

The Insufficiency of Formal Design Methods - the necessity of an experimental approach for the understanding and control of complex MAS

Bruce Edmonds
Centre for Policy Modelling
Manchester Metropolitan University
Aytoun Building, Manchester, M1 3GH, UK
<http://cfpm.org/~bruce>

Joanna Bryson
Department of Computer Science
University of Bath
Bath, BA2 7AY, UK
<http://www.cs.bath.ac.uk/~jjb>

Abstract

We highlight the limitations of formal methods by exhibiting two results in recursive function theory: that there is no effective means of finding a program that satisfies a given formal specification; or checking that a program meets a specification. We also exhibit a ‘simple’ MAS which has all the power of a Turing machine. We then argue that any ‘pure design’ methodology will face insurmountable difficulties in today’s open and complex MAS. Rather we suggest a methodology based on the classic experimental method – that is ‘scientific foundations’ for the construction and control of complex MAS.

1. Some Limitations of Formal Methods

We start by looking at the limitations of formal methods. Although these *are almost certainly already known*, they are worth reviewing because they are *so* under-publicised¹, many will not be aware of them.

The idea of formal methods is to write specifications of software in a formal language. This formal language is often of a logical or set-theoretic nature. This has two undisputed advantages: *firstly*, that the formal specification is unambiguous and, *secondly*, the specifications can themselves be syntactically manipulated (e.g. in formal proofs). This formal language is thus a sort of *lingua franca* for software engineers – it allows them to communicate and manipulate specifications of software. However, like any language, there are fundamental difficulties that arise when attempting to translate to and from other languages. Here there are the two such problems: one of relating the specifications to executable code and another of relating informal natural language specifications with the formal specifications. The first will be dealt with in this section and the second in the next.

¹ We searched the literature on formal methods for something which listed these limitations so as to simply reference them, but did not find any!

Given the nature of computation and program code two major questions present themselves. The first is whether there is any systematic or effective way of getting from a given specification to a system of software – we will call this the “programming problem”. The second is whether there is any systematic or effective means of checking whether a given software system meets a given specification – we will call this the “checking problem”.

1.1 The Programming Problem

A particular case of the first question is whether there is any effective method of producing a *program* from a formal specification – single programs are components of software systems, and if we can’t do this for programs we can’t do it for systems of software. To formalise this we will assume the Church-Turing thesis ([1] page 67), namely that “effective method” means a program (e.g. a Turing machine). Thus this question can be reduced to the following: *is there a program that, given a specification, will output a program that meets that specification?*

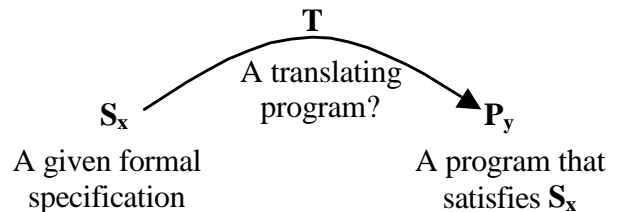


Figure 1. A supposed program from formal specifications to appropriate programs

The answer is, of course, “no”. Even when we know that there *is* a program that satisfies a specification, there is no computation that will take us from formal specifications to such programs. This is shown below.

The ‘halting problem’ is an undecidable problem [13], (that is it is a question for which there does not exist a program that will answer it, say outputting 1 for yes and 0 for no). This is the problem of whether a given program will eventually come to a halt with a given input. In other

words whether $P_x(y)$, program number x applied to input y , ever finishes with a result or whether it goes on for ever. Turing proved that there is no such program [13].

Define a series of problems, LH_1, LH_2 , etc., which we call ‘limited halting problems’. LH_n is the problem of ‘whether a program with number n and an input n will ever halt’. The crucial fact is that each of these is *computable*, since each can be implemented as a finite lookup table². Call the programs that implement these lookup tables: PH_1, PH_2 , etc. respectively. Now *if* the specification language can specify each such program one can form a corresponding enumeration of formal specifications: SH_1, SH_2 , etc.

