Beyond the Design Stance: The Intention of Agent-Based Engineering

Bruce Edmonds and Joanna J. Bryson
CPM and MMU; University of Bath
United Kingdom

October 17, 2003

Abstract

1 Introduction

This paper is a manifesto in progress. We are two researchers who frequently use agents in anger to do our jobs. Yet we have been having trouble finding value in most of the papers that get published in the field of Multi Agent Systems (MAS). This paper represents our attempt to understand why not.

We will start with the most basic question: Why do people use Multi Agent Systems? One answer is that it's not clear they do. We still don't see very many industrial applications of agent technology beyond robotics and AI for computer games, and neither of those use anything like what we generally know as MAS. But there are several good reasons that someone might:

- Because a system needs to be agile and responsive to unpredictable requests and opportunities. This is what the agent-based approach has been promising and providing since at least 1986 (Brooks, 1986), but does not require multiple agents.
- Because parts of this system are distributed across an environment in such a way that communication between parts needs to be taken into consideration. The simple function call is not a good abstraction either due to delays (which would remove the responsiveness) or unreliability (which might freeze an entire program when only part of its functionality is absent, thus reducing opportunism and recovery options.) This is the main engineering reason to think of a system as composed of more than one agent, and to worry about agent communication protocols.
- Because programmers, like all people (Cosmides and Tooby, 1992), are actually pretty good at thinking about entities as social agents with intentions and obligations. Since old programming methods didn't work very well in the dynamic, responsive and distributed environment, the agent-based approach has been developed to make it easier for programmers to get this right.

Our concern is that these solid engineering reasons for developing MAS-based software engineering methodologies seem to have been getting lost in recent years. Instead, the agent community seems to be wandering in a morass of formal systems and theoretical papers. For five years there was a fairly successful series of conferences (beginning with Johnson, 1997) which emphasised working agents and the techniques required to build them, but this has been engulfed into the largely theoretical AAMAS conference, where the vast majority of consistently working systems described seem to be in the minority of papers dedicated to robotics — normally considered a far more difficult engineering domain than straight software.

So what has happened? We believe that for some reason the Agents community has regressed back from the discoveries of eXtreme and agile software toward the older design stance, oddly neglecting the theoretical work proving that such approaches are impossible (Harel, 2003).

We begin by discussing the *Design Stance* — the over-reliance on upfront specifications (also known as *creationism*[1]) — and why it seems completely contradictory that this has infested the MAS community. We then discuss the nine main

[1] "**creationism** n. The (false) belief that large, innovative designs can be completely specified in advance and then painlessly magicked out of the void by the normal efforts of a team of normally talented programmers. In fact, experience has shown repeatedly that good designs arise only from evolutionary, exploratory interaction between one (or at most a small handful of) exceptionally able designer(s) and an active user population — and that the first try at a big new idea is always wrong. unfortunately, because these truths don't fit the planning models beloved of **management**, they are generally ignored." (Raymond, 1992, p. 111)

tools of software systems engineering, how they are currently being used by MAS, and suggestions about how we think their use could be improved.

2 The Design Stance

The primary goal in engineering IT systems can be expressed as:

> To produce IT systems that work well in practice once deployed in their operational context.

One particular strategy for approaching this goal is what can be called the design stance. The design stance can be divided into three stages:

1. To agree the goals for the IT system;
2. To write a specification that would meet these goals;
3. To implement a system that meets this specification.

To push through the design stance one has to ensure three things: that the identified goals are the appropriate ones for the final operating context; that the specification is such that it will meet the identified goals once deployed; and that the implemented system will work according to its specification in practice.

Such an approach works well in relatively simple, static and analysable situations. However such situations are becoming increasingly rare. Today's systems are not closed encapsulated processing units designed to a special and well-defined job, but increasingly interact with many other systems designed by completely different programmers for disparate (and sometimes unknown) purposes.

