EPOS 2006

Epistemological Perspectives on Simulation

II Edition
University of Brescia, Italy
October 5-6, 2006

Algorithmic Analysis of Production Systems used as Agent-Based Social Simulation Models

Jim Doran*

Algorithmic analysis of models is quite common in general, but is rarely attempted in the context of agent-based social simulation. We explore the algorithmic analysis of simulation models that take the form of production systems as these are defined in computer science. Several implemented analysis algorithms for a particular type of production system are discussed, including algorithms for agent discovery and for model abstraction. Examples of the use of these algorithms are given and their potential briefly considered.

Key Words: model analysis, production systems, agent discovery, model abstraction

* Department of Computer Science, University of Essex, Colchester CO4 3SQ, UK

1. Introduction: Model Analysis

Agent-based social simulation relies upon the precise specification of a model in, typically, a computer programming language such as C++ or Prolog, or a multi-agent system oriented software environment such as SWARM or SDML, and then the repeated execution of the model on a computer to acquire sufficient experimental data for reliable conclusions about its behaviour to be drawn. However, as has long been recognised, this is not the only possible way to proceed (e.g. Doran and Gilbert, 1994, section 1.2.2; Teran, Edmonds, and Wallis, 2001; Gilbert and Troitzsch, 2005). A model's structure can sometimes be directly analysed to obtain information about its behaviour without its ever being executed. This can be a much faster means of obtaining properties of a large and complex model than are systematic experimental trials (see Doran, 2005, 2006). Furthermore, such analysis is arguably more natural in the sense that it corresponds more directly to the mental processes of human beings when they reflect upon dynamic systems and their behaviour.

However, it is also true that direct analysis of computer programs is very difficult, often intractable. This prompts consideration of *production systems* (in the computer science sense e.g. Wulf et al, 1981 pp 550-555; Russell and Norvig, 1995, p 314). In effect, production systems are programs in a certain class of specialised "programming languages" (CLIPS is a well known example – see, for example, Giarratano and Riley, chapters 7-10) that have full computational power but are simple in structure and hence relatively easy to analysis. We emphasise that production systems can capture complex social phenomena, including agents that learn and communicate, just as well (or as badly) as any other program[1].

Here we define a class of production systems, and hence production system models, and use this definition to seek to design algorithms of practical use that analyse realistic models, particularly as regards the agents that they contain, the abstractions that they support, and the interactions between agents and abstraction.

2. Production Systems

DEFINITION: A *production system (PS)* as we define it (definitions elsewhere vary in detail) comprises three parts:

> A *set of variables* each with its own associated possible *value set*. No variable may be associated with (i.e. bound to) more than one of its values at the same time. A set of bindings that associates a value with each variable is a *state*.
>
> A *set of rules* each in the form
>
> set of variable/value bindings => single variable/value binding

[1] We avoid mathematical logic based representations as they introduce complexity that seems unnecessary for our purposes.

in a suitable syntax. A rule has a “left hand side” (LHS) – the binding set -- and a “right hand side” (RHS) – the single binding. Rules are not to be interpreted to specify logical entailment, but rather consequence in time. Given a state, the rules generate a successor state.

An *execution procedure* that matches all rule LHSs to the existing state variable/value bindings and executes in parallel those rules that match by asserting the corresponding RHS bindings. Hence the current set of bindings is updated and a successor state generated. Bindings that are not updated by a rule execution persist unaltered to the successor state (compare *frame axioms* and their use, Russell and Norvig, 1995, page 206).

Importantly, we specify that *matching rules are executed in parallel*. The “conflict resolution” issue prominent in alternative formulations of production systems is thus not an issue here. It is replaced by the need to avoid assertion of contradictory bindings (see later).

The execution procedure is invoked repeatedly, and a production system thus generates a sequence of states. More abstractly, a production system implements repeated invocation of a many-one mapping over the set of possible states.

The production systems we have defined have three key properties. *Firstly*, there are no intra-rule variables, that is, rule LHSs do not have unbound variables that are bound during matching and the bindings then used on the RHS. *Secondly*, since matching rules are executed in parallel, the rule set of a PS must never set contradictory bindings, that is, it must never happen that as one rule sets X/a another sets X/b. *Thirdly*, it is assumed that value sets are finite and relatively small. Thus we must work with quantitative variables as if they were qualitative and design rules sets accordingly.