The question now is whether there is any way of computationally finding PH_n from the specification SH_n . But if there *were* such a way we could solve Turing’s general halting problem in the following manner: *first* find the maximum of x and y (call this m); *then* compute PH_m from SH_m ; *and finally* use PH_m to compute whether $P_x(y)$ halts. Since we know the general halting problem is uncomputable, we also know that there is no effective way of discovering PH_n from SH_n *even though for each SH_n an appropriate PH_n exists!*

Thus the only question left is whether the specification language is sufficiently expressive to enable SH_1, SH_2 , etc. to be formulated. Unfortunately the construction in Gödel’s famous incompleteness proof [9] guarantees that any formal language that can express even basic arithmetic properties will be able to formulate such specifications.

1.2 The Checking Problem

The checking problem is apparently less ambitious than the programming problem – here we are given a program and a specification and have ‘only’ to check whether they correspond. The question here is whether there is any effective or systematic way of doing this.

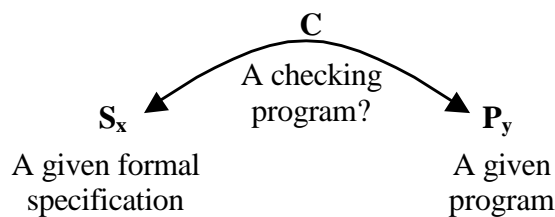


Figure 2. A supposed program for checking whether a program satisfies a given specification

Again the answer is in the negative. To demonstrate this we can reuse the limited halting problems defined in the last subsection. The counter-example is whether one can computationally check (using C) that a given program P meets the specification SH_n . In this case we will limit

ourselves to programs, P , that implement $n \times n$ finite lookup tables with entries: $\{0,1\}$.

Now we can see that if there *were* a checking program C that, given a program and a formal specification would tell us whether the program met the specification, we could again solve the general halting problem. We would be able to do this as follows: *first* find the maximum of x and y (call this m); *then* construct a sequence of programs implementing all possible finite lookup tables of type: $m \times m$; $\{0,1\}$; *then* test these programs one at a time using C to find one that satisfies SH_n (we know there is at least one: PH_m); *and finally* use this program to compute whether $P_x(y)$ halts. Thus there is no such program, C .

1.3 An Apparently Simple MAS

In order to emphasise how simple a MAS can be and still be beyond the power of formal methods, consider the following systems: ‘GASP’ systems (Giving Agent System with Plans). These have n agents, labelled: $1, 2, 3$, etc., each of which has a changable integer store and a finite number of fixed specified plans. Each time interval the store of each agent is incremented. Each plan is composed of a (possibly empty) sequence of ‘give instructions’ and finishes with one ‘test instruction’. Each ‘give instruction’, G_a , has the effect of giving 1 unit to agent a (if the store is non-zero). The ‘test instruction’, $JZ_{a,p,q}$, determines the plan that will be executed next time period as plan p if the store of agent a is 0, otherwise plan q .

Thus ‘all’ that happens in this class of GASP systems is the giving of tokens with value 1 and the testing of other agents’ stores to see if they are zero to determine the next plan. However GASP systems have all the power of Turing machines, and hence can perform any formal computation at all – the proof is outlined in the appendix. Since GASP machines are this powerful, many questions about it are not amenable to any systematic decision procedure. For example, if M is any non-empty proper subset of all possible GASP machine indexes, Rice’s theorem ([1] page 105) holds, so that determining whether an arbitrary given GASP machine is a member of M is uncomputable.

Thus the conclusion of this section is that if the specification language can deal with basic arithmetic then, in general, there is no effective or systematic way of either getting from a specification to a program that satisfies it or checking that a program satisfies a given specification. Since *almost all* realistic applications *require* such arithmetic this conclusion holds for almost all classes that include these. The example of GASP machines shows how simple such MAS can be. Similarly, [18] shows the infeasibility of some finite design problems.

² Of course, discovering what the correct entries of such a table is an insurmountable problem, but this is the point.

2. Some Limitations of Formal Specification

The primary goal in engineering IT systems can be stated as follows: *to produce IT systems that work well in practice once deployed in their operational context.*

One particular strategy for approaching this is what can be called the formal specification strategy (FSS). This is divided into three basic stages:

- ?? To agree the goals for the IT system;
- ?? To write a specification that would meet these goals;
- ?? To implement a system that meets this specification.

