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 J. 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 exhibit a simple MAS which has all the power of a Turing machine. We argue that any pure design methodology will face insurmountable difficulties in today's open and complex MAS. We recommend instead a methodology based on experimental method – scientific foundations for MAS construction and control.

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-publicised[1], 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*, 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 it. Here there are the two such problems: one of relating the specifications to executable code and another of relating informal specifications to the formal specifications (this will be dealt with in the next section).

Given the nature of computation and program code two questions present themselves. The first is whether there is any 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 effective means of checking whether a given software system meets a given specification – 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 ([6] 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?*

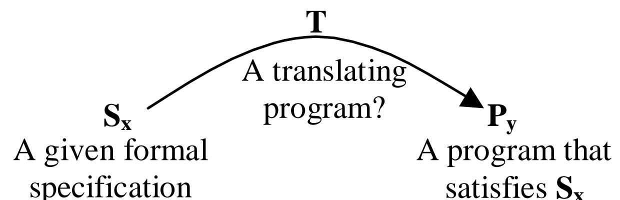

Figure 1. A supposed program from formal specifications to appropriate programs.

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.

[1] We searched the formal methods literature for a list of these limitations (so we could just reference them) with no success.

Turing [18] proved that the 'halting problem' is an undecidable problem (where *undecidable* means there provably cannot exist a program or algorithm to answer it). This is the problem of whether a given program will eventually come to a halt with a given input – whether $\mathbf{P_x(y)}$, program number **x** applied to input **y**, ever finishes with a result or whether it goes on forever. No program can determine this [18].

Define a series of problems, $\mathbf{LH_1}$, $\mathbf{LH_2}$, etc., which we call 'limited halting problems'. $\mathbf{LH_n}$ is the problem of 'whether a program with number $\leq$**n** and an input $\leq$**n** will ever halt'. Each of these *is computable*, since each can be implemented as a finite lookup table[2]. Call the programs that implement these tables: $\mathbf{PH_1}$, $\mathbf{PH_2}$, etc. respectively. Now *if* the specification language can specify each such program one can form a corresponding enumeration of formal specifications: $\mathbf{SH_1}$, $\mathbf{SH_2}$, etc.

The question now is whether there is any way of computationally finding $\mathbf{PH_n}$ from the specification $\mathbf{SH_n}$. 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 $\mathbf{PH_m}$ from $\mathbf{SH_m}$; *and finally* use $\mathbf{PH_m}$ to compute whether $\mathbf{P_x(y)}$ halts. Since we know the general halting problem is uncomputable, we also know that there is no effective way of discovering $\mathbf{PH_n}$ from $\mathbf{SH_n}$ *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 $\mathbf{SH_1}$, $\mathbf{SH_2}$, etc. to be formulated. Unfortunately, the construction in Gödel's famous incompleteness proof [12] guarantees that any formal language that can express even basic arithmetic properties will be able to formulate such specifications. Thus for any useful specification language, the programming problem is also undecidable.

2.1. The Checking Problem

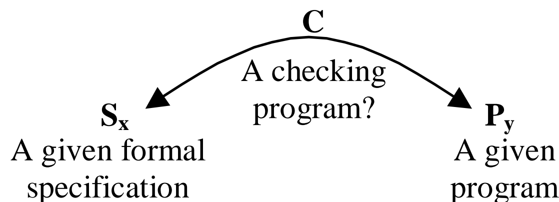

Figure 2. A supposed program for checking whether a program satisfies a specification.

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.

Again the answer is 'no'. To show 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 program **P** meets the specification $\mathbf{SH_n}$. In this case, we will limit ourselves to programs, **P**, that implements $\mathbf{n\times n}$ lookup tables with entries from $\mathbf{\{0,1\}}$.

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: $\mathbf{m\times m\rightarrow\{0,1\}}$; *then* test these programs one at a time using **C** to find one that satisfies $\mathbf{SH_n}$ (we know there is at least one: $\mathbf{PH_m}$); *and finally* use this program to compute whether $\mathbf{P_x(y)}$ halts. Thus there is no such program, **C**.

