Some papers and information from the
Munich Simulation Group

DMASS: A Distributed Multi-Agent System for Simulation in Social Science

SMASS: A Serial Multi-Agent System for Simulation in Social Simulation

SMASS.2 - A Sequential Multi-Agent Social Simulation System. An updated version

Towards Computational Institutional Analysis: Discrete Simulation of a 3P Model

Urban's Rule for Generating a List of other Lists

DMASS: A Distributed Multi-Agent System for Simulation in Social Science

Wolfgang Balzer

Karl Brendel

University of Munich
Institut für Philosophie, Logik und Wissenschaftstheorie
April 1996

The aim of this paper is to describe a first version of a program for a multi-agent system which we call DMASS in the following for the sake of reference.[1] DMASS is a properly distributed system running under BRAIN AID PROLOG[2] on transputer systems. The intended applications of the program are qualitative simulations of social systems in a micro-macro setting in which each actor of the social system is represented by one transputer.

We do not know any other system running on distributed hardware which aims at this kind of applications. Therefore, we believe, the description of DMASS is of some interest even if we do not yet have thoroughly worked out simulation results.[3]

Existing qualitative, social simulations mainly are performed in a cellular automata environment[4] and programmed and run in serial environments.[5] We claim that the existing simulation results can be reproduced in DMASS without much effort and that DMASS is an instrument for social simulation much more flexible and smooth than cellular automata. A detailed comparison of DMASS with cellular automata simulations cannot be made here for reasons of space.

Our second claim deals with the treatment and administration of time in DMASS as compared with serial systems. DMASS in this respect is not only much easier and much simpler to program but also avoids biases which at the present state of standard operating systems are practically unavoidable in serial programs.

The salient feature of DMASS is a minimal, restricted conceptual apparatus to deal with comprehensive, social phenomena. Its architecture primarily aims at providing a flexible frame in which the user may himself insert variable 'rules of behavior'

[1] We are indebted to G.Gerl, J.Sander, J.Urban and J.Tazarki for helpful suggestions, to K.Ritter for permission to use a first generation transputer system in his research lab at the Technical University of Munich, and to R.Schöne and W.Schultz for technical assistance.

[2] This is the only PROLOG version we know running on distributed hardware. The package is developed by M.Ostermann, F.Bergmann and G.von Walter, see (Ostermann et al., 1995).

[3] We are presently performing simulations at a large transputer system at the computing center PC^2 of the University of Paderborn. We are indebted to M.Ostermann and B.Bauer for assistance in using this computer.

[4] See, for instance, (Cohors-Fresenborg, 1977) for this notion.

[5] Interesting simulation results have been achieved in such settings, for instance by (Hegselmann, 1994) and (Liebrand & Messick, 1992).

for various forms of human action and interaction, including various forms of messages, and study the macro-effects of these micro-items. DMASS was developed by application of a bottom up strategy: only those notions which proved essential for the intended simulations were included. This results in a lack of almost all features present in BDI architectures.[6]

1 Transputer Systems and BRAIN AID PROLOG

A transputer system consists of a set of independent processors (transputers) among which there are hard wired connections. Each transputer supports eight processes which can be identified in a program. If we neglect the particular features of transputers a transputer system may be imagined just as a net of ordinary computers which are interconnected with each other. However, only one of these -the 'server'- is distinguished as an interface to the user while the other, 'normal' transputers are not directly accessible to the user. Thus, if the user wants to place some input in one of the normal transputers, he has to load that input to the server together with a command that makes the server send the input as a message via internal wire to the intended receiver.

A transputer program therefore typically consists of two parts. One part regulates the input-output behavior of the server, while the second part contains those programs which are downloaded -via server- to the normal transputers. Input typically consists of different programs, one for each normal transputer. The server takes these programs and sends them to the respective normal transputers for which they are intended, together with a command to store them.[7] When the program is finished the server collects the results from each normal transputer (if there are any), and gives them as output to the user.

BRAIN AID PROLOG[8] or BAP, for short, is a variant of PROLOG which runs on transputer systems. On each single transputer BAP is just like PROLOG. Thus each transputer has a data-base of atomic facts together with a set of rules which are formulated in the well known Horn-clause form.[9]

In addition to this, in BAP each transputer has a 'mailbox' to which messages can be sent by the other transputers. Each transputer systematically empties its

[6] See (Rao & Georgeff, 1991).

[7] A nice feature of the PROLOG environment is that these programs are PROLOG terms and thus have the structure of ordinary messages as exchanged during the simulation runs.

[8] (Ostermann et al., 1995).

[9] An easy introduction to PROLOG is found in (Clocksin & Mellish, 1984), see also (Derantsart et al., 1996).

mailbox in 'first in - first out' order, and deals with the messages it finds there. More precisely, the transputer considers the first message of the stack in the mailbox (which is the first it received earlier). The content of the message is given by a PROLOG term. When the transputer has read this term it immediately takes it as a command and tries to execute it. For instance, if the message content was a predicate, say, $retract(value(peter, 2, 3))$ the transputer will execute this PROLOG predicate, and retract the entry $value(peter, 2, 3)$ from its set of facts.

In BAP, each transputer also can send out messages with a distinct address. This is done by a command $send(Sender, Receiver, Content)$ in which $Sender$ and $Receiver$ identify the sender and the receiver of the message,[10] respectively, and $Content$ is the message content which can be any PROLOG term. When a transputer comes to execute this predicate, the message -together with the identifications- is sent to the transputer identified in the predicate as $Receiver$. There, it is added to the stack of messages in the receiver's mailbox, as the last member.

The notion of a 'term' in PROLOG is rather comprehensive. For instance, each PROLOG program as well as each atomic fact is a term. As PROLOG does not impose any substantial restriction on the logical types of its expressions the content of messages sent back and forth thus can be arbitrarily rich. In this respect BAP seems to outperform all other available programs for distributed systems.

2 Transputers Representing Social Agents

Our philosophy in using a transputer system for social simulations is to represent each social agent by one transputer:

$$\text{one agent} - \text{one transputer.}$$

Actually, a transputer supports eight processes so one might contemplate to increase the number of agents by representing each agent by one (and always the same) process on one (and always the same) transputer. This strategy,[11] however, has disadvantages the clarification of which, we think, is illuminating.

Social agents are not only agents, that is, individuals with sensory surfaces and certain intellectual capabilities. They are at the same time *social* beings. It is not very clear to-day what this means precisely, but one thing *is* clear: an essential feature of 'sociality' rests in the ability of exchanging information, of communicating, and of influencing one another. In a simulation environment this feature reduces to the

[10] More precisely, these are names for transputers in the system.

[11] The strategy is also made attractive by the high cost of transputer hardware in comparison to 'ordinary' hardware.

exchange of messages, more precisely, to the ability of sending out and receiving messages which bear some more or less definite content. Thus there are two basic components for a system of social agents, both equally important. On the one hand, there are the individuals with their sensory and cognitive abilities. In a very reduced form such an individual is represented on the computer by a knowledge base consisting of a set of statements and of rules for changing this set in the light of incoming, new information.

On the other hand there is the 'world' of interaction, communication and influence. In order to represent this world of interaction, the knowledge base must be split up and its parts relativized to the individuals, and another component must be added to the system: another set of rules which 'simulate' the exchange of pieces of knowledge among the 'different' knowledge bases of the individuals. In the real world communication is highly contingent on the content of the messages exchanged and on the respective states of the senders and receivers. The 'rules of communication' are elusive and difficult to identify. To know them means to know the structure of the social world in which the persons live.

A distributed program represents this situation rather realistically. There are practically no explicit rules for communication. The only thing regulated is the 'physical' exchange of information between the transputers. The content as well as the address and time of a message is entirely under control of the sender, and of its respective internal state. In the same way, each receiver of a message is 'free' to react to it or not.

A serial program in principle might be constructed in the same fashion. However, in such a program there is no necessity of explicitly representing messages and their exchange. The effect of one person sending a message and another person reacting to it may directly be stated as a rule of the program. As a matter of programming it seems tedious and redundant to stick to the pattern of message exchange, and there is a continuous drive to make 'shortcuts'. But the previous discussion should have made clear that each such 'shortcut' amounts to introducing some explicit rule of the social system, a rule which is unlikely to be generally valid in the system.

What becomes visible is a tension between real, 'social' rules for communication and individual patterns of behavior in communication. Few real rules are known at the moment, so it seems good strategy to avoid them in a computer representation. On the other hand, the individual patterns of communication are numerous, and if they are represented by program rules the program becomes overloaded and unrealistically fine grained. In serial programs there is this tendency to work with program rules as a means of governing processes of interaction, and as a result there is another tendency of working with rather homogeous rules, rules which work across a wide range of individuals and situations. This is bound to create a simplistic, and sterile picture of communication in a group of actors.

In this respect, a transputer system is closer to reality, simply because in it, real messages are exchanged. The content of each message, as well as the decision of

sending it or reacting to it, is contingent on the particular state of the individual. So whatever 'rules' are governing the exchange of messages: they must be implicit in the rules representing the individuals.

If we would not represent each person by one transputer but by one process we would face the same bias as do serial programs. We would explicitly have to describe by program rules the effect of one person's internal state on that of another person which corresponds to information exchange.

3 DMASS and the Background Model

DMASS captures just one feature of a comprehensive model of social institutions described in (Balzer, 1990).[12] According to this model a social institution is a system in which, very roughly, a hierarchical structure among groups is induced by individual exertions of power or influence. The basic relation capturing exertions of power is a four place relation $power(i, a, j, b)$, reading 'person i by doing action a exerts power over person j so that j does action b'. An alternative, and somewhat more neutral interpretation is obtained by replacing exertion of power by exertion of influence. In reality, power is exerted in many different ways, and with many different kinds of actions a, b.[13]

In DMASS, actions are treated in an abstract way, and are represented by numbers. More precisely, an action is represented by a pair (I, O) of natural numbers representing, respectively, the value or amount of *input* and *output* of the action thus represented. I is interpreted as the effort, or the value of the effort, it takes for a particular person to perform the action (I, O), and O is interpreted as (the value of) the result of the action for the person. An exertion of power involving two actions and two actors, thus is represented by six components:

$$(i, j, I_i, O_i, I_j, O_j).$$

i is called the *determining* agent and j the *subordinate* agent, and $I_i, ..., O_j$ are i's and j's in- and outputs of the actions they perform which together make up the exertion of power. Note that this format is very similar to that of exchange. In fact, it can be shown that exchange is a limit case of exerting power.[14]

In this format for power or influence, DMASS does not differentiate between different types of power or influence, nor does it differentiate between different kinds of

[12] See also (Balzer, 1993).
[13] Compare (Balzer, 1992).
[14] (Balzer, 1994).

actions. There is just one homogenous kind of power and one abstract type of action which has different 'degrees' or 'tokens' as given by the numbers I, O.

There is a finite set of individuals which can perform actions of the kind described, and which are grouped into neighbourhoods. Each person i has a fixed set of neighbours, and the transputer representing that person has stored these neighbours as facts of the form $neighbour(i, j)$ (read: 'j is a neighbour of i'), for finitely many j. Each person has a degree of 'physical strength'[15] which is represented by a number s, $1 \leq s \leq 10$. Neighbours and strengths of each person remain fixed throughout the simulation.

Moreover, each person has an 'account' which in an abstract way expresses the value of the person's possessions. For person i this value is stored in the form $value(i, w)$ where w is a natural number. This value changes over time but as the time elapsing in DMASS simply is machine time there is no need to use an explicit argument for time. If w is replaced by some new w' in the course of interaction, the 'old' entry $value(i, w)$ is deleted from, and the 'new' one, $value(i, w')$, is inserted in i's database. In the beginning, each actor i has an 'initial endowment', some entry $value(i, w)$ is present in i's database.

Now the interaction taking place in DMASS is very simple. Each person tries to increase the value of its account by exerting power over suitable neighbours. The only and main rule is that physical strength determines who can be a determining agent in a given pair of actors. More precisely, each actor i chooses one of his neighbours j, and checks whether he (i) is stronger than j. If so, i can enforce actions on j, and j *must* perform them. i first chooses some 'token' of exertion of power $(i, j, I_i, O_i, I_j, O_j)$.[16] He then performs 'his' part of the event, namely (I_i, O_i) and then 'forces' j to perform 'his' (i.e. j's) part (I_j, O_j). In due course both agents actualize their accounts accordingly. i subtracts from his present value the effort I_i and adds the output of the event, O_i, that is, i replaces $value(i, w)$ by $value(i, w')$ where $w' = w - I_i + O_i$. j proceeds in the same way with his value w_j given by $value(j, w_j)$, and his action (I_j, O_j). Basicly, that's all.

On the machine, this process involves just one message: when i has found a neighbour j and a suitable power-token $(i, j, I_i, O_i, I_j, O_j)$ he sends a message to j telling j to perform 'his' (j's) part (I_j, O_j): $perform(I_j, O_j)$. If j finds this message in his mailbox, he will perform, i.e. adjust his account according to the I_j, O_j-values.

Even this rather simplistic approach provides many opportunities for variation: number of neighbours, physical strengths of the actors, the distribution of these strengths in neighourhoods, initial endowments and their distributions in neighbourhoods, as well as the precise rules according to which an actor chooses among his neighbours and finds a power-token to enforce.

[15] This degree may also be interpreted more abstractly as 'social' strength or wealth, or social capital.

[16] The precise ways of how a neighbour and an action-token are chosen may be varied in various directions which need not be discussed here.

4 The Program

Here we provide the full program with detailed comments which should enable the reader to follow and understand the program. Knowledge of PROLOG notation is presupposed.

The program is described in two parts. In section A, we describe the operation of the central, organizing node, node 1. This node is operative in preparing the system for the simulation, in particular getting necessary facts and data from the user, sending these to the other nodes, telling them when to begin their individual operation and when to stop. Moreover, it loads all the necessary programs in the databases of the individual nodes. At the end of the simulation, node 1 tells all other nodes to report their relevant data to node 1 from where they are given to the user.

PREDICATE simulation.

```
simulation :-   assert_facts,            (1)
.               collect_data(N),         (2)
.               create_individuals(N),   (3)
.               distribute_data(N),      (4)
.               load_individuals(N)      (5)
.               start_simulation(N),     (6)
.               stop_simulation(N),      (7)
.               show_results(N).         (8)
```

This predicate exclusively deals with node 1 which is used as the 'coordinator' feeding the initial data to the 'ordinary' nodes, and collecting data from the 'ordinary' nodes when the simulation is stopped. Node 1 does not act as an ordinary individual. 'Ordinary' nodes, different from node 1, will be referred to as 'individuals' in the following.

(1) assert_facts asks for two facts the user has to feed in. These are:
a) The number of individuals taking part in the simulation. This number is denoted by Number_of_individuals, or briefly, in the explanations, by N. individuals(N) is added as a fact to node 1's database.
b) The runtime, i.e. the number of seconds the simulation is going to run, denoted by runtime(R), where R is a non-negative integer. After R seconds, the simulation will stop, the values then present in each node are sent to node 1 and stored there for display. runtime(R) is added as a fact to the database of node 1.

(2) collect_data asks for the data defining the initial state of each indidivual which are interactively obtained from the user. These data are:
a) the strength of each individual I, denoted by strength(I,J), where J is an integer, $1 \leq J \leq N$ (N = Number_of_individuals),

b) the value W of each individual I's account, displayed in the form value(I,W) where W is an integer.

(3) create_individuals initializes those nodes acting as individuals. This is a feature characteristic for the transputer system.

(4) distribute_data to each indidivual sends the initial data relevant for that indidivual. These data are present in node 1 because of (2), and are sent from there to each individual. Moreover, this predicate to each inidvidual sends a number of rules (predicates) which the individual is to store in its data-base, and which during simulation govern the 'behavior' of each indidivual. After that, each individual is prepared with its own set of facts and rules, and thus ready to start interacting with other individuals.

(5) load_individuals loads all the predicates defined in section B below to the database of each individual, and sets each individual in a mode in which it can be activated by means of the 'start' message from node 1. This mode is set by calling in each individual the predicate my_action_loop described in Sec.B.

(6) start_simulation triggers the beginning of action of each individual. Each individual K starts to execute what we call its 'main loop' (see below). Roughly, K checks its neighbours and their strengths, compares them with its own strength, and then on the basis of some heuristics, tries to find a neighbour and an action token such that performing the action token with that neighbour increases K's value account (possibly in a maximal way).

(7) stop_simulation stops the operation of the individuals after R seconds, (R from 1-b). Each individual quits its main loop and then sends the value present in its account to node 1 where it is stored.

(8) show_results displays the results of the value accounts of the individuals when the simulation has stopped. Other information also might be shown with this predicate.

We present the details in the order in which they are needed when the program is running.

An auxiliary predicate needed for output on the screen is

```
PREDICATE phh(voidlist). phh([ ]) :- nl.
phh([H|T]) :- write(H), write(' '), phh(T).

PREDICATE assert_facts.

assert_facts :-
phh['Type', in, a, non-negative, integer, 'N', followed, by, a, dot, '.', and, 'RETURN.'
'N', denotes, the, number, of, nodes, you, want, to, 'simulate.' '1 ≤ N ≤ 20.']),
read(N),
asserta(individuals(N)),
phh['Type', in, a, non-negative, integer, 'R', followed, by, a, dot, '.', and 'RETURN.'
'R', denotes, the, number, of, seconds, the, system, will, 'run.']),
read(R),
```

```
asserta(runtime(R)),
fail.
```

The number N of individuals, individuals(N), and the runtime R -in seconds-, runtime(R), are added to the database of node 1.

PREDICATE collect_data(integer).

```
collect-data(N) :-
collect_strengths(N),
collect_values(N).

collect_strengths(N) :-
database(individuals(N)),
phh(['Type', in, a, list, of, 'N', pairs, of, numbers, for, strengths, of, the, form, '[i,j]',
with, '1 ≤ i ≤ N', and, '0 ≤ j ≤ 10.', 'Each', pair, should, be, enclosed, in, brackets,
'[ ].', '(Type', '[0,0]', when, 'finished).']),!,
repeat,
write('Enter Strength-list: '),
read([Individual|[Strength|[ ]]]),
( Individual = 0, Strength = 0 ;
asserta(strength(Individual,Strength)), fail ).
```

The pairs strength(I,J) are put in by the user and added to the database of node 1.

```
collect_values(N) :-
database(individuals(N)),
phh(['Type', in, a, list, of, 'N', pairs, of, values, of, the, form, '[i,v]', with, '1 ≤ i
≤ N', and, an, integer, 'v.', 'Each', pair, should, be, enclosed, in, brackets, '[ ].',
'(Type','[0,0]', when, 'finished).']), !,
repeat,
write('Enter Value list: '),
read([Individual|[Value|[ ]]]),
( Individual = 0, Value = 0 ;
asserta(value(Individual,Value)), fail ).
```

The list of values of the form value(I,V) is fed in by the user and stored in the database of node 1.

PREDICATE create_individuals(integer).

```
create_individuals(N) :-
database(strength(I,_)),
I ≤ N,
create(msg(I,8)),
fail.
create_individuals(_) :- !.
```

create(msg(I,8)) is BRAIN-AID specific. It initializes process number 8 on node I. Process 8 is used by default.

PREDICATE distribute_data(integer).

```
distribute_data(N) :-
distribute_strength_data(N),
distribute_values(N),
define_neighbours(N).
```

To each individual K there are sent the following data: data about K's strength, about the value of K's account, and data about the names (numbers) and strengths of K's neighbours. Each individual has exactly four neighbours, and these are defined by a certain scheme which is defined by define_neighbours and need not be described here.

PREDICATE distribute_strength_data(integer).

```
distribute_strength_data(N) :-
database(strength(Individual,Strength)),
Individual ≤ N,
send_msg((Individual,8),asserta(strength(Individual,Strength))),
fail.
distribute_strength_data(_) :- !.
```

The predicate iterates throught the database by getting all facts of the form strength(I,S), checks whether I ≤ N, and then tells individual (node) I to add the fact strength(I,S) to it's (I's) database. This is done with the help of the BRAIN AID item send_msg.

