Using the Experimental Method to Produce Reliable Self-Organised Systems

Bruce Edmonds

Centre for Policy Modelling
Manchester Metropolitan University
cfpm.org/~bruce

Abstract. The 'engineering' and 'adaptive' approaches to system production are distinguished. It is argued that producing reliable self-organised software systems (SOSS) will necessarily involve considerable use of adaptive approaches. A class of apparently simple multi-agent systems is defined, which however has all the power of a Turing machine, and hence is beyond formal specification and design methods (in general). It is then shown that such systems can be evolved to perform simple tasks. This highlights how we may be faced with systems whose workings we have not wholly designed and hence that we will have to treat them more as natural science treat the systems it encounters, namely using the classic experimental method. An example is briefly discussed. A system for annotating such systems with hypotheses, and conditions of application is proposed that would be a natural extension of current methods of open source code development.

1. Introduction

Zambonelli and van Dyke (in parallel with others) have pointed out that, increasingly, a different *kind* of computer system will be required if we are to meet many of society's needs [19] and these are starting to be developed. In this paper I go further[1] and argue that in parallel with different *kinds of system* we will need a *different kind of approach* to producing such systems – an approach which places more emphasis on natural scientific approaches than has been usual in multi-agent systems. In other words, that design and engineering (in a sense I will make clear) must make more room for adaptation and experiment.

I start by distinguishing what I call the 'engineering' and 'adaptation' approaches and their how they are applied (both separately and together). I then discuss some of the limitations of the engineering method by considering an apparently simple class of MAS that nonetheless is, in general, intractable to methodical and effective design methods (the limitations of the adaptation method being fairly obvious). In contrast I show that these systems can be adapted to serve defined (albeit simple), purposes

[1] To be clear, it is not that Zambonelli and van Dyke in [19] don't see a need for a change in method as well as the change in system type, but that they do not see such a need to depart from the engineering approach to the extent I am suggesting.

using an evolutionary algorithm. These two sections lead on to the conclusion that we will necessarily have to develop and deploy systems for which there is no complete understanding based on its design. Such systems (and many others) will have be understood as we do with other 'ready-made' systems in the natural world: by hypothesis and experiment. I then sketch how such a natural science of self-organised systems may be used to achieve fallible but high levels of reliability and (relatively) safe system reuse. I end by giving a short example to illustrate this before I conclude.

2. Two Approaches for Obtaining Useful Systems

There are two basic ways of getting a useful system: by designing and then implementing it so as to construct it (what I will call the "engineering approach"); or by taking some existing system and then manipulating it until it is good enough (what I will call the "adaptive approach"). I briefly explain these approaches which are then illustrated in figure 1, before considering their combination.

2.1 The Engineering Approach

The engineering approach seeks to develop a series of methods so that the resulting construction is as useful as possible when the construction is finished. For example the processes by which a steel girder is made is such that, probably, it will have certain physical characteristics when made (torsion strength etc.). This approach focuses on what can be done *before* the system has been constructed, thus it concentrates upon developing methodologies and practices to obtain to its goals. These methods may be based to some extent upon an underlying theory of systems and system construction, but on the whole they are systemisations of what has been found to work in the past. Thus essentially one relates what one wants to methods that have been found in the past to produce this and makes a plan and then implements it to achieve the result.

2.2 The Adaptive Approach

The adaptive approach takes an existing system and seeks to interact with the system, going through a cycle of testing its current properties and changing it, until it is acceptable. For example, one may train a dog so that it acquires the behaviours and habits that you need to guard your house (barking at strangers etc.). As with the engineering approach, this may be based upon some theory of the system or it may just be a matter of trial and error. This approach focuses on what can be done with a system *after* it is constructed and is done by comparing current properties against the desired properties and deciding what changes might move it from having the former to achieving the later.

