public Toy1() {
        super("Toy_Model_1");
        addAllParams(Param.createPars(this , "[a-z]*"));
        groupUngroupedParams("Main");
    }
}

```

Still in the top class, the method `setup()` should be overridden to perform all the model's initialization code. In the code below, we start by removing all possible objects from a previous execution. Next, we create a `GridSpace` object that is added to your toy world as its single space, and an `AgentPopulation` object that is added to that space as a sub-population of the top-level population. We then define an `Agent` subclass — the only `Body` objects with your toy world. `Agent` objects put themselves at a random site in the world, and define their basic behavior as moving to a neighbor site at each simulation time step. Finally, a set of such agents is created and added to the created `AgentPopulation`.

```

public void setup() {
    clearAll();
    addSpace(new GridSpace(width, height, true));
    AgentPopulation pop = new AgentPopulation();
    getSpace().getTopPop().addMember(pop);

    class MyAgent extends Agent {
        final List allSites = getSpace().getAllSites();

        MyAgent() {
            setSite((Site) allSites.get(Global.getRandom().nextInt(allSites.size())));
        }

        public void act() {
            List ss = getSpace().getAllNeighbors(getSite(), range, dkind);
            Site s = (Site) ss.get(Global.getRandom().nextInt(ss.size()));
            setSite(s);
        }
    }
    for (int i = 0; i < nAgents; i++) {
        pop.addMember(new MyAgent());
    } ....
}

```

Still in the method `setup()`, we create two viewers. One to visualize the `GridSpace` object created, used to watch agents navigating on the GUI screen. The second viewer is utilized to visualize a non-sense data time series. The viewers are added to the toy world so they can be visualized at the GUI. The data object is added to the toy world so it can be exported to external programs.

```

GridSpaceViewer gsp = new GridSpaceViewer(getSpace());
addViewer(new Viewer("The_Space", gsp, getSpace()));
ts = new TimeSeries("ats");
addDataObject(ts);
Plot plt = new Plot(ts, Plot.LINES);
addViewer(new Viewer("A_Time_Series", plt, ts));
dv = new DataVector("adv");

```

```
    addDataObject(dv);  
}
```

Finally, in the `main()` method the model is started by creating an instance of your model class. The `start()` starts the GUI, that in turn controls the program's main thread. The GUI then decides when to call `setup()` and when to run a simulation for a certain number of time steps, and runs (repetitions), and for which specified parameter settings.

```
public static void main(String [] args) throws IOException {  
    Toy1 model = new Toy1();  
    model.start(args);  
}
```

3.2 Another Toy World: A Simple Cooperation Game

In the following, we describe how to implement a simple cooperation game. In our game agents are grouped at each time step with a subset of other agents, in order to play a cooperation game and to learn from the resulting payoffs. The actual number of agents playing a game is set in a `TaskEnv` object. If it is 1 then the agent's behavior target is to learn to select the appropriate response for the `TaskEnv`. If it is a positive number, *then the focus of the agent is to decide which partners he wants to play the game with. The partners are selected based on the strength of the social tie that the agent has with them.

Below we show the code for a subclass of `SimpleTaskEnv`, which in turn is a subclass of `TaskEnv`. `SimpleTaskEnv` is useful when a single (ideal) response is to be associated with a `TaskEnv` context. The context here is coded as a `Long` value. The `setupPayoffs()` method is used to setup the benefits and costs of agents participating in the game. Here, R is a parameter for the reward and I a parameter for the cost or investment. If the action chosen by an agent is equal to the response associated with the `TaskEnv` *then that is considered as a cooperation act and costs I . If the response differs the cost is 0. The benefits from each player action take a value proportional to the number of cooperators (Axelrod, 1985, 1997; Boyd & Richerson, 1985).

```
class MyTaskEnv extends SingleTaskEnv {  
    public MyTaskEnv(long c, long r, int n) {  
        super(new Long(c), new Long(r), n);  
    }  
  
    public void setupPayoffs(List ags) {  
        int coop = 0;  
        for (Iterator i = ags.iterator(); i.hasNext(); ) {  
            Agent ag = (Agent) i.next();  
            LongAction a = (LongAction) ag.getLastAction();  
            if (a.getResponseID() == getResponseID()) {  
                coop++;  
                ag.setCost(I);  
            } else {  
                ag.setCost(0);  
            }  
        }  
        for (Iterator i = ags.iterator(); i.hasNext(); ) {  
            Agent ag = (Agent) i.next();  
            ag.setBenefit(R*coop/getNAgents());  
        }  
    }  
}
```

The method `perform()`, codes for the sequence of events performed at each collective action. Next is a very typical sequence: First, the actions chosen by each agent set in the `TaskEnv` are computed with `setupActions()`; Then with method `setupPayoffs()` the payoffs to each agent are computed and memorized; Next, it is noted that agents have used all the time available in the current time step (so they do not perform more than a single action per time step); Finally, the strength of the social ties is updated based on the payoff obtained. As discussed next, this makes agents interact more with agents they had positive experiences with — here modelled as payoffs.