PREDICATE distribute_values(integer).

```
distribute_values(N) :-
database(value(Individual,Value)),
Individual ≤ N,
send_msg((Individual,8),asserta(value(Individual,Value))),
fail.
distribute_values(_).
```

The predicate iterates through the database, gets all facts of the form value(I,S), checks whether I ≤ N, and tells individual (node) I to add the fact 'value(I,S)' to it's (I's) database. This is done with the help of the BRAIN AID item send_msg.

PREDICATE define_neighbours(integer).

The effect of this predicate is to add to each individual I's database the following facts.
a) neighbours(I,[Neighbour1,Neighbour2,Neighbour3,Neighbour4]), where Neighbour1, ..., Neighbour4 are the four neighbours of I.
b) neighbours_strength(I,
[[StrengthNeighbour1,Neighbour1], [StrengthNeighbour2,Neighbour2],

[StrengthNeighbour3,Neighbour3], [StrengthNeighbour4,Neighbour4]]),
i.e. a list of four pairs of strength-values together with the names (numbers) of the corresponding neighbours. Each pair [StrengthNeighbourJ,NeighbourJ] expresses that neighbour J has strength S = StrengthNeighbourJ. 'strength(J,S)' is present in node 1's database.

```
define_neighbours(N) :-
database(strength(Individual,_)),
Individual ≤ N,
neighbours(Individual,[Neighbour1,Neighbour2,Neighbour3,Neighbour4]),
database(strength(Neighbour1,StrengthNeighbour1)),
database(strength(Neighbour2,StrengthNeighbour2)),
database(strength(Neighbour3,StrengthNeighbour3)),
database(strength(Neighbour4,StrengthNeighbour4)),
send_msg((Individual,8),asserta(neighbours(Individual,
[Neighbour1,Neighbour2,Neighbour3,Neighbour4]))),
send_msg((Individual,8),asserta(neighbours_strength(Individual,
[
[StrengthNeighbour1,Neighbour1], [StrengthNeighbour2,Neighbour2],
[StrengthNeighbour3,Neighbour3], [StrengthNeighbour4,Neighbour4]
]))),
fail.
define_neighbours(_) :- !.
```

The predicate iterates through node 1's database, gets all facts of the form strength(I,S), checks whether I ≤ N, and then does the following. It calls the predicate neighbours(I,List) defined below under (∗) which gives I's four neighbours in the list 'List'. For each such neighbour 'NeighbourJ' node 1 then gets from its database the strength of NeighbourJ. Node 1 then, for each individual I so obtained, tells I (by means of BRAIN AID send_msg) to add the two predicates neighbours(I,[Neighbour1,Neighbour2, Neighbour3,Neighbour4]), and neighbours_strength(I,[[StrengthNeighbour1,Neighbour1], [StrengthNeighbour2,Neighbour2], [StrengthNeighbour3, Neighbour3], [StrengthNeighbour4,Neighbour4]]) to it's (I's) database.

(∗) PREDICATE neighbours(integer,voidlist).

We assume the individuals to be connected in a 'circular' topology. The natural neighbours of individual I are just the individuals I-2, I-1, I+1, I+2. However, this assignment does not work at the point where the 'last' individual, N, is put next to the 'first', 1. At this point some extra definitions are needed which are expressed by the predicate node(I,J), to be read: if I+1,I+2,I-1,I-2 are in the scope of 1,...,N then J is I, if not then J is an appropriate substitute falling within that scope.

PREDICATE node(integer,integer).

```
node(Node,Correct_Node_Nr) :-
```

database(individuals(N)),
Node = 0, Correct_Node_Nr = N ;
Node = -1, Correct_Node_Nr = N - 1 ;
Node = N + 1, Correct_Node_Nr = 1 ;
Node = N + 2, Correct_Node_Nr = 2 ;
Node ≠ 0, Node ≠ -1, Node ≠ 17, Node ≠ 18,
Correct_Node_Nr = Node.

The number N of individuals used is taken from the database.

neighbours(Individual,[Neighbour1,Neighbour2,Neighbour3,Neighbour4]) :-
Candidate1 is Individual - 2, node(Candidate1,Neighbour1),
Candidate2 is Individual - 1, node(Candidate2,Neighbour2),
Candidate3 is Individual + 1, node(Candidate3,Neighbour3),
Candidate4 is Individual + 2, node(Candidate4,Neighbour4).

I's second neighbour, Neighbour2, for example is determined as follows. As a candidate, Candidate2 is defined by Candidate2 = I - 1. It is then checked whether Candidate2 falls within the scope of 1,...,N (by means of the node-predicate). If not, the node-predicate corrects the value of Candidate2, and sets it to Neighbour2.

PREDICATE load_individuals(integer).

This predicate refers to a file called node_predicates, described in section B below. In this file all the predicates to be uses by each individual are stored.

load_individuals(N) :-
database(strength(I,S)),
exec(I,(load(node_predicates), my_action_loop)).

The predicate iterates through all individuals I present in terms of strength-data in the database of node 1. For each such individual I, it triggers the execution of (load(node_predicates), my_action_loop) by individual I. 'ecex' is a BRAIN AID predicate, and leads to the execution of

(load(node_predicates),my_action_loop).

This leads to the following. Node I loads down all the predicates from the file node_predicates, described in section B, to its database, and then executes my_action_loop. The latter is one of the predicates included in the file node_predicates, and thus present in I's database, after that file has been loaded down.

PREDICATE start_simulation(integer).

start_simulation(N) :-
phh(['The', simulation, can, now, 'begin.', 'Type', 'start', followed, by, 'RETURN', to, start, the, 'simulation.']),
read(_),

start_individuals(N).

The user is asked to start the simulation. If he does, the predicate start_individuals is activated which is described below.

PREDICATE start_individuals(integer).

```
start_individuals(N) :-
database(strength(Individual,_)),
Individual ≤ N,
send_msg((Individual,8)),start),
fail.
start_individuals(_) :-
write('Simulation started! ... and running ...'), nl, !.
```

This predicate runs through node 1's database and gets all facts of the form strength(I,_), i.e. all individuals. Each such individual then is told to start operating by sending it the message 'start'. When all individuals have been treated, node 1 informs the user that the simulation has started.

All individuals now interact with each other. These interactions are independent of node 1 and described in part B below. Meanwhile, node 1 checks the runtime set in (1) by means of the predicate stop_simulation.

PREDICATE stop_simulation(integer).

```
stop_simulation(N) :-
database(runtime(Runtime)),
time(CurrentTime),
OutTime is CurrentTime + (Runtime ⋆ 100),
check_for_timeout(OutTime),
write('Stopping Simulation!'), nl,
database(strength(Individual,_)),
Individual ≤ N,
send_msg((Individual,8), quit),
fail.
stop_simulation(_) :- !.
```

This predicate gets the runtime from node 1's database: 'Runtime' which has been put in under (1). It then checks the actual time with the predicate 'time' of BRAIN AID, and sets CurrentTime to the actual time. Now OutTime, the time at which the simulation is to be stopped, is defined by OutTime = CurrentTime + (Runtime ⋆ 100). This value is handed over to the predicate check_for_timeout. The predicate check_for_timeout(OutTime) which is described below continuously checks whether the present time -i.e. the time of the inbuilt clock- is identical with OutTime. If the present time becomes greater than OutTime, check_for_timeout succeeds. The stop_simulation predicate tells the user that the simulation is stopped. It then tells

each individual (by retrieving the strengths in its database as in other instances before) to quit. This is done by sending the message 'quit' to each individual. An individual getting this message will stop its operation according to a procedure which is independent of node 1 and therefore is described in section B below.

PREDICATE check_for_timeout(integer).

```
check_for_timeout(OutTime) :-
repeat,
time(CurrentTime),
( CurrentTime ≥ OutTime, write('Reached timeout!'), !
; fail ).
```

The following loop is repeated. Current time is retrieved from the inbuilt clock by means of the time-predicate. If CurrentTime is ≥ the OutTime calculated before, the cut prevents further repetition, the predicate succeeds, and we can continue in the above predicate stop_simulation. If CurrentTime < OutTime the loop is repeated. Thus the loop will operate until current time becomes greater than the OutTime set before.

PREDICATE show_results(integer).

```
show_results(N) :-
database(strength(Individual,_)),
Individual ≤ N,
rec_msg(_,value(Individual,Value)),
str_int(IStr,Individual),
str_int(VStr,Value),
conlist(['Value account of individual ',IStr,' is ',VStr,'n'], Info),
write(Info),
fail.
show_results(_) :- !.
```

This predicate iterates through all individuals and then waits until from individual I it gets the message 'value(I,V)'. This is done by means of the rec_msg-predicate in BRAIN AID which succeeds only when the required message is received. The individuals and their values are then displayed on the screen.

B: MODULE nodepredicates.

Here we describe the programs which have to be loaded to each individual node different from node 1. These programs are loaded to the individual nodes by means of load_individuals in (5) above. The file 'nodepredicates' is then sent by means of BRAIN AID 'exec' to each individual. In the following description, we assume that this has happended and that all programs of the file 'nodepredicates' are available in the database of each individual. In the following description we imagine to look at

the programs available on node K, K=2,...,N.

The predicates -in the order in which they will occur when the program runs- are:

PREDICATE nodepredicates :-

my_action_loop,	(9)
treat_goal(_,_),	(10)
insort(_,_,_),	(11)
next_receiver(_,_),	(12)
start,	(13)
action,	(14)
perform(_,_,_,_,_),	(15)
ran(_,_,_)	(16)
quit.	(17)

PREDICATE my_action_loop.

This predicate is used to control the interplay of the individuals. In its present form, there is pretty much control.

```
my_action_loop :-
asserta(action(off)),
repeat,
( database(action(off)) ;
database(action(on)), action ),
rec_msg(Client,Goal),
( Goal = quit ;
treat_goal(Client,Goal), fail ).
```

The predicate begins by writing 'action(off)' in K's database. As long as this entry is present in K's database, K is not allowed to take action; it can only react to messages. The predicate then starts a loop which is repeated until the clause 'shutdown' is reached in which case the predicate succeeds. The first disjunction, database(action(off)) or database(action(on)), works as follows. If 'action(off)' is found in the database, the clause succeeds. In this case the individual is in a 'passive' mode. It calls the BRAIN AID item rec_msg(Client,Goal) which is executed only if the individual receives a message 'Goal' from individual 'Client.' As long as there is no message, the individual (K) will not do anything. When a message is received, two cases are considered, according to the content of the message. First case: the message is 'shutdown' which means that individual K is told to stop its operation.

Second case: the Goal received is one which is admitted in the treat_goal-predicate. In this case, the goal is executed as prescribed by treat_goal, see below. After that, the node K fails, and repeats the repeat-loop. A change in the database from 'action(off)' to 'action(on)', i.e. a switch from the passive to an active mode, occurs when the 'start' predicate is called (see below).

PREDICATE treat_goal(void,void).

This predicate is used in order to avoid a feature built in in BRAIN AID, namely that in a rec_msg-loop all information is retracted from the database before the message is executed. This would lead to the destruction of all information which each node uses in order to participate in interactions. treat_goal avoids this by specifying the different possible forms of goals which node K can receive in a message, and by specifying K's reaction to each such goal.

```
1: treat_goal(_,asserta(X)) :- asserta(X),!.
2: treat_goal(Client,retract(X)) :-
     retract(X),
     send_msg(Client,retract(X,success)),!.
treat_goal(Client,retract(_)) :- send_msg(Client,retract(_,failure)).
3: treat_goal(_,start) :- start,!.
4: treat_goal(_,perform(A,B,C,D,E,F)) :- perform(A,B,C,D,E,F),!.
5: treat_goal(Client,send_strength) :-
     database(strength(_,MyStrength)),
     send_msg(Client,strength(_,MyStrength)).
6: treat_goal(Client,quit) :- quit(Client),!.
```

treat_goal covers six cases. In case 1 the message received tells to asserta(X), in which case X is added to K's database. In case 2 the message is retract(X). In this case K attempts to retract X from its database. If X is in fact in K's database, this succeeds, and K returns to the Client, from which K got the message that X was successfully retracted. If X is not in K's database, retract(X) cannot succeed. In this case K uses the other avaliable option and returns to Client that retraction has failed. In case 3 the message is 'start'. In this case K proceeds to the start-predicate, and thereby starts its main loop, see below. In case 4, the message received tells 'perform', i.e. to perform an action type specified by the list [A,...,F]. In this case the action type is performed according to the perform-predicate described below. In case 5, the message received says that K should send information about its strength to Client. In this case, K retrieves its strength MyStrength from the database and sends the message 'strength(K,MyStrength)' to Client. In the final case, the message tells K to quit. K then goes to the quit-predicate which -in its present version (to be emended) simply sends the actual value of K's account to node 1.

PREDICATE insort(voidlist,voidlist,voidlist).

```
insort([ ],[ ],_).
insort([X|L],M,P) :- insort(L,N,P), insortx(X,N,M,P).
```

insort(L,N,P) sorts a list L according to an ordering relation P, and returns a newly, ordered list N. In this, it refers to insortx, see below.

```
insortx(X,[A|L],[A|M],N) :-
```

```
P =.. [N,A,X], call(P), !,
insortx(X,L,M,N).
insortx(X,L,[X|L],_) :- !.
```

The following predicate insort_list(L,N,P) sorts a list of lists L according to some order relation P, and returns a new list of lists N. This is done by comparing only the first elements of each list in L.

```
insort_list([ ],[ ],_).
insort_list([X|L],M,P) :-
insort_list(L,N,P), insortx_list(X,N,M,P).
```

The predicate insortx_list is defined below.

```
insortx_list(X,[A|L],[A|M],N) :-
order1(A,X,N), !,
insortx_list(X,L,M,N).
insortx_list(X,L,[X|L],_) :- !.
```

The order referred to in the previous predicate is defined as follows.

```
order1([A|_],[B|_],N) :- P =.. [N,A,B], call(P).
```

PREDICATE next_receiver(integer,integer).

This predicate randomly selects one of K's four neighbours with whom K will interact in the next step.

```
next_receiver(Individual,Neighbour) :-
database(neighbours(Individual, [Neighbour1,Neighbour2,Neighbour3,Neigh-
bour4])),
random(Number),
Selector is Number mod 4,
( Selector = 0, Neighbour is Neighbour1;
Selector = 1, Neighbour is Neighbour2;
Selector = 2, Neighbour is Neighbour3;
Selector = 3, Neighbour is Neighbour4 ).
```

This predicate first gets from K's database the fact neighbours(K, [N1,...,N4]) stating the list [N1,...,N4] of K's four neighbours. Next, a random number 'Number' is chosen, and taken modulo 4, the result is called Selector. Thus Selector is one of the numbers 0,1,2,3 picked randomly. The next receiver then is defined as neighbour J where J is the value of Selector, i.e. the random number from the set 0,1,2,3.

PREDICATE start.

```
start :-
get_my_ids(Individual,_),
database(neighbours_strength(Individual,NeighboursStrengthList)),
```

```
retract(neighbours_strength(Individual,NeighboursStrengthList)),
insort_list(NeighboursStrengthList,SortedNeighboursStrengthList,'<'),
asserta(neighbours_strength(Individual,SortedNeighboursStrengthList)),
retract(action(off)),
asserta(action(on)).
```

When node K executes 'start', it gets its BRAIN AID identification, which is (K,8). It then gets from its database the list of the strengths of its four neighbours which has been stored in the database by means of (2) and (4) in Sec.A. This list is sorted according to increasing strength, such that the first item in the list is the weakest neighbour. Technically, the ordering is achieved by the predicates defined under (11) above. The original list of strengths is retracted from the database and replaced by the new, sorted one. Then K changes its action mode. The fact 'action(off)' indicating passive mode is replaced by 'action(on)' in the database. K is now in the active mode, and in the repeat-loop of my_action_loop in (9) 'action' will be reached. K now executes the action predicate described below.

PREDICATE sort(voidlist,voidlist).

```
order(N,H) :- integer(N), integer(H), H < N.
sort(L1,L2) :- permutation(L1,L2), sorted(L2), !.
permutation(L,[H|T]) :-
append(V,[H|U],L), append(V,U,W),
permutation(W,T).
permutation([ ],[ ]).

sorted(L) :- sorted(0,L).
sorted(_, [ ]).
sorted(N,[H|T]) :- order(N,H), sorted(H,T).
```

PREDICATE action.

This prediacte describes the main loop which each individual can execute. It uses the above standard predicate for sorting. K is the individual under consideration.

```
action :-
database(strength(K,S)),
database(neighbours_strength(K,List)),
List = [[SN1,N1],[SN2,N2],[SN3,N3],[SN4,N4]],
ran(I1,0,10) , ran(I2,0,9),
I2 - I1 < S - SN1 , asserta(aux(N1,SN1,I1,I2)),
nextto([SN1,N1],[SN2,N2],List),
( I2 - I1 < S - SN2 , asserta(aux(N2,SN2,I1,I2)) ;
S - SN2 ≤ I2 - I1 ),
nextto([SN2,N2],[SN3,N3],List),
( I2 - I1 < S - SN3 , asserta(aux(N3,SN3,I1,I2)) ;
```

```
S - SN3 ≤ I2 - I1 ),
nextto([SN3,N3],[SN4,N4],List),
( I2 - I1 < S - SN4 , asserta(aux(N4,SN4,I1,I2)) ;
S - SN4 ≤ I2 - I1 ),
calculate_values(I1,I2),
findall(Wi1, aux1((Ni,I1,Wi1,I2,Wi2), List1),
sort(List1, List2),
member(X,List2),
aux1(N,I1,W1,I2,W2),
send_msg(msg(N,8),
perform(K,I1,W1,I2,W2)),
database(value(Node,Worth)),
NewWorth is Worth + W1,
retract(value(Node,Worth)),
asserta(value(Node,NewWorth)).

calculate_values(I1,I2) :-
database(aux(Ni,SNi,I1,I2),
( SNi < S , Wi1 is I2 * SNi, Wi2 is 10 - (2 * I2) , asserta(aux1(Ni,I1,Ei1,I2,Wi2)) ;
S ≤ SNi , Wi1 is 10 - (2 * I1) , Wi2 is I1 * S, asserta(aux1(Ni,I1,Wi1,I2,Wi2)) ),
fail.
calculate_values(I1,I2).

PREDICATE perform(integer,integer,integer,integer,integer).

perform(I,I1,W1,I2,W2) :-
database(value(N,W)),
W3 is (W + W2)-I2,
retract(value(N,W)),
asserta(value(N,W3)).
```

An individual K getting this order by a message is told to perform the action-type [I1,W1,I2,W2] in the position of the subordinate agent. K subsequently adjusts its value-account by adding the value W2 of the action-type to its previous value W, and subtracting I2. Individual I is the 'sender' of the order which is not used in the present version.

```
PREDICATE ran(integer,integer,integer).

ran(X,A,Z) :-
Z ≥ A, D is Z - A, M is D / 2, N is D - M,
random(P1), random(P2),
X is (a + (P1 mod (M + 1)) + (P2 mod (N + 1)).
```

References

Balzer, W. 1990: A Basic Model for Social Institutions, *Journal of Mathematical Sociology* 16, 1-29.

Balzer, W. 1993: *Soziale Institutionen*, Berlin: de Gruyter.

Balzer, W. 1992: A Theory of Power in Small Groups, in: H. Westmeyer (ed.), *The Structuralist Program in Psychology*, Bern: Hogrebe, 191-210.

Balzer, W. 1994: Exchange versus Influence: A Case of Idealization, in B.Hamminga & N.B.de Marchi (eds.), *Idealization VI: Idealization in Economics*, Poznan Studies in the Philosophy of the Sciences and the Humanities 38, Amsterdam: Rodopi, 189-203.

Cohors-Fresenborg, E. 1977: *Mathematik mit Kalkülen und Maschinen*, Braunschweig: Vieweg.

Clocksin, W. F. & Mellish, C. S. 1984: *Programming in PROLOG*, Berlin etc.: Springer, 2nd ed.

Derantsart, P., Ed-Dbali, A., Cervoni, L. 1996: *PROLOG: The Standard*, Berlin etc. Springer.

Hegselmann, R. 1994: Solidarität in einer egoistischen Welt: eine Simulation, in: J.Nida-Ruemelin (Hrsg.), *Praktische Rationalität*, Berlin: de Gruyter, 349-90.