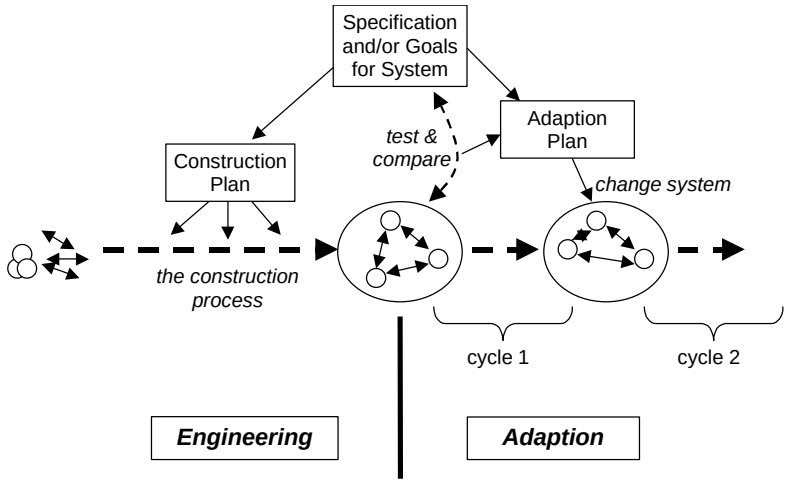

Fig. 1. An illustration of the engineering and adaptation phases before and after system creation

2.3 Combining the Two Approaches

Of course, the two approaches are usually used together, and this is shown in figure 1. Thus however carefully a steel girder is constructed using established methods, it is tested for flaws before being used. Similarly one often has to make an initial system in order to be able to start adapting it and one often employs the engineering approach when one wants to structurally adapt parts of an existing system. Furthermore these approaches are often combined at different levels: engineering a bridge uses basic design forms which have been developed by a process of adaptation; and adapting the design of a car uses pre-engineered parts.

In the production of software systems, one typically first applies the engineering approach and then follow this with the adaptation approach – "10% implementation and 90% debugging", as the adage goes. However, this is not always the case for sometimes these occur in different combinations. For example, one might be faced with a legacy system, in which case one might be limited to the adaptation approach plus engineering additional wrappers, interfaces etc. If the adaptation process fails to get the system up-to-scratch one might be forced to re-engineer substantial sections of the system. Also these approaches might be used at different levels: thus one might *engineer* a mechanism to *adapt* some software; or *train* a human to *engineer* a compiler; or *construct* code from a higher level language etc.

2.4 Using the Approaches Separately

Despite the fact that these two approaches are most effectively used together, there are large sections of computer science dedicated to eliminating the need for one or other of them. Thus genetic programming and other techniques in Machine Learning minimises the engineering phases at the object level, starting with randomised systems and adapting them from there. Similarly the formal methods community seems to wish to eliminate the adaptation phase and reduce system production to purely the engineering phase. This unfortunate trend has been exacerbated by two factors: *firstly*, the split between the AI and ML communities with their different conferences, journals, approaches, traditions etc. and, *secondly*, the formalist trend in computer science which attempts to reduce the adaptation phase by making the engineering phase a formal science akin to mathematics or logic.

3. The Insufficiency of Engineering for SOSS

Elsewhere [6], Joanna Bryson and I criticise an over-reliance on formal design methods, where the engineering approach is focussed on to the exclusion of adaptation. There I show a number of formal results, which are basically simple corollaries of Gödel [8] and Turing [18]. These can be summarised as follows: for a huge range of specification languages (e.g. those that essentially include arithmetic):

1. *There is no general systematic or effective method that can generate or find a program to meet a given specification.*
2. *There is no general systematic or effective method that, given a formal specification and a program, can check whether the program meets that specification.*

Where "general systematic of effective method" means one that could be implemented with a Turing Machine. These results hold for the overwhelming majority of classes of systems, including all those which include integer arithmetic. This illustrates the 'gap' between formal specifications and programs – a gap that will not be bridged by automation.