```

public void perform() {
    setupActions();
    setupPayoffs();
    updateByPayoff();
    useTime();
    updateAllTiesStrength(0);
}
}

```

Below, we define an `Agent` subclass for the agents in our toy model. In the constructor method, the type of control for the agent to use is set. We use a `ListControl` with a maximum list size set to `NA`. After some more parameter settings of the `ListControl` object, a single initial action is added to the control. This allows the `ListControl` to derive other actions from this first one by mutation. Next a social network is created and added to the list of the agent social networks.

```

class MyAgent extends Agent {
    public MyAgent() {
        setControl(new ListControl(NA));
        getControl().setLearningRate(A);
        ((ListControl) getControl()).setMutationRate(PM);
        LongAction a = new LongAction(0, 0, 0.1);
        a.setValidBits(1);
        ((ListControl) getControl()).addAction(a);
        addSocialNet(new SocialNet(this, 0));
        setQuality(Global.getRandom().nextGaussian() * 10);
    }...
}

```

In the method `act()`, we code what each agent does. First, a set of partners is selected using a `Selector` object by invoking a method that performs a probabilistic selection of a certain number of agents. The class `BondCriteria` implements the `Criteria` interface, and defines the probability of an agent to be selected as a partner by another agent — which defines the context of selection. This probability is the score obtained by calling method `psel()` in `MyAgent` class. The score is 0 if the agent already performed an action in the current time step, or the strength of the bond with the other agent otherwise.

```

public void act(TaskEnv te) {
    List partners = Selector.selectXDups(pop.getAllMembers(), BondCriteria.crit,
        te, getThisList(), te.getNAgents() - 1);
    if (partners.size() < te.getNAgents() - 1) {
        return;
    }
    te.clearAgents();
    te.addAgent(this);
    te.addAllAgents(partners);
    te.perform();
}

public double psel(Agent ag) {
    if (!hasFreeTime()) {
        return 0;
    }
}

```

```

    }
    return getSocialNet(0).getTieStrength(ag);
}
}

static class BondCriteria implements Criteria {
    static BondCriteria crit = new BondCriteria();

    public double score(Object obj, Object ctx) {
        MyAgent ag1 = (MyAgent) obj;
        MyAgent ag2 = (MyAgent) ctx;
        return ag1.psel(ag2);
    }
}

```

Finally, an `AgentPopulation` subclass is defined to assign `TaskEnv` to agents. Here a random `TaskEnv` is selected from the ones created in the constructor of the `AgentPopulation` subclass. This is performed in the `actOne()` which is invoked for every agent that has free time. In the constructor the scheduling policy is also set, and the agents are created.

```

class MyAgentPopulation extends AgentPopulation {
    MyAgentPopulation() {
        setSchedulingTimeUsage(Clock.SCHD_FREETIME);
        for (int i = 0; i < nAgents; i++) {
            addMember(new MyAgent());
        }
        for (int i = 0; i < nTaskEnv; i++) {
            addTaskEnv(new MyTaskEnv(i, i, i + 1));
        }
    }