To succeed with this one has to ensure that: the identified goals are the appropriate ones for the final operating context; the specification is such that it will meet the identified goals once deployed; and 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 unitary processing units designed for a special and well-defined job, but increasingly are interacting with many other systems designed by completely different programmers for disparate (and sometimes unknown) purposes. The cost of human input into and interpretation of output from IT systems means that it is more efficient to take the input from other IT systems and send the output to other IT systems where this is possible. For example, an automatic trading system may take information from a variety of sources, including the stock index calculated by other IT systems. If there are enough such connected trading systems then there may occur feedback loops between the system that calculates the index and those that do the trades. This, in fact, is thought to have occurred, forcing the stock-market authorities to institute a new rule that breaks this autonomous feedback loop.

Systems, now that they are cheap, pervasive and have been around for a while, become embedded into the human practice that has, in turn, adapted 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 effect 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 fraud will almost inevitably mean 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/environments.

Messy environments are the rule – neat ones increasingly the exception. A consequence of their prevalence and the primary systems goal is that our methodology for constructing IT systems should be so that they work well as part of messy systems. This does not *necessarily* mean they will themselves be messy when considered in isolation, but it may well mean that they are not very neat. Trying to keep them neat so that FSS can be retained is putting the cart before the horse. We should not be afraid of messy environments, but they do require a recognition that in such cases the FSS is, at best, in need of supplementary strategies and, at worst, inapplicable.

For the above reason, and others the precise context of operation may be partially unknown to the designers of the 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 FSS 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 you won't know that the specification meets the stated goals³.

Of course, nobody attempts to employ FSS as their *only* strategy for anything but toy problems. There is always some debugging that turns out to be necessary, however carefully one designs and implements a system and usually a substantial process of trial and adaptation 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 of the process of producing working MAS. For example in [17], there are no sections on validation – it 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 logic somehow compensates for the difficulty of validation⁴.

In several of Jennings's and Wooldridge's papers they suggest ways of limiting the complexity of the MAS, so as to limit the difficulties of *designing* a MAS. For example in [19], the authors suggest (among others), not to:

- ?? have too many agents (i.e. more than 10);
- ?? make the agents too complex;
- ?? allow too much communication between your agents.

³ These do not exhaust the possibilities, of course. For example one might have many intermediate levels, but each layer will add some level of uncertainty. The point is that the “distance” between goals and implementation is great however one does it.

⁴ A more careful and detailed review of this book is [6].

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 stages in considering MAS we will call it the “Pure Design Approach” (PDA). 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 me that with MAS there will be an even smaller proportion of effort spent on the stages characterised by the design stages. The PDA 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%. This paper can be seen as a call to restore the balance by rigorous application of the classical experimental method – MAS needs more than maths, logic and philosophy to make it work, it needs *scientific* foundations.

3. Some Software Production Strategies

Looking 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. We discuss a selection of them below, looking at their applicability and robustness. The argument here is for a balanced approach utilising all of the possible strategies, rather than focussing on a few.

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 must be effectively accurate in terms of the effects of the hidden detail it stands for.

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.

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. To ensure complete compatibility one needs a complete standard about the interactive behaviour of each part which, in open systems, is rarely available (and in fact not 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 common tasks and their complexity. Conceptually messy standards but which are found to meet needs abound, since they work.

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. 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 [14].

Formalisation: formalisation can help eliminate ambiguity and facilitate automation. This makes it a natural way of expressing and enforcing standards. As we have sought to show above, its role in facilitating automation is over-hyped. 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. There is an inescapable trade-off between the expressiveness of a formal systems and the ease with which it can be automated. Even simple systems (such as the GASP systems above) can be beyond automation.

Transparency: the transparency of a system is the ease with which the behaviour of a part may be understood and controlled using an accessible analogy, 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. In other words, it is not necessary for the model to be universal, covering all possible behaviours under all circumstances. We discuss the transparency of the agent concept below.

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. This sort of redundancy require two things, multiplicity and diversity, to achieve maximum robustness. If the different parts of a systems are essentially the same then the robustness to uncertain conditions is reduced. This is another reason why the sort of limitations on MAS necessary to preserve a FSS might be counter-productive.