Systems that are cheap, pervasive and mature become embedded into a human practice that, in turn, adapts to their presence. This embedding means that the IT system cannot be considered in isolation from the web of other systems it is part of. In other words, no separate, off-line analysis of needs and specifications is possible. Any implemented system will affect the human practice it interacts with in somewhat unpredictable ways, so that any goals an IT system was designed for might become out-dated due to its own introduction. For example, introducing an IT system to help identify and combat fraud will almost inevitably mean (if it is successful) that the kind of fraud attempted will change.

Environments or systems which are not easily analysable, that are rapidly changing in unpredictable ways, which are composed of lots of different overlapping aspects, with multiple uses and goals, which are not closed to outside interaction, and which are not designed in a unitary way we will call *messy systems or environments*. Messy environments are the rule — neat ones very much the exception. The combined consequence of messy environment's prevalence and the primary systems engineering goal is that our methodology for constructing IT systems should be such that they work well as parts of a messy system. This does not necessarily mean the IT systems will themselves be messy when considered in isolation, but it may well mean that they are not very simple or neat. Trying to keep them neat so that the design stance is conserved is putting the cart before the horse. We should not be afraid of messy systems, but they do require a recognition that the design stance is, at best, in need of supplementary strategies to complement it and, at worst, largely inapplicable.

For the above reason, and others (such as simple ignorance and quickly changing situations) the precise context of operation may be partially unknown to the designers of an IT system. This means that either the identified goals will not be precisely correct or that the goals need to be specified at a very abstract level, such as *responding to changing demands of browsers*. Trying to meet very abstract goals using the design stance is difficult, the *distance* between the implementation and the goals is just too great. Either one chooses a high-level (i.e. quite abstract) level of specification, in which case one can never guarantee that the implementation meets the specification, or a lower-level specification, in which case one won't know that the specification meets the stated goals[2].

3 The Design Stance in MAS

Of course, nobody attempts to employ the design stance as their only strategy for anything but toy problems. At the least there is some degree of debugging that turns out to be necessary, however carefully one designs and implements a system. Usually a substantial process of trial and adaption takes place after the initial construction. However, reading much of the MAS literature one might be forgiven for thinking that the design steps are the by far the most important part

[2]These options do not fully exhaust the possibilities, of course. For example one might have many levels of specification, but each layer might add some level of uncertainty as it interacts with other parts — any level of specification depends upon assumptions about the context inherited from other layers. The point is that the *distance* between goals and implementation is great however one does it.

of the process of producing working MAS. This is perhaps epitomised by the work of Michael Wooldridge and Nick Jennings. In Wooldridge's book, *Reasoning about Multi-agent Systems* (Wooldridge, 2000), there are no sections on validation. He does briefly point out the difficulties of validating the BDI-framework for MAS but immediately follows it with the phrase: "Fortunately we have powerful tools to help us in our investigation" and goes on to discuss BDI-type logics. Thus he gives the impression that the formal machinery of a kind of logic somehow compensates for the difficulty of validation[3].

In several of Jennings' and Wooldridge's papers they suggest ways of limiting the complexity of the MAS, so as to limit the difficulties designing a MAS presents. For example, Wooldridge and Jennings (1998) suggest the following (among others):

- Do not have too many agents (i.e. more than 10);
- Do not make the agents too complex;
- Do not allow too much communication between your agents.

These criteria explicitly rule-out the application of MAS in any of the messy environments where they will need to be applied. They hark back to the closed systems of unitary design that the present era has left way behind. To the extent that there is an exclusive emphasis on the design stance in considering MAS we will call it the *Pure Design Stance* (PDS).

There is an adage that programming is 10% design and implementation and 90% debugging — an adage that is relevant even when the most careful design methodology is used. It seems likely to us that with MAS there will be an even smaller proportion of effort spent on the stages characterised by the design stance. The PDS is obviously a 'straw man', however the arguments herein hold to the extent that there is an overemphasis on the 10% and a passing over of the 90%. We believe the best way to restore the balance by rigorous application of the classical experimental method. *MAS needs statistics more than logic to make it work.* It needs scientific foundations[4].