The foregoing three requirements are imposed to make algorithmic analysis more tractable. They are both important and quite demanding. The burden of meeting them falls upon the designer of the rule set ie the “programmer”.

3. Two Examples of Production System Models

DEFINITION: A *production system model* is a production system that is being used as a model.

As a first example of a PS model, consider the following small set of rules that model an aircraft landing (or crash landing!):

```
flight_status/in_air & engines/running &
pilot_status/ok => flight_status/landed

flight_status/landed => engines/not_running

engines/running & fuel/n => fuel/(n-1)  (100 >= n >=
1)

fuel/0 => engines/not_running
```

```
engines/not_running => altitude/ground-level
```

A plausible initial state is:

```
flight_status/in_air
engines/running
fuel/100
pilot_status/ok
```
[contrast with `pilot_status/faulty`]

Given this initial state, then the next state will be:

```
flight_status/landed
engines/running
fuel/99
pilot_status/ok
```

From these rules, it is easy to see that in all circumstances the aircraft must arrive at ground level with engines not running, whether or not pilot_status is initially ok.

A second example PS model, this time modelling in outline a lecture being given, is the following *standard lecture model:*

```
lecturer/speaking-well & content/knowledgeable
=> class/very-interested

lecturer/speaking-poorly & content/knowledgeable &
class/bored   => class/somewhat-interested

lecturer/speaking-poorly & content/knowledgeable &
class/very-interested   => class/somewhat-interested
lecturer/speaking-well   & content/ignorant  & class/bored
=> class/somewhat-interested

lecturer/speaking-well & content/ignorant  &
class/very-interested    => class/somewhat-interested

lecturer/speaking-poorly & content/ignorant    =>
class/bored

class/bored => lecturer/speaking-well
                                //the lecturer tries
                            harder!

lecturer/ speaking-poorly & class/somewhat-interested
=> class/bored

content/ignorant  & class/somewhat-interested  =>
class/bored
```

A possible initial state for this model is:

```
lecturer/speaking-poorly
content/ignorant
class/very-interested
```

It is reasonably easy to see by inspection of these rules that whenever *content* is *ignorant* there will be an oscillation in the class's level of interest between *somewhat-interested* and *bored*.
In this model there are, intuitively, just two agents: the lecturer and the class (collectively). In the model each of these agents comprises a single variable, which is counter-intuitively simple. In general an agent within such a model will consist of a subset of the system's variables, some of them to be interpreted as representing aspects of the agent's mental state, with associated rules.

4. Analysis Algorithms

Working from the foregoing definition of a production system, algorithms have been designed, programmed (in C) and tested to:

(a) Execute a production system
(b) Discover the prerequisite initial conditions of a specified state of a PS
(c) Discover the stable states of a given PS
(d) Generate "powerful" abstractions of a given PS model.
(e) Identify the (minimal) agents in a given PS model

Algorithm (a), RUN PS, executes a PS and is relatively straightforward. In pseudo-code:

```
Repeat
{
     Match all rule LHSs against current state and set
     RHS bindings of matching rules into current
     state, overwriting existing bindings as required
 }
```

Algorithm (b), PRECURSOR, uses "backward chaining" to find the precursor states of a particular PS model state. In the aircraft-landing model, for example, it enables algorithmic discovery of the fact that from all initial states the aircraft arrives at ground level with engines still. It also enables discovery of all initial states for the lecture model that lead to the class ending the lecture "very-interested". The algorithm is more complex than at first appears because of the need to handle the "frame problem" (Russell and Norvig, 1995, page 207).

Algorithm (c), STABLE STATES, finds stable states for the production system, that is states that once reached are never left. Finding stable points is a standard and useful method of analysis for systems of differential equations. Indeed, systems of differential equations have much in common with production systems. This is illustrated by the fact that the standard methods of simulating systems of differential equations (e.g. Runge-Kutta) proceed by first replacing them by in effect a production system.

A stable state of the standard lecture model turns out to be:

```
lecturer/speaking-well
content/knowledgeable
class/very-interested
```

5. Abstraction of a PS Model

A model of a lecture can be created at many levels of abstraction. A more refined model than that given above might include representations of a chairman, of slides and slide showing, and of a clock. More interesting and complex (and more obviously social) are lecture models that include representations of the mental states of the individual members of the class, of learning processes in individuals, and of interactions between class members. In fact, an *elaborated lecture model* has been created and experimented with at approaching this last level of detail. This is a production system with over one hundred rules. It includes simple representations of slides being shown, of (three) class members learning, and even of two of the class members playing the little game of "paper, scissors and stone".