Adaptively: inflexible systems can be honed so that they are very efficient at what they do, however it may

mean that, out of its intended context, it becomes useless or even counter-productive. Systems that have the ability to adapt to prevailing circumstances require more infrastructure but may be more reliable. In particular simple learning and decision making abilities may allow a part to function in effective ways unforeseen by its designer. Such adaptivity comes at the cost of precise control – an ability to cope in unforeseen circumstances generally involves the ability to behave in unforeseen ways. Limiting agents so that their behaviour is predictable (or even, in extreme cases, provable) can mean that needed adaptivity is ruled-out.

Testing: however carefully one designs and implements a system one can never be sure of the resulting behaviour – the only way to be sure is 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 these are the case. 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. In more complex cases the individual systems may work as desired but not interact with its environment in an acceptable way.

The approach to engineering MAS over the last decade has emphasised the first five of these: abstraction; automation; standardisation; modularity and formalisation. The chosen abstractions chosen have been dominated by a 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 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 – i.e. the mechanisms of adaptivity are limited to those of inference. Techniques for testing and debugging have not significantly developed from those of traditional non-agent programming techniques.

This paper argues that the balance needs to be restored by more emphasis on the tactics in the second half of the list: transparency; redundancy; adaptivity and testing. The abstraction may be the agent if there is a clear analogy with social actors (see next section). What is important about this actor representation is that the representational analogy is powerful and transparent (in the sense above) so that the analogy of this entity as a social actor can guide our programming and experimentation with it. The main engine of automation is the simulation – a platform for computational experiments. This is usually agent-based simulation since components with the sophistication of agents are sometimes required. The modularity is often

much less well-defined than in many MAS. There are often entities such as agents and teams, but there are also entities such as groups, societies, institutions etc. that are less easy to ascribe precise boundaries. The transparency is often provided by applying mechanisms found in the social or biological domains (other domains are also possible) – 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. The 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 key 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 a scientific approach.

The sort of systems we have to deal with can be characterised by several kinds of complexity: what we call “syntactic”, “semantic” and “analytic” complexity [7]. 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 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 shown by the effectiveness of pseudo-random number generators or (a distributed example) many cellular automata (e.g. [15]).

Semantic complexity is when 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 or systems designed by different people, 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.

Analytic complexity is when it is not possible to completely analyse a system into a set of independent parts. In other words, any consideration of separate parts will necessarily lose some of the behaviour that it would

display when part of the 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 in the human institution that it is deployed in. In such a case the IT system is embedded in the wider system such that it can not be separated from it to aid analysis without changing the behaviour of both the IT system and the wider system [5]. A consequence of such embedding can be that the formal off-line process of design and implementation is inappropriate.

The presence of these kinds of complexity suggests that a science of MAS may be similar to zoology, in that 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. That *a priori*, foundationalist studies 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. There is a lot of resistance to such suggestions since it indicates that there will be no theoretical ‘short-cut’ to success, and thus that progress will be much slower than some might have hoped and require a lot more painstaking empirical work.

4. The Roots of the Agent Concept

It is worth considering for a moment the roots of the agent concept. It may be the case that interacting with humans may be facilitated by having some human characteristics (the “like me” test of [2]), but the claim for the utility of the agent concept goes far beyond user interfaces, games and social simulations. The question is: why would one, in other situations, *wish* to deal with a chunk of code in a similar way to a human or animal actor, and it what way does this help in the construction of complex systems? I.e. why the *agent* abstraction rather than one of the other possibilities (e.g. ‘patterns’)?

If the agent concept is to have effective leverage in aiding the software production process it needs to be a good analogy for guiding programmers. In other words our intuitions about how social actors might interact in complex social processes can help direct our programming of such actors in the form of artificial agents. Otherwise there is no fundamental reason why a particular chunk of code should have *any* particular set of properties – one would expect each chunk to have exactly those properties as is appropriate for it. In other words, without an effective analogy with real social actors there is *nothing* that is common to *all* entities which might be called “software agents”. It is the ability to think about agents as

social actors which gives the agent concept its meaning and is the source of its potential power.