3.1. 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 changeable 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', $\mathbf{G_a}$, has the effect of giving 1 unit to agent **a** (if the store is non-zero). The 'test instruction', $\mathbf{JZ_{a,p,q}}$, 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 ([6] page 105) holds, so that determining whether an arbitrary given GASP machine is in M is uncomputable.

In conclusion, if a specification language can deal with basic arithmetic then 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. Of course, *almost all* realistic applications

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

require arithmetic. The example of GASP machines shows how simple such MAS can be.[3]

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:

- Agree the goals for the IT system;
- Write a specification that would meet these goals;
- 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 other systems designed by different programmers for disparate (and sometimes unknown) purposes.

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 an IT system cannot be considered in isolation from the web of other systems it is part of – thus no separate, off-line analysis of needs and specifications is possible. Any implemented system will affect the human practice it interacts with in unpredictable ways. 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 (e.g. spam filters).

Environments or systems which are not easily analysable, that change rapidly in unpredictable ways, which are composed of lots of different overlapping aspects, with multiple uses and goals, which are open 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 increasingly the exception. A consequence of the prevalence of messiness and the primary IT systems engineering goal is that *any methodology for constructing IT systems should work well with messy systems.* Trying to keep systems neat so that FSS can be retained, while desirable and in some cases perhaps possible, is putting the cart before the horse.

[3] C.f.[13]; also [23] shows some finite design problems infeasible.

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 (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. If one uses many levels each will add uncertainty – the distance between goals and implementation remains great.

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 [16]. 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 [22], 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 Wooldridge gives the impression that the formal machinery of a logic somehow compensates for the difficulty of validation (see [9]).

Several of Jenning's and Wooldridge's papers suggest limiting the complexity of MAS, so as to limit the difficulties of *designing* a MAS. For example in [24], 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.

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 that with MAS there will be an even smaller proportion of effort spent on the stages characterised by design. The

PDA is obviously a "straw man", however our arguments hold to the extent that there is an overemphasis on the 10% and a passing over of the 90%. This paper is a call to restore the balance – to acknowledge the irreplaceable role of iterative, experimental methods. MAS needs more than maths, logic and philosophy to make it work, it needs *scientific* foundations.

3. Some Software Production Strategies

A host of strategies have been developed to produce IT systems adequate to the messy systems they will be deployed in. 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 focusing 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 – the analogies used must be accurate in terms of the effects of the hidden details they stand 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 a system is composed of many parts (e.g. agents or modules) one can get the parts working together in a basic way by agreeing some standards concerning the form and content of their interaction. However, one cannot ensure the complete compatibility of these interactions using such a standard for the reasons given earlier in Section 1. In fact, strict standards are not even desirable since they overly constrain development. There is always a tension between the flexibility delegated to the parts and the effectiveness of a standard in controlling interaction. The most robust standards are not the result of *a priori* thought but arise, at least substantially, from actual practice. Many elegant invented standards are ignored due to their subtle inappropriateness for common tasks or their complexity. Yet conceptually messy standards found to meet needs flourish.

Modularity: where different roles and tasks are fairly well separable, these can be delegated to different parts of the system. If a 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 [19].

Formalisation: formalisation can reduce 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. Whether automation is possible 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 system 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 implement several different independent processes in parallel. If one mechanism fails, possibly another will work. Social and biological systems abound in this kind of redundancy. This sort of redundancy requires two things to achieve maximum robustness: multiplicity and diversity. If the redundant strategies are essentially the same then their system's robustness to uncertain conditions is reduced – another reason why limitations on MAS to preserve a FSS might be counter-productive.

Adaptability: inflexible systems can be honed so that they are very efficient. However, out of their intended context, they may become useless or even counter-productive. Systems with the ability to adapt to prevailing circumstances require more infrastructure but may in some circumstances be more reliable. Simple learning and decision-making abilities may allow a part to function in effective ways unforeseen by its designer. However, such adaptability comes at the cost of precise control – an ability to cope in unforeseen circumstances generally implies an ability to behave in unforeseen ways. Limiting agents so that their behaviour is predictable (or even provable) can rule out needed adaptability.

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 [16]. In complex cases an individual system may work as designed but not interact with its environment in an acceptable way. 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 [4]. The design of a system (if known) provides a source of suggestions for hypotheses to be

checked, but can never be sufficient on its own, for the reason that they are seldom of a form that allows the direct prediction of behaviour. Debugging is a simple case of such testing.