Several researchers have investigated algorithmic abstraction of models, notably Zeigler (1990), Fishwick (1995), and Ibrahim Y and Scott P (2004). It is standard to see models that are abstractions of a base model as the outcome of behaviour preserving homomorphisms. The difficulty is to decide what we mean by "good" abstractions and actually to find them given an original base model. Our approach here is unusual in that we see abstraction as a process best applied to a production system without reference to any interpretation it may have as a model or to any particular type of structure (e.g. agents) that an interpretation may see within it.

More specifically, a set (or "space") of alternative abstractions of a given base model may be generated by sequences of abstraction steps. Here we specify an *abstraction step* to be the combination of two variables into one, with the value set of the new combined variable comprising the result of applying a randomly chosen[2] many-one mapping to the cross-product set of the value sets of the two original variables.

In the work reported here (algorithm (d) above, ABSTRACTION) these abstraction steps are used within a (heuristically guided) *hill-climbing search* (see Russell and Norvig, 1995, page 112). Successive abstraction steps are chosen and appended to the existing sequence of adopted abstraction steps. The choice between alternative abstraction steps is made by always selecting the abstraction step that is assessed as most promising.

The *promise* of an abstraction step is essentially measured as the (inverse of the) number of indeterminate (ie one-many) abstracted state successions to which the proposed abstraction step (and preceding step sequence) gives rise from a

[2] Fully random choice is weak. Experimentation with heuristically guided but partially random choice ongoing.

sample of basic model state trajectories generated for the purpose. The rationale for this approach is that we accept a "class", say, as an abstraction of a set of "students" if we find the concept of a class useful as a means to simple and sound predictions and explanations.

In pseudo-code, ABSTRACTION is:

```
    Input the base model

    Until no further abstraction is possible
    {
        Generate a set, S, of alternative abstraction
steps

        Select and adopt the most promising abstraction
        step from S by testing each possible abstraction
        step on an ad hoc sample of state trajectories.
    }
```

Applying ABSTRACTION to the standard lecture model yields the following simple abstract model that highlights the possible oscillation:

```
V/x  => V/x
V/p  => V/q
V/q  => V/p
```

This model has just one variable, and it is apparent that either the state is constant (V/x) or it alternates between V/p and V/q.

Finding powerful abstractions potentially provides insight into the essential and significant behaviour of a model and of the phenomena that it models (compare Gilbert, 2006). The existing algorithm can generate a range of more abstract models from the standard lecture model listed above and highlights key behaviours. Experimental application of ABSTRACTION to the *elaborated lecture model* is ongoing.

The strategic difference between this approach and those mentioned earlier is that we define abstraction operations at the "fine-grained" level – that is, at a level below that at which such macro-phenomena as agents are naturally considered -- and that we have deployed a standard search procedure to locate good abstracted models in the space of possible abstracted models.

6. Agents in a PS Model

Can a PS model contain (representations of) agents? The answer is surely yes (see earlier). With each agent in the model interpretation there may be associated a subset of the model's set of variables, some of them reflecting presumed "mental" and "emotional" attributes. A subset of the rules may also be associated with a particular agent. Other variables and rules will represent interactions and the agents' collective environment. It is thus natural to ask the following question. Can the agents that a model creator(s)'s interpretation of the model suggests to be present within it, be detected by an "objective" algorithm that has no access to the interpretation?

For example, in the foregoing lecture model there are intuitively just two agents, the lecturer and the class. But would an objective analysis of the production system itself reveal these two agents. In other words, do the agents in an agent-based model have a real computational existence beyond the interpretation read into the model by its designer and users? Should they? Can there be other agents in the model that its creator is not aware of, that is, which are in some sense emergent from the model's structure? We suggest that the ability to discover the minimal agents formally present in a PS model enables comparison with the set of agents the model's designer *conceives* as present, possibly leading either to modification of the model or of the model design concept.