To illustrate how simple such systems can be, I defined a particular class of particularly simple MAS, called GASP systems (Giving Agent System with Plans). These are defined as follows. There are **n** agents, labelled: **1, 2, 3**, etc., each of which has an integer store which can change and a finite number of plans (which do not change). Each time interval the store of each agent is incremented by one. Each plan is composed of: a (possibly empty) sequence of 'give instructions' and finishes with a single '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' is of the form $\mathbf{JZ_{a,p,q}}$, which has the effect of jumping (i.e. designating the plan that will be executed next time period) to plan **p** if the store of agent **a** is zero and plan **q** otherwise. 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. This is illustrated in figure 2.

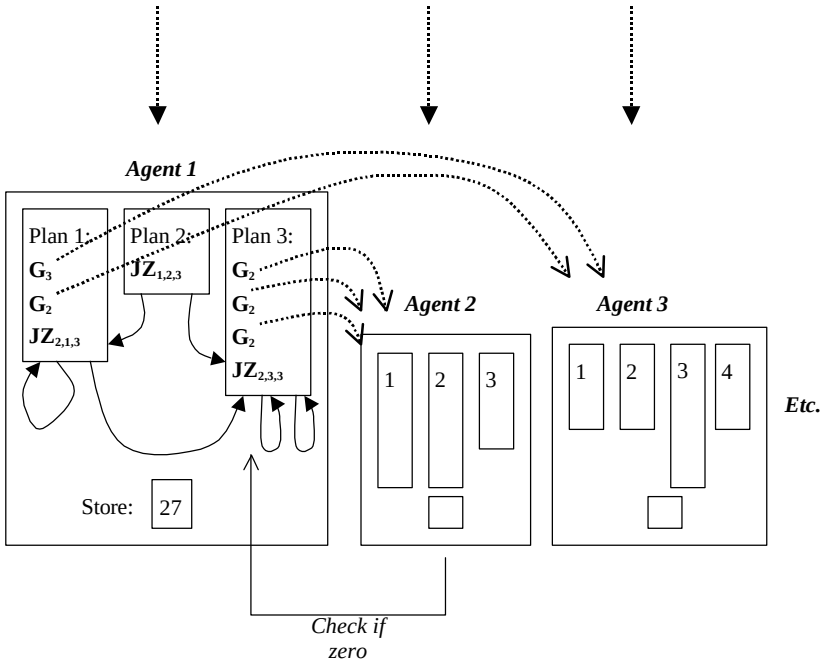

Fig. 2. An illustration of the working of GASP MAS: Each agent has a single store and a fixed number of very simple plans composed of a list of "give one" instructions and a final one of "if agent ***x***'s store is zero the go to plan ***a*** next, else plan ***b***".

However GASP systems have the same power as Turing machines, and hence can perform *any* formal computation at all (a proof outline of this can be found in [6]). Since GASP systems are this powerful, many questions about them are not amenable to any systematic decision procedure. In particular, the above two results hold. Thus formal design methods can not provide a complete solution for system construction, and may only be effective for relatively simple systems.

Part of the problem seems to be the illusion that computational systems are predictable, simply because at the micro-level, each step of a computation is predictable. However, as the example of GASP systems shows, this is not the case. For even though working out what may happen next at any given stage is simple, it is impossible to compute many general aspects of their behaviour, from whether two machines will have the same effect in terms of their stores to whether a given machine will ever stop [4]. Thus we must give up the over-ambitious aim of complete reliance on the engineering approach when we consider MAS of even minimal complexity, and certainly for self-organised systems.

4. Producing Self-Organised Software Systems (SOSS)

Since we can not totally rely on *designing* self-organised MAS we need to consider also using adaptation as a *principle* method of useful system production, and not just as an after-thought to "fine tune" and "debug" systems we have already engineered. To show the possibility of this I have evolved GASP systems to perform some simple tasks. These use a simple and untuned evolutionary algorithm with small populations of simple GASP systems over relatively short time runs, but nonetheless develop the desired properties. Of course, people have been evolving computational systems for about 40 years. The purpose of this section is to show: (1) that this can be done in very simple but effective ways with systems that are Turing-complete[2]; and (2) that this can be done with a MAS.

