# 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 [18], [20], [22], [24] 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 [3], [15], [29]. A third, recent approach uses object-oriented software like [14] or [30]. 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 [11] 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 [30], [31] 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 [6] and [9].

## 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 [12], [26]. 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 [1]. 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 [4], [5]. 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 [7]. 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 [18], [22], [24].

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 [2], [23], [34], [35].

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' [8] 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 [25], [16], [28]. 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 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.[1]

## 5 RULES OF BEHAVIOR

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

[1]In a first, properly distributed version of SMASS, called DMASS [10] 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.

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 [13], SANP [21] or the proposal in [19] 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 $t+1$.[2]

[2]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.

## 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 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 [32] (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 [33] 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

[1] P. Agre, 1997: Computation and Human Experience, Cambridge/Mass., Cambridge University Press.

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

[3] R.Axelrod, 1984: The Evolution of Cooperation, New York: Basic Books.

[4] W.Balzer, 1985: Incommensurability, Reduction, and Translation, Erkenntnis 23, 255-67.

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

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

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

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

[9] W.Balzer, 1993: Soziale Institutionen, Berlin: de Gruyter.

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

[11] K.M.Carley and M.J.Prietula, 1994: ACTS Theory: Extending the Model of Bounded Rationality, in K.M.Carley and M.J.Prietula (eds.), Computational Organization Theory, Hillsdale NJ: Lawrence Erlbaum, 55-87.

[12] P.R.Cohen, H.J.Levesque, 1990: Intention is Choice with Commitment, AI 42, 213-61.

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

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

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

[16] A.I.Goldman, 1970: A Theory of Human Action, Engelwood Cliffs: Prentice Hall.

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

[18] R.Hegselmann, 1996: Cellular Automata in the Social Sciences, Perspectives, Restrictions, and Artefacts, in [15], 209-33.

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

[20] W.B.G.Liebrand and D.M.Messick, 1996: Computer Simulations of Sustainable Cooperation in Social Dilemmas, in [15], 235-47.

[21] Man Kit Chang and Carson C.Woo, 1992: SANP: A Communication Level Protocol for Negatioations, in E.Werner and Y.Demazeau (eds.) Decentralized A.I.-3, Amsterdam: Elsevier, 31-54.

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

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

[24] A.Nowak and M.Lewenstein, 1996: Modelling social Change with Cellular Automata, in [10], 249-85.

[25] 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.

[26] A.S.Rao, M.P.Georgeff, 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.

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

[28] R.Schiffer, 1987: Remnants of Meanings, Cambridge/Mass.: MIT Press.

[29] R.Schüssler, 1990: Kooperation unter Egoisten: Vier Dilemmata, München: Oldenbourg.

[30] SWARM: http://www.santafe.edu/projects/swarm/.

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

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

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

[34] M.J.Wooldridge, N.R.Jennings (eds.), 1991: Intelligent Agents, Berlin: Springer.

[35] M.Wooldridge, J.P.Mueller, M.Taube (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 [32] 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]). type_of_neighbourhood(schellin,moore,1).
type_of_neighbourhood(takeweak,von_Neumann,2). choose_run(1).
% 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(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)),!.
```

% 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.
```