4 Then Why is the Design Stance Ever Taken?

So why is the Design Stance used at all? Historically, the origins are obvious. Large expensive systems were developed that turned out to be not what the users actually wanted. Also historically, software engineering and languages weren't what they are today. People wrote large, monolithic programs in abstruse languages. There was no such thing as code libraries that could be linked to, let alone objects, relational databases, or scripting languages. In fact, source code was often lost or thrown away after a program proved effective enough, because 'fixing' old complicated programs often introduced more errors than were corrected.

In those days, the only way to make what was in hindsight seen to be the 'right' program was to start all over again. Unsurprisingly, it was suggested that if only a program was properly designed from the beginning, when it was just on paper (and managers could actually read and understand it), then all those expensive hours of programming, debugging and installing the initial, incorrect program could be avoided. The mantra was, the earlier you fix a mistake, the less it costs. As time goes by, the cost of fixing a bug goes up exponentially.

Beck (2000) recounts this story, but points out that now-a-days, although it is obviously still cheapest to get the general problem right before anyone has started working, the rest of the curve has been flattened by new engineering techniques. Beck (2000) claims the cost of fixing is in fact roughly constant, not exponential.

What Beck doesn't bring out enough is why these new techniques and tools were developed. The problem with any new product is that the users can't know exactly what they need, how they will use it, what it will cost to build, or how much money it will make or save (Hammer and Champy, 1993). Users cannot be expected to be able to tell which features are simple to add or alter (require changing one line of code or even a parameter) and which are difficult[5]. Users also can't know how much their business process will change, or fail to change, when they are handed new capabilities. Thus it is not simply that the pressure for the Design Stance has reduced because of new tools and techniques which allow programs to be rapidly constructed, neatly documented, easily maintained and simply installed. Rather, these tools and processes have been developed due to pressure caused by the intractability of the Design Stance.

It should be said that we are not arguing a new, trendy idea here. Trendy, yes, but Frederick Brooks said in 1975 "Always build one system to throw away," for just these reasons Brooks (1975). He also cautioned against the 'second system syndrome': the deeply fallacious belief that, even once you have delivered a working system, that you now

[3](Edmonds, 2002) gives a more careful and detailed review of this book.

[4]This theme will be developed significantly further in a forthcoming longer version of this paper. In the meantime, see Bryson et al. (2001) and Edmonds (2003a).

[5]In fact, even experts cannot always remember what is computationally tractable, or they wouldn't swear by UML.

understand the requirements well enough to add all the features you thought of but were holding back on before. Even a working system is not a good enough design to be confident that another system built from that template will work right or be what's wanted. Contemporary agile programming methods only reflect a clearer understanding of what good programmers have known for a surprisingly long time (see also Sussman, 1973).

Given all this, then why is there this overemphasis on aspects of a Pure Design Stance in agent research? Historical happenstance doesn't explain its continued favour in the presence of better paradigms. That the PDS agenda should be taken up and continued requires some explanation. We can only offer guesses (warning: some of these are brutal):

- First, academics covet the clothing of a respectable science. This makes the formal machinery of logic very attractive. Despite the fact that MAS' foundations are only loosely based in philosophy and social phenomena via loose analogies, the structure of logic makes it look as if the foundations of a science are there.
- The fact that software is deterministic at the micro-level gives credence to the old functionalist dream of certainty — in this case certainty in designed software systems. As mathematicians have previously found that this is not possible even with totally reliable formal system. This is demonstrated by the fall of many elements of the Hilbert programme (by Gödel and others). Of course, for most engineers this dream is not a danger, because it was never possible, even conceptually. Most engineers are attempting to develop techniques to produce reliable systems in a chaotic world — they act to increase reliability. Software engineers do the reverse — they produce only somewhat reliable systems from very reliable components.
- The pure design stance fits well into existing academic/consultant and managerial practices. It is important to many managers that they maintain at least the illusion of control, thus they favour fixed goals, roles and procedures to obtain this. They treat software in the same way and thus seek to constrain the software in a similar way. An off-line design process suits the academic/consultant — they can thus limit contact with the company and delimit the problem they are working on. A formal specification allows the consultant to ignore many of the uncertainties in producing software that works in practice in the context of operation and instead simply produce software that meets the agreed specification. In this sense the PDS can be seen as the application of a rigid managerial and planning style (as was practiced in the old Soviet Union) to software.
- Another pressure for formalism is that, for some people, it's fun. Some people just enjoy doing logic. MAS gives them another arena to get their papers published which may actually be more open than the true mathematical journals, and also nets more funding because it claims to be practical.
- Finally one of the reasons that the pure design stance is championed by some academics is the fact that they limit the problems they address to simple, even toy, situations. It seems sadly rare that a novel methodology or architecture has been tried in a real complex application before its novelty or elegance gains popularity among other academics.

