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

[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).

at providing a flexible frame in which the user may himself insert variable 'rules of behavior' 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

[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).

empties its 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

[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.

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 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]

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

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 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). Basiclly, 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

[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.

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.

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, 'RE-
TURN.' '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 'RE-
TURN.' '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 $\leq$ 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 $\leq$ 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 $\star$ 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 $\geq$ 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 indi-

vidual 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 accord-

ing 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 $\geq$ 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

W.Balzer, 1990: A Basic Model for Social Institutions, Journal of Mathematical Sociology 16, 1-29.
W. Balzer, 1993: Soziale Institutionen, Berlin: de Gruyter.
W.Balzer, 1992: A Theory of Power in Small Groups, in: H. Westmeyer (ed.), The Structuralist Program in Psychology, Bern: Hogrebe, 191-210.
W.Balzer, 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.
E.Cohors-Fresenborg, 1977: Mathematik mit Kalkülen und Maschinen, Braunschweig: Vieweg.
W.F.Clocksin & C.S.Mellish, 1984: Programming in PROLOG, Berlin etc.: Springer, 2nd ed.
P.Derantsart, A.Ed-Dbali, L.Cervoni, 1996: PROLOG: The Standard, Berlin etc. Springer.
R.Hegselmann, 1994: Solidarität in einer egoistischen Welt: eine Simulation, in J.Nida-Ruemelin (Hrsg.), Praktische Rationalität, Berlin: de Gruyter, 349-90.
W.B.G.Liebrand and D.M.Messick, 1992: Computer Simulation of the Relation Between Individual Heuristics and Global Cooperation in Prisoner's Dilemmas, Social Science Computer Review 11, 301-12.
M.Ostermann, F.Bergmann, G.v. Walter, 1995: Brain Aid Systems, Dokumentation von BRAIN AID PROLOG, erhältlich von M.Ostermann, Bogenstr.9, 52080 Aachen.

A.S.Rao & M.P.Georgeff, 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.