The evolutionary algorithm was extremely simple. A population of GASP systems were evolved. Each generation 1/3rd of the GASPs with the best fitness were preserved unchanged, the 1/3rd with worst fitness were culled, and the best 2/3rds mutated (with a 10% chance of any number in any plan being replaced by a new random number of the appropriate range) and entered into the population. This is called "Evolutionary Programming" [7] it can be seen as sort-of stochastic hill-climbing algorithm on a population. The algorithm is illustrated in figure 3.

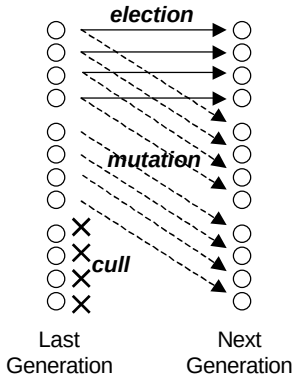

Fig. 3. The simple evolutionary algorithm applied to evolve GASPs: each generation the GASPs are ranked; the top 1/3 elected; the top 2/3 mutated; and the bottom 1/3 culled.

This does not produce "open-ended" evolution, as can occur in Genetic Programming [11, 12], since the length of plans, the number of plans and agents is fixed. This could be fixed by including an operator to possibly increase these – this would probably result in the discovery of more sophisticated solutions [15].

---

[2] An interesting approach to evolving Turing Complete machines is [16,**Error! Reference source not found.**].

4.1 Task 1: Long periodic pattern development

To show that GASP systems producing outputs of increasing complexity can be evolved, I defined the fitness function as the period that the GASP system settled down into (if it did, the maximum otherwise) in terms of changes in the agents' stores. Thus each generation I ran each of 24 GASP systems for 500 time periods and at the end determined the period of repetition of the system. That is how far back one has to go to reach the same pattern as the last one. If there was no evidence of any such pattern (i.e. if the GASP does settle down to any repetitive behaviour so the time of the onset of this behaviour + the period of the repetition is > 500) it was accorded the maximum fitness and the evolution was halted.

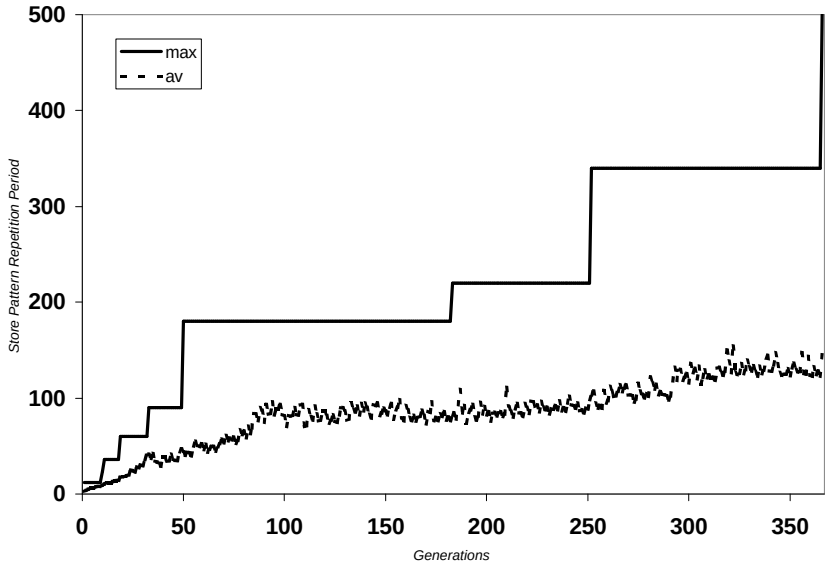

Fig. 4. The evolution of a GASP with a resulting repetitative period of over 500 time periods in 366 generations, with a population of 24 GASPs, each with 10 agents, each with 5 plans, each of which have 'give lists' of up to 3 instructions long (plus a "test for next" instruction).