However, defining in computational terms just what is an agent in the context of a PS is difficult, and widely recognised to be so. Standard textbook definitions are either too specialised or too ambiguous to offer much help[3]. We define a *minimal agent* in a production system by reference to the notion of "influence" as follows (adapted from Doran, 2002):

> One variable in a production system is said to *influence* another when there exists at least one rule that refers to the former on its LHS and binds the latter on its RHS.
>
> A *minimal agent* is a non-empty subset of the variables of a production system. These variables must each be one of *input*, *output*, or *internal* variables where the meaning of these agent variable types is defined by reference to influence.
>
> - An input variable is influenced only by variables external to the agent and influences only output variables and/or variables internal to the agent.
> - An output variable is influenced only by input variables and/or internal variables of the agent and influences only variables external to the agent.
> - An internal variable is influenced by, or influences, only other variables of the agent i.e. internal, input or output variables.
>
> An agent must contain at least one input variable and one output variable, and must include ALL variables that are internal consequent on these input and output variable sets.

The rationale for this definition is to express a minimum intuitive requirement for an agent: an agent "perceives", "decides", and "acts". A minimal agent according to this definition may be very simple, no more than an input and an output, or it may be highly complex and "cognitive" – neither is precluded. Typically, the agents (or representations of agents depending on your viewpoint)

[3] E.g. "An *agent* is a computer system that is *situated* in some *environment* and that is capable of *autonomous* action in this environment in order to meet its design objectives" (Wooldridge, 2002, p.15)

in current agent-based social simulation studies rarely comprise more than a few rules and variables. Notice that according to this definition two agents may or may not be disjoint – consider that the audience in a lecture is both a single agent and made up of many agents. Agents may also be nested.

Algorithm (e), FIND AGENTS, is able to discover the agents, in this sense, in a production system. However, at present it is limited to finding agents that have exactly one input variable and exactly one output variable. In outline pseudo-code FIND AGENTS is as follows:

```
Foreach possible pair of variables V_input, V_output

     {Find the set of variables S1 directly or indirectly
          influenced by V_input
     Find the set of variables S2 that directly or
          indirectly influence V_output
     If V_input is not itself a member of S1
          and V_output is not itself a member of S2
     then, provided the intersection of 1 and S2 is not
null,
          the union of S1 and S2 is an agent
     }
```

When applied to the standard lecture PS model given earlier, this algorithm finds no agents as is to be expected. There are only three variables in the model, and its structure is too simple for it to have agent content. However, when the model is developed so that explicit perceptual input and appearance output variables are associated with the class (for details see Appendix A), then the three variables now constituting the class are indeed recognised as an agent, as also are the variables that together constitute the "environment" of the class.

Using the above definition it is possible to make precise, for example, how two or more minimal agents may be viewed as a single agent (compare Bosse and Treur, 2006). It is an open question how best to strengthen the definition to render the agents it delimits more clearly cognitive whilst keeping the definition sufficiently precise for purpose.

7. Agents and Abstraction

It is natural to ask under what conditions agents existing in a PS model are preserved under the abstraction process. A plausible conjecture is that if abstraction steps are constrained always to use pairs of variables that are either both within or both outside every agent in the model, those agents will either be preserved or reduced to a single variable (thereby ceasing to be an agent) by abstraction. Agents will never be introduced by the abstraction step. However, were two variables one within and one outside an agent to be combined, then this would be likely to destroy agenthood. Hence it seems that there do exist conditions under which abstraction preserves agents until they are reduced to single variables. The significance of this insight is that if the original set of agents in a model is an important part of our view of the target system to which the

model relates, in practice we may wish an abstraction process to have this agent-preserving property. More generally, we can ask for each type of macro-structure that we find of interest in the PS model under what conditions it is preserved under abstraction.

8. Discussion and Conclusion

This paper has addressed automatic analysis of a model to gain insights into its behaviour. Few would dispute that in principle such analysis can be useful. But the practical value of the lines of attack presented in this paper depends not just upon the computational feasibility of the algorithms discussed, which has been shown at least for simple cases, but also on the possibility of creating non-trivial PS models in the first place. There is no doubt that constructing PS models of the type we have considered is potentially laborious and intricate, primarily because there are no intra-rule variables, and because the value sets must be small. Some systematisation of the model creation process looks essential. Models with "spatial" structure may be relatively easy to construct. There are other options. It may be possible to specify models in a higher-level language and then automatically compile them down to the simple PS level considered here. Or it may be possible to "lift" the analysis algorithms somewhat. If either of these options proves effective, then the burden upon the model creator(s) will be correspondingly reduced.

It is trivial that if models are faulty, then algorithmic analysis of them will yield faulty "insights". Thus the algorithms presented here rely for their practical value on the validity of the models to which they are applied. Nothing has been said in this paper about model validation. We have concentrated entirely on ways of finding out what is the behaviour of a given model from its computational form.