5 System-Building Strategies and MAS

So what does it take to build a successful system? To step back and look more generally at the problem of producing IT systems that will be adequate to the messy systems they are to be deployed in, we can list a host of strategies developed to help in this task. They include:

1. *Abstraction:* using abstractions that stand for a lot of detail and are underpinned by well understood analogies may enable a programmer to achieve the desired behaviour by working only at the level of such abstractions. For this to work the analogy used by the programmer to make decisions about the use of the abstractions must be effectively accurate in terms of the effects of the hidden detail that such use has. Useful abstractions are often underpinned by algorithms that allow automation.
2. *Automation:* when there are many steps that are amenable to automation (e.g. as in compilation of a formal language to machine code), many possible translation errors can be avoided. Useful automation depends upon there being some good way of understanding the process being automated, so that one knows when to use it and can understand the results when it is used.
3. *Standardisation:* when you have a system that is composed of many parts (e.g. agents or modules) you can get the parts working together in a basic way by agreeing some standards concerning the form and content of their interaction. One can not ensure the complete compatibility of the parts interacting using such a standard, since unpredictable interactions can always occur between the parts (especially when a standard allows interaction). To ensure complete compatibility one needs a complete standard about the interactive behaviour of each part. In open

systems, such a standard is rarely available or even achievable, and in practice not even desirable since it would overly constrain the development of each part.

There is always a tension between the flexibility delegated to the parts and the effectiveness of a standard in controlling the interaction. The most robust standards are not the result of *a priori* thought but arise, at least substantially, from actual practice. There are many invented standards with elegant structures that are ignored due to their subtle inappropriateness for frequent tasks and their complexity. Conceptually messy standards that meet immediate needs abound.

4. *Modularity:* where different roles and tasks are fairly well separable, these can be delegated to different parts of the system. If this task is in great demand it may be abstracted, standardised and distributed. This can simplify the design of the overall system as well as provide a potentially reusable piece of code and functionality. However, in many systems that have (in the broadest sense) evolved over time, such modularity may not be neat — many parts will have multiple roles, and many roles may be distributed across many kinds of parts. This is why the most reliable source of modular decomposition is probably around resources, rather than tasks (Bryson and Stein, 2001). Examples of resources include specific specialised representations and the data they hold, or specific pieces of equipment.

5. *Formalisation:* formalisation can help eliminate ambiguity and facilitate automation. This makes it a natural way of expressing and enforcing standards. Unfortunately, its role in facilitating automation is over-rated. Although a formal specification may suggest the prospect of automation this depends upon the ease with which formal expressions can be written that reflect the real situation whilst retaining their amenability to automation. Many formal systems are only mechanisable in simple cases — in other cases formalisation may not be good route to automation. There is an inescapable trade-off between the expressiveness of a formal systems and the ease with which it can be automated.

6. *Transparency:* the transparency of a system is the ease with which the behaviour of a part may be understood and controlled using an accessible model or theory. This model can be partial, as long as it is a good guide to behaviour. For example its predictions may be only negative or probabilistic, and the scope over which it is effective might be limited to certain circumstances. In other words, it is not necessary for the model to be universal, covering all possible behaviours under all circumstances.

7. *Redundancy:* one way of attempting to ensure an outcome in messy circumstances is to have different parallel processes in place to do this. If one mechanism fails then it is possible that another will work. Social and biological systems abound in this kind of redundancy, where each new mechanism does not completely replace an existing mechanism (unless it is very costly) but works in parallel with it. Major computer systems also often exploit some redundancy in hardware, e.g. arrays of hard disks with data encoded to guard against single disk failures. Redundancy require two things to achieve maximum robustness: multiplicity and diversity. If the different parts of a systems are essentially the same then the robustness to uncertain conditions is reduced, in other words deep heterogeneity is a benefit.