Liebrand, W.B.G. & Messick, D. M. 1992: Computer Simulation of the Relationl Between Individual Heuristics and Global Cooperation in Prisoner's Dilemmas, *Social Science Computer Review* 11, 301-12.

Ostermann, M., Bergmann, F. & von Walter, G. 1995: Brain Aid Systems, Dokumentation von BRAIN AID PROLOG, erhältlich von M.Ostermann, Bogenstr.9, 52080 Aachen.

Rao, A. S. & Georgeff, M. P. 1991: Modelling Rational Agents within a BDI Architecture, in J.Allen et al. (eds.), Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufman, 473-84.

SMASS: A SERIAL MULTI-AGENT SYSTEM FOR SOCIAL SIMULATION

Wolfgang Balzer
University of Munich

1 THE GOAL OF SMASS

SMASS is a simple simulation program which can *flexibly* deal with many *different* forms of individual behavior. The combination of these features: simplicity, different rules of behavior for one individual actor, and flexibility for the user to switch between different applications with different rule sets are rarely found in existing programs.

In the literature we find essentially four kinds of social simulations. On the one hand, there are programs -usually working in a cellular automata setting- like (Hegselmann, 1996), (Liebrand & Messick, 1996), (Mark, 1998), (Nowak & Lewenstein, 1996) which investigate the macro effects occurring when all individuals perform one single rule of behavior. Second, there are simulation studies of populations of individuals in the spirit of evolutionary game theory in which all members of a population perform one single rule of behavior, and in which the relative success of such a population in comparison with other populations is at stake (Axelrod, 1984), (Schuessler, 1990). A third, recent approach uses object-oriented software, like (Epstein & Axtell, 1996) or SWARM. Here, different rules ('methods') can be added to each agent ('object'). These systems being implemented in variants of object-oriented C, are difficult to program, and switching between rules is not straightforward. Last not least, there are systems including a full cognitive apparatus for each agent thus providing room for each actor's applying many different rules. Such systems, exemplified best by (Carley & Prietula, 1994) are far from being simple.

In any real social system a person may perform actions of several different kinds, and the person usually has some choice of which kind of action to engage in. In order to become more realistic, social simulations therefore must include the possibility of behaving according to different rules. As this may drive the system towards great complexity and make it difficult to use, special attention has to be given to securing flexibility and simplicity of application. One of the main goals in the design of SMASS is to strike a balance between simplicity and flexibility of changing the rules on the one hand and still cover a space large enough to comprise the most central types of human social interaction on the other hand.

In SMASS each individual commands several different rules of behavior among which it can choose in a given situation. But the user of SMASS is not restricted to those rules which presently are implemented. SMASS is built such that, first, among

a range of implemented rules, the user may choose any arbitrary subset and run a simulation in which the actors are restricted to rules from that subset, and such that, second, it is relatively easy to invent and plug in new rules of behavior without changing SMASS's global 'architecture'. A comparison with object-oriented systems providing similar facilities SWARM, (Uhrmacher, 1997) still has to be made.

Up to now I did not concentrate on finding new, 'interesting' simulation results, I was focusing on the development of the frame program and on reproducing different existing programs in that frame. At the present stage, SMASS is able to simulate systems of actors which can perform quite a number of different kinds of actions but I have not yet carried out systematic simulations with mixed sets of actions. What I did up to now in applying SMASS is to reproduce several existing 'one-rule' studies by simply inactivating all other rules of behavior present in the system. I think that this is enough indication of the potential of SMASS and justifies the description of some of its major components even in the absence of new, 'interesting' simulation results. My long term goal in developing the system is to perform simulation studies of realistic social systems, like social institutions. A comprehensive formal theory of social institutions which can serve as a basis for implementation is described in (Balzer, 1990) and (Balzer, 1993).

2 WHAT TRIGGERS INDIVIDUAL ACTION: BDI VERSUS RULE GOVERNED BEHAVIOR

A widely used approach to the construction of multi-agent systems is the so-called BDI-approach (from belief, desire, intention). In a BDI model each actor is equipped with components representing his or her beliefs, desires and intentions (Cohen & Levesque 1990), (Rao & Georgeff, 1991). I use the term 'BDI-approach' here in a broad sense so as to encompass approaches which do not use all three of these items. The BDI-approach has its sources in 'classical' Bayesian decision theory and in the possible worlds approach of formal logics. Main applications of BDI systems are in the technical construction of robots which can interact with other robots and humans. The BDI approach also is attractive to social scientists for it centrally uses notions which humans themselves use in natural language to reason about their actions.

However, I am not the first to argue that the BDI-model is deficient for the simulation of social systems, and that it has to be enriched by components for purely rule or norm-based behavior (Agre, 1997). For this, there are at least four arguments, a theoretical, a methodological, a practical, and an 'internal' argument.

The theoretical argument focusses on the central 'actor-loop' in which input from the environment is used to produce output-behavior of an agent. In the BDI-model

this loop *always*, i.e. for *each* action, includes some use of the BDI apparatus. In other words, there is no action that was not preceded by some deliberation. This assumption lies at the heart of the rational choice approach. But in sociology there also is another paradigm, that of *homo sociologicus*, which has emerged from work of, for instance, Durkheim and Parsons. *Homo sociologicus* does not always use his BDI apparatus, a great deal of his behavior consists in following rules and norms. The picture roughly is this. An actor has stored many rules and norms which can be applied in respective, specific situations, and which are given together with specific conditions of applicability. Whenever a person finds herself in a situation fitting to some application condition of a rule she will apply that rule and behave accordingly.

These approaches lead to different actor loops. In the 'pure' BDI case the actor in each loop takes a decision according to the principles of Bayesian decision theory (or some weakened version thereof), whereas *homo sociologicus* needs not decide in each loop. In some loops he may simply check which of his rules applies in the respective situation. This check is not a decision, it is a process of fitting or of recognition or an application of a production system.

These two different kinds of loops cannot be realized simultaneously, a person cannot at the same time choose an action on the basis of decision theory and 'choose' the same action by matching the situation against the application conditions of her stored rules. In the second case it is not appropriate to speak of a choice at all. It seems to me that both kinds of behavior (BDI guided and rule guided) are incommensurable in a precise sense (Balzer, 1985), (Balzer,1989). Without going into details, the point is this. In a decision there have to be at least two alternatives among which one can choose whereas in a rule guided situation it may well be that only one application condition of one rule matches against the situation. Defenders of the BDI approach will say that the case in which there is just one alternative still is a case of choice. In defending the sociological paradigm I would say that if there is no alternative there is no choice, and there is no decision. What seems to be a quarrel about the use of words for a trivial borderline case, in fact indicates a central schism between two ways of looking at the social world (Balzer, 1992a). Because of the far reaching implications of looking at things either way there is no easy way to settle this dispute -as we know from the work of Kuhn and Feyerabend.

In spite of the difficulty of this controversy it is not difficult to enrich the actor-loop so that both types of behavior (BDI guided and rule guided) can be realized by the same agent in $different$ situations. The person simply may decide in some loops, and act on the basis of rules in other loops.

The methodological argument againts the pure BDI model points to the well known fact that belief, desire and intention are mental predicates and are not directly observable. Moreover, the theory-guided access to them also is severely restricted and can be achieved only in laboratory situations from which there is no reliable inference to behavior in other, real-life situations. Therefore, the use of BDI components yields a restriction when simulation results are to be compared with real data, for the latter

do not cover the central parts of the BDI-model in a reliable way.

The practical argument consists in pointing to simulation programs, like SMASS, in which rule- and norm-based behavior is primary, and 'rational', BDI features are secondary. The system at present can reproduce many of the existing simulations of cluster- and group-formation on game theoretical or other principles (Hegselmann, 1996), (Mark, 1998), (Nowak & Lewenstein, 1996).

Finally, the 'internal' argument can point to the fact that rule- and/or norm-based behavior has proponents both in sociology (as documented, say, by the classics of Durkheim and Parsons) and increasingly also in DAI, for instance (Altermann & Zito-Wolf, 1993), (Müller et al., 1996), (Wooldridge & Jennings, 1991), (Wooldridge et al., 1995).

In SMASS, a person has no explicit BDI apparatus. Her beliefs, desires and intentions are present only implicitly, namely in her rules of behavior which may -but need not- refer to principles of (bounded) rationality. Primarily, an actor's behavior in SMASS is social-habitual, or 'situated' as some authors call it. Each actor has a character which is given by a list of weights, one for each mode of behavior. If the system is run, say, with 5 modes the character of actor A has the form $[C_1, ..., C_5]$. In each given situation, if the person is in the active state, she chooses one of the five modes (action-types) $M_1, ..., M_5$, say M_i, with probability C_i, and then tries to perform an action of the chosen type according to the rule of behavior which is present for that type. If, for instance, M_1 is 'rest' and C_1 is 0.5 the person is very lazy.

The characters are automatically created by means of discrete distributions whose weights either can also be automatically created or set 'by hand'. In a future version the choice of modes will be achieved by a mixture of character and a set of social norms (see 'check-environment' in the 'kernel' predicate, line (10) in the appendix).

3 ACTIONS: TYPES AND TOKENS

In dealing with a multiplicity of actions of different types the first problem to overcome is given by the token-type distinction. An actor usually can perform different action tokens of one type of action. The type 'exerting power' (Balzer, 1992a) can be realized by many different action tokens, ranging from Hans' beating Fritz here and now to Helmut Kohl's signing the 're-unification' treaty for West-Germany in 1989. The type 'playing a prisoner's dilemma game' can be realized by Marco and Pietro being questioned by the public prosecutor here and now or by my quarrelling with my neighbour about dealing with the garbage ten years ago. The type 'exchange' may be realized by my buying some cherries from grocer Kuhnt at 12.12.1984 or by my signing a document at the notary's office of Thaler in 3.3.1986. In first approximation,

tokens may be seen as 'elements' making up a 'class' which is the 'type'. However, as well known from the philosophy of action this view ultimately is untenable the reason being that the notion of action is soaked with that of propositional attitudes and in the domain of propositional attitudes the principle of extensionality breaks down (Goldman, 1970), (Quine, 1961), (Schiffer, 1987).[1] Put differently, the problem is that one action token under two different descriptions may 'belong to' two different types. A standard example are the two action types 'I wanted to shoot the thief' versus 'I wanted to shoot my drunken friend' both expressing the same token which roughly consists in my intentionally pulling the trigger after having heard noise in my home late at night and having seen some person moving in the dark. One reaction to this opaqueness of action tokens has been to deny them the status of scientifically reputable entities which in turn, and ultimately, would mean to give up any science dealing with human actions.

An alternative reaction is to acknowledge the contested status of action tokens but nevertheless use them with sufficient precaution in social theorizing; this is the philosophy for SMASS. Yet another problem arises at once. The number of action tokens that may be relevant in a social system usually is so large that it is practically impossible to store all possible tokens in a computer's memory. In humans, the tokens are not explicitly represented. Rather, they are created in the course of action and interaction. In the real course of events in which action tokens get realized there is an irreduzible element of chance. No action token in the real world can be completely described before it has taken place. In SMASS, action tokens accordingly are introduced and created in the course of processing and in their creation random elements are used with varying degree depending on the action type and the level of abstraction.

So SMASS uses action types *and* action tokens. The type of an action is given by a certain syntactic format which may vary from action-type to action-type. By contrast, action tokens are created in each situation when the actor has decided to perform an action of a distinct type. The creation of a token of that type is part of the rule which governs that type of behavior. In most cases a token is created by a mixture of random elements and of some deliberation in the way of bounded rationality.

The action type 'exertion of power', for instance, is linked to a syntactic scheme of the form $[A, B, IA, OA, IB, OB]$, with variables A, B, IA, OA, IB, OB. A denotes the person exerting power, B the person over whom power is exerted, IA, OA denote the input and output (integer) values of the interaction for A, and IB, OB the in- and output values for B. If A and B are interpreted, from such a type a token is created by assigning concrete numbers to IA, OA, IB, OB. Intuitivly, A may choose some token in which 'her' input-output relation IO-IA is large irrespective of the values IB, OB, as long as it is feasible for B to perform such a token. If power is exerted, for instance, by A's ordering B to carry a load for A then the value of A's input is just the value, or cost in this case, of her uttering the order while A's output value is the

[1] This insight did not have much impact on AI or so it seems.

value -which we may conceive in monetary terms- of the load being carried. On the other side, B's input value is the value (cost) of his effort or labour of carrying the load, and B's output value is his cost of getting exhausted, tired and loosing part of his lifetime. Of course, representing these 'values' numerically is a strong idealization, but its not worse than other such representations lying at the heart of exchange- and game theory. Integers are used for reasons of simplicity.

4 ACTIVE AND REACTIVE BEHAVIOR

An actor in SMASS is in one of two different states. In the *active* state he is free to perform any action he may choose. In the *reactive* state, she *must* react according to some protocol that has been released in a previous interaction. For instance, in a previous interaction of exchange her partner may have agreed to exchange definite quantities of definite commodities, and may have himself adjusted his state so that, for him, the exchange is done. This partner then has released a protocol telling her that he has done his part and that she now has to do her part. When she finds this protocol in a given situation she *must* act accordingly.

SMASS gives priority to the reactive states (see the predicate for 'kernel', lines 10-13 in the appendix). Whenever an actor is called up in a simulation run at a given time he first checks whether any protocols have been activated for him, that is, he is in the reactive state. When he finds such activated protocols he will execute them and this is all he will do at that time. He gets into the active state only when at the time he is called up no protocols are activated for him. In an extreme case an actor at all times may be the target of many protocols and thus never become really active. Such cases do not seem unrealistic, though.[2]

5 RULES OF BEHAVIOR

The different types of action which the actors can perform in SMASS are called *modes*.

[2] In a first, properly distributed version of SMASS, called DMASS (Balzer & Brendel, 1996) which runs on transputer systems, this case can create a bottleneck because some actors may be waiting for reactions of others who never react due to their being fully engaged in satisfying the wants of other actors.

For each mode SMASS contains a rule of behavior. When an actor is in the active state he will choose a certain mode of behavior. The rule for that mode then is called up and regulates the actor's performance. Typically, a rule of behavior for a given mode comprises different steps which need not be independent of each other. Also, the order in which these steps are carried out may be different for different rules, and in some rules, steps may be missing.

Step 1: The person chooses another actor with whom she will interact; in some cases several other actors are chosen.

Step 2: A token of the mode (action-type) is chosen or created. This is the token which the person ultimately will perform (if she succeeds).

Step 3: She checks whether interaction with the actor(s) chosen in 1) is feasible. In exchange, for instance, if a token is chosen representing the purchase of a certain quantity of a certain good, a partner must have enough of that good.

Step 4: If the preceding steps have succeeded the person 'performs' the action token. She adjusts her own state accordingly, and she releases a protocol that triggers the partner's corresponding reactions. This release is done by noting a fact which activates a particular protocol that is already programmed. When a protocol is activated it will be carried out in future actions by her partners and/or herself.

Each protocol used here is part of a corresponding mode and its code belongs to that for the mode. Various protocols that have been proposed in the literature, like the contract net protocol (Davis & Smith, 1983), SANP (Chang & Woo, 1992) or the proposal in (Kreifelts & v.Martial, 1991) can potentially be used in SMASS in connection with different modes. A further advantage of SMASS is that, being written in PROLOG, the message contents which are handed over in the protocols practically don't need any specific format. Any PROLOG term -which may be a whole PROLOG program itself- may serve as message content.

Not all rules involve a protocol. One rule of behavior presently implemented is 'bodily exercise'. Performing an action of that type simply changes the actor's bodily strength. When all four steps -up to the activation of the protocol- succeed the actor has acted in the given mode, he has performed an action-token of the kind represented by the given mode.

6 UPDATING

SMASS provides means for synchronous and asynchronous updating. Each mode has 'its' own updating procedure which consists of two parts. One part belongs to the core

program and is called up at the end of each period of time (see the 'adjust' examples in the appendix). The second part is implicit in the rules of behavior linked with each mode. The second part may be absent for modes which require strict synchronous updating. For instance, in the mode 'exchange' asynchronous updating is quite natural. After each exchange both partners adjust their endowments. However, even in this case at the end of a period a synchronous updating is called up which replaces each actor's final (in the given period t) endowment by the same (initial) endowment for the following period $t + 1$.[3]

7 FLEXIBLE CHOICE OF SYSTEMS OF RULES

A salient feature of SMASS is its flexibility of picking a set of modes (action-types), and running a simulation with just these modes. This flexibility has two aspects. First, the user may choose any subset of those modes which are already implemented, that is, for which corresponding rules of behavior and adjustment have been programmed. As a special case of this aspect, the user may pick just one single mode, and in this way repeat or vary many of the 'one rule' simulation studies found in the literature. Second, the user may himself formulate new rules of behavior and add them to the system by programming and adding corresponding rules of behavior and update rules. If new variables are introduced which require new initial data, it is also necessary to add these data or some program creating them automatically. Adding a new mode may require from two hours work in simple cases up to several hours or even a couple of days for complicated modes involving complicated protocols, provided the programmer is fit in PROLOG and has got accustomed with SMASS.

SMASS contains a module for administrating the links between the modes on the one hand and the rules of behavior, the rules of updating, and the variables on the other hand. In this module a list of links between the modes chosen by the user and their corresponding rules of behavior, updating, and variables is created when SMASS is started The main program is formulated invariantly relative to these link lists. All procedures in the main programm can work with different such link-lists, and the execution of the main program takes the same form for different lists of links.

Put differently, when the user has chosen a particular set of modes to be simulated, SMASS will create a corresponding link-list, and then run the main program. When, in the next session the user changes the set of modes he wants to use, SMASS

[3] It may be noted that in the distributed version of the system, DMASS, all the problems of updating simply vanish. A kind of complementary problem arising in the distributed system, namely to report the agents' states to some external device at the end of certain real-time periods, is less likely to cause artificial effects, and much easier to handle technically.

creates new link-lists, and with these runs the main program which is the same as before (and in all other sessions).

8 A BRIEF SYNTHESIS

SMASS is written in SWI-PROLOG (Wielemaker, 1993) (which is free under UNIX, for instance in the LINUX package and also can be freely used with permission for academic applications under DOS) and laid down in several files. The 'main' file, SIM, which actually is among the smallest, contains the module for the creation of link-lists and the main program. A second file, $PARA$, contains parameters which must be set by the user before the simulation is started. The parameters include things like the number of runs (for statistics), the number of periods for one simulation run, the number of actors, the type of neighbourhoods used, and/or the number of neighbours, the number of commodities, as well as game parameters (payoffs), gridwidth (for cellular modes), ranges of initial physical strengths of the actors and other ranges for other variables. Most importantly, $PARA$ must contain a list of the modes which are chosen by the user. This is just a list of numbers picked from the total number of modes implemented (16 at present). Also, $PARA$ contains lists of weights for the creation of characters.

For each mode or rule of behavior, the file $RULES$ contains the code governing that rule including code for adjustments in updating as far as there are special adjustment for the rule.

When SIM is started, $PARA$ is consulted, and on the basis of the parameters present in $PARA$, initial data are automatically created. The code for data creation resides in another file $CREATE$ which is consulted when needed. Alternatively, if a file $DATA$ of initial data is provided by the user, $SMASS$ will take the data from $DATA$ and will not create his own data. The data in any case are stored in an external file also called $DATA$. Next the link module creates link-lists for the modes it finds in the $PARA$ file. With these link-lists the main program is started. In the main program, in each statistical run the files $DATA$ and $RULES$ are consulted, and a simulation run is started. A simulation run simulates a fixed number of periods of time (as laid down in $PARA$). In each period, each actor is picked once (in random order) and activated. He checks whether there are protocols for him to perform, and if so, he executes them. If there are no protocols he is set in the active state. He chooses one of the modes (those which the user has picked for the present run) according to his character, and then acts in that mode along the lines of the rule of behavior for that mode which is present in $RULES$. This may involve the release of a protocol to be executed in the next period. In either case, when having reacted to a protocol,

or acted, the actor is done and the next actor is called up. At the end of the period, individual updating is executed for each of the modes as laid down in *RULES* and global updating is made as described in *SIMUL*. When the set number of periods is finished, all facts created in these periods are deleted, and the next statistical run is started. At the end of each simulation run (and if wanted, at the end of each period), the relevant data and new facts are stored in a external file *RESULTS*.