    protected void actOne(Agent ag) {
        TaskEnv te = getTaskEnv(Global.getRandom().nextInt(getAllTaskEnvs().size()));
        ag.act(te);
    }
}

```

A point that should draw the reader's attention at this stage, is that the code for this simple cooperation is evenly distributed between the defined `MyTaskEnv`, `MyAgent`, and `MyAgentPopulation` subclasses. As a rule of thumb: `Population` subclasses should code operations (or data) relative to a large population of agents; `TaskEnv` subclasses should code for operations relative to a temporary assemblies of agents to perform a collective action (usually for the duration of single time step); `Agent` subclasses should code operations specific to a single agent. Although this may seem unnecessarily cumbersome, it allows the different model parts to be logically separated and respective of (or suggestive of) the ontology that the modeler has in mind. For simple models the advantages from following this practice might not be too substantial, but for more complex models the gains in ease of design and clarity may be substantial. Moreover, specific models may not need to subclass all three classes.

3.3 Higgs's Mimetic Transition

Paul G. Higgs's mimetic transition model is an agent-based model of gene-meme co-evolution, and was designed to study the conditions under which the capacity for learning memes can evolve, even if there is no mechanism to distinguish between memes that increase biological fitness and those that decrease it instead (Higgs, 2000). The model uses non-overlapping generations of agents which have two biological parents and

a set of cultural parents from which they learn memes, and do so with a probability proportional to their learning ability. Agents are able also, with some small probability, to invent new memes. Higgs's results show that, for a wide range of parameter values, the capacity for learning evolves in a phase-transition, where the capacity for learning evolves initially very slowly, but, after a critical point, increases very fast. Although we have several theoretical reservations about Higgs's model, we consider useful, for illustrative purposes, to show here how such model can be implemented in ETHOS (see (Aunger, 2000) for general reservations on the use of the concept of *meme* in modelling human social behavior and culture).

We model this by defining an `Agent` subclass implementing most of the model code. Following Higgs's specification, each agent is defined to comprise a learning ability, and both a biological and a cultural fitness value. The constructor with two parameter specifies how an agent obtains its learning ability from two parent agents. The list of memes an agent knows about is stored as a list of attributes.

```

class MyAgent extends Agent {
    double biofit = 0;
    double cultfit = 0;
    double l = L0;

    MyAgent() {}

    MyAgent(MyAgent par1, MyAgent par2) {
        super(par1, par2);
        double l = (par1.l + par2.l) / 2;
        if (Global.getRandom().nextDouble() <= U) {
            l = l + (Global.getRandom().nextGaussian()*SD_DL + DL);
        }
    } ...
}

```

In the method `act()`, which is called at every simulation time step by the containing `Population`, the list of cultural teachers for an agent is created. First, its biological parents are added to the list. Next, a set of K randomly selected members using the roulette-wheel method of the `Selector` object is added to the list of teachers. The method `learnFromTeachers()` performs a two part nested iteration over all agents (the teachers) and all memes, to garner which memes are learnt by the agent. This is followed by a probabilistic check to watch if the agent is able to invent a new meme. Finally, the agent's biological and cultural fitness is computed based on the fitness of the set of all memes known to it.

The memes proper, are trivially defined in a class with two members — the cultural and biological contribution of the meme (not shown). The values are set randomly according to the model's specification.

```

public void act() {
    if (getParent1() == null) { //first generation
        return;
    }
    Population pop = getParent1().getPopulation();
    List teachers = new ArrayList();
    teachers.add(getParent1());
    teachers.add(getParent2());
    List teachers2 = Selector.selectXDups(pop.getAllMembers(),
        CulturalFitnessCriteria.crit, null, null, K);
    teachers.addAll(teachers2);
    learnFromTeachers(teachers);
    if (Global.getRandom().nextDouble() <= PINV) {
        addAttr(new Meme());
    }
    computeFitness();
}
}

```

Two criteria are defined: one for agent's cultural fitness and another for the agent's biological fitness. This is coded below, very similarly to the code presented in the previous cooperation game example.

```
static class CulturalFitnessCriteria implements Criteria {
    static CulturalFitnessCriteria crit = new CulturalFitnessCriteria();

    public double score(Object obj, Object ctx) {
        MyAgent ag = (MyAgent) obj;
        return ag.cultfit;
    }
}

static class BioFitnessCriteria implements Criteria {
    static BioFitnessCriteria crit = new BioFitnessCriteria();

    public double score(Object obj, Object ctx) {
        MyAgent ag = (MyAgent) obj;
        return ag.biofit;
    }
}
```

An AgentPopulation subclass is defined to perform the inter-generational step and to gather statistics. Below, we see that the BioFitnessCriteria criteria is used to select $2 \times N$ agents to be the parents of the next generation. After they are so selected they are removed from the population, and the offspring is created to replace them using the two argument constructor of MyAgent.