8. *Adaptivity:* inflexible systems can be honed so that they are very efficient at what they do, but this may come at the cost of being useless out of a very specific context. Systems that have the ability to adapt to prevailing circumstances require more infrastructure but may be more generally available. When well defined and constrained, simple learning and decision making abilities may allow a part to function in effective ways unforeseen by its designer while still letting it be consistently reliable in its intended role. Of course, in general the tradeoff for adaptability is unpredictability, but this can be addressed not only through design, but also through redundancy (c.f. Opitz and Maclin, 1999).

9. *Testing:* however carefully one designs and implements a system one can never be sure of the resulting behaviour — the only way to be sure it to try it. This means that one has to determine the behaviour *experimentally* — exploring the behaviour, making hypotheses about the behaviour and testing these to see if they are likely true. The design of the system (if known) provides a suitable source of suggestions for hypotheses to be checked, but can never be sufficient on their own, for the reason that they are seldom of a form that allows the direct prediction of behaviour. Debugging is the simplest such case of such testing — the system does not work as intended so that hypotheses need to be developed as to why it is not and tested, once such a hypothesis is confirmed then one can try to change the behaviour to what is desired. In the more complex case the individual system works as desired but does not interact with its environment in an acceptable way. If even in simple unitary environments the effort of getting an IT system to work is 10% implementation and 90% debugging, then in messy environment the debugging effort is likely to be even more predominant.

The approach to engineering MAS over the last decade has emphasised the first five of these: abstraction; automation; standardisation; modularity and formalisation. The abstractions chosen have been dominated by the relatively small set of Beliefs, Desires and Intentions (and other closely related ones). Automation has been focused on verification and compilation techniques. The standardisations has resulted in a host of protocols for communication. The modularity tends to be decomposed in terms of tasks and roles. This is of course the 'agent', now supplemented by holonic agents and teams. The formalisation of choice has been logic. The adaptivity is often only that of delaying planning until the moment of choice; that is, the mechanisms of adaptivity are limited to those of inference. Techniques for testing and debugging have not significantly changed or developed from traditional non-agent programming techniques, and often seem to be lagging behind the state-of-the-art in software engineering or systems.

6 Making MAS Work

To be clear, we are not saying that controlling and understanding open complex MAS systems is impossible. We are saying that nature of the control and understanding will be closer to that found in the natural sciences (e.g. physics, biology) rather than the formal sciences (logic and mathematics). The understanding of such systems will be via experimentally validated theories rather than formally established properties. This view is pessimistic about the extent that certainty is possible for any but the smallest software components, but this simply reflects the nature of open complex systems. Engineers designing bridges have no certainty about how their constructions will fare — there is no possibility of proving the properties of a design from the laws of physics. Rather they rely on techniques and styles that have been extensively tried (e.g. multiple approximate calculations of stress, extensive simulation) and careful monitoring of the results as they are built, keeping ready to fix any problems if they emerge. Our task is far more difficult than for the structural engineer because we have not developed a good enough understanding of social and organisational behaviour, which are the components we are attempting to build with.

Thus we believe more emphasis needs to be placed on the tactics in the second half of the list above: transparency; redundancy; adaptivity and testing, because these support and come from an experimental science approach to social systems — which is what MAS are an artificial version of.

The *transparency* of the actor-agent analogy is of vital importance because it is one of the principle reasons that MAS might be useful. More support and tools need to be developed for computation experiments on and observation of MAS systems. The concept of modularity needs to be expanded from only agent and teams to entities such as groups, societies, institutions etc. that are less easy to ascribe precise boundaries. Transparency is often provided by applying mechanisms found in the social or biological domains[6]. An understanding of how they may work in their source domains provides a useful starting point for understanding how they may work in artificial domains.