However there is a problem with this. Except in very simple cases (ants?) we do not *know* very much about how social actors abilities to organise are related to their individual properties. The full complexities of this “macro-micro” link are only starting to be uncovered; some examples can be found in past papers of this conference and some of its attendant workshops (e.g. ESOA [12] and MABS [10]). Here the central question of how particular social or organisational mechanisms ‘play-out’ in societies of agents is being experimentally investigated. Here the analogy between agents and actors is much more explicit.

5. An Experimental Approach

One might reasonably ask what sort of foundations can we provide for our software components, if formal foundations are not feasible. The answer must lie in the validation rather than the verification of code. A complex software system may behave somewhat like its design, but one can not rely on this. The intended behaviour of a system is only a hypothesis about the system behaviour that has to be checked by experimentation. Thus the hypothesis that the system is behaving as designed must compete against other hypothesis about the system – it is the sensible place to start, but for complex systems it is likely to be wrong. However we need *some* basis for using algorithms when constructing complex MAS.

We suggest that chunks of code (including agents and agent systems) that are intended for reuse (either conceptually or verbatim) should be accompanied by a set of *testable hypotheses* about that chunk of code’s behaviour. Each of these would be of such a nature that they can be checked by rerunning the code and comparing the results (or output) of the code against their predictions. Ideally each of these should be such that the proportion of data sets which satisfy the hypothesis out of the total number of possible data sets should tend to zero when the program is tested for longer runs (or larger size).

As these hypotheses are tested they can be appended with data concerning the results of these tests, in particular: the parameter ranges of those tests, number of runs and significance of the results. In this way people who are considering re-using the code will know over which ranges they can rely on the code in respect to those properties they need. One should not re-use code if one has to rely on properties that are not one of the testable hypotheses or if they plan to run it with parameters that are not within a range that has been tested – such people have a choice whether to test the code in the ranges they require themselves or else to look for some other code.

Over time code or algorithms that are reliable can be established through cooperative and distributed testing.

This would fit in very well with how public domain code such as Linux is developed and maintained, except that the reliance put on code could be based on explicit rather than implicit information. Such a process is very close to that used in the natural science – hypotheses are tested in experiments so that only those hypotheses that survive many attempts at disconfirmation are trusted. The process of natural science has a good record at producing useable knowledge that is applicable to complex constructions (e.g. bridges), there is no reason why a similar kind of reliability can't be built up for systems of software.

5.1 An example: evolutionary algorithms

The properties of many search algorithms are not amenable to formal proof due to their stochastic nature and yet are being applied in complex MAS – recent examples include using a Genetic Algorithm (GA) to sequence contracts between self-interested agents [4] or tag-based evolution for the control of loading agents [8].

The “No Free Lunch” theorems [16] tell us that, *in general*, no search algorithm will be better than any other. The moral of this is that to gain any efficiency one has to exploit some specific properties of the class of search spaces one is concerned with. Further the stochastic nature of many search algorithms means that proving their properties is unlikely. Rather this is primarily an *empirical matter*. For, even though it properties might be confirmed by results from greatly simplified and approximate analytic formulations, it is never certain that the simplifications and approximations that were necessary to make these tractable did not significantly distort the representation of that system.

If the code and specification for such a search module were published along with a database of hypotheses of average and/or worst case performance w.r.t. different classes of problem (along with confidence statistics, parameter settings concerning size etc.), then software engineers who wished to use this in their system would be able to make evidence-based judgements on the suitability for their purposes. Academics (or other interested parties) might wish to either extend this database of results to new classes of problems or parameter ranges so as to aid engineers make such decisions or, alternatively, seek to disconfirm them by careful simulation experiment. Further some might dare to make ‘second-order’ hypotheses about the properties of problem classes that results in certain minimum levels of performance, which might themselves be subject to experimental investigation.

Thus for a GA, an entry in the database of hypotheses might be of the form: for a problem space with a single solution, bitstrings of size n , a GA with mutation of 10%, population m , will find the solution with probability distribution: $D(n,m,t)$, where t is the generation.