The ease with which a GASP may be evolved to exhibit long periodic behaviour is strongly related to the number of agents and plans. A similar population of 24 GASPs of 10 agents, each with 10 plans achieved a periodic behaviour of greater than 1000 iterations in only 24 generations. In similar experiments I was able to evolve GASPs with periodic behaviour with high prime factors (there is an example in the appendix).

All that this shows is that it is feasible to evolve GASPs of increasing complexity. The next task is more difficult and more suited to the distributed nature of GASPs.

4.2 Task 2: Anti-avalanche defence

The next task chosen is better suited to the nature of GASP systems, that is the distribution of their stores. The task here is to distribute its stores among its agents so that half of them have stores that are greater than those generated by an accumulating score to which shifting avalanches contributed to. One can think of the agents piling up the defences to keep out increasing piles of snow resulting from the avalanches. These avalanches are generated by a self-organised critical system and is known to produce avalanches whose distribution follows a power-law, and which is very difficult to predict [1]. The task of the GASP is to redistribute the units that are fed evenly (one to each agent) to the correct places to counteract the accumulating results of the avalanches. This is a continual race – the GASP is evaluated over its success at maintaining this over 25 cycles, but each time there may be a different pattern of inputs to the avalanche and a different pattern of avalanches. The overall set-up is illustrated in figure 4.

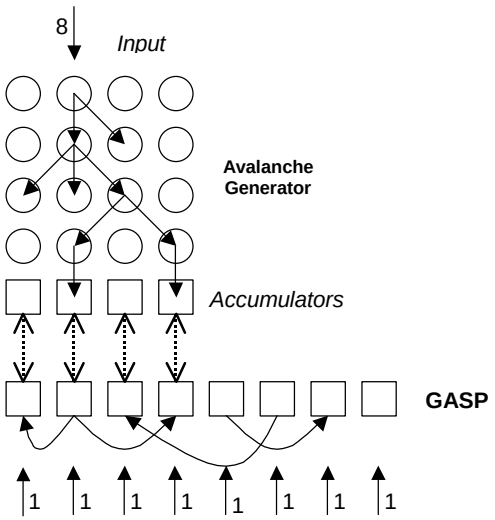

Fig. 5. An illustration of the target problem – the job of the agents in the GASP is to have more in their store that the corresponding accumulators receiving the results of the avalanches

The avalanche generator is a version of the basic 'sand-pile model' investigated by Per Bak and others [1]. It comprises of a set of piles, such that when a pile gets above a critical height it topples over onto adjoining piles, possibly causing them to topple etc. Units are constantly added, in this case to a pile along the 'top' edge. In this version when piles topple the units 'fall' randomly onto the three piles in the next row down in the adjoining columns (as illustrated in figure 4). The result is that the avalanche generator outputs, on average, the same number of units as was input but in irregular avalanches of various sizes. This makes it a difficult task to learn because

the best GASPs in the long term will be those that ignore the particularities that give selective advantage in a single generation, but rather learns a more general strategy.

To the advantage of the adaptive approach I set up the evolution so that the problem it is trying to solve changes during the evolution. The two versions of the problem are the 'variable input' and the 'fixed input' problem. In the variable input problem the input to the avalanche generator remains at a certain column position for a random number of iterations (in the range [1,10]) and then relocates to another randomly chosen position. This means that the avalanches will result with more being accumulated in the columns adjoining to the input position wherever it is, so in the variable problem this will change every now and then. In the fixed input problem the input is always at the first column, so there will be more long-term bias to the same output accumulators. Thus the variable input problem is more difficult to solve.

The GASPs were evolved against the variable input problem for the first 100 generations, then against the fixed input problem for 100 generations and back again to the variable input problem for the last 100 generations. In each generation the GASP is evaluated against 30 iterations of the GASP and avalanche generator. Each generation the avalanche generator is differently initialised with piles of random height below the critical height so the exact avalanche patters will be different every time – thus this is far from a static problem! Figure 6 show the success of this evolution over 31 runs.