Redundancy is often an inevitable result of applying mechanisms found in the social and biological fields, since these abound in redundant systems. *Adaptivity* is often critical with learning taking a key role — often exploiting situations where learning can be gradual and complex, whilst decision making needs to be simple and immediate. *Testing* in the form of *post hoc* theorising and experimentation with MAS dominates all else: this is the key to the scientific approach to understanding complex systems.

The sort of systems we have to deal with can be characterised by several kinds of complexity: syntactic, semantic, and analytic complexity. If a computational system is *syntactically complex* then there is no easy prediction of the resulting behaviour from the initial set-up of the system. In other words, the the computational *distance* between initial conditions and outcomes is too great to be analytically bridgeable using any short-cut. The only real way to get the outcomes is to run the system. The difficulty in bridging this gap means that there are at least two *views* of the system: that of the set-up of the system and that of the resulting behaviour. That such syntactic complexity can exist is evidenced by the effectiveness of pseudo-random number generators or, if one prefers a more distributed example, many cellular automata, e.g. those described by Stephen (1986).

Semantic complexity describes the case where any formal representation of a system is necessarily incomplete. Thus any formal theory is limited in its applicability to a restricted domain or context. Clearly, in simple computational systems there is sometimes, in theory, a complete formal representation — the code itself. However, this may well not be the case in open systems or those designed on multiple sites, when no adequate representation of the effective code may be available. Even where it is available, the presence of syntactic complexity may make this representation useless for controlling the outcomes, thus there still may be no useful and complete formal representation — *effective* semantic complexity. The presence of semantic complexity means that instead of a single representation of the outcomes one has an incomplete patchwork of context-dependent models. Semantic and Syntactic Complexity are discussed in greater depth by Edmonds (2003b).

Analytic complexity is when it is not possible to completely analyse a system into a set of independent parts. In

[6]Other source domains are also possible, but less common.

other words, any consideration of an isolated part will necessarily miss some of the behaviour that it would display when operating in the complete system. One cause of this is due to the process of *embedding*, where the rest of the system adapts to the behaviours of the part. This commonly occurs when the IT system is the part and the wider system is the human institution in which it is deployed. In such a case the IT system can not be separated from the wider system to aid analysis without changing the behaviour of both the IT system and the wider system. A consequence of such embedding can be that the classic (and often used) formal off-line process of design and implementation is difficult and inappropriate.

The presence of these kinds of complexity suggests that a science of MAS may be more akin to zoology or psychology than at least folk notions of theoretical physics[7]. That is, there may be lots of essentially different kinds of agents, teams, trust, modes of communication etc. There may not be a single methodology, architecture, type of framework, formalisation or theory that covers them all. It may be that lots of observation and exploration is necessary before any abstraction to theory is feasible. *Thata prori*, foundationalist studies based only on philosophy will be, at best, irrelevant and, at worst, misleading. That abstraction will only be possible as and where hypotheses are shown to be successful in experiments and practice.

7 Conclusion

There is a tendency for failing research programmes to retreat away from the real challenges of their field of study and into purely formal and abstract questions that are only of interest to academics in the field. The real issues of our field is surely how MAS can work when applied in their messy and open operational contexts, for example in human social systems or the web. Purely abstract concerns such as *a priori* logical deconstructions of trust and the like should have to prove their worth in terms of ultimate application before being acceptable as relevant.

Science started to progress when it abandoned reliance on *a priori* systems of thought (how authorities such as Aristotle thought systems should be) for observational and experimental methods such as were established by Bacon and Galileo. Bacon (1561–1626) urged that we must not only observe nature in the raw but also "twist the lion's tail" (Hacking, 1983, p. 149). We think that MAS will prosper when it remembers this lesson.

In conclusion, despite a great deal of research, we believe that the promise of MAS is far from realised, and we are worried that it is getting further from being realised due to the direction of the majority of work in this area. The majority of working applications being used in earnest that we know about are actually-agent based modelling systems, which are having a significant impact in both science and public policy (Cederman and Rao, 2001; Hemelrijk, 2002; Veselka et al., 2002), in games and in military simulations. These applications use architectures and protocols not just designed but developed and evolved in response to real needs, problems and data. We do believe there are valid reasons for using the MAS approach which we enumerated in the introduction, but we think a new direction is needed. MAS needs a renaissance, a set of techniques developed in the forge of practice rather than the armchair of theory. We need to twist the lion's tail.