With *RESULTS* one can do the usual statistics and take the output, or *RESULTS* itself, as input for a graphics program for the visualization of the results. I use XPCE (Wielemaker, 1996) for this purpose which also is programmed in PROLOG. Permission to use XPCE under UNIX is free for academic applications, the DOS version costs about 200 dollars.

REFERENCES

Agre, P. 1997: *Computation and Human Experience*, Cambridge/Mass., Cambridge University Press.

Alterman, R. and Zito-Wolf, R. 1993: Agents, Habitats, and Routine Behavior, IJCAI-93, Vol.1, Chambery, France, 305-10.

Axelrod, R. 1984: *The Evolution of Cooperation*, New York: Basic Books.

Balzer, W. 1985: Incommensurability, Reduction, and Translation, *Erkenntnis* 23, 255-67.

Balzer, W. 1989: On Incommensurability, in K.Gavroglu et al. (eds.), *Imre Lakatos and Theories of Scientific Change*, Dordrecht, Kluwer, 287-304.

Balzer, W. 1990: A Basic Model of Social Institutions, *Journal of Mathematical Sociology* 16, 1-29.

Balzer, W. 1992a: Game Theory and Power Theory: A Critical Comparison, in: T. Wartenberg (ed.), *Rethinking Power*, Albany, SUNY Press, 56-78.

Balzer, W. 1992b: A Theory of Power in Small Groups, in H.Westmeyer (ed.), *The Structuralist Program in Psychology*, Bern, Hogrefe and Huber, 191-210.

Balzer, W. 1993: *Soziale Institutionen*, Berlin: de Gruyter.

Balzer, W. and Brendel, K. 1996: DMASS: A Distributed Multi-Agent System for Social Simulation, manuscript.

Carley, K. M. and Prietula, M. J. (eds.) 1994: *Computational Organization Theory*, Hillsdale NJ: Lawrence Erlbaum.

Carley, K. M. and Prietula, M. J. 1994: ACTS Theory: Extending the Model of Bounded Rationality, in (Carley and Prietula, 1994), 55-87.

Cohen, P. R. and Levesque, H. J. 1990: Intention is Choice with Commitment, *Artificial Intelligence* 42, 213-61.

Davis, R. and Smith, R. G. 1983: Negotiation as a Metaphor for Distributed Problem Solving, *Artificial Intelligence* 20, 63-109.

Epstein, J. M. and Axtell, R. 1996: *Growing Artificial Societies*, Cambridge MA: MIT Press.

Gilbert, G. N. and Doran, J. (eds.), 1994: *Simulating Societies*, London: UCL Press.

Goldman, A. I. 1970: *A Theory of Human Action*, Engelwood Cliffs: Prentice Hall.

Hegselmann, R., Mueller, U. and Troitzsch, K. G. (eds.), 1996: *Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View*, Dordrecht: Kluwer.

Hegselmann, R. 1996: Cellular Automata in the Social Sciences, Perspectives, Restrictions, and Artefacts, in (Hegselmann, Mueller and Troitzsch, 1996), 209-33.

Kreifelts, T. and von Martial, F. 1991: A Negotiation Framework for Autonomous Agents, in Y.Demazeau and J.-P.Müller (eds.), *Decentralized A.I.-2*, 71-87.

Liebrand, W. B. G. and Messick, D. M. 1996: Computer Simulations of Sustainable Cooperation in Social Dilemmas, in (Hegselmann, Mueller and Troitzsch, 1996), 235-47.

Chang, M. K. and Woo, C. C. 1992: SANP: A Communication Level Protocol for Negatioations, in E.Werner and Y.Demazeau (eds.) *Decentralized A.I.-3*, Amsterdam: Elsevier, 31-54.

Mark, N. 1996: Beyond Individual Differences: Social Differentiation from First Principles, paper read at the 1996 ASA conference, New York.

Mueller, J. P., Wooldridge, M. J. and Jennings, N. R. (eds.), 1996: *Intelligent Agents III*, Berlin: Springer.

Nowak, A. and Lewenstein, M. 1996: Modelling social Change with Cellular Automata, in (Carley and Prietula, 1994), 249-85.

Quine, W. v. O. 1961: Two Dogmas of Empiricism, in W. v. O. Quine, *From a Logical Point of View*, Cambridge/Mass.: Harvard University Press, 20-46.

Rao, A. S. and Georgeff, M. P. 1991: Modelling Rational Agents within a BDI-Architecture, in R.Fikes, E.Sandewall (eds.), *Proceedings of Knowledge Representation and Reasoning* (KR+R-91), Morgan Kaufmann, 473-84.

Schelling, T. C. 1971: Dynamic Models of Segregation, *Journal of Mathematical Sociology* 1, 143-86.

Schiffer, R. 1987: *Remnants of Meanings*, Cambridge/Mass.: MIT Press.

Schüssler, R. 1990: *Kooperation unter Egoisten: Vier Dilemmata*, München: Oldenbourg.

SWARM: http://www.santafe.edu/projects/swarm/.

Uhrmacher, A. M. 1997: Concepts of Object- and Agent-oriented Simulation, *Transactions of the Society for Computer Simulation International* 14, 59-68.

Wielemaker, J. 1993: SWI-Prolog 1.8, Reference Manual, University of Amsterdam, Dept. of Social Science Informatics.

Wielemaker, J. 1996: Programming in XPCE/Prolog, University of Amsterdam, Dept. of Social Science Informatics.

Wooldridge, M. J. and Jennings, N. R. (eds.), 1991: *Intelligent Agents*, Berlin: Springer.

Wooldridge, M., Mueller, J. P. and Taube, M. (eds.), 1995: *Intelligent Agents II*, Berlin: Springer.

APPENDIX: AN MINIATURE EXAMPLE

Here is the code of SMASS restricted to the application of three very simple rules. All features of the handling of data output and technical details pertaining to test and debugging are omitted as well as the automated creating of link lists mentioned in Sec.8. Compare (Wielemaker, 1993) for explanations of the built-in PROLOG predicates which can not be given here. Loading the different files in a directory accessible to PROLOG, compiling SIM, and entering 'start' plus return should make SMASS running. The results can be seen in a file $RESULTS$ that will be newly created by SMASS, and the data which were created and used should be present in a new file $DATA$. Changing the list of arguments of the 'modes' predicate in line /* 1 */ will produce simulations in which only those rules present by names in that list are applied.

% File $PARA$ (parameters for SMASS)

```
/* 1 */ modes([takeweak,donothin,schellin]). runs(2). periods(10).
actors(47). use_old_data(no). gridwidth(8).
variables_in_rule(donothin,[wealth]).
variables_in_rule(schellin,[location_schelling,schelling_colour]).
variables_in_rule(takeweak,[location,neighlist,strength,wealth_weak]).
weights(donothin,1,[101]). weights(schellin,1,[101]).
weights(takeweak,1,[101]).
% modes([takeweak,donothin,schellin]) in line (1) specifies the list
% L=[takeweak,donothin,schellin] of modes (action-types)
% which are used in the present simulation.
```

% File SIM (core program)

```
/* 2 */ start :- consult(para), consult(pred), consult(rules),
( delete_file(results) ; true), use_old_data(X),
( X = no, create_data; X = yes, consult(data) ), begin.
create_data :- ( delete_file(data) ; true ), consult(createex),
modes(L), make_global_data(L), make_variable_list(L,L1),
length(L1,E), ( between(1,E,N), nth1(N,L1,VAR), make(VAR), fail
; true ), !.
make_global_data(L) :- actors(AS), make_characters(AS,L).
make_variable_list(L,L1) :- asserta(variable_list([])), length(L,E),
( between(1,E,X), nth1(X,L,M), variables_in_rule(M,L2),
build_variable_list(L2), fail ; true ), variable_list(L1),!.
build_variable_list(L2) :- variable_list(L), append(L,L2,L3),
retract(variable_list(L)), asserta(variable_list(L3)),!.
```

```
% R denotes a statistical run, T a period of time. RR is the number of
% statistical runs, TT that of periods of time for each simulation run,
% AS the fixed number of actors. RESULTS is the external file in
% which the raw data are written. All dynamical variables (those whose
% values may change during a simulation run) are written in the format
% fact(R,T,var(X1,...,Xn)) where R,T are as above, and 'var' varies in
% the names of variables attached to the given modes (like 'wealth',
% 'strength' etc.). The loop in (4) produces TT executions of 'kernel',
% that is, one simulation run covering TT periods. After each such run
% in (4) all dynamical facts are deleted, and the original data are
% reconsulted in (3) for the next run.
```

```
% (7) loops over all actors. In each step one actor A is randomly drawn
% from list L (6) and activated. After 'activate(R,T,A)' (9) is
% executed actor A had his opportunity in period T as described in the
% following, A is deleted from the actor list (10), and another actor is
% called up in (7).
```

```
begin :- runs(RR), periods(TT),
( between(1,RR,R), mainloop(R,TT), fail; true ), !.
mainloop(R,TT) :-
/* 3 */ consult(data), findall(X,fact(0,0,X),L), length(L,E),
( between(1,E,Z), nth1(Z,L,FACT), append(results),
write(fact(R,1,FACT)), write('.'), nl, told, retract(fact(0,0,FACT)),
asserta(fact(R,1,FACT)), fail; true ), append(results), nl, told,
/* 4 */ ( between(1,TT,T), kernel(R,T), fail; true ),
/* 5 */ retract_facts,!.
retract_facts :- ( fact(X,Y,Z), retract(fact(X,Y,Z)), fail; true ).
kernel(R,T) :- actors(AS), findall(I,between(1,AS,I),L),
/* 6 */ asserta(actor_list(L)),
```

```
/* 7 */ ( between(1,AS,N), choose_and_activate_actor(R,T,N), fail;
true ), retract(actor_list(L1)),
/* 8 */ adjust(R,T),!.
choose_and_activate_actor(R,T,N) :- actor_list(L), length(L,E),
/* 9 */ Y is random(E)+1, nth1(Y,L,A), activate(R,T,A), delete(L,A,L1),
retract(actor_list(L)), asserta(actor_list(L1)),!.

% When actor A gets activated he first checks his environment (10). This
% yields a possibility for immediate reactions to external (non-social)
% changes which are not implemented in the present version. Next, A
% executes protocols (11), if there are such for her. The protocols
% 'protocol(M,A,R,T)' for all modes M are found in the RULES file, and
% succeed only if a previous entry has been made in the fact base
% signalling that the protocol should be executed by A in the next
% period. If no protocols are activated in this way A switches to the
% active state. She chooses a mode M and acts in that mode (12), (13).
% The predicates 'act_in_mode(M,A,R,T)' are found in the file RULES.
% When in a given period all actors have 'acted' once, in (8) the
% adjustment is called up. First, in (14) for each mode an individual
% adjustment is made, if necessary. The corresponding predicates are
% found in the RULES file. Second, in (15) a global adjustment is made.
% This includes writing all facts present in the RESULTS file.

/* 10 */ activate(R,T,A) :- check_environment(R,T,A),
/* 11 */ ( execute_protocols(R,T,A)
;
/* 12 */ choosemode(R,T,A,M),
/* 13 */ ( act_in_mode(M,A,R,T) ; true)
),!.
check_environment(R,T,A) :- true.
/* 11 */ execute_protocols(R,T,A) :- protocol(M,A,R,T).
/* 8 */ adjust(R,T) :- modes(L), length(L,E), actors(AS),
/* 14 */ ( between(1,E,X), individual_adjust(X,R,T,AS,L), fail; true ),
/* 15 */ global_adjust(R,T), append(results), nl, told,!.
individual_adjust(X,R,T,AS,L) :- nth1(X,L,Z),
/* 14 */ ( between(1,AS,A), adjust(Z,A,R,T), fail ; true),!.
/* 15 */ global_adjust(R,T) :- T1 is T+1, repeat,
( fact(R,T,FACT), retract(fact(R,T,FACT)), asserta(fact(R,T1,FACT)),
append(results), write(fact(R,T1,FACT)), write('.'), nl, told,
fail; true ),!.

% The character C of A is found in the file DATA which was downloaded
% in (2).
```

```
choosemode(R,T,A,M) :- fact(R,T,character(A,C,SUM)), length(C,K),
modes(L), Z is random(SUM*1000)+1, asserta(aux_sum(0)),
between(1,K,X), do1(X,Z,C,Y), Z =< Y , nth1(X,L,M),
retract(auxsum(SS)),!.
do1(X,Z,C,Y) :- aux_sum(S), nth1(X,C,C_X), Y is S + (C_X * 1000),
retract(aux_sum(S)), asserta(aux_sum(Y)),!.
```

% File *PRED* (auxiliary predicates)

% Create random numbers normally or discretely distributed, as well as
% von Neumann- and Moore neighbourhoods.

```
normal_distribution(N,AS,L,U,SI) :- MU is L + (0.5 * (U-L)),
( between(1,AS,A), determine_nd_value(N,MU,SI,L,U,A), fail; true),!.
determine_nd_value(N,MU,SI,L,U,A) :- repeat, X is random(10001)+1,
X4 is (1/10000)*(((X-1)*U)+(10001-X)*L), W is integer(X4),
PI is pi, X1 is 2*(PI*(SI*SI)), X2 is (1 / sqrt(X1)),
X3 is (-((W-MU)*(W-MU))) / (SI*SI), Y is X2 * exp(X3),
W1 is random(10001)+1, Z is (W1-1)/10000, Z =< Y, between(L,U,W),
asserta(nd_expr(N,A,W)), !.
make_discrete_distribution(N,AS,EX,LIST) :-
( between(1,AS,A), determine_dd_value(N,A,EX,LIST), fail; true ),!.
determine_dd_value(N,A,EX,LIST) :- X is random(100)+1,
between(1,EX,Z), nth1(Z,LIST,W_Z), X < W_Z, assert( dd_expr(N,A,Z)),!.
calculate_sum(L,S) :- asserta(counter(0)), length(L,E),
( between(1,E,X), auxpred(L,X) , fail ; true), counter(S),
retract(counter(S)).
auxpred(L,X) :- nth1(X,L,N), counter(C), C1 is C+N,
retract(counter(C)), asserta(counter(C1)), !.
make_nbh(moore,N,I,J,L) :- gridwidth(G), ( N=1, moore_nbh_1(G,I,J,L)
; 1 < N, moore_nbh_1(G,I,J,L2), asserta(auxlist(I,J,L2)),
length(L2,K), ( between(1,K,X), mnbh(X,L2,N,G,I,J), fail ; true),
auxlist(I,J,L5), delete(L5,[I,J],L6), sort(L6,L),
retractall(auxlist(A,B,L8))
),!.
mnbh(X,L2,N,G,I,J) :- nth1(X,L2,Y), Y=[I1,J1], N1 is N-1,
make_nbh(moore,N1,I1,J1,L3), auxlist(I,J,L4), append(L4,L3,L5),
retract(auxlist(I,J,L4)), asserta(auxlist(I,J,L5)), !.
moore_nbh_1(G,I,J,L) :- recalculate_neg(G,I,1,Im),
recalculate_neg(G,J,1,Jm), recalculate_pos(G,I,1,Ip),
recalculate_pos(G,J,1,Jp),
L = [ [I,Jm],[Im,Jm],[Im,J],[Im,Jp],[I,Jp],[Ip,Jp],[Ip,J],[I,Jm] ].
recalculate_neg(G,I,H,I1) :- X is I-H, ( ( 0 < X, I1 is X
; 0 =:= X, I1 is G ) ; X < 0, I1 is (G+I) - H ),!.
```

```
recalculate_pos(G,I,H,I1) :- X is I+H, ( ( I < G, X =< G, I1 is X
; I =:= G, (H > 0, I1 is H; H =:= 0, I1 is G) ) ; I < G, X > G,
I1 is (H+I)-G ),!.
make_nbh(von_Neumann,N,I,J,L) :- gridwidth(G), ( N=1,
von_Neumann_nbh_1(G,I,J,L) ; 1 < N, von_Neumann_nbh_1(G,I,J,L2),
asserta(auxlist(I,J,L2)), length(L2,K),
( between(1,K,X), vNnbh(X,L2,N,G,I,J), fail ; true),
auxlist(I,J,L5), delete(L5,[I,J],L6), sort(L6,L),
retractall(auxlist(A,B,L8)) ),!.
vNnbh(X,L2,N,G,I,J) :- nth1(X,L2,Y), Y=[I1,J1], N1 is N-1,
make_nbh(von_Neumann,N1,I1,J1,L3), auxlist(I,J,L4), append(L4,L3,L5),
retract(auxlist(I,J,L4)), asserta(auxlist(I,J,L5)), !.
von_Neumann_nbh_1(G,I,J,L) :- recalculate_neg(G,I,1,Im),
recalculate_neg(G,J,1,Jm), recalculate_pos(G,I,1,Ip),
recalculate_pos(G,J,1,Jp), L = [ [I,Jm],[Im,J],[I,Jp],[Ip,J] ].
decompose(Y,I,J,G) :- between(1,G,Z), Y =< Z*G, Z1 is Z-1, I is Z,
J is Y-(Z1*G),!.
```

% File *CREATE* (generates characters and data)