The approach to engineering MAS over the last decade has emphasised the first five of these strategies: abstraction, automation, standardisation, modularity and formalisation. The chosen abstractions 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. Standardisation has resulted in a host of protocols for communication. The modularity is generally the agent, now supplemented by holonic agents and teams. The formalisation of choice has been logic.

Adaptability is often limited to delaying planning until the moment of choice – the mechanisms of adaptability are therefore limited to those of inference. Techniques for testing and debugging have not significantly developed from those of traditional non-MAS programming techniques.

4. Towards a Science of MAS Development

This paper argues that more emphasis must be placed on the tactics in the second half of the list: transparency, redundancy, adaptability and testing. Further, we suggest that 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.

The main engine of automation in experimental MAS methodology is the simulation – a platform for computational experiments. Modularity in agent-based simulations 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, cultures, societies, institutions and parties for which it is less easy to ascribe precise boundaries. Transparency is often provided by applying mechanisms found in the social or biological domains, though other domains are also possible. An understanding of how such mechanisms may work in their source domains provides a useful starting point for understanding what they might contribute 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. Adaptability is similarly common, with learning taking a key role. For many agents in real-time domains learning can be gradual and complex, but decision making needs to be simple and immediate [5]. Testing in the form of post hoc theorising and experimentation 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 [10]. 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 many cellular automata (e.g. [20]).

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 should be (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 may still 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 a separate part would necessarily loose some of the behaviour that it would display when part of the system. One cause of this is due to the process of embedding – when the rest of the system adapts to the behaviours of the part. For example, this commonly occurs when the part is an IT system and the wider system is the human institution in which it is deployed. As described earlier, humans adapt rapidly to new systems frequently employing them in unexpected ways. A consequence of such embedding is that formal off-line design and implementation is inadequate [1].

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 prori*, 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. Progress will be slower than some might have hoped and require a lot more painstaking empirical work.

5. 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 [7]), 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. In what way does this help in the construction of complex systems? Why the *agent* abstraction rather than one of the other possibilities?

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 must be able to help direct our programming of similar actors in the form of artificial agents. 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 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 [17] and MABS [14]). 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. As a consequence, the findings of these papers may be of more utility to MAS developers than many formalisms.

6. 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 [4]. A complex software system may behave somewhat like its design, but one cannot rely on this. *The intended behaviour of a system is only a hypothesis about the system's behaviour – it must be checked by experimentation.* The hypothesis that the system is behaving as designed must compete against other hypothesis – it is the sensible place to start, but for complex systems it is likely to be wrong.

Nevertheless, we need *some* basis for reusing 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[4]. Each of these should 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. Achieving the desired output randomly should also be improbable in the long term.

As these tests are run they should be annotated 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. Developers may avoid re-using code if they have to rely on properties that are not testable or parameters outside a tested range. Alternatively, developers may write new tests themselves or run existing tests over the parameters they require.

Over time, code and algorithms that are reliable can be established through co-operative and distributed testing. This is similar to 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. spacecraft). There is no reason why a similar kind of reliability can't be built up for software.

7. 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 – e.g. tag-based evolution for the control of loading agents [11].

The "No Free Lunch" theorems [21] 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

[4] See further unit tests [1].

spaces one is concerned with. Further, the stochastic nature of many search algorithms means that proving their properties is unlikely (expect in artificially simple cases). Rather this is primarily an *empirical matter*.

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 it in their system would be able to make evidence-based judgements on its suitability for their purposes. Users or academics might wish to either extend this database of results to new classes of problems or parameter ranges so as to aid engineers or, alternatively, to seek to disconfirm them by careful experiment. Some might dare to make 'second-order' hypotheses about the properties of problem classes that results in certain minimum levels of performance. These meta-hypotheses would then be themselves subject to experimental investigation.

To return to our example of a GA, an entry in the database of hypotheses might be of the form: for a random problem space with a single solution of size **n**, a GA with mutation of 10%, size **m**, will find the solution with probability distribution: D(n, m, t), where **t** is the number of generations.

At the moment although archetypal fitness landscapes for which GA variants are postulated, there is no systematic recording of the conditions of application of hypotheses for others to refer to. This impedes the use of these algorithms by others.