```
class MyAgentPopulation extends AgentPopulation {
    MyAgentPopulation() {
        for (int i = 0; i < N; i++) {
            addMember(new MyAgent());
        }
    }

    public void act() {
        super.act();
        getStats();
        List ags = Selector.select(getAllMembers(),
            BioFitnessCriteria.crit, null, null, 2*N);
        removeAllMembers();
        for (int i = 0; i < N; i++) {
            Agent ag = new MyAgent((MyAgent) ags.get(i*2), (MyAgent) ags.get(i*2 + 1));
            addMember(ag);
        }
    }
}
```

The process of gathering statistics is straightforward. Below, three statistical variables are employed to compute the mean of three different statistics. Following the Higgs paper: the mean biological fitness of the population, the mean number of memes learnt, and the mean learning ability. Once these three values are computed they are added to three TimeSeries objects, that can be exported for plotting in another program or displayed on a viewer in the ETHOS GUI.

```
public void getStats() {
    Var vfit = new Var();
    Var vnmemes = new Var();
    Var vlabil = new Var();
    for (int i = 0; i < N; i++) {
        vfit.add(((MyAgent) getMember(i)).biofit);
    }
}
```

```

        vnmemes.add(((MyAgent) getMember(i)). getAllAttrs(). size());
        vlabil.add(((MyAgent) getMember(i)). l);
    }
    fit.add(vfit.getValue());
    nmemes.add(vfit.getValue());
    labil.add(vfit.getValue());
}
}

```

Although Higgs’s model is a relatively simple one, its implementation is far from trivial. The use of ETHOS abstractions allows the model to be expressed in a natural form (together with the basic JAVA language infrastructure). By using well tested code such as `Selector` and `AgentPopulation` objects, and a pre-set domain ontology for agents and populations of agents, the solution to the implementation unfolds in a straightforward way.

3.4 Modelling Human Mate Choice

We also re-implemented a set of human mate choice models, developed by us in the past to study how human mating demographic patterns could be “grown” from the bottom-up, by using plausible behavior rules inspired by an evolutionary functional analysis of the task domain (Simão & Todd, 2002, 2003). Our models assume agents with a finite life-span, and a (normally distributed) one-dimensional quality attribute. An agent’s goal is to try to mate with a high quality partner as early as possible in its life. Mating requires mutual acceptance, and involves a minimal courtship period during which agents may switch partner. Comparing several behavioral strategies, we found that the best types of strategies involve a combination of partner switching during courtship, and the setting of a minimal aspiration level threshold, to avoid mating with low quality partners (Simão & Todd, 2002). Using these strategies provided an account of several population level patterns, such as reasonably high levels of assortative mating for most parameter settings, and the right-skewed bell distribution of age at marriage/mating time found in a wide range of cultures.

To model this with ETHOS, we defined two `AgentPopulation` subclasses to hold a defined `Agent` class. The constructor of the `MyAgent` class defines two `SocialNet` objects. One, identified by the integer `SNET_AQ`, is used to maintain the list of agents of the opposite sex that the agent knows about. This list has a maximum size of *NS*. The method `trim()` from the base class is overridden to ensure that an agent’s date is not removed from its list of acquaintances. The second social net maintains at most one member — the agent’s present* date. The sex of agents is defined as an integer set to be 0 or 1. Agents also have their one-dimensional quality attributed when they are created. This is used as primary information whenever other agents decide whether or not to date the agent.

```

class MyAgent extends Agent implements Comparable {
    CourtStrategy s = new CourtStrategy(this);
    ...
    MyAgent(int sex) {
        super();
        setSex(sex);
        setQuality(getRandomQuality());
        setMaxAge(L);
        addSocialNet(new SocialNet(this, SNET_AQ, NS) {
            protected void trim() {
                Agent ag = getMember(Global.getRandom().nextInt(getMaxSize()));
                if (ag != getDate()) {
                    tbreak(ag);
                }
            }
        });
        addSocialNet(new SocialNet(this, SNET_DATE));
    }
}

```

```

}

MyAgent getDate() {
    return (MyAgent) getSocialNet(SNET_DATE).getMember(0);
} ...

```

Additionally, each agent maintains a reference to an object that is its strategy. In the `act()` method the strategy object is used to decide which of the agents of the opposite sex referenced in the `TaskEnv` object the agent should propose to. Still in the `act()` method, a `Selector` object is used to add a random agent of the opposite sex to the social network using a criteria that gives all agents an equal probability of choice (given they are not known already).