% The characters for the actors are created and written to the *DATA*
% file. For each variable initial data are created and written to the
% *DATA* file.

```
make_characters(AS,L) :- build_up_characters(AS,L), export_results(AS).
build_up_characters(AS,L) :- length(L,E),
( between(1,E,X), make_distribution(X,L,E,AS), fail ; true ),
( between(1,AS,A), collect_characters(L,E,A), fail ; true),
retractall(dd_expr(M1,M2,M3)), !.
make_distribution(X,L,E,AS) :- nth1(X,L,M), weights(M,EX,LIST),
make_discrete_distribution(M,AS,EX,LIST), retract(weights(M,EX,LIST)),!.
collect_characters(L,E,A) :- asserta(character(A,[])),
( between(1,E,X), nth1(X,L,M), add_character(M,A), fail; true ),!.
export_results(AS) :- ( between(1,AS,A), export(A), fail; true),!.
export(A) :- character(A,L2), calculate_sum(L2,SUM), append(data),
write(fact(0,0,character(A,L2,SUM))), write('.'), nl, told,
retract(character(A,L2)),!.
add_character(M,A) :- dd_expr(M,A,C), character(A,L1), append(L1,[C],L2),
retract(character(A,L1)), asserta(character(A,L2)),!.
make(wealth) :- actors(AS), domain_of_wealths(L,U), sigma_wealths(SI),
normal_distribution(wealth,AS,L,U,SI),
( between(1,AS,A), nd_expr(wealth,A,W), append(data),
write(fact(0,0,wealth(A,W))), write('.'), nl, told,
retract(nd_expr(wealth,A,W)), fail ; true ),!.
```

```
make(wealth_weak) :- actors(AS), domain_of_wealth_weak(L,U),
sigma_wealth_weak(SI), normal_distribution(wealth_weak,AS,L,U,SI),
( between(1,AS,A), nd_expr(wealth_weak,A,W), append(data),
write(fact(0,0,wealth_weak(A,W))), write('.'), nl, told,
retract(nd_expr(wealth_weak,A,W)), fail ; true ),!.
make(strength) :- actors(AS), weights(strength,LIST),
expressions(strength,EX),
make_discrete_distribution(strength,AS,EX,LIST),
( between(1,AS,A), dd_expr(strength,A,W), append(data),
write(fact(0,0,strength(A,W))), write('.'), nl, told,
retract(dd_expr(strength,A,W)), fail; true ),!.
make(location) :- actors(AS), gridwidth(G), G1 is G * G,
findall(X,between(1,G1,X), L), asserta(cell_list(L)),
( between(1,AS,A), locate(A), fail ; true), retractall(cell_list(L2)),!.
locate(A) :- cell_list(L), length(L,E), X is random(E)+1, nth1(X,L,Y),
gridwidth(G), decompose(Y,I,J,G), append(data),
write(fact(0,0,location(A,I,J))), write('.'), nl, told,
asserta(fact(0,0,location(A,I,J))), delete(L,Y,L1),
retract(cell_list(L)), asserta(cell_list(L1)),!.
make(neighlist) :- type_of_neighbourhood(TYPE,DEGREE), actors(AS),
( between(1,AS,A), make_neighbourhood(A,TYPE,DEGREE), fail ; true).
make_neighbourhood(A,T,D) :- fact(0,0,location(A,I,J)),
make_nbh(T,D,I,J,L), length(L,E), asserta(aux_list(A,[ ])),
( between(1,E,X), collect_neighbours(A,X,L), fail ; true),
aux_list(A,L2), sort(L2,L3), append(data),
write(fact(0,0,neighlist(A,L3))), write('.'), nl, told,!.
collect_neighbours(A,X,L) :- nth1(X,L,Z), Z=[I,J],
fact(0,0,location(N,I,J)), aux_list(A,L1), append(L1,[N],L2),
retract(aux_list(A,L1)), asserta(aux_list(A,L2)),!.
make(location_schelling) :- gridwidth(G), actors(AS),
L=[ [1,1],[1,G],[G,1],[G,G] ], asserta(auxlist([ ])),
( between(1,AS,A), schelling_locate(A,G,L), fail; true),
retractall(auxlist(LL)),!.
schelling_locate(A,G,L) :- auxlist(L1), repeat, I is random(G)+1,
J is random(G)+1, not member([I,J],L1), not member([I,J],L),
asserta(fact(0,0,schelling_loc(A,I,J))), append(data),
write(fact(0,0,schelling_loc(A,I,J))), write('.'), nl, told,
append(L1,[ [I,J] ],L2), retract(auxlist(L1)), asserta(auxlist(L2)),!.
make(schelling_colour) :- actors(AS),
( between(1,AS,A), set_colour(A), fail ; true),!.
set_colour(A) :- fact(0,0,schelling_loc(A,I,J)), N is I+J,
N1 is N mod 2, ( N1 =:= 0, append(data),
```

```
write(fact(0,0,colour(A,white))), write('.'), nl, told;
append(data), write(fact(0,0,colour(A,black))), write('.'), nl,
told ), !.

% File RULES

% RULE 1: 'donothin'. The person intentionally does not do anything. A
% fixed sum (3 * E) is deducted from her wealth in each period.

domain_of_wealths(50,500). sigma_wealths(20). exist_min(20).
act_in_mode(donothin,A,R,T) :- feasible(donothin,A,R,T),
chooseaction(donothin,A,R,T), perform(donothin,A,R,T),!.
feasible(donothin,A,R,T) :- fact(R,T,wealth(A,W)), exist_min(E),
E1 is 3 * E, W1 is W-E1, W1 > 0,!.
chooseaction(donothin,A,R,T) :- true,!.
perform(donothin,A,R,T) :- fact(R,T,wealth(A,W)), W1 is W-5,
retract(fact(R,T,wealth(A,W))), asserta(fact(R,T,wealth(A,W1))),!.
protocol(donothin,A,R,T) :- fail.
adjust(donothin,A,R,T):- true.

% RULE 2: 'schellin'.
% One of the first programs leading to emergent patterns, here: the
% clustering of persons of equal colour. For explanation consult
% (Schelling,1971).

act_in_mode(schellex,A,R,T) :- feasible(schellex,A,R,T),
chooseaction(schellex,A,R,T), perform(schellex,A,R,T),!.
feasible(schellex,A,R,T) :- true.
chooseaction(schellex,A,R,T) :- gridwidth(G),
fact(R,T,schelling_loc(A,I,J)), scan_neighbourhood(A,G,I,J,R,T,ANSWER),
( ANSWER=yes ; calculate_move(A,R,T,G) ),!.
scan_neighbourhood(A,G,I,J,R,T,ANSWER) :- make_nbh(moore,1,I,J,L),
findall(N,neighb(N,L,R,T),L1), length(L1,E1),
findall(N1, equal_colour(N1,A,L,R,T), L2), length(L2,E2),
( ( ( E1 =< 2, 1 =< E2; 3 =< E1, E1 =< 5, 2 =< E2 )
; 6 =< E1, E1 =< 8, 5 =< E2 ), ANSWER=yes ; ANSWER=no),!.
neighb(N,L,R,T) :- member([I,J],L), fact(R,T,schelling_loc(N,I,J)).
equal_colour(N,A,L,R,T) :- member([I,J],L), fact(R,T,schelling_loc(N,I,J)),
fact(R,T,colour(N,CN)), fact(R,T,colour(A,CA)), CA=CN.
calculate_move(A,R,T,G) :- G1 is G * G, between(1,G1,X),
decompose(X,I,J,G), not fact(R,T,occupied(B,I,J)),
not fact(R,T,schelling_loc(B1,I,J)),
scan_neighbourhood(A,G,I,J,R,T,ANSWER), ANSWER=yes,
asserta(fact(R,T,occupied(A,I,J))),!.
perform(schellex,A,R,T) :- true.
```

```
protocol(schellex,A,R,T) :- fail.
adjust(schellex,A,R,T) :-
( fact(R,T,occupied(A,I,J)), fact(R,T,schelling_loc(A,IA,JA)),
retract(fact(R,T,schelling_loc(A,IA,JA))),
asserta(fact(R,T,schelling_loc(A,I,J))),
retract(fact(R,T,occupied(A,I,J)))
; true
),!.

% RULE 3: 'takeweak' (take from the weaker).
% Each actor tries to find a neighbour which is physically weaker, and
% to take away some part of that persons wealth. The amount taken away
% is randomly chosen from a pre-specified range (3 * SS). In (17) a
% protocol is formulated which will be performed by the addressee in
% the next period. The message content for that protocol is 'handed
% over' by asserting it in (16) and by being read in (17) by the
% addressee. More complicated protocols basically work in the same way,
% all necessary regulations being described and handed over (perhaps
% several times) as 'messages' in the way of the example.

exist_min_weak(20). domain_of_values(20). domain_of_wealth_weak(100,500).
sigma_wealth_weak(40). expressions(strength,4).
weights(strength,[10,50,90,100]). type_of_neighbourhood(moore,1).
act_in_mode(takeex,A,R,T) :- feasible(takeex,A,R,T),
chooseaction(takeex,A,R,T), perform(takeex,A,R,T).
feasible(takeex,A,R,T) :- exist_min_weak(MIN),
domain_of_values(SS), fact(R,T,neighlist(A,L)), length(L,E),
fact(R,T,strength(A,SA)), between(1,E,X),
investigate(R,T,X,L,MIN,SA,SS).
investigate(R,T,X,L,MIN,SA,SS) :- nth1(X,L,N), fact(R,T,strength(N,SN)),
SN < SA, fact(R,T,wealth_weak(N,WN)), W1 is WN-(3 * SS), !, MIN =< W1,
T1 is T+1, not fact(R,T1,give_the_stronger(N,Y)),
asserta(neighb(N,WN)).
chooseaction(takeex,A,R,T) :- true.
perform(takeex,A,R,T) :- neighb(N,WN), retract(neighb(N,WN)),
fact(R,T,wealth_weak(A,WA)), domain_of_values(SS), S1 is 3 * SS,
X is random(S1), WA1 is WA+X, retract(fact(R,T,wealth_weak(A,WA))),
asserta(fact(R,T,wealth_weak(A,WA1))), T1 is T+1,
/* 16 */ asserta(fact(R,T1,give_the_stronger(N,X))).
/* 17 */ protocol(takeex,A,R,T) :- fact(R,T,give_the_stronger(A,X)),
fact(R,T,wealth_weak(A,WA)), W1 is WA-X,
retract(fact(R,T,wealth_weak(A,WA))),
asserta(fact(R,T,wealth_weak(A,W1))),
retract(fact(R,T,give_the_stronger(A,X))).
```

```
adjust(takeex,A,R,T) :- true.
```

SMASS.2 - A Sequential Multi Agent Social Simulation System

An updated version

I found in the paper from 2000 a 'bug'. One line in the file 'para' was not printed in the paper, namely 'type_of_neighbourhood(moore,1).' As my program had this line, and therefore ran smoothly, it was not a real bug but only a 'type-setter' bug. Sorry, but this was not my fault.

In the last years the SWI program was several times updated, some commands were changed. So I had to change the SMASS program in some places. I changed some names of the predicates for better reading.

I added some small graphic programs. So the results of a simulation can be seen in some pictures. The graphic programs could be substantially shortened.

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```
:- multifile fact/3 .

start:-
consult(para), consult(pred), consult(rules), consult(display),
check_inconsistent_parameters,
( exists_file(results), delete_file(results)
;
true
),
( exists_file(data), delete_file(data)
;
true
),
modes(L), make_list_of_variables(L,LV),
make_reduced_list_of_variables(L,LV,RLV),
use_old_data(X),
( X = no, create_data(L,RLV)
; X = yes, consult(data)
),
begin.

check_inconsistent_parameters :-
    actors(AS), gridwidth(G), G1 is G*G,
    ( AS =< G1
    ;
    G1 < AS, write(too_many_actors_for_the_grid), fail
    ).
```

```
make_list_of_variables(L,LV) :-
    asserta(list_of_variables([])), length(L,E),
    ( between(1,E,NM), build_list_of_variables(NM,L), fail ; true ),
    list_of_variables(LV), retract(list_of_variables(LV)),!.

build_list_of_variables(NM,L) :-
    nth1(NM,L,M), variables_in_rule(M,LM),
    list_of_variables(LVdyn), append(LVdyn,LM,Lnew),
    retract(list_of_variables(LVdyn)), asserta(list_of_variables(Lnew)),!.

make_reduced_list_of_variables(L,LV,RLV) :-
    length(LV,EV), asserta(list_of_variables([])),
    ( between(1,EV,NV), minimize_list_of_variables(NV,LV), fail ; true),!,
    list_of_variables(RLV), retract(list_of_variables(RLV)),!.

minimize_list_of_variables(NV,LV) :-
    list_of_variables(LVdyn),
    nth1(NV,LV,V),
    ( member(V,LVdyn)
    ;
    append(LVdyn,[V],LVnew),
    retract(list_of_variables(LVdyn)),
    asserta(list_of_variables(LVnew))
    ),!.

create_data(L,RLV) :-
    consult(create),
    make_global_data(L),
    length(RLV,EV),
    ( between(1,EV,NV), nth1(NV,RLV,VAR), make(VAR), fail ; true ),
    retractall(int_loc(WW1,WW2,WW3)), /* should be generalized */
    !.

make_global_data(L) :- actors(AS), make_characters(AS,L).
```

```
begin :-
    runs(RR), periods(TT),
    ( between(1,RR,R), mainloop(R,TT), fail; true ), !,
    make_pictures.

mainloop(R,TT) :-
    consult(data),
    findall(X,fact(0,0,X),L), length(L,E),
    ( between(1,E,Z), nth1(Z,L,FACT), append(results),
    write(fact(R,1,FACT)), write('.'), nl, told,
    asserta(fact(R,1,FACT)), retract(fact(0,0,FACT)), fail
```

```
    ; true
    ), append(results), nl, told,
    ( between(1,TT,T), kernel(R,T), fail; true ),
    retract_facts,!.
retract_facts :- ( fact(X,Y,Z), retract(fact(X,Y,Z)), fail; true ).
kernel(R,T) :-
    prepare_indivi(R,T),
    actors(AS), findall(I,between(1,AS,I),L),
    asserta(actor_list(L)),
    ( between(1,AS,N), choose_and_activate_actor(R,T,N), fail
    ;
    true
    ), retract(actor_list(L1)),
    adjust(R,T),!.
prepare_indivi(R,T) :-
    modes(LM), length(LM,EM),
    ( between(1,EM,NM), nth1(NM,LM,MOD), prepare(R,T,MOD), fail ; true),!.
choose_and_activate_actor(R,T,N) :-
    actor_list(L), length(L,E),
    Y is random(E)+1, nth1(Y,L,A), activate(R,T,A), delete(L,A,L1),
    retract(actor_list(L)), asserta(actor_list(L1)),!.
activate(R,T,A) :- check_environment(R,T,A),
    ( execute_protocols(R,T,A)
    ;
    choosemode(R,T,A,M),
    ( act_in_mode(M,A,R,T) ; true)
    ),!.
check_environment(R,T,A) :- true.
execute_protocols(R,T,A) :-
    protocol(M,A,R,T).
adjust(R,T) :- modes(L), length(L,E), actors(AS),
    ( between(1,E,X), individual_adjust(X,R,T,AS,L), fail; true),
    global_adjust(R,T), append(results), nl, told,!.
individual_adjust(X,R,T,AS,L) :-
    nth1(X,L,Z),
    ( between(1,AS,A), adjust(Z,A,R,T), fail ; true),!,
    adjust(Z,R,T).
global_adjust(R,T) :-
    T1 is T+1, repeat,
    ( fact(R,T,FACT), retract(fact(R,T,FACT)), asserta(fact(R,T1,FACT)),
```

```
    append(results), write(fact(R,T1,FACT)), write('.'), nl, told,
    fail
    ; true
    ),!.
choosemode(R,T,A,M) :-
    fact(R,T,character(A,C,SUM)), length(C,K),
    modes(L), Z is random(SUM * 1000)+1, asserta(aux_sum(0)),
    between(1,K,X), do1(X,Z,C,Y), Z =< Y , nth1(X,L,M),
    retract(aux_sum(SS)),!.
do1(X,Z,C,Y) :-
    aux_sum(S), nth1(X,C,C_X), Y is S + (C_X * 1000),
    retract(aux_sum(S)), asserta(aux_sum(Y)),!.
```

```
make_pictures :- modes(LM), length(LM,EM),
    consult(results),
    ( between(1,EM,NM), nth1(NM,LM,MOD), make_picture(MOD), fail ; true),!.
```

```
% create.pl
% This module generates characters and data in SMASS
```

```
:- dynamic fact/3 .
make_characters(AS,L) :-
    build_up_characters(AS,L), export_results(AS).
build_up_characters(AS,L) :-
    length(L,E),
    ( between(1,E,X), make_distribution(X,L,E,AS), fail ; true ),
    ( between(1,AS,A), collect_characters(L,E,A), fail ; true),
    retractall(dd_expr(M1,M2,M3)), !.
make_distribution(X,L,E,AS) :-
    nth1(X,L,M), weights(M,EX,LIST),
    make_discrete_distribution(M,AS,EX,LIST), !.
collect_characters(L,E,A) :-
    asserta(character(A,[])),
    ( between(1,E,X), nth1(X,L,M), add_character(M,A), fail; true ),!.
add_character(M,A) :-
    dd_expr(M,A,C), character(A,L1), append(L1,[C],L2),
    retract(character(A,L1)), asserta(character(A,L2)),!.
```

```
export_results(AS) :-
    ( between(1,AS,A), export_res(A), fail; true),!.
export_res(A) :-
    character(A,L2), calculate_sum(L2,SUM),
    append(data),
    write(fact(0,0,character(A,L2,SUM))), write('.'), nl, told,
    retract(character(A,L2)),!.
```

```
make(wealth_dono) :-
    actors(AS), domain_of_wealth_dono(L,U),
    sigma_wealth_dono(SI), normal_distribution(wealth_dono,AS,L,U,SI),
    ( between(1,AS,A), nd_expr(wealth_dono,A,W), append(data),
    write(fact(0,0,wealth_dono(A,W))), write('.'), nl, told,
    retract(nd_expr(wealth_dono,A,W)), fail
    ; true
    ),!.
make(wealth_take) :-
    actors(AS), domain_of_wealth_take(L,U),
    sigma_wealth_take(SI), normal_distribution(wealth_take,AS,L,U,SI),
    ( between(1,AS,A), nd_expr(wealth_take,A,W), append(data),
    write(fact(0,0,wealth_take(A,W))), write('.'), nl, told,
    retract(nd_expr(wealth_take,A,W)), fail
    ; true
    ),!.
make(strength) :-
    actors(AS), weights(strength,LIST),
    expressions(strength,EX),
    make_discrete_distribution(strength,AS,EX,LIST),
    ( between(1,AS,A), dd_expr(strength,A,W), append(data),
    write(fact(0,0,strength(A,W))), write('.'), nl, told,
    retract(dd_expr(strength,A,W)), fail
    ; true
    ),!.
make(location) :-
    actors(AS), gridwidth(G), G1 is G*G,
    findall(X,between(1,G1,X), L), asserta(cell_list(L)),
    ( between(1,AS,A), locate(A), fail ; true),
    retractall(cell_list(L2)),!.
locate(A) :-
    cell_list(L), length(L,E), X is random(E)+1, nth1(X,L,Y),
```

```
    gridwidth(G), decompose(Y,I,J,G),
    append(data), write(fact(0,0,location(A,I,J))), write('.'), nl, told,
    asserta(int_loc(A,I,J)),
    delete(L,Y,L1),
    retract(cell_list(L)), asserta(cell_list(L1)),!.
make(colour) :-
    actors(AS),
    ( between(1,AS,A), set_colour(A), fail ; true),!.
set_colour(A) :-
    int_loc(A,I,J),
    N is I+J,
    N1 is N mod 2, append(data),
    ( N1 =:= 0, write(fact(0,0,colour(A,white))), write('.'), nl
    ;
    write(fact(0,0,colour(A,black))), write('.'), nl
    ), told, nl, !.
```

```
% display.pl
% The programs displaying some actions relativ to a given action type.
```

```
:- dynamic fact/3 .
make_picture(donothin) :-
    VV1 = 'Picture-', concat(VV1,donothin,Vd1),
    new(@donothin, picture(Vd1)), send(@donothin, open),
    asserta(object_list(donothin,[])), height(donothin,H),
    send(@donothin, display, new(AXISUP, line(5,5,5,175,first))),
    send(@donothin, display, new(AXISRIGHT, line(5,175,350,175,second))),
    periods(TT),
    make_objects(donothin,TT),
    make_other_objects(donothin),
    activate_picture(donothin,R,H). /* NM = 1, MOD = donothin, R a run */
make_objects(donothin,TT) :-
    height(donothin,H),
    ( between(1,TT,A), make_dots(donothin,A,H), fail; true),!.
make_dots(donothin,A,H) :-
    A1 is 5+5*A,
    send(@donothin, display, new(BT, circle(4)), point(A1,50)),
    object_list(donothin,OL), append(OL,[BT],OL1), retract(object_list(donothin,OL)),
    asserta(object_list(donothin,OL1)),!.
```

```
make_other_objects(donothin) :-
    send(@donothin, display, new(@W11, text('actor')), point(20,175)),
    send(@W11, flush),
    send(@donothin, display, new(@W12, text('number')), point(60,175)),
    asserta(obj(donothin,@W12)),
    send(@W12, flush),!.
activate_picture(donothin,R,H) :-
    object_list(donothin,OL), actors(AS),
    ( between(1,AS,A), display_run(donothin,A,OL,R,H), sleep(1), fail; true),!.
display_run(donothin,A,OL,R,H) :-
    update(donothin,A),
    periods(TT),
    ( between(1,TT,T), make_wealth_dono(R,T,A,OL,H), fail
    ; true
    ),!.
make_wealth_dono(R,T,A,OL,H) :-
    fact(R,T,wealth_dono(A,WA)), magnify_by(MAG), W is MAG*WA, Y1 is H-W,
    nth1(T,OL,OT), X1 is 5+5*T,
    get(OT, position, point(X,Y)),
    send(OT, position, point(X1,Y1)),
    send(OT, fill_pattern, colour(red)),
    send(OT,flush),!.
```

% schellin

```
make_picture(schellin) :-
    VV2 = 'Picture-', concat(VV2,schellin,Vd2),
    new(@schellin, picture(Vd2)), send(@schellin, open),
    periods(TT),
    make_objects(schellin,TT),!,
    make_other_objects(schellin),
    choose_run(R),
    activate_picture(schellin,R),!.
make_objects(schellin,TT) :-
    height(schellin,H), /* 2 = NM , in para */
    gridwidth(G), size_of_boxes(S), GG is G*G,
    asserta(object_list(schellin,[])),
    ( between(1,G,I), make_row(I,G,S), fail; true),!.
make_row(I,G,S) :-
```