6. Conclusion

The limitations of formal methods have been known since Gödel. We should not be trying to construct a new Hilbert Programme for agent systems, attempting to formally mechanise the programming process – this is doomed to fail except in the simplest of cases. Rather we should seek *scientific* foundations for agent systems – that is, foundations based on the classic experimental method. Although this means that we will have to rid ourselves of the illusion that we can *fully* understand our own code, it does offer the real possibility of reliable software systems. To support this we will have to improve the methodology and technology for testing and adapting software (the ‘90%’) to match the effort spent on the supporting the specification and implementation of software (the ‘10%’).

7. Acknowledgements

Thanks to the participants of the ABSS SIG meeting of AgentLink II at Barcelona, 2003 for their comments upon the talk that eventually grew into this paper.

8. References

- [1] Cutland, N. J. (1980) *Computability*. CUP.
- [2] Dautenhahn, K. (1997) I could be you – the phenomenological dimension of social understanding. *Cybernetics and Systems*, **25**:417-453.
- [3] Davis, M. (ed.) (1965) *The Undecidable*. New York: Raven.
- [4] Dutta, P.S. and Sen, S. (2002) Optimal sequencing of individually rational contracts. In *Proc. of the 1st Int. joint Conference on Autonomous Agents and Multiagent System*, Bologna, Italy. ACM Press, 607-612.
- [5] Edmonds, B. (1998) Social Embeddedness and Agent Development. UKMAS'98, Manchester, Dec. 1998. <http://cfpm.org/cpmrep46.html>
- [6] Edmonds, B. (2002). A review of “Reasoning about Rational Agents”. *J. of Artif. Societies and Social Simul.* **5**(1). <http://jasss.soc.surrey.ac.uk/5/1/reviews/edmonds.html>
- [7] Edmonds, B. (2003). Towards an ideal social simulation language. In Sichman, J. et al (eds.), *Multi-Agent-Based Simulation II: 3rd Int. Workshop, (MABS02)*, Revised Papers, pages 104-124, Springer, LNAI, **2581**.
- [8] Hales, D. and Edmonds, B. (2002) Groups and organizations: Evolving social rationality for MAS using “tags”. In *Proc. of the 1st Int. joint conference on Autonomous Agents and Multiagent System*, Bologna, Italy. ACM Press, 497-503.
- [9] Gödel, K. (1931) Über formal unentscheidbare Sätze der Principia Mathematica und verwandter System I. *Monatshefte Math. Phys.* **38**: 173-198. Translation in [3].
- [10] Hales, D. et al (eds.) (2003) *Multi-Agent Based Modelling III (MABS 2003)*. Springer, LNAI, **2927**.
- [11] Moss, S. and Edmonds, B. (1994) Economic Methodology and Computability: Some Implications for Economic

- Modelling, IFAC Conf. on Computational Economics, Amsterdam, 1994. <http://cfpm.org/cpmrep01.html>
- [12] Serugendo, G. Di M., et al. (2003) 1st Int. Workshop on Eng. Self-Org. Applications, AAMAS'03, Melbourne, Australia.
- [13] Turing, A. M. (1936) On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc.* **42**:230-65; **43**:544-6. Reprinted in [3].
- [14] Wimsatt, W. (1972). Complexity and Organisation. In Scavenger and Cohen (eds.), *Studies in the Philosophy of Sciences*. Dordrecht: Riddle, 67-86.
- [15] Wolfram, S. (1986) Random sequence generation by cellular automata. *Adv.s in Applied Math.*, **7**:123-169.
- [16] Wolpert, D. (1996) The lack of a priori distinctions between learning algorithms. *Neural Computation*, **8**:1341-1390.
- [17] Wooldridge, M. (2000) *Reasoning about Rational Agents*. Cambridge, MA: MIT Press.
- [18] Wooldridge, M. (2000) The Computational Complexity of Agent Design Problems. *Proc. of the 4th Int. Conf. on MultiAgent Systems*, IEEE Computer Society, 341-348.
- [19] Wooldridge, M. and Jennings, N. (1998). Pitfalls of agent-oriented development. In Sycara, K. P. and Wooldridge, M., (eds.), *Proc. of the 2nd Int. Conf. on Autonomous Agents*, Minneapolis, USA. ACM Press, pages 385-391.

9. Appendix – Proof Outlines

In the outlines below we use some standard results in recursive function theory, which we quote from [1].