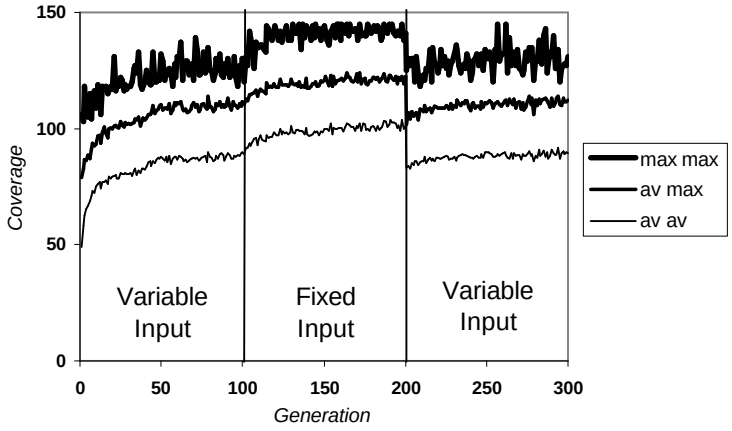

Fig. 6. Statistics showing the extent that the evolved GASPs covered the incoming avalanches (max=150). These are over 31 runs of the evolutionary algorithm, each with a population of 12 GASPs where each GASP is evaluated over 30 iterations. Bottom line shows the average over the 31 runs of the average coverages, next line the average of the maximum coverages and the top line the maximum of maximum coverages.

As you can see, the GASPs evolve over the first 100 generations until they have learned to cover the avalanches to a certain extent. Then when the problem unexpectantly changes at generation 100 and becomes easier they quickly adapt to

this. Finally when the problem is switched back to the variable input problem at generation 200 they have to relearn to cope with this (although this is much quicker than for the first time). This illustrates how an adaptive system (involving continual evolution) may be able cope with the unexpected better than an 'one-off' solution (however constructed). Simply taking the current best GASP is a crude way of using the learning achieved by the whole system, there are better ways (e.g. [15]).

One can imagine this sort of system being applied to combat fraud where the type of fraud is being continually innovated. Beating a system that continually evolves is much more difficult than beating a static target. If the fraudsters (or virus writers!) invent systems to continually evolve their agents this might be the only effective defence. This is being investigated in the sub-field of artificial immune systems [3].

5. Putting the Production of SOSS onto a Sound Basis

If I am right that many SOSS will be evolved to a considerable extent rather than purely designed, and that formal methods will not be able to ensure that such systems meet their specification, then we are left with a problem.

This problem is: *how are we to ensure that the systems we produce will perform satisfactorily when they are deployed in their operating context*?

The answer I suggest is this: *by systematically applying the classic experimental method used in the natural sciences*.

In other words, that we should make *explicit testable hypotheses* about the important characteristics of the systems we produce (by whichever means) and test these *experimentally* to determine: (1) their reliability and (2) their scope (i.e. the conditions under which they hold. These hypotheses should accompany the systems' publication, and be used by those who are considering using that system.

In addition to the hypotheses should be sets of conditions under which it has been tested. Thus if a system has been run repeatedly using a certain range of parameters and other settings, in certain conditions and it was found that in these circumstances the hypotheses held, them these circumstances should be appended to the hypotheses. As the system is tested in more circumstances this set should grow. When someone who wants to use the system for the properties listed in the hypotheses they should check that the circumstances it will be deployed under are covered by one of those that are listed as having been tested. If they are not, the person has the choice of either testing it themselves (and adding to the list if successful) or choosing another system. In this way there will be a slow co-evolution of the code, the hypotheses and the list of conditions as a result of the interaction of those using the system.