```
    ( between(1,G,J), make_column(I,J,S), fail ; true),!.
make_column(I,J,S) :-
    I1 is (I-1)*S, J1 is (J-1)*S,
    send(@schellin, display, new(E, box(S,S)), point(I1,J1)),
    object_list(schellin,L), append(L,[E],L1), asserta(object_list(schellin,L1)),
    retract(object_list(schellin,L)),!.
make_other_objects(schellin) :-
    send(@schellin, display, new(@W21, text('period')), point(20,162)),
    send(@W21, flush),
    send(@schellin, display, new(@W22, text('number')), point(60,162)),
    asserta(obj(schellin,@W22)),
    send(@W22, flush),!.
activate_picture(schellin,R) :-
    gridwidth(G),
    actors(AS), periods(TT), object_list(schellin,L),
    ( between(1,TT,T), display_period(schellin,R,T,AS,G,L), sleep(1), fail ; true),
    retract(object_list(schellin,L)),!.
display_period(schellin,R,T,AS,G,L) :-
    G1 is G*G,
    update(schellin,T),
    ( between(1,G1,X), nth1(X,L,OB), send(OB, fill_pattern, colour(white)),
    send(OB, flush), fail
    ; true
    ),!,
    ( between(1,AS,A), display(schellin,R,T,G,L,A), fail ; true),!.
display(schellin,R,T,G,L,A) :-
    fact(R,T,colour(A,C)),
    fact(R,T,location(A,I,J)), N is (I-1)*G+J, nth1(N,L,OB),
    ( C = black, X = colour(black)
    ;
    C = white, X = colour(blue)
    ),
    send(OB, fill_pattern, X), send(OB, flush),!.
```

```
% picture for takeweak
```

```
make_picture(takeweak) :-
    VV3 = 'Picture-', concat(VV3,takeweak,Vd3),
    new(@takeweak, picture(Vd3)), send(@takeweak, open),
```

```
    send(@takeweak, display, new(AXISUP, line(5,5,5,160,first))),
    send(@takeweak, display, new(AXISRIGHT, line(5,160,400,160,second))),
    asserta(object_list(takeweak,[])), height(takeweak,H),
    periods(TT), choose_run(R),
    make_objects(takeweak,TT,R),
    make_other_objects(takeweak),
    activate_picture(takeweak,R).

make_objects(takeweak,TT,R) :-
    actors(AS), asserta(oblist(takeweak,[])),
    ( between(1,AS,B), make_cell(B,R), fail ;true),
    send(@takeweak,flush),
    oblist(takeweak,L), asserta(oblist(takeweak,L)), !.

make_cell(B,R) :-
    J is 140, I is 5 + (B-1)*20, I1 is I + 10,
    send(@takeweak, display, new(E, box(10,20)), point(I,J)),
    send(@takeweak, display, new(F, box(10,20)), point(I1,J)),
    send(@takeweak, flush),
    oblist(takeweak,L), append(L,[[E,F]],L1),
    asserta(oblist(takeweak,L1)),
    retract(oblist(takeweak,L)),!.

make_other_objects(takeweak) :-
    send(@takeweak, display, new(@W1, text('period')), point(20,162)),
    send(@W1, flush),
    send(@takeweak, display, new(@W2, text('number')), point(60,162)),
    asserta(obj(takeweak,@W2)),
    send(@W2, flush),!.

activate_picture(takeweak,R) :-
    choose_run(R), make_maxima(R), /* R the run in para */
    periods(TT), actors(AS),
    ( between(1,TT,T), depict_period(takeweak,T,AS,R), fail ; true),!,
    ask_for_end, destroy(Answer),!.

depict_period(takeweak,T,AS,R) :-
    update(takeweak,T),
    ( between(1,AS,A), update(takeweak,T,A,R), fail; true),!,
    send(@takeweak,flush), sleep(1),!.

update(MOD,T) :- obj(MOD,O), send(O,string,T), send(O,flush),!.

update(takeweak,T,A,R) :-
    oblist(takeweak,OBL1), nth1(A,OBL1,[OA1,OA2]),
    fact(R,T,strength(A,SA)), SA1 is 5*SA,
    get(OA1, position, point(X,Y)), Y1 is 160 - SA1,
    send(OA1, position, point(X,Y1)), send(OA1,height,SA1),
```

```
    send(OA1, fill_pattern, colour(green)),
    send(OA1,flush),
    fact(R,T,wealth_take(A,V1)), max_val(MAX),
    get(OA2, position, point(X2,Y2)),
    ( V1 =< 0, V is 0; V is (V1/MAX)*20 ), Y3 is 160-V,
    send(OA2, position, point(X2,Y3)),send(OA2,height,V),
    send(OA2, fill_pattern, colour(blue)),
    send(OA2,flush),!.
make_maxima(R) :-
    findall(V,fact(R,T,wealth_take(A,V)),L1), sort(L1,L2),
    length(L2,E2), nth1(E2,L2,MAX1), asserta(max_val(MAX1)),!.
ask_for_end :-
    new(@d, dialog('Display')),
    send(@d, append, new(TI, text_item(type_End, ''))),
    send(@d, append, button(ok, message(@d,return,TI?selection))),
    get(@d, confirm, Answer),
    send(@d, destroy),!.
destroy(Answer) :- send(@takeweak, destroy).
```

% para.pl

```
runs(1).
periods(50).
actors(7).
gridwidth(4).
exist_min(20).
size_of_boxes(10).
modes( [takeweak,donothin,schellin ]).
modes([donothin,schellin,takeweak]).
use_old_data(no).
variables_in_rule(donothin,[wealth_dono]).
variables_in_rule(schellin,[location,colour]).
variables_in_rule(takeweak,[location,strength,wealth_take]).
weights(donothin,2,[50,101]).
weights(schellin,1,[101]).
weights(takeweak,1,[101]).
type_of_neighbourhood(schellin,moore,1).
type_of_neighbourhood(takeweak,von_Neumann,2).
choose_run(1).
```

```
domain_of_wealth_dono(50,500).
sigma_wealth_dono(20).
height(donothin,175).
magnify_by(0.2).
height(schellin,170).
domain_of_wealth_take(100,200).
sigma_wealth_take(30).
expressions(strength,2).
weights(strength,[70,101]).
domain_of_values(20).
height(takeweak,200).
```

% pred.pl

```
:- dynamic fact/3 .

normal_distribution(N,AS,L,U,SI) :-
    MU is L + (0.5 * (U-L)),
    ( between(1,AS,A), determine_nd_value(N,MU,SI,L,U,A), fail; true),!.

determine_nd_value(N,MU,SI,L,U,A) :-
    repeat, X is random(10001)+1,
    X4 is (1/10000) * (((X-1) * U)+(10001-X) * L), W is integer(X4),
    PI is pi, X1 is 2*(PI * (SI * SI)), X2 is (1 / sqrt(X1)),
    X3 is (-((W-MU)*(W-MU))) / (SI*SI), Y is X2 * exp(X3),
    W1 is random(10001)+1, Z is (W1-1)/10000, Z =< Y, between(L,U,W),
    asserta(nd_expr(N,A,W)), !.

make_discrete_distribution(N,AS,EX,LIST) :-
    ( between(1,AS,A), determine_dd_value(N,A,EX,LIST), fail; true ),!.

determine_dd_value(N,A,EX,LIST) :-
    X is random(100)+1,
    between(1,EX,Z), nth1(Z,LIST,W_Z), X < W_Z, assert( dd_expr(N,A,Z)),!.

calculate_sum(L,S) :-
    asserta(counter(0)), length(L,E),
    ( between(1,E,X), auxpred(L,X) , fail ; true), counter(S),
    retract(counter(S)).

auxpred(L,X) :-
    nth1(X,L,N), counter(C), C1 is C+N,
    retract(counter(C)), asserta(counter(C1)), !.

make_nbh(moore,N,I,J,L) :-
    gridwidth(G),
```

```
    ( N=1, moore_nbh_1(G,I,J,L)
    ; 1 < N, moore_nbh_1(G,I,J,L2), asserta(auxlist(I,J,L2)),
    length(L2,K), ( between(1,K,X), mnbh(X,L2,N,G,I,J), fail ; true),
    auxlist(I,J,L5), delete(L5,[I,J],L6), sort(L6,L),
    retractall(auxlist(A,B,L8))
    ),!.
mnbh(X,L2,N,G,I,J) :-
    nth1(X,L2,Y), Y=[I1,J1], N1 is N-1,
    make_nbh(moore,N1,I1,J1,L3), auxlist(I,J,L4), append(L4,L3,L5),
    retract(auxlist(I,J,L4)), asserta(auxlist(I,J,L5)), !.
moore_nbh_1(G,I,J,L) :-
    recalculate_neg(G,I,1,Im),
    recalculate_neg(G,J,1,Jm), recalculate_pos(G,I,1,Ip),
    recalculate_pos(G,J,1,Jp),
    L = [[I,Jm],[Im,Jm],[Im,J],[Im,Jp],[I,Jp],[Ip,Jp],[Ip,J],[Ip,Jm]].
recalculate_neg(G,I,H,I1) :-
    X is I-H, ( ( 0 < X, I1 is X
    ; 0 =:= X, I1 is G ) ; X < 0, I1 is (G+I)- H ),!.
recalculate_pos(G,I,H,I1) :-
    X is I+H, ( ( I < G, X =< G, I1 is X
    ; I =:= G, (H > 0, I1 is H; H =:= 0, I1 is G) ) ; I < G, X > G,
    I1 is (H+I)-G ),!.
make_nbh(von_Neumann,N,I,J,L) :-
    gridwidth(G),
    ( N=1,
    von_Neumann_nbh_1(G,I,J,L) ; 1 < N, von_Neumann_nbh_1(G,I,J,L2),
    asserta(auxlist(I,J,L2)), length(L2,K),
    ( between(1,K,X), vNnbh(X,L2,N,G,I,J), fail ; true),
    auxlist(I,J,L5),
    delete(L5,[I,J],L6),
    sort(L6,L),
    retractall(auxlist(A,B,L8))
    ),!.
vNnbh(X,L2,N,G,I,J) :-
    nth1(X,L2,Y), Y=[I1,J1], N1 is N-1,
    make_nbh(von_Neumann,N1,I1,J1,L3), auxlist(I,J,L4), append(L4,L3,L5),
    retract(auxlist(I,J,L4)), asserta(auxlist(I,J,L5)), !.
von_Neumann_nbh_1(G,I,J,L) :-
    recalculate_neg(G,I,1,Im),
    recalculate_neg(G,J,1,Jm), recalculate_pos(G,I,1,Ip),
    recalculate_pos(G,J,1,Jp), L = [[I,Jm],[Im,J],[I,Jp],[Ip,J]].
```

```
decompose(Y,I,J,G) :-
    between(1,G,Z), Y =< Z*G, Z1 is Z-1, I is Z,
    J is Y-(Z1*G),!.
```

```
% rules.pl
% The action rules in SMASS
```

```
:- dynamic fact/3 .
```

```
% RULE 1: 'donothin'. The person intentionally does not do anything.
```

```
domain_of_wealths(50,500).
sigma_wealths(20). exist_min(20).
prepare(R,T,donothin) :- true.
act_in_mode(donothin,A,R,T) :-
    feasible(donothin,A,R,T),
    chooseaction(donothin,A,R,T), perform(donothin,A,R,T),!.
feasible(donothin,A,R,T) :-
    fact(R,T,wealth_dono(A,W)), exist_min(E),
    E1 is 3*E, W1 is W-E1, W1 > 0,!.
chooseaction(donothin,A,R,T) :- true,!.
perform(donothin,A,R,T) :-
    fact(R,T,wealth_dono(A,W)), W1 is W-5,
    retract(fact(R,T,wealth_dono(A,W))),
    asserta(fact(R,T,wealth_dono(A,W1))),!.
protocol(donothin,A,R,T) :- fail.
    adjust(donothin,A,R,T):- true.
    adjust(donothin,R,T) :- true.
```

```
% RULE 2: 'schellin'.
% The person move to a better location, if she likes the
% new ambience. This rule is synchronous because an actor can move only
% once in a period.
```

```
prepare(R,T,schellin) :-
    asserta(occupied([])), gridwidth(G),
```

```
    G1 is G*G, findall(X,between(1,G1,X),CL),
    actors(AS),
    asserta(list222(CL)),
    ( between(1,AS,A), subtract_cell(A,CL,R,T,G), fail ; true),!,
    list222(LFree), retract(list222(LFree)), asserta(free_cells(LFree)),!.
subtract_cell(A,CL,R,T,G) :-
    fact(R,T,location(A,I,J)),
    ( member(B,CL), decompose(B,I,J,G), list222(LFree),
    subtract(LFree,[B],LFreenew), retract(list222(LFree)), asserta(list222(LFreenew))
    ;
    true
    ),!.
act_in_mode(schellin,A,R,T) :-
    feasible(schellin,A,R,T),
    chooseaction(schellin,A,R,T), perform(schellin,A,R,T),!.
feasible(schellin,A,R,T) :- true.
chooseaction(schellin,A,R,T) :-
    gridwidth(G),
    fact(R,T,location(A,I,J)), scan_neighbourhood(A,G,I,J,R,T,ANSWER),
    ( ANSWER=yes ; calculate_move(A,R,T,G,schellin) ),!.
calculate_move(A,R,T,G,schellin) :-
    free_cells(LFree), length(LFree,EFree),
    asserta(pot_cells([])),
    ( between(1,EFree,X), check_cell(X,R,T,A,G,LFree,schellin), fail ;true),!,
    pot_cells(LC), length(LC,EE), retract(pot_cells(LC)),
    asserta(yes_list([])),
    ( between(1,EE,NC), nth1(NC,LC,C), enter(C), fail ; true),!,
    yes_list(LYES), retract(yes_list(LYES)),
    occupied(Locu),
    ( LYES = [] /* does nothing */
    ;
    LYES ≠ [], compare1(LYES,Locu,LL), /* ≠ is not a PROLOG operation */
    ( LL ≠ [] /* does nothing */
    ;
    LL ≠ [], nth1(1,LL,IntC), nth1(1,IntC,B), nth1(2,IntC,I), nth1(3,IntC,J),
    IntCnew = [A,I,J],
    append(Locu,[IntCnew],Locunew), retract(occupied(Locu)),
    asserta(occupied(Locunew))
    )
    ),!.
compare1(LYES,Locu,LL) :-
```

```
    length(LYES,EL), asserta(list111([])),
    ( between(1,EL,Z), compare_cell(Z,LYES,Locu), fail ; true),!,
    list111(LL), retract(list111(LL)),!.
compare_cell(Z,LYES,Locu) :-
    nth1(Z,LYES,C), nth1(1,C,B), nth1(2,C,UU), nth1(1,UU,I), nth1(2,UU,J),
    U = [X,I,J],
    ( member(U,Locu) /* nothing is done */
    ;
    list111(LL), append(LL,[U],LLnew), retract(list111(LL)), asserta(list111(LLnew))
    ),!.
enter(C) :-
    yes_list(LYES), nth1(3,C,ANS),
    ( ANS = yes , append(LYES,[C],LYESnew), retract(yes_list(LYES)),
    asserta(yes_list(LYESnew))
    ;
    true
    ),!.
check_cell(X,R,T,A,G,LFree,schellin) :-
    nth1(X,LFree,B), decompose(B,I,J,G),
    scan_neighbourhood(A,G,I,J,R,T,ANSWER), pot_cells(LL),
    append(LL,[[B,[I,J],ANSWER]],LLnew), retract(pot_cells(LL)),
    asserta(pot_cells(LLnew)),!.
scan_neighbourhood(A,G,I,J,R,T,ANSWER) :-
    type_of_neighbourhood(schellin,TW,TG),
    make_nbh(TW,TG,I,J,L),
    findall(N,neighb(N,L,R,T),L1), length(L1,E1),
    findall(N1, equal_colour(N1,A,L,R,T), L2), length(L2,E2),
    (
    ( ( E1 =< 2, 1 =< E2; 3 =< E1, E1 =< 5, 2 =< E2 )
    ;
    6 =< E1, E1 =< 8, 5 =< E2
    ), ANSWER=yes
    ;
    ANSWER=no
    ),!.
neighb(N,L,R,T) :-
    member([I,J],L), fact(R,T,location(N,I,J)).
equal_colour(N,A,L,R,T) :-
    member([I,J],L), fact(R,T,location(N,I,J)),
    fact(R,T,colour(N,CN)), fact(R,T,colour(A,CA)), CA=CN.
```

```
perform(schellin,A,R,T) :- true.
    protocol(schellein,A,R,T) :- fail.
adjust(schellin,A,R,T) :-
    occupied(LL),
    ( LL=[]
    ;
    LL ≠ [] ,
    ( nth1(X,LL,[A,I,J]),
    fact(R,T,location(A,Iold,Jold)),
    retract(fact(R,T,location(A,Iold,Jold))),
    asserta(fact(R,T,location(A,I,J)))
    ;
    true
    )
    ),!.
adjust(schellin,R,T) :-
    free_cells(LFree), retract(free_cells(LFree)),
    occupied(LL), retract(occupied(LL)),!.
```

```
% RULE 3: 'takeweak'.
% The person takes wealth from a weaker person. Here
% we find an asynchronous rule of a strange -but realistic- way of cheating.

prepare(R,T,takeweak) :- true.
act_in_mode(takeweak,A,R,T) :-
    feasible(takeweak,A,R,T),
    chooseaction(takeweak,A,R,T), perform(takeweak,A,R,T).
feasible(takeweak,A,R,T) :-
    exist_min(MIN),
    domain_of_values(SS),
    fact(R,T,location(A,IA,JA)),
    make_nbh(TW,GW,IA,JA,NHA),!,
    length(NHA,E), asserta(neighlist([])),
    ( between(1,E,Y), locate(Y,R,T,NHA), fail; true),!,
    neighlist(NHL), append(results), write(neighlist(A,i,NHL)), write('.'),
    nl, told,
    length(NHL,E1),
    retract(neighlist(NHL)),
    fact(R,T,strength(A,SA)),
    asserta(n_strength([])),
```