Let the specification language, \mathbf{L} , be that of a standard first order classical logic with arithmetic: having symbols: $0, 1, +, ?, =, ?, ?, ?, ?, ?$, as well as variables: $\mathbf{x}, \mathbf{y}, \mathbf{z}, \dots$ and brackets (all with the standard semantics). To formalise the programming and checking problems we need to effectively enumerate statements in this language: $\mathbf{S}_1, \mathbf{S}_2$, etc.; and programs: $\mathbf{P}_1, \mathbf{P}_2$, etc.. $\mathbf{P}_x(\mathbf{y})$ represents the functions that results from program with index \mathbf{x} applied to input \mathbf{y} ; if the program halts it will output the value $\mathbf{P}_x(\mathbf{y})$ and $\mathbf{P}_x(\mathbf{y})?z$? $\mathbf{P}_x(\mathbf{y})$ halts with resulting value \mathbf{z} . We use the following ([1] page 145) due to Gödel[9]:

Suppose that $M(x_1, \dots, x_n)$ is a decidable predicate. Then it is possible to construct a statement $?(x_1, \dots, x_n)$ of \mathbf{L} that is a formal counterpart of $M(x_1, \dots, x_n)$ in this sense: for any $a_1, \dots, a_n? N: M(x_1, \dots, x_n)$ holds iff $?(x_1, \dots, x_n)$ does

Now the predicate $H_n(x, y, z, t)$? $\mathbf{P}_x(\mathbf{y})$ halts in \mathbf{t} or fewer steps resulting in the value \mathbf{z} , is decidable ([1] page 88), so by the above there is a statement $\mathbf{h}(x, y, z, t)$ in \mathbf{L} such that $H(x, y, z, t)$ is true iff $\mathbf{h}(x, y, z, t)$ is. So define \mathbf{SH}_n as $?z?t(\mathbf{h}(x, y, z, t))?x?n?y?n$. The effectiveness of the enumeration of statements in \mathbf{L} means that there is a computable function $?(\mathbf{n})$ such that $\mathbf{S}_{?(\mathbf{n})}$ is \mathbf{SH}_n .

\mathbf{PH}_n is defined as a program that implements an $\mathbf{n}?n$ lookup table whose entries are 0 or 1, where the number at column number x and row y is 1 if $\mathbf{P}_x(\mathbf{y})$ halts and 0 otherwise. This is computable by the Church-Turing Thesis ([1] page 67) due to its finite nature.

There is no computable function, \mathbf{PT} , such that for all \mathbf{n} and $\mathbf{x}, \mathbf{y}?n$, $\mathbf{P}_{\mathbf{PT}(\mathbf{n})}(\mathbf{x}, \mathbf{y})=1$ if $\mathbf{P}_x(\mathbf{y})$ halts (and 0 o.w).

$\mathbf{P}_{\mathbf{PT}(\max(\mathbf{x}, \mathbf{y}))}(\mathbf{x}, \mathbf{y})=1$ if $\mathbf{P}_x(\mathbf{y})$ halts and 0 if it does not. $\mathbf{P}_{\mathbf{PT}(\max(\mathbf{x}, \mathbf{y}))}(\mathbf{x}, \mathbf{y})=?^2_U(\mathbf{PT}(\max(\mathbf{x}, \mathbf{y})), \mathbf{x}, \mathbf{y})$, where $?^2_U$ is the universal binary function which is computable ([1] page 86). If \mathbf{T} was computable then $?^2_U(\mathbf{PT}(\max(\mathbf{x}, \mathbf{y})), \mathbf{x}, \mathbf{y})$ would also be, but this decides the halting problem which is impossible [13].

There is no computable function, $\mathbf{T}(\mathbf{n})$, such that if binary predicate, $\mathbf{S}_n?L$ then $\mathbf{P}_{\mathbf{T}(\mathbf{n})}(\mathbf{y})?z$ iff $\mathbf{S}_n(\mathbf{y}, \mathbf{z})$ holds.

Suppose there was such, then $\mathbf{T}(\mathbf{n})=\mathbf{PT}(\mathbf{n})$ would be computable, which would contradict the previous lemma.