We finally conclude that to the extent that valid PS models can be created, and analysis algorithms of the type described above applied to them – and analysis of large models comprising hundreds of rules is quite tractable – then these techniques promise to be useful alongside (not, of course, instead of) the standard methods of agent-based social simulation.

References

Bosse T., and Treur J. (2006) Formal Interpretation of a Multi-Agent Society As a Single Agent, *JASSS***,** 9(2) http://jasss.soc.surrey.ac.uk/9/2/6.html

Doran, J. E. (2002) Agents and MAS in STaMs. In: *Foundations and Applications of Multi-Agent Systems: UKMAS Workshops 1996-2000, Selected Papers* (eds. M. d'Inverno, M. Fisher, M. Luck, and C. Preist), Springer Verlag, LNAI 2403, 131-151.

Doran J. E. (2005) Iruba: an Agent Based model of the Guerrilla War Process. In: *Representing Social Reality*, Pre-Proceedings of the Third Conference of the European Social Simulation Association (ESSA), Koblenz, Sept 5–9, 2005, ed Klaus G Troitzsch, Folbach. Pp 198-205.

Doran J. E. (2006) Modelling a Typical Guerrilla War *Proceedings IEEE DIS2006 Workshop (Knowledge Systems for Coalition Operations strand),* Prague, June 2006.

Doran J. E. and Gilbert N., (1994). Simulating Societies: an Introduction. In: *Simulating Societies* (Eds. N. Gilbert and J.E. Doran) UCL Press, pp 1-18

Fishwick, P.A., (1995) *Simulation Model Design and Execution.* Prentice-Hall.

Giarratano J and Riley G (1989) *Expert Systems: Principles and Programming*, PWS-Kent Pub. Co. Boston.

Gilbert N., (2006) Kiss and Tell: in praise of abstraction. EPOS 2006, Brescia

Gilbert N. and Troitzsch K., (2005) *Simulation for the Social Scientist* UCL Press: London (2^{nd} ed.)

Ibrahim Y. and Scott P. (2004) Automated Abstraction for Rule-Based Multi-Agent Systems. Proc. *ESSA 2004,* Valladolid, Spain

Russell S. and Norvig P. (1995) *Artificial Intelligence, a modern approach.* Prentice Hall.

Teran O., Edmonds B., and Wallis S. (2001) Mapping the Envelope of Social Simulation Trajectories In: *Multi-Agent-Based Simulation* (eds. S Moss and P Davidsson) Springer LNAI 1979 pp. 229-243

Wooldridge M. (2002) *MultiAgent Systems* Wiley and Sons: Chichester.

Wulf W.A., Shaw M., Hilfinger P.N. and Flon L. (1981) *Fundamental Structures of Computer Science,* Addison-Wesley

Zeigler B. P. (1990) *Object-Oriented Simulation with Hierarchical, Modular Models: intelligent agents and endomorphic systems.* Academic Press.

APPENDIX A

There follows the rule set for the standard lecture model with added structure. This structure takes the form of a variable *classp* that mediates perceptions made by the class and a variable *classa* that mediates the external appearance of the class. The three variables *classp, class* and *classa* are then collectively recognised as an agent by FIND AGENT. The variables *classa*, *lecturer*, *content* and *classp* are also collectively recognised as an agent (in effect, the agent's "environment"), itself a sequential combination of two agents.

```
lecturer / speaking-well & content / knowledgeable =>
classp / well-k

classp  / well-k => class / very-interested

lecturer / speaking-poorly & content / knowledgeable =>
classp / poorly-k

classp / poorly-k & class / bored  =>
class / somewhat-interested

classp / poorly-k & class / very-interested   =>
class / somewhat-interested

lecturer / speaking-well & content / ignorant  =>
classp / well-i

classp / well-i & class / bored => class / somewhat-interested

classp / well-i & class / very-interested  =>
class / somewhat-interested

lecturer / speaking-poorly & content / ignorant   => classp /
poorly-i

classp / poorly-i => class / bored

class / very-interested => classa / very-interested

class / somewhat-interested  => classa / somewhat-interested

class / bored => classa / bored

classa / bored => lecturer / speaking-well

classp / poorly-k & class / somewhat-interested  =>
class / bored

classp / poorly-i & class / somewhat-interested  =>
class / bored

classp / well-i & class / somewhat-interested => class / bored

classp / poorly-i & class / somewhat-interested =>
class / bored
```