```
    ( between(1,E1,X), investigate(R,T,X,NHL,MIN,SA,SS), fail; true),!,
    n_strength(LL), length(LL,E2), retract(n_strength(LL)),
    ( E2 = 0, fail
    ;
    0 < E2,
    sort(LL,LLnew), length(LLnew,E3), nth1(E3,LLnew,WW),
    WW = [NB,SB],
    asserta(neighb(NB))
    ),!.
locate(Y,R,T,NHA) :-
    nth1(Y,NHA,[I,J]), fact(R,T,location(B,I,J)),
    neighlist(LL), append(LL,[B],LLnew), retract(neighlist(LL)),
    asserta(neighlist(LLnew)),!.
investigate(R,T,X,NHL,MIN,SA,SS) :-
    n_strength(LL),
    ( nth1(X,NHL,NB), fact(R,T,strength(NB,SB)),
    SB < SA,
    fact(R,T,wealth_take(NB,WB)), W1 is WB-(3*SS), !,
    MIN =< W1, append(LL,[[NB,SB]],LLnew),
    retract(n_strength(LL)), asserta(n_strength(LLnew))
    ;
    true
    ),!.
chooseaction(takeweak,A,R,T) :- true.
perform(takeweak,A,R,T) :-
    neighb(NB), retract(neighb(NB)),
    fact(R,T,wealth_take(A,WA)), domain_of_values(SS), S1 is 3*SS,
    X is random(S1), WA1 is WA+X, retract(fact(R,T,wealth_take(A,WA))),
    asserta(fact(R,T,wealth_take(A,WA1))), T1 is T+1,
    asserta(fact(R,T1,give_to_the_strong(NB,X))).
protocol(takeweak,A,R,T) :- fact(R,T,give_to_the_strong(A,X)),
    fact(R,T,wealth_take(A,WA)), exist_min(MIN), W1 is max(WA-X,MIN),
    retract(fact(R,T,wealth_take(A,WA))),
    asserta(fact(R,T,wealth_take(A,W1))),
    retract(fact(R,T,give_to_the_strong(A,X))).
adjust(takeweak,A,R,T) :- true.
    adjust(takeweak,R,T) :- true.
```

Towards Computational Institutional Analysis: Discrete Simulation of a 3P Model

Alain Albert
Département des sciences administratives, Université de Québec a Hull, Canada
Wolfgang Balzer
Institut für Philosophie, Logik und Wissenschaftstheorie, Universität München, Germany

Abstract: A 3P model (production, predation,protection) which can be game theoretically solved for two actors is generated to n actors and studied by means of discrete simulations. The simulation confirm robust incentives for actors to produce and predate in an institution free enviroment, whereas protection activity is not significantly related to the ability for protection. The model is criticized for its neglect of predators predating on each other, and for its inability to reproduce real-life proportions of producers and predators and the times these spend on the three activities.

Key words: simulation, discrete simulation, 3p model, economics, game theory

Introduction

In the last decade, social scientists have shown growing interest in the formal analysis of social institutions.[1] Economists, sociologists, political scientists and philosophers of science have contributed to this formal and mathematical modelling of institutions (their emergence, dynamic properties and stability).

At the same time computer simulations of social phenomena shifted from 'traditional' numerical simulations based on mathematical equations to agent-based, discrete event simulations. This new computational approach to modelling and simulating social phenomena has given birth to several new fields of research such as computational organization theory (Prietula & Carley, 1994), (Prietula, Carley & Gasser, 1998), computational sociology (Bainbridge et al., 1994), computational anthropology (Doran, 1995, Dean at el., 1998), computational social psychology (Nowak & Vallacher, 1998) and last but not least , computational economics (Tesfatsion, 1998).

The aim of this paper is to contribute to this new research agenda by adding computational institutional analysis or briefly CIA[2] to the list of computational social

[1] The formal approach based on deductive reasoning is sometimes opposed to the descriptive approach of the 'old' institutionalist school of the Commons variety (Commons, 1934). However, such an opposition between a theoretically driven 'new' institutionalism and an 'anti-theoretic' old institutionalism does not seem adequate; see (Hodgson, 1998) who stresses the early institutionalists' concern for theoretical issues.

[2] This field of research has something in common with its more famous counterpart. Computational Institutional Analysis is at the *center* of economic analysis, it is (artificial) *intelligence* based and, finally, it is *agent* based. Needless to say, we do not pursue the same objectives.

fields. As we envision it, CIA combines the formal modelling of social institutions with new methods of doing agent-based computational social science. The present paper is a step in this direction. We take up an economic, game theoretical model and investigate its potential for the understanding of institutions by means of simulation.

One way of theorizing about social order is by classifying actors in terms of the types of the actions they perform. Going back to a very basic, almost 'prehistoric' level, three broad types of of activities which seem to be promising for this task are production, predation and protection, where predation is understood to comprise all forms of taking away things or resources from a person against that person's will, and protection means to protect *own'sown* possessions, resources and body. The proportions in which members engage in these activities may be used to draw distinctions between different forms of social organization - whether historical-empirical or merely conceptual. In a society of slave holders the slaves do not engage in protection, whereas the peasants in a peasant society do so. Conversely, the slave owners spend quite some effort on protection, much more than does a leader in a rural society. More precisely, the approach consists of looking at the *proportions of time* which a person devotes to production, predation and protection, to use these proportions for a classification of the persons, and to analyze the relative sizes of the classes to obtained. A person spending almost all her time on production thus may be classified as a *producer*, while it is not easy to find a natural label for, say, a person devoting her time equally to production, predation and protection eventhough the kind of such persons is determined theoretically in a precise way.

This approach may be pursued by starting from simple, 'institution-free' economic models in which the optimal or equilibrium distribution of persons' times is studied in a game theoretic setting. We here generalize a simple one-good, two-agent hobbesian model studied in (Houba & Weikard, 1995) which deals with the optimal allocation of actors' times on the three kinds of activities: *p*roduction, *p*redation, and *p*rotection, this is why we speak of a 3P model. On the basis of his utility function which depends on the amounts of time *all* actors spend on each activity, each single actor tries to optimally distribute a fixed, total amount of time among the three types of activities. As game theoretic analysis becomes very difficult, if not practically impossible, for numbers of actors greater than two, simulation offers itself as the natural tool to be used.

We introduce the generalized 3P model, and describe how this is simulated in a discrete event setting. We then explore its potential by investigating the connections between actors' abilities to produce, predate and protect, and the percentages in which these abilities are present in the population. These connections found in different simulations are critically discussed in the light of corresponding, presystematic expectations. We describe some expected, 'nice' results, but also a number of unexpected results indicating deficiencies of the present, basic model. In spite of these negative results we believe that the model has a great potential for modifications and refinements.

A first positive result is that predation is 'robust' in the sense that actors who are best at predating (i.e. whose ability for predating exceeds that for production and predation) in most cases spend almost all their time on predation. Moreover, the time spent on predation increases sharply with an increase of the ability to predate, and does not much depend on variation of the other abilities. This finding points to a natural incentive which theoretically could back Hobbes' state of nature. A second positive result is that production time also increases with an increase of the ability to produce, though the degree of increase varies with other parameters, in particular with the coefficients for the other abilities and the percentage of producers in the population. This also indicates a natural incentive, and the variability of increase opens the way for studying the systematic effects of other, 'external' parameters on the incentive to produce.

Negatively, we first found that actors which are best at production not only tend to spend increasingly more time on protection when the number of predators increases (which is naturally expected), but in many cases they spend the overwhelming part of their time on protection, even with moderate numbers of predators.

This is of interest to CIA for the following reasons. On the one hand, as far as we know, such a pattern of activities (spending almost all time on protection) is not observed in civilized societies nor does the limited knowledge of pre-history indicate such behavior. On the other hand, this effect points to a feature which is not covered by 'standard' economic approaches to social institutions, namely their functional role in weakening the protective capacity of 'producers' (whether to the benefit of 'law and order' or of 'predators' can be left open). The excessive time spent on protection we found in many simulations shows that this functional role is not captured by the present model (and by other economic models of the hobbesian variety). In this respect, power-centered models based on the interactions between a larger set of agents (autocrats, bureaucrats, bandits, and producers ...)[3] would probably give a more realistic picture of the social institutions we are trying to model and simulate. However, in the present paper we adhere to the 'KISS' principle advocated by (Axelrod, 1997).

Another negative result is that protection time in most cases does not monotonically increase with protection ability. A first interpretation is that the ability for protection is dominated by the other two abilities, and thus not really an independent variable. This interpretation is also supported by the intuitive observation that the abilities for predation and protection in a pre-historic environment are closely related to similar kinds of bodily skills and strengths.

Finally, in simulations where abilities were lognormally distributed in the population, we were not able to produce patterns of time proportions corresponding to presystematic, real-life expectations, like, say, 70% of the population spending 80% of time on production and 30% spending 80% of time on predation.

[3] Examples of such economic models may be found in (Usher, 1993) or (Wintrobe, 1998), see also (Balzer, 1990).

1 The Basic Hobbesian 3P Model

It may seem strange to start an analysis of social institutions by modelling an institution-free hobbesian world. But as noted by (Wolff, 1996) in his analysis of Hobbes' state of nature, 'To understand why we have something, it is often a good tactic to consider its absence'. Hence, one way to examine how social institutions emerge and what type of social interactions (exchange based vs. power based) underlie these institutions, is to start from an institution-free setting of which Hobbes' account is perhaps the most famous example.

Since Bush's pioneering work (Bush, 1976) there have been numerous articles and books[4] devoted to the modelling of conflictual anarchy of the hobbesian variety.[5] We here study a simple representative of the hobbesian variety of models in order to show how such a model, when generalized to a multi-agent computational world, may give rise to interesting features that could (practically) not be found by paper and pencil. However since, as pointed out by (Binmore, 1998), computer simulations are not a substitute for deductive reasoning based on sound theoretical microeconomics or game theory we shall first give a brief account of the theoretical model underlying our multi-agent simulations.

In the hobbesian world, there are no property rights or social norms to regulate agent interactions. In order to survive in such a world, individual agents undertake three basic types of activities: they produce, they use force to steal (predate) and they protect themselves against the predatory activities of others.

People are not equal in their abilities for doing so. Some are stronger than others, some are better at producing than at stealing. Depending on their relative abilities individuals produce, steal and protect themselves by equating the marginal returns of these three basic activities. The results of an actor's marginal calculus depend on the behavior of the other agents with whom she interacts. In most approaches this interactive behavior is modelled by Cournot-Nash type assumptions but a few models use Stackelberg type (leader-follower) assumptions.

Adopting the two-persons generalization of (Houba & Weikard, 1995) of Bush's original model, let us consider two persons i, j. Let P_{i1}, P_{i2}, P_{i3} be the production, predation and protection functions of individual i (those of j are obtained by interchanging i and j).

(1) Production: $P_{i1} = f_1(a_{i1}, t_{i1})$

[4] A critical review of these models is found in (Albert, 1999).

[5] We are reluctant to use the term 'anarchy' in connection with conflictual models opposing bandits (predators) to peasants (producers) because this tends to confirm the widespread prejudice that anarchy *implies* fighting or a hobbesian state of nature. Originally, anarchy only means absence of domination. Though predation, robbery and exploitation are compatible with the absence of domination, they are by no means *implied* by such absence, as Hobbes made us believe. See (Flap, 1985) for a counter example. The hobbesian state of nature in which everyone fights everyone is only one among many other conceptual - including less frightening - alternatives. See also the comments of (Dowd, 1997) on Hirshleifer's model (Hirshleifer, 1995) of conflictual anarchy.

(2) Predation: $P_{i2} = f_2(a_{i2}, t_{i2}, P_{j1}, P_{j3}), i \neq j$
(3) Protection: $P_{i3} = f_3(a_{i3}, t_{i3})$,

where the $a_{ip} > 0$ are individual parameters for, respectively, the productive (p=1), predatory (p=2) and protective (p=3) capacities of individual i, called *abilitiy coefficients* in the following, and t_{ip} denotes the time devoted by individual i to activity number p. Whereas the production and protection functions (1 and 3) depend only on i's own parameters and variables, the predation function (2) includes arguments that do not only depend on i's own capacities and time devoted to predation. The predation function also depends on the other person's time and capacities devoted to production and protection. The more j produces the more i can steal from him, but the more j protects himself the more costly is it to steal from him.

Each individual k has a utility function U_k it seeks to maximize. A simple form for U_i suggested by (Houba & Weikard, 1995) is this:

(4) $U_i = U_i(t_{i1}, t_{i2}, t_{i3}, t_{j1}, t_{j2}, t_{j3}) = P_{i1} + P_{i2} - P_{j2}, j \neq i$

Thus U_i is equal to what i is able and willing to produce (captured by P_{i1}) plus what she is able and willing to steal from j (captured by P_{i2}) minus what is stolen from her by the other actor j (captured by P_{j2}).

In (Houba & Weikard, 1995) the functions f_1, f_2 and f_3 are generally specified as follows. For $k = i, j$,

(7) $f_{k1}(a_{k1}, t_{k1}) = a_{k1}t_{k1}$ and $f_{k3}(a_{k3}, t_{k3}) = a_{k3}t_{k3}$
$f_{i2}(a_{i2}, t_{i2}, P_{j1}, P_{j3}) = a_{i2}(t_{i2})^{\alpha_i} P_{j1}(1 - P_{j3})$, and
$f_{j2}(a_{j2}, t_{j2}, P_{i1}, P_{i3}) = a_{j2}(t_{j2})^{\alpha_j} P_{i1}(1 - P_{i3})$.

Using (7) we obtain the following general expressions for U_i and U_j.

(8) $U_i(t_{i1}, t_{i2}, t_{i3}, t_{j1}, t_{j2}, t_{j3}) = a_{i1}t_{i1} + a_{i2}(t_{i2})^{\alpha_i} a_{j1}t_{j1}(1 - a_{j3}t_{j3}) - a_{j2}(t_{j2})^{\alpha_j}$
$a_{i1}t_{i1}(1 - a_{i3}t_{i3})$,
$U_j(t_{j1}, t_{j2}, t_{j3}, t_{i1}, t_{i2}, t_{i3}) = a_{j1}t_{j1} + a_{j2}(t_{j2})^{\alpha_j} a_{i1}t_{i1}(1 - a_{i3}t_{i3}) - a_{i2}(t_{i2})^{\alpha_i}$
$a_{j1}t_{j1}(1 - a_{j3}t_{j3})$.

Each actor k seeks to maximize his utility subject to the constraint that $t_{k1}+t_{k2}+t_{k3} \leq T$ where T is the total amount of time available in the period considered which, for reasons of simplicity, is set equal to 1 for both actors. Clearly, both actors are strategically interdependent since in (8) i's utility depends on the times chosen by j and conversely. The resulting game can be analytically solved for two actors.

2 The General Model

We generalize this model to the case of n actors as follows, retaining the assumption of one single good that is produced by everyone. Each of the n actors i ($i = 1, ..., n$) has a utility function U_i depending on the $3n$ times which all actors spend on the

three activities: production, predation and protection. For each i, the times i spends on production, predation and protection, respectively, are denoted by t_i^1, t_i^2 and t_i^3. Thus i's distribution of time on the three activities is given by $\vec{t_i} = (t_i^1, t_i^2, t_i^3)$ and i's utility function may be written as $U_i = U_i(\vec{t_1}, ..., \vec{t_n})$. When the time distributions of the other actors $j, j \neq i$, are held constant, we simply write $U_i = U_i(\vec{t_i})$. We assume that i's utility function has the following form

$$
\begin{aligned}
(9)\ U_i(\vec{t_1}, ..., \vec{t_n}) &= a_{i1}t_{i1} \\
&+ a_{i2}(t_{i2}/(n-1))^{\alpha_i}\Sigma_j(a_{j1}t_{j1}(1-a_{j3}t_{j3})) \\
&- min(1, (\Sigma_j a_{j2}(t_{j2}/(n-1))^{\alpha_j}))a_{i1}t_{i1}(1-a_{i3}t_{i3})
\end{aligned}
$$

where $0 < \alpha_i < 1$, $0 \leq a_{i1}, a_{i2}, a_{i3}$ and $a_{i1} + a_{i2} + a_{i3} = 1$ for $i = 1, ..., n$. The ability coefficient a_{ip} expresses the 'ability' or 'efficiency' with which actor i performs activity number $p (p = 1, 2, 3)$ for production, predation, protection), and t_{ip} is the time i spends on activity p. The three components of U_i in (9) may be interpreted as follows. The first component $a_{i1}t_{i1}$ represents the amount of the single good which i produced, depending on her productive ability a_{i1} and the time t_{i1} she spent on production.

The second component may be best understood if we rewrite it as $(n-1)$ [$a_{i2}(t_{i2}/(n-1))^{\alpha_i}(1/(n-1))\Sigma_j a_{j1}t_{j1}(1-a_{j3}t_{j3})$]. $a_{i2}(t_{i2}/(n-1))^{\alpha_i}$ is the 'weight' of i's activity of predating when i predates one of the n other actors, on the assumption that i splits his 'predation time' equally on all other actors. The average, 'non-protected' production of some actor thus predated by i is $(1/(n-1))\Sigma_j a_{j1}t_{j1}(1-a_{j3}t_{j3})$. So $a_{i2}(t_{i2}/(n-1))^{\alpha_i}(1/(n-1))\Sigma_j a_{j1}t_{j1}(1-a_{j3}t_{j3})$ is i's utility from predating one 'average' fellow actor. In order to obtain i's total utility this expression has to be taken $n-1$ times.

In the third part, $(t_{j2}/(n-1))^{\alpha_j}$ gives the 'size' or 'weight' of that part which j can take away from i's non-protected product $a_{i1}t_{i1}(1-a_{i3}t_{i3})$ on the assumption that j spends her 'predation time' t_{j2} equally on all other actors. Thus the third part refers to the sum of all parts which are taken away from i's non-protected product by all the other actors. Since in the case of more than two actors the sum of all 'weights' may be greater than 1 we have to take the minimum of this sum and 1 in order to prevent a change of sign in the third component.

As an analytic treatment of these general equations is very difficult, if not practically impossible, the best way to proceed is by simulation. We use a discrete event simulation shell called SMASS (Sequential Multi-Agent System for Social Simulation) written in PROLOG (Balzer, 1999). This shell executes simulation runs over a fixed number N of periods such that in each period, each actor is called up for action exactly once. The task of implementation in this shell reduces to the formulation and implementation of a rule of behavior according to which each actor acts when called up in a period T.

3 The Simulation

As the above analytic model is static, we have to find a way using a dynamical simulation in order to obtain the static distributions of actors' times devoted to the three different activities. This is done as follows. The model's total time interval which is captured in one simulation run, is represented by the number N of all periods over which the simulation is run. Assuming that each actor in each period acts just once we can count the numbers m_1, m_2, m_3 of periods in which he produces, predates, or engages in protection, so $N = m_1 + m_2 + m_3$. We identify these numbers m_1, m_2, m_3 with the times t_1, t_2, t_3 an actor spends on the three activities in the solution of the analytical model.

A second problem is to formulate a rule of behavior expressing the maximization assumptions which in the analytic model are applied to the equations (1)-(4) and (5) and (6) above. In principle, one could try to just let each actor solve the above equations and distribute her time according to that solution. This is impractical, however, because we consider more than two actors, and for larger numbers we simply wouldn't know how to solve the equations. We therefore formulate a different basic rule of behavior as a substitute for the assumptions of the analytic model.

To this end during the course of the simulation a 'history' is built up recording in each period T the numbers of periods every single actor spent on each of the three activities up to the present period T. Thus if actor i is called up in period T her history $\vec{h_{i,T}}$ will consist of three numbers $\vec{h_{i,T}} = (h_{i1,T}, h_{i2,T}, h_{i3,T})$ such that $h_{i1,T} + h_{i2,T} + h_{i3,T} = T$ and each $h_{ip,T}$ is the nmuber of periods in which i performed activity number $p (p = 1, 2, 3)$. Such a history gives the distribution of the times i spent on the three activities.

Instead of the utilities $U_i(\vec{t_1}, ..., \vec{t_n})$ derived from the 'final' proportions of times we now may consider utilities derived from the relative proportions of times spent up to a given period T, i.e. utilities depending on the actors' histories up to T

$U_i(T) = U_i(\vec{h_{1,T}}, ..., \vec{h_{n,T}})$, where $\vec{h_{i,T}} = (h^1_{i,T}, h^2_{i,T}, h^3_{i,T})$

We apply the following rule of behavior. An actor i in period T calculates the marginal utilities for each of the three activities, and chooses that activity which yields highest marginal utility. The marginal utilities are those actor i would derive from spending one more period on production, predation or protection, given that up to period T he spent the times $(h^1_{i,T}, h^2_{i,T}, h^3_{i,T})$ on these activities. i's marginal utility for *production* in period T is thus defined by

(10) $U_i(\vec{h_{1,T}}, ..., (h^1_{i,T} + 1, h^2_{i,T}, h^3_{i,T}), ..., \vec{h_{n,T}}) - U_i(\vec{h_{1,T}}, ..., \vec{h_{n,T}})$.

The marginal utilities for predation and protection are obtained in the same way by adding in (10) one period to $h^2_{i,T}$ and $h^3_{i,T}$, respectively.[6]

In (10) the other actors' histories enter in the calculation of i's marginal utilities;

[6] As periods are represented by integers, the natural unit here is 1.

these are taken as they are found at the time of execution in period T.[7]

In the analytical model a solution or state of equilibrium is a list of time distributions $(\vec{t_1}, ..., \vec{t_n})$ (a 'state') satisfying a condition of maximality or equilibrium. In the simulation such a state corresponds to the actors' $final$ histories $(\vec{h_{1,N}}, ..., \vec{h_{n,N}})$ where N denotes the total number of periods for which the simulation is run. While the simulation is running, the histories $\vec{h_{i,T}}$ steadily change when T grows from 1 to N. However, we can say that the system in state $(\vec{h_{1,T}}, ..., \vec{h_{n,T}})$ has become $stable$ if the fractions $h_{ip,T'}/T'$ do not change significantly for all T' such that $T \leq T'$. For instance, when the final distribution of i's time is (0.5,0.5,0) - i.e. i spent half of her time on producing and half of it on predating - then for $N = 100, \vec{h_{i,N}} = (50/100, 50/100, 0)$. When the system has become stable, say in period 70, then $\vec{h_{i,70}} = (35/70, 35/70, 0)$ and these fractions will show only insignificant deviations for $T > 70$. As the system operates with integers, they cannot remain strictly identical because, say, for $N = 100$, in each period one of the history's components will be increased by 1/100.

The states which are stable in this sense may be taken as the analogues of analytic solutions. For all simulations performed we found that 100 periods were sufficient to reach a stable state when deviations were allowed up to $\epsilon = 0.02$. The stable state in most cases was reached between periods number 60 and 80.

4 Simulation Results

We performed a number of simulations in order to explore the space of possibilities given by variations in the parameters: numbers of actors, ability coefficients, exponents, and initial distributions of predators and producers in the population. This is a huge space and it does not seem a good idea to try to explore it fully systematically. We varied several items in more systematic fashion, but only so within relatively narrow boundaries. Each simulation run captured 100 periods and in all runs a stable state was reached. Each simulation was repeated ten times with the same initial data. The results reported here are the mean values over these repetitions, deviations from these means were usually in the order of 0.01.

Even within a homogenous population of completely identical actors, slightly different results are observed for different actors. This effect is due to the multi-agent character of the simulation in which it makes a difference, for instance, whether in a period one of the few predators is called up at the beginning or towards the end of the period, i.e. before most other actors have chosen their activities and acted, or after that. However, these individual differences usually are not significant, deviations being smaller than 0.02, and usually much smaller. For this reason, we do not differentiate in the following description between single actors, and just report the results for one arbitrary, representative member of each sub-population.[8]

[7] This amounts to asynchronous updating.

[8] As a warming up exercise we simulated the Houba-Weikard 2-actor model with the coefficients

In a first series of simulations we used ability coefficients that are lognormally distributed in the population. As these coefficients consist of three components whose interdependency is difficult to judge empirically we used a mix of two different random processes to create them. We first created lognormally distributed numbers b_i - one for each actor i - within the interval [0,1]. We then split the 'rest' $1 - b_i (\geq 0)$ randomly into two parts $b_i = a_i + c_i$, and used $[a_i, b_i, c_i]$ as coefficients of actor i. Each run was repeated 10 times.

Defining 'producers' i as those actors whose ability for producing, a_2^i, is strictly greater than that for predating, a_2^i, and predators as all other actors, the population split up into x% of producers and (1-x)% of predators. With varying numbers of actors x varied in the interval [40,60].

The means of the ability coefficients and the time profiles did vary with variations of the number of actors, but this effect is mainly due to the fact that for a different number of actors, the ability coefficients are newly created in the random way described earlier. Table 1 summarizes some results.

Table 1

number of actors	10	50	100
percentage of	50	42	43
producers	.	.	.
mean producer:	.	.	.
ability coeffs	(0.36,0.10,0.53)	(0.51,0.14,0.33)	(0.46,0.13,0.40)
time profiles	(0.55,0.17,0.27)	(0.23,0.57,0.18)	(0.18,0.64,0.17)
variances	(0.007,0.041,0.021)	(0.035,0.174,0.052)	([0.038,0.180,0.060)
of time	.	.	.
mean predator:	.	.	.
ability coeffs	(0.18,0.58,0.22)	(0.16,0.59,0.23)	(0.14,0.60,0.25)
time profiles	(0.13,0.86,0)	(0.01,0.98,0)	(0.01,0.98,0)
variances	(0.009,0.009,0)	(0,0,0)	(0,0,0)
of time	.	.	.

Remarkably, in populations of more than 40 actors, a 'mean producer' spends more time on predating than on producing. This means that several single producers, i.e. actors who are more able to produce than to predate, nevertheless spend more time on predation, which, for them, is the inferior activity. This result clashes with the assumption of rationality underlying the model. However, we can interpret it as showing that the incentive for predation which is incorporated in the form of the untility function is much stronger than that for production so that it surpasses the *prima facie* incentive given in terms of the ability coefficients.

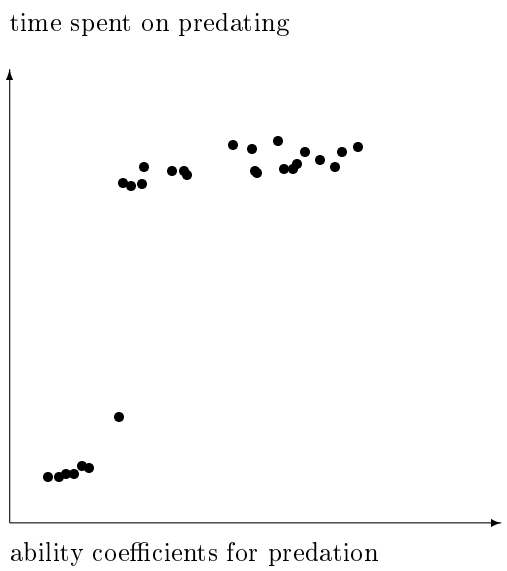
By contrast, the 'mean predators' do not spend much time on producing even-

$[2, 0, 1]$ for the producer and $[1, 1, 0]$ for the predator. This yields the expected Nash equilibrium at $(0, 0.3968, 0.2063)$ - the remaining times being uniquely determined by the time constraint - for the predator, which in this case also can easily be computed by hand.

though they have a non-negligable coefficient for production. Moreover, the 'mean predators' do hardly spend any time on protection, which in many simulation means that no single predator does so. This outcome conflicts with the intuition - external to the model - that predators also should predate on their 'fellow' predators. Given the high percentage of predators in the population (often more than 50%), one would want to see a substantial amount of time spent by predators on protecting themselves against each other. However, this incentive is not captured by the model. The third, negative part of the utility function depends multiplicatively on the actor's own product ($a_{i1}t_{i1}$) in (9) which, for predators is neglegible. According to (9) a predator spending no time on production has nothing to protect. In reality, even in the basic case in which all products - whether produced or robbed - are consumed in the same period, there is the possibility of one predator taking away from another one the good which the latter just robbed from a third person.

Looking at how each single time component t_r (e.g. the time spent on predation, r=2) depends on a single ability coefficient a_s (e.g. the coefficient for production, s=1), we arranged the coefficients that are present in the population in an increasing order so that for the set $\{i_1, ..., i_n\}$ of actors we got $a_s(i_1) < ... < a_s(i_n)$. When for each $a_s(i_j)$ we plot the corresponding time $t_s(i_j)$ in a diagram the dependence (in a population of 100 actors) can be graphically depicted as in figure 1.

Figure 1



Distinguishing increase (+) from decrease (-), and degrees of the strengths of the connections (1 = strong and regular, 2 = weak and regular, 3 = irregular) all the dependencies are summarized in table 2.

Table 2

.	increasing coefficient for	.	.
.	production	predation	protection
time spent on	.	.	.
production	+,2	-,1	+,3
predation	-,2	+,1	-,3
protection	+,3	-,1	+,3

These connections do not change when they are restricted to the two subpopulations of producers and predators.

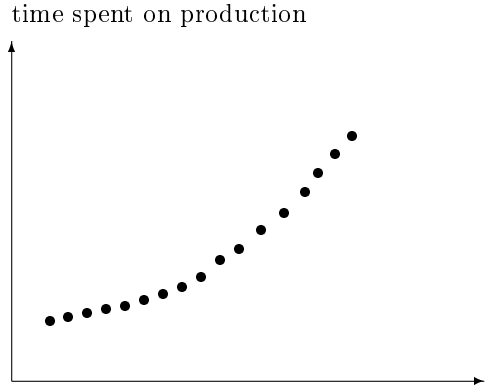
The absence of a regular increase of protection time with an increase of protection ability (even in the subpopulation of producers) we find unsatisfactory. As producers' product increases over time (in the simulation), and as there are many predators, producers should have a strong incentive for protection which is also in accordance with the form of the utility function (9).

In these simulations on might suspect that the results depend on the initial creation of lognormally distributed ability coefficients. In order to control for this we conducted a second series in which we focused on one ability coefficient. When this was fixed, the percentages of producers, predators and protectors (defined in terms of abilities), as well as all the other coefficients were varied randomly. The random creation of the 'other' parameters was repeated 20 and 50 times. Doing the simulation for different values of the focused ability coefficient, like 0.2, 0.25, 0.3, ..., 1, and plotting the times spent on one activity against the focused coefficient, we obtained qualitatively the same results as in the first series. Figure 2 shows some dependencies for the series 0.2, 0.25, 0.3, ..., 1 of coefficients number s on the x-axis and times number r on the y-axis.

In a third series, we investigated the sensitivity of the model in dependence of the absolute numarical values of the ability coefficients. Instead of normalized ability coefficients (adding up to 1) we used larger numbers, and studied the system's behavior for different, fixed sets of coefficients and proportions of producers and predators. We started with normalized coefficients, multiplied them by 10, 20, 30 and gauged the (1-...) expressions in (9) to the absolute values, e.g. when using coefficients adding up to 10, the '1' was replaced by '10'. In a population of 20 actors we ran all combinations of coefficients (0.8,0,9.2), (0.4,0.4,0.2), (0.1,0.1,0.8) for producers, (0,1,0), (0,0.7,0.3),(0.3,0.4,0.3) for predators and percentages 100, 80, 60, 40, 20 of producers in the population.

There was no significant variantion of predators' times in dependence on the absolutesizes of ability coefficients, and variantion for producers was relatively small, never exceeding 30%. We may say that the model is moderately robust with respect to the absolute sizes of ability coefficients.

Figure 2



We also varied the exponents α_i attached to predation times. In (9), and in the earlier simulations these exponents had been uniformly set equal to 1/2. In a fourth series of exponents 1/2 were replaced by smaller and larger values (0.2,0.4,0.8,1), but still each acots's utility function was calculated with the same exponents. Running the simulation in the setting of series 3 above we found that the times of predators are hardly affected by changes of the exponents. The main effect observed for producers was that when their percentage in the population decreases below a threshold, they split their times nearly equally on production and predation. The only effect of varying exponents is that this threshold decreases with growing exponent, but also with decreasing predating ability of the predators.

In a final series we tried to reproduce 'reasonable' empirical time distributions as found in existing populations. For example, in a slave holder society (Knight, 1977), a first guess for time distributions would be (1,0,0) for slaves (which form, say 40 percent of the population), and (0,0.6,0.4) for non-slaves (making up 60 percent of the population). That is, slaves spend all their time of production, while non-slaves split their time on 60% of predation and 40% of protection. We started a search program which tried to find ability coefficients for which the time distributions resulting in a simulation with such coefficients fitted with the times and percentages fixed beforehand.

This resulted in complete failure. For none of three 'reasonable', initial time distributions and percentages the program found coefficients such that the simulation results would fit with the given times and percentages. Even if we admit that the search algorithm used is perhaps very inefficient this indicates that the model in its present form is not sufficiently flexible.[9]

[9] Of course, this does not mean that this kind of of fitting is the only kind of validation prodcedure

Conclusion

First simulations with a multi-agent model in which actors optimize the time distributions for production, predation and protection yield insight in the rational, non-institutionalized incentives for engaging in each of these activities. We found the predation is 'robust' in the sense that actors who are best at predating in lostbcases spend almost all their time on predation. This points to a natural incentive which theoretically could back Hobbes' state of nature. A second positive result is that production time also increases with the increase of the ability to produce, though the degree of increase varies with other parameters, in particular with the coefficients for the other abilities and the percentage of producers in the population. This also indicates a natural incentive, and the variability of increase opens the way for studying the systematic effects of other,'external' parameters on the incentive to produce.

Negatively, we found that protection time in most cases does not monotonically increase with protection ability. A first interpretation is that the ability for protection is dominated by the two other abilities, and thus does not really an independent variable. This interpretation is also supported by the intuitive observation that the abilities for predation and protection in a pre-historic enviroment ar closely related to similar kinds of bodily skills and strengths.

We were not able with the present model to produce 'real life' time distributions and percentages of producers and predators. This may be have two reasons. First, the model's basic equation (9) may be too rigid or too restricted. In future research we will use variations of the model with different exponents and different overall forms of (9) to find 'solutions' which reproduce given, plausible time distributions and percentages. In particular, the absence of predation among predators in (9) has to be removed.

A second reason for failure may be the neglect of institutional features. Broadly speaking, institutions seem to produce and to stabilize certain patterns of time distributions and percentages which do not naturally occur in the institution-free state. We hypothesize that the present model allows to incorporate some such institutional features, which we hope to find and inclcude in the picture.

References

Albert, A. 1999: Les modèles économiques d'anarchie: une revue de la litérature, manuscript.

Axelrod, R. 1997: *The Complexity of Cooperation*, Princeton NJ: Princeton UP.

Bainbridge, R. et al. 1994: Artificial Social Intelligence, *Annual Review of Sociology* 20, 401-36.

for the model.

Balzer, W. 1990: A Basic Model of Social Institutions, *Journal of Mathematical Sociology*, 16, 1-29.

Balzer, W. 1993: *Soziale Institutionen*, Berlin: de Gruyter.

Balzer, W. 1996: On the Measurement of Action, in: R.Hegselmann et al. (eds.), *Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View*, Dordrecht: Kluwer, 141-56.

Balzer, W. 1999: SMASS: A Sequential Multi-Agent System for Social Simulation, to appear in the Proceedings of the 1997 Dagstuhl Conference, R.Suleiman et al. (eds.).

Balzer, W. & Brendel, K. 1996: DMASS: A Distributed Multi-Agent System for Social Simulation, manuscript.

Binmore, K. 1998: Review of R.Axelrod, The Complexity of Cooperation, *Journal of Artifical Societies and Social Simulation*, http://www.soc.surrey.ac.uk/JASSS/1/1/review.html

Bush, W. C. 1976: The Hobbesian Djungle or Orderly Anarchy, in: A. T. Denazu & R. J. Mackay (eds.) *Essays on Unorthodox Economic Strategies*, Blacksburg VA: University Publications, 27-37.

Commons, J. R. 1934: *Institutional Economics: Its Place in Political Economy*, New York: Macmillan.

Congleton, R. D. 1997: Political Efficiency and Equal Protection of the Law, *Kyklos* 50, 485-505.

Dean, J. S. 1998: Understanding Anasazi Culture Change through Agent Based Modelling, http://www.santafe.edu/sfi/publications/working-papers/98-10-094.pdf

Doran, J. 1995: Simulating Prehistoric Societies: Why and How?, Aplicationes Informaticas in Arqueologia: *Teorias e Sistemas* 2, Bilbao, 40-55.

Dowd, K. 1997: Anarchy, Warfare and Social Order: Comment of Hirshleifer, *Journal of Political Economy*, 105, 648-51.

Flap, H. 1985: Conflict, loyaliteit en gewald, Dissertation, University of Utrecht (Netherlands).

Gilbert G. N. & Doran, J. (eds.), 1994: *Simulating Societies*, London: UCL Press.

Hegselmann, H., Mueller, U. & Troitzsch, K. G. (eds.), 1996: *Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View,* Dordrecht: Kluwer.

Hirshleifer, J. 1995: Anarchy and its Breakdown, *Journal of Political Economy* 103, 26-52.

Hodgson, G. M. 998: The Approach of Institutional Economics, *Journal of Economic Literature* 36, 166-92.

Houba, H. & Weikard, H. 1995: Interaction in Anarchy and the Social Contract: A

Game-theoretic Perspective, Working Paper # TI 95-186, Tinbergen Institute.

Hurwicz, L. 1996: Institutions as Families of Game Forms,*The Japanese Economic Review* 47, 113-32.

Knight, F. W. 1977: The Social Structure of Cuban Slave Society in the Ninetheen Century, in: V. Rubin & A. Tuden (eds.), *Comparative Perspectives on Slavery in New World Plantation Societies*, Annals of the New York Academy of Sciences, Vol.292, New York, 297-306.

Mauss, M. 1960: Essai sur le don. Forme et raison de léchange dans les sociétés archaiques, in: M. Mauss, *Sociologie et Anthropologie*, Paris: Presses Universitaires de France. 145-279. (orig.1923-24).

Nowak, A. & Vallacher, R. 1998: Toward Computational Social Psychology: Cellular Automata and Neural Network Models of Interpersonal Dynamics, in: *Connectionist Models of Social Reasoning and Social Behavior*, S. J. Read and I. C. Miller (eds.), Mahmaw NJ: Lawrence Erlbaum Publishers, 277-311.

Prietula, M. & Carley, K. 1994: Computational Organization Theory: Autonomous Agents and Emergent Behavior, *Journal of Organizational Computing* 4, 41-83.

Prietula, M., Carley, K. & Gasser, L. (eds.), 1998: *Simulating Organizations: Computational Models of Institutions and Groups*, Cambridge MA: MIT Press.

Rider, R. 1993: War, Pillage and Markets, *Public Choice* 75, 149-56.

Schotter, A. 1981: *The Economics of Institutions*, Cambridge: Cambridge University Press.

Testart, A. 1998: L'esclavage comme institution, *L'Homme* 145, 31-69.

Tesfatsion, L. 1998: Agent-Based Computational Economics: A Brief Guide to the Literature, http://www.econ.iastate.edu/tesfatsi/ace.htm

Tuomela, R. 1995: *The Importance of Us*, Stanford: Stanford UP.

Usher, D. 1993: *The Welfare Economics of Markets. Voting and Predation*, Michigan University Press.

Wielemaker, J. 1993: SWI-Prolog 1.8, Reference Manual, University of Amsterdam, Dept. of Social Science Informatics.

Wielemaker, J. 1996: Programming in XPCE/Prolog, University of Amsterdam, Dept. of Social Science Informatics.

Wintrobe, W. 1998: *The Political Economy of Dictatorship*, Cambridge: Cambridge University Press.

Wolfesperger, A. 1995: *Économie Publique*, Paris: Presses Universitaires de France.

Wolff, J. 1996: *An Introduction to Political Philosophy*, Oxford, Oxford University Press.

Urban's Rule

Programmed by *Joseph Urban*

The cartesian product of two sets represented as lists.
The cartesian product is the empty set, if one of the input sets are empty.