8. Conclusion

The limitations of formal methods have been known since Gödel. We should not be 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 experimental method. Although this means that we will have to give up 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 ('90%' of any software project) to match (or reduce) the effort devoted to the '10%' – the specification and implementation of MAS.

9. Acknowledgements

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

10. References

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

[2] Bryson, J. J. (2004) Modular representations of cognitive phenomena in AI, psychology and neuroscience, *Visions of Mind*, D. Davis (ed.), Idea Group Inc.

[3] Bryson, J. J. & Hauser, M. D. (2002) What monkeys see and don't do, *AAAI Spring Symposium on Safe Learning Agents*, M. Barley & H. W. Guesgen (eds.) AAAI.

[4] Bryson, J. J., Lowe W. & Stein, L. A. (2000) Hypothesis Testing for Complex Agents, *NIST Workshop on Performance Metrics for Intelligent Systems,* A. M. Meystel & E. R. Messina (eds.) 233-240. NIST.

[5] Bryson, J. J. & Stein, L. A. (2001) Modularity and Design in Reactive Intelligence, *IJCAI*, 1115-1120, Morgan Kaufman.

[6] Cutland, N. J. (1980) *Computability*. Cambridge University Press.

[7] Dautenhahn, K. (1997) I could be you – the phenomenological dimension of social understanding. *Cybernetics and Systems*, **25**:417-453.

[8] Davis, M. (ed.) (1965) *The Undecidable*. New York: Raven.

[9] 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

[10] 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**.

[11] Hales, D. & Edmonds, B. (2002) Groups and organizations: Evolving social rationality for MAS using "tags". In *Proc. of the 1st Int. joint conf. on Autonomous Agents and Multiagent Systems*, Bologna, Italy. ACM Press, 497-503.

[12] Gödel, K. (1931) Uber formal unentscheidbare Sätze der Principia Mathematica und verwandter System I. *Monatschefte Math. Phys.* **38**: 173-198. Translated in [8].

[13] Harel, D. (2003) Computers Ltd.: What they *really* can't do. Oxford.

[14] Hales, D. & al. (eds.) (2003) *Multi-Agent Based Modelling III (MABS 2003)*. Springer, LNAI, **2927**.

[15] Moss, S. & Edmonds, B. (1994) Economic Methodology and Computability, IFAC Conf. on Computational Economics, Amsterdam, 1994.

[16] Parnas, D. L. (1985) Software Aspects of Strategic Defense Systems, *American Scientist,* **73**(5): 432-440.

[17] Serugendo, G. Di M., & al. (2003) 1st Int. Workshop on Eng. Self-Org. Applications, AAMAS'03, Melbourne, Australia.

[18] 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 [8].

[19] Wimsatt, W. (1972). Complexity and Organisation. In Scavenger and Cohen (eds.), *Studies in the Philosophy of Sciences*. Dordrecht: Riddle, 67-86.

[20] Wolfram, S. (1986) Random sequence generation by cellular automata. *Adv.s in Applied Math.*, **7**:123-169.

[21] Wolpert, D. (1996) The lack of a priori distinctions between learning algorithms. *Neural Computation*, **8**:1341-1390.

[22] Wooldridge, M. (2000) *Reasoning about Rational Agents*. Cambridge, MA: MIT Press.

[23] Wooldridge, M. (2000) The Computational Complexity of Agent Design Problems. *Proc. of the 4th Int. Conf. on MultiAgent Systems)*, IEEE Computer Society, 341-348.

[24] 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.

11. Appendix – Proof Outlines

In the outlines below we use some standard results in recursive function theory, which we quote from [6].