```

public void act(TaskEnv te) {
    if (Global.getRandom().nextDouble() <= Y) {
        Agent ag = (Agent) Selector.selectOne((getSex() == 0 ? sex1 : sex0).getAllMembers(),
            Selector.EquiCriteria.crit, null, getSocialNet(SNET_AQ).getAllMembers());
        getSocialNet(SNET_AQ).addMember(ag);
    }
    for (Iterator i = te.getAllAgents().iterator(); i.hasNext(); ) {
        MyAgent ag = (MyAgent) i.next();
        int action = s.getAction(getDate(), ag);
        if (action == 1) {
            props.add(ag);
        }
    }
    Collections.sort(props);
}
}

```

The first subclass of `AgentPopulation` we define is used to represent each sex sub-population. It takes as arguments in the constructor the identifier of the sex and the number of agents created. At each simulation time step, it selects the set of agents an agent will interact with by adding them to a `TaskEnv`. It also performs population wide operations such as removing from the population mated pairs of agents, updating agents strategies, and killing agents that exceed the set maximum age.

```

class MyAgentPopulation1 extends AgentPopulation {
    int sex;
    TaskEnv te = new TaskEnv();

    MyAgentPopulation1(int sex, int n) {
        for (int i = 0; i < n; i++) {
            addMember(new MyAgent(sex));
        }
        this.sex = sex;
    }

    public void actOne(Agent ag) {
        te.clearAgents();
        MyAgent ag0 = (MyAgent) ag;
        List asex = sex == 0 ? sex0.getAllMembers() : sex1.getAllMembers();
        List ags = Selector.select(asex, InteractionCriteria.crit, null,
            (ag0.getDate() != null ? ag0.getDate().getThisList() : null), 1);
        te.addAllAgents(ags);
        ag.act(te);
    }
}

```

```

public void act() {
    super.act();
    if (sex == 0) {
        removeMated();
    }
    updateStrats();
    kill();
}
}

```

The second subclass of `AgentPopulation` is used to reference two sub-populations, each representing one sex. Its only operation, other than passing control to its sub-populations, is to pair up agents as a result of mutual proposal and acceptance. (For brevity's sake, we omit here the details of the proposal making and the matching algorithm. The full specification can be found in (Simão & Todd, 2002).)

```

class MyAgentPopulation2 extends AgentPopulation {
    MyAgentPopulation2() {
        sex0 = new MyAgentPopulation1(0, (int) R*P);
        sex1 = new MyAgentPopulation1(1, P);
        addMember(sex0);
        addMember(sex1);
    }