8 Acknowledgements

We would each like to say that anything you really disliked about this paper was written by the other. Thanks also to Scott Moss for discussion about these sorts of ideas, to David Hales and Will Lowe for helping us develop some practical ways forward and to members of the ABSS SIG of AgentLink II for their comments.

References

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA.

Brooks, Jr., F. P. (1975). *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley, Reading, MA.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23.

Bryson, J. J., Lowe, W., and Stein, L. A. (2001). Hypothesis testing for complex agents. In Meystel, A. M. and Messina, E. R., editors, *NIST Workshop on Performance Metrics for Intelligent Systems*, pages 233–240, Washington, DC. NIST Special Publication 970.

[7] In practice, very, very few real-world physical systems have been analysed completely to the ground either, for just the same kind of reasons.

Bryson, J. J. and Stein, L. A. (2001). Modularity and design in reactive intelligence. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 1115–1120, Seattle. Morgan Kaufmann.

Cederman, L.-E. and Rao, M. P. (2001). Exploring the dynamics of the democratic peace. *Journal of Conflict Resolution*, 45(6):818–833.

Cosmides, L. and Tooby, J. (1992). Cognitive adaptations for social exchange. In Barkow, J. H., Cosmides, L., and Tooby, J., editors, *The Adapted Mind*, pages 163–228. Oxford University Press.

Edmonds, B. (2002). A review of "reasoning about rational agents" by michael wooldridge. *Journal of Artificial Societies and Social Simulation*, 5(1). `http://jasss.soc.surrey.ac.uk/5/1/reviews/edmonds.html`.

Edmonds, B. (2003a). Beyond the design stance — losing some theoretical control in return for success in practice via complex adaptive systems. Talk at the Agent-Based Social Simulation Special Interest Group Meeting, slides available from `http://www.ide.bth.se/~pdv/ABSS/schedule.html`.

Edmonds, B. (2003b). Towards an ideal social simulation language. In Sichman, J., Bousquet, F., and Davidsson, P., editors, *Multi-Agent-Based Simulation II: Third International Workshop, (MABS02), Revised Papers*, pages 679–690, Berlin. Springer.

Hacking, I. (1983). *Representing and Intervening*. Cambridge University Press.

Hammer, M. and Champy, J. A. (1993). *Reengineering the Corporation: A Manifesto for Business Revolution*. Harper Collins Publishers Inc., New York, NY.

Harel, D. (2003). *Computers Ltd.: What They* Really *Can't Do*. Oxford University Press.

Hemelrijk, C. K. (2002). Self-organization and natural selection in the evolution of complex despotic societies. *Biological Bulletin*, 202(3):283–288.

Johnson, W. L., editor (1997). *Proceedings of the First International Conference on Autonomous Agents*. ACM press.

Opitz, D. and Maclin, R. (1999). Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198.

Raymond, E., editor (1992). *The New Hacker's Dictionary*. MIT Press, Cambridge, MA. available online.

Stephen (1986). Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7:123–169.

Sussman, G. J. (1973). A computational model of skill acquisition. Technical Report AITR-297, MIT Artificial Intelligence Laboratory.

Veselka, T. D., Boyd, G., Conzelmann, G., Koritarov, V., Macal, C., North, M., Schoepfle, B., and Thimmapuram, P. (2002). Simulating the behavior of electricity markets with an agent-based methodology: The electricity market complex adaptive system (EMCAS) model. In *The U.S. Association for Energy Economics, 2002 USAEE Conference*, Vancouver, BC.

Wooldridge, M. and Jennings, N. (1998). Pitfalls of agent-oriented development. In Sycara, K. P. and Wooldridge, M., editors, *Proceedings of the Second International Conference on Autonomous Agents*, Minneapolis, USA. ACM Press.

Wooldridge, M. J. (2000). *Reasoning about Rational Agents*. MIT Press, Cambridge, MA.