Let the specification language, **L**, be that of a standard first order classical logic with arithmetic: having symbols: 0, 1, +, ×, =, ∀, ∃, ¬, ∧, ∨, →, as well as variables: **x**, **y**, **z**, … 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.. $\mathbf{P_x(y)}$ represents the functions that results from program with index **x** applied to input **y**; if the program halts it will output the value $\mathbf{P_x(y)}$ and $\mathbf{P_x(y)\downarrow z} \equiv$ '$\mathbf{P_x(y)}$ halts with resulting value **z**'. We use the following ([6] page 145) due to Gödel [12]:

*Suppose that $M(x_1,\ldots,x_n)$ is a decidable predicate. Then it is possible to construct a statement $\sigma(\mathbf{x_1},\ldots,\mathbf{x_n})$ of **L** that is a formal counterpart of $M(x_1,\ldots,x_n)$ in this sense: for any a1,…an∈N: $M(x_1,\ldots,x_n)$ holds iff $\sigma(\mathbf{x_1},\ldots,\mathbf{x_n})$ does*

Now the predicate $H_n(x,y,z,t) \equiv$ '$\mathbf{P_x(y)}$ halts in **t** or fewer steps resulting in the value **z**', is decidable ([6] page 88), so by the above there is a statement **h(x,y,z,t)** in **L** such that H(x,y,z,t) is true iff **h(x,y,z,t)** is. So define $\mathbf{SH_n}$ as $\exists z \exists t(\mathbf{h(x,y,z,t)}) \wedge \mathbf{x} \leq \mathbf{n} \wedge \mathbf{y} \leq \mathbf{n}$. The effectiveness of the enumeration of statements in **L** means that there is a computable function $\theta(\mathbf{n})$ such that $\mathbf{S_{\theta(n)}}$ is $\mathbf{SH_n}$.

$\mathbf{PH_n}$ is defined as a program that implements an **n**×**n** lookup table whose entries are 0 or 1, where the number at column number x and row **y** is 1 if $\mathbf{P_x(y)}$ halts and 0 otherwise. This is computable by the Church-Turing Thesis ([6] page 67) due to its finite nature.

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

$\mathbf{P_{PT(max(x,y))}(x,y)}=1$ if $\mathbf{P_x(y)}$ halts and 0 if it does not. $\mathbf{P_{PT(max(x,y))}(x,y)}=\psi^2_U(\mathbf{PT}(\max(\mathbf{x},\mathbf{y})),\mathbf{x},\mathbf{y})$, where ψ^2_U is the universal binary function which is computable ([6] page 86). If **T** was computable then $\psi^2_U(\mathbf{PT}(\max(\mathbf{x},\mathbf{y})),\mathbf{x},\mathbf{y})$ would also be, but this decides the halting problem which is impossible [18].

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

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

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

Let: $\mathbf{PPH_{n,1}}$, $\mathbf{PPH_{n,2}}$, etc. be an enumeration of programs that implement all possible $\mathbf{n}\times\mathbf{n}\rightarrow\{0,1\}$ lookup tables. Now by the effectiveness of program enumeration and the Church-Turing thesis there is a computable function: $\phi(\mathbf{n},\mathbf{m})$ such that $P_{\phi(n,m)}$ is $\mathbf{PPH_{n,m}}$. Now suppose there was such a **C**, then $\mu\mathbf{n}(\mathbf{C}(\phi(\max(\mathbf{x},\mathbf{y}),\mathbf{n}),\theta(\max(\mathbf{x},\mathbf{y}))))$ is computable (where μ is the minimisation function 10 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) ([6] 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 **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 junp to instruction step **q** [15]. 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 **m** instructions: $\mathbf{i_1}$, $\mathbf{i_2}$, …, $\mathbf{i_m}$ which refers to registers $\mathbf{R_1}$, …, $\mathbf{R_n}$, we construct a GASP system with **n**+2 agents, each of which has **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 **s** (**s**∈{1,…,**m**}) in agent number **a** (**a**∈{1,…,**n**}) is determined as follows: there are four cases depending on the nature of instruction number **s**:

1. $\mathbf{i_s}$ is $\mathbf{S_a}$: plan **s** is ($\mathbf{J_{a,s+1,s+1}}$);
2. $\mathbf{i_s}$ is $\mathbf{S_b}$ where **b**≠**a**: plan **s** is ($\mathbf{G_{n+1}}$, $\mathbf{J_{a,s+1,s+1}}$);
3. $\mathbf{i_s}$ is $\mathbf{DJZ_{a,q}}$: plan **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 **b**≠**a**: plan **s** is ($\mathbf{G_{n+1}}$, $\mathbf{J_{a,q,s+1}}$).

Thus each plan **s** in each agent mimics the effect of instruction **s** in the AURA program with respect to the particular register that the agent corresponds to.