    public void act() {
        super.act();
        pairUp(); //make new pairs
    }
}

```

Comparing the implementation of our mate choice model(s) using `ETHOS` with the one constructed initially from scratch we found some interesting tradeoffs. On the one hand, using `ETHOS` required us to be aware of `ETHOS` meta-model abstractions, and how to properly use them. On the other, the existence of a conceptual landscape from which objects can be picked and mixed simplifies the cognitive effort in developing non trivial models. Moreover, the code tends to be much shorter and the possibility of design and/or of implementation errors much smaller.

4 Discussion

`ETHOS` currently adds three main modelling constructs to the ones provided by widely used MAS frameworks for agent-based modelling, such as `SWARM` (Minar et al., 1996), `REPAST` (Collier, 2002), and `ASCAPE` (Brookings, 2000). Namely, behavior selection, social influence, and relationship management. It also provides a simpler and yet flexible event management scheme than these frameworks.

The behavior or action selection mechanism can be as simple or as sophisticated as the modeler desires. It can be fully implemented in the code of the agents class, as is done in the current framework, or it can be implemented through control objects that implement non-trivial action selection mechanisms (Bryson, 2000b, 2000a). At this stage `ETHOS` provides only a simple evolutionary strategy based object (`ListControl`), but others can be easily added. This type of mechanisms allow a large class of models to be implemented, such as game-theoretical ones with learning (Young, 1998).

The social influence mechanism is accommodated by having control objects update their state based not only on the reference agent payoff, but also on others' payoff or other information. Although further experience is required to see which types of social influence are more suited to study which phenomena, this

is clearly a key feature to support in modelling culture transmission and behavioral traditions (Heyes & Bennett G. Galef, 1996). For example, in a recent paper Noble and Franks studied the relationship between environmental structures and imitation and other simpler forms of learning (Noble & Franks, 2003). They found that imitation does not always benefit the individual, and other forms of social learning might be used instead. As far as ETHOS is concerned, the motivating drive is to provide a principled manner for modelling inter-agent influence and communication, without requiring ad hoc implementations by model developers. As different mechanisms with well documented semantics are provided, this will allow modelers to experiment with them and see in what scientifically relevant ways that affects their simulations' results.

Social relationships are partly supported in some of the MAS mentioned above through generic graph-like data structures. Ethos takes this a step further. Giving a semantic interpretation to agents' social relationships, it permits the automation of some processes, such as parent-offspring relationships management, or the dynamics of social network growth. Here too, additional new features might be incorporated as we gain experience using ETHOS. Such features might include analysis of social structure, and automatic generation of networks with a certain type of topology (e.g. small-world networks (Watts, 1999)).

Finally, comparing the ETHOS event dispatching scheme with that of other MAS, we believe ETHOS to propitiate a balanced tradeoff between simplicity of use and flexibility of operation. The event scheduling scheme of SWARM and REPAST, which is based on the scheduling of events for arbitrary points in time, while flexible, tends to produce event driven code. As most system programmers will attest, this tends to produce code hard to understand and prone to error (Tanenbaum, 1987; Coulouris, Dollimore, & Kindberg, 1994). In contraposition to that, ASCAPE uses a mechanism of simple iteration of behavior rules at the grain level of agent aggregates. As illustrated by the model examples presented here this may become over restrictive. Sometimes control is required simultaneously both at the level of the individual and at the level of the aggregate (population). While this can also be implemented in ASCAPE by having aggregates with only one agent we find that preternatural. While ASCAPE provides a domain ontology that maps naturally to computational abstractions, ETHOS takes the inverse approach. It maps a domain ontology which is natural for the social scientist and maps it into the computational infrastructure.

While all features mentioned above can potentially be implemented on top of these other MAS, providing them "off the shelf" simplifies modelers' work. In any case, we plan to bridge ETHOS's core features with other MAS because some users might prefer (to continue) to use them.

To keep testing the usefulness of ETHOS's main abstractions (possibly extending and refining them), we plan to implement additional models of human social behavior, either from the literature or designed specifically for our future studies. The design of an editor to specify models without requiring complete knowledge of the JAVA programming language would also be an important next step. Our design philosophy is that a feature should be incorporated in ETHOS only if it used in a wide range of models, and if its availability considerably simplifies model development and testing. Although scientifically useful models should be kept simple, it is our working experience that some types of models benefit from additional abstractions than just those provided by current MAS. Still, further work is required to glean which of ETHOS's features are most useful. ETHOS is currently at a prototyping stage, but a beta version of it can be downloaded from for testing: <http://centria.di.fct.unl.pt/~jsimao/ethos>.

5 Conclusions