There is no computable function, \mathbf{C} , such that $\mathbf{C}(\mathbf{n}, \mathbf{m})=1$ iff, $?y, z (\mathbf{P}_n(\mathbf{y})?z$ iff $\mathbf{S}_m(\mathbf{y}, \mathbf{z}))$ holds (o.w. 0).

Let: $\mathbf{PPH}_{n,1}, \mathbf{PPH}_{n,2}$, etc. be an enumeration of programs that implement all possible $\mathbf{n}?n$ $\{0,1\}$ lookup tables. Now by the effectiveness of program enumeration and the Church-Turing thesis there is a computable function: $?(\mathbf{n}, \mathbf{m})$ such that $\mathbf{P}_{?(\mathbf{n}, \mathbf{m})}$ is $\mathbf{PPH}_{n,m}$. Now suppose there was such a \mathbf{C} , then $?n(\mathbf{C}(\max(\mathbf{x}, \mathbf{y}), \mathbf{n}), ?(\max(\mathbf{x}, \mathbf{y})))$ is computable (where $?$ is the minimisation function [1] page 43-45) but also the function $\mathbf{T}(\mathbf{n})$ – a contradiction.

GASP machines can emulate any Turing Machine.

The class of Turing machines is computationally equivalent to that of unlimited register machines (URMs) ([1] page 57). That is the class of programs with 4 types of instructions which refer to registers, $\mathbf{R}_1, \mathbf{R}_2$, etc. which hold positive integers. The instruction types are: \mathbf{S}_n , increment register \mathbf{R}_n by one; \mathbf{Z}_n , set register \mathbf{R}_n to 0; $\mathbf{C}_{n,m}$, copy the number from \mathbf{R}_n to \mathbf{R}_m (erasing the previous value); and $\mathbf{J}_{n,m,q}$, if $\mathbf{R}_n=\mathbf{R}_m$ jump to instruction number \mathbf{q} . This is equivalent to the class of AURA programs which just have two types of instruction: \mathbf{S}_n , increment register \mathbf{R}_n by one; and $\mathbf{DJZ}_{n,q}$, decrement \mathbf{R}_n if this is non-zero then if the result is zero jump to instruction step \mathbf{q} [11]. Thus we only need to prove that given any AURA program we can simulate its effect with a suitable GASP system. Given an AURA program of \mathbf{m} instructions: $\mathbf{i}_1, \mathbf{i}_2, \dots, \mathbf{i}_m$ which refers to registers $\mathbf{R}_1, \dots, \mathbf{R}_n$, we construct a GASP system with $\mathbf{n}+2$ agents, each of which has \mathbf{m} plans. Agent \mathbf{A}_{n+1} is basically a dump for discarded tokens and agent \mathbf{A}_{n+2} remains zero (it has the single plan: $(\mathbf{G}_{n+1}, \mathbf{J}_{a+1,1,1})$). Plan \mathbf{s} ($\mathbf{s} \{1, \dots, \mathbf{m}\}$) in agent number \mathbf{a} ($\mathbf{a} \{1, \dots, \mathbf{n}\}$) is determined as follows: there are four cases depending on the nature of instruction number \mathbf{s} :

1. \mathbf{i}_s is \mathbf{S}_a : plan \mathbf{s} is $(\mathbf{J}_{a,s+1,s+1})$;
2. \mathbf{i}_s is \mathbf{S}_b where $\mathbf{b}?a$: plan \mathbf{s} is $(\mathbf{G}_{n+1}, \mathbf{J}_{a,s+1,s+1})$;
3. \mathbf{i}_s is $\mathbf{DJZ}_{a,q}$: plan \mathbf{s} is $(\mathbf{G}_{n+1}, \mathbf{G}_{n+1}, \mathbf{J}_{a,q,s+1})$;
4. \mathbf{i}_s is $\mathbf{DJZ}_{b,q}$ where $\mathbf{b}?a$: plan \mathbf{s} is $(\mathbf{G}_{n+1}, \mathbf{J}_{a,q,s+1})$.

Thus each plan \mathbf{s} in each agent mimicks the effect of instruction \mathbf{s} in the AURA program with respect to the particular register that the agent corresponds to.