```
cart([],_,[]):-!.
cart(_,[],[]):-!.
cart([A|ARest],BL,CL):-
    create_tuple(A,BL,TList),
    cart(ARest,BL,CInterim),
    append(TList,CInterim,CL).

create_tuple(_,[],[]).
create_tuple(A,[B|BRest],[[A,B]|TRest]):-
    create_tuple(A,BRest,TRest).
```

A cartesian product of n sets is respresented as lists.
A cartesian product is the empty set, if one of the input sets are empty.
The predicate is undefined, if by mistake input list contains only one list.

```
n_cart(NList,[]):- member(NList,[]),!.
n_cart(NList, undef):- length(NList,1),!.
n_cart([AL,BL|Rest],CL):-


    % take the first two sets and create the cartesian product of these
    % two sets first

    cart(AL,BL,C),

    % now extend the initial tuples by processing the remaining lists
    % so that you get a list of n-tuples which is the cartesian product

    extend_cart_tuple([C|Rest],CL).

extend_cart_tuple([C|[]],C).
extend_cart_tuple([TupleList,Next|Rest],CL):-
    help_extend_cart_tuple(TupleList,Next,ETupleList),
    extend_cart_tuple([ETupleList|Rest],CL).

help_extend_cart_tuple([],_,[]).
help_extend_cart_tuple([T|TupleList],List,ExtTupel):-
    extend_tuple(T,List,EList),
    help_extend_cart_tuple(TupleList,List,IList),
    append(EList,IList,ExtTupel).
```

```
extend_tuple(_,[],[]).
extend_tuple(T,[E|Rest],[ExtT|ERest]) :-
    append(T,[E],ExtT),
    extend_tuple(T,Rest,ERest).

% predicates for testing the cartesian product of two sets:

test(C):-
    cart([1,2],[3,4],C).

test0(C):-
    n_cart([[1],[3,4],[1,6,7]],C),
    assert(prot(C)).

test(A,B,C):-
    n_cart([[1,2,3],[2],[4,5],[6,7,8],[9]],A),
    assert(prot(A)),
    n_cart([[1,2],[3,4],[5,6],[7,8],[9,10],[11,12,13,14,15]],B),
    assert(prot(B)),
    n_cart([[1,2],[],[5,6]],C),
    assert(prot(C)).
```