In this article we described ETHOS, a MAS framework devised to support the development of agent-based models of human social behavior and culture change. We described its main abstractions and how they fit together. To test the usefulness of the framework we presented several examples that use its abstraction, including two models published in the literature. We have argued that ETHOS's features simplify the modelling process of a wide range of agent-based models intended to study human social behavior.

6 Acknowledgments

This work was partially supported by a PRAXIS XXI Ph.D. scholarship, and project FLUX, funded by FCT/MCES, Portugal. We would like to acknowledge the scientific and intellectual support and/or comments to earlier draft versions of this paper to Peter Todd, Paulo Gama Mota, Nuno Preguiça, and João Sousa.

REFERENCES

- Aunger, R. (Ed.). (2000). *Darwinizing culture: the status of memetics as a science*. Oxford University Press.
- Axelrod, R. (1985). *The evolution of cooperation*. New York: Basic Books.
- Axelrod, R. (1997). *The complexity of cooperation: Agent-based models of competition and collaboration*. Princeton/NJ: Princeton University Press.
- Back, T. (January 1996). *Evolutionary algorithms in theory and practice: Evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press.
- Bandura, A. (1977). *Social learning theory*. N.J.: Prentice-Hall, Inc.
- Bandura, A. (1985). *Social foundations of thought and action: A social cognitive theory*. N.J.: Prentice-Hall, Inc.
- Barkow, J. H., Cosmides, L., & Tooby, J. (Eds.). (1992). *The adapted mind: Evolutionary psychology and the generation of culture*. New York: Oxford University Press.
- Beyer, H.-G. (2001). *Theory of evolution strategies*. Springer Verlag.
- Boyd, R., & Richerson, P. J. (1985). *Culture and the evolutionary process*. Chicago: University of Chicago Press.
- Brookings, R. S. (2000). Ascape: An agent based modeling framework in java. (<http://www.brook.edu/es/dynamics/models/ascape/>)
- Bryson, J. (2000a). Cross-paradigm analysis of autonomous agent architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 12(2), 161–184.
- Bryson, J. (2000b). Hierarchy and sequence vs. full parallelism in action selection. In J.-A. Meyer, A. Berthoz, D. Floreano, H. Roitblat, & S. W. Wilson (Eds.), *From animals to animats 6: Proceedings of the sixth international conference on simulation of adaptive behaviour*. Cambridge, MA: MIT Press/Bradford Books.
- Collier, N. (2002). Repast: An extensible framework for agent simulation. (<http://repast.sourceforge.net/projects.html>)
- Coulouris, G., Dollimore, J., & Kindberg, T. (1994). *Distributed system: Concepts and design (second edition)*. Wokingham: Addison-Wesley.
- Deacon, T. W. (1998). *The symbolic species: The co-evolution of language and the brain*. New York: W.W. Norton & Company.
- Donald, M. (1993). *Origins of the modern mind: Three stages in the evolution of culture and cognition*. New York: Harvard Univ Press.
- Doran, J. E. (2000). Trajectories to complexity in artificial societies: Rationality, belief, and emotions. In T. A. Kohler & G. J. Gumerman (Eds.), *Dynamics in human and primate societies — agent-based modeling of social and spatial processes* (pp. 89–105). New York: Oxford University Press.

- Durham, W. H. (1990). *Coevolution: Genes, culture, and human diversity*. Stanford, CA: Stanford University Press.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object oriented software*. Addison-Wesley.
- Gilbert, N. (2000). Modeling sociality: The view from europe. In T. A. Kohler & G. J. Gumermman (Eds.), *Dynamics in human and primate societies — agent-based modeling of social and spatial processes* (pp. 355–371). New York: Oxford University Press.
- Goldberg, D. E. (1989). *Genetic algorithms: in search, optimization & machine learning*. Addison Wesley.
- Heyes, C. M., & Bennett G. Galef, J. (Eds.). (1996). *Social learning in animals: The roots of culture*. San Diego: Academic Press.
- Higgs, P. G. (2000). The mimetic transition: a simulation study of the evolution of learning by imitation. *Proceedings of The Royal Society of London*(267), 1355–1361.
- Laland, K. N., Odling-Smee, J., & Feldman, M. W. (2000). Niche construction, biological evolution, and cultural change. *Behavioral and Brain Sciences*, 23, 131–175.
- Minar, N., Burkhart, R., Langton, C., & Askenazi, M. (1996). The swarm simulation system: A toolkit for building multi-agent simulations. (<http://www.swarm.org/index.html>)
- Mithen, S. (1996). *The prehistory of the mind: The cognitive origins of art, religion and science*. London: Thames and Hudson.
- Noble, J., & Franks, D. W. (2003). Social learning mechanisms compared in a simple environment. In R. K. Standish, M. A. Bedau, & H. A. Abbass (Eds.), *Artificial life viii: Proceedings of the eighth international conference on artificial life*. Cambridge, MA: MIT Press.
- Nowak, A., & Vallacher, R. R. (1998). *Dynamical social psychology*. New York: Guilford Press.
- Reed, E. S. (1996). *Encountering the world: Toward an ecological psychology*. New York: Oxford University Press.
- Simão, J. P., & Pereira, L. M. (2003). Ethos: A mas framework for modeling human social behavior and culture. In *Proceeding of the 4th workshop on agent-based simulation* (pp. 87–92). SCS European Publishing House.
- Simão, J. P., & Todd, P. M. (2002). Modeling mate choice in monogamous mating systems with courtship. *Journal of Adaptive Behavior*, 10(2).
- Simão, J. P., & Todd, P. M. (2003). Emergent patterns of mate choice in human populations. *Journal of Artificial Life (special issue)*, (to appear).
- Sperber, D. (1996). *Explaining culture: A naturalistic approach*. Oxford: Blackwell.
- Tanenbaum, A. S. (Ed.). (1987). *Operating systems: Design and implementation*. Englewood Cliffs/NJ: Prentice-Hall Software Series.
- Watts, D. J. (Ed.). (1999). *Small worlds*. Princeton/NJ: Princeton University Press.
- Young, H. P. (1998). *Individual strategy and social structure: An evolutionary theory of institutions*. Princeton/NJ: Princeton University Press.