One can imagine some sort of open-access distributed repository and database for such systems, hypotheses and conditions of application. Programmers (or system growers!) would place their systems in the repository with the normal documentation and some hypotheses and tested conditions of application. Others would test it under new conditions as they needed to and add this information to the database. Useful systems that were found to be reliable under sufficiently wide conditions would get to be used and test a lot – systems whose scope was found to be narrow would be passed over. Eventually new versions of these systems would be made and the process

continue. Such a system would be a natural add-on to the distributed way some open-source code is developed.

It should be now clear how this is simply an application of the 'classic' scientific experimental method. The world of software systems is one about which hypotheses are made, tested and developed. The crucial test of a system is not its relation (if any) to a designer's intentions for it but its proven performance in terms of the hypotheses about it. This marks a shift of emphasis away from verification to validation.

Now, of course, the method of construction and/or the process of adaptation are good sources for these hypotheses about system behaviour, but they are neither necessary (the only sources) nor sufficient (they can't be relied upon to be correct). Other hypotheses might come about solely from observing their behaviour (and maybe internal workings). Some others might be special cases of more general hypotheses concerning identified *classes* of system. A broad and important source for such hypotheses originate from other fields such as biology (e.g. Evolutionary Computation) or sociology (e.g. reputation-based mechanisms).

Thus there is a loose relation between: the plan of construction; the theory about the system; and any adaptation plan. For example: the system theory may be suggested by the construction plan; the adaptation plan may be informed by the system theory; the success of an adaptation plan may suggest a system theory; or a construction plan may be informed by the system theory. This set of relations is shown in figure 7.

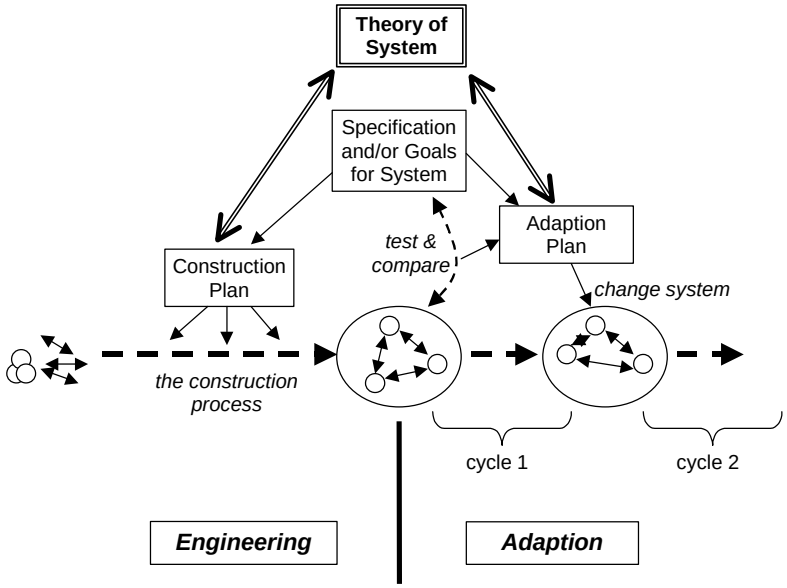

Fig. 7. An illustration of the relation of theory to the engineering and adaptation approaches.

Of course, as in science, *once* a theory has become established (by being extensively and independently tested), it can then be used to *deduce* things about the systems concerned. Formal deduction has a role with respect to *whole* complex and self-organised systems, but one that comes into its own only *after* a system theory has been experimentally established.

6. Example Cases

6.1 Hypothesising about systems in evolutionary computation

There are areas of computing where something like an experimental method is widely applied, e.g. the field of Evolutionary Computation (EC). For example [13] proposes several hypotheses about the causes of bloat in GP populations and then tests them experimentally. This is indicative of the field. Whilst there are a few formal results and models (mostly of fairly simple cases and systems), the majority of the work could be described as experimental. Furthermore, in the sense that types of system are produced whose properties are broadly know and which are successfully applied in other systems and combined with other systems, it is successful.

However, more generally the hypothesising in evolutionary computation is usually: (1) specific to performance on a particular set of problems and (2) does not include the scope under which the hypotheses are found to hold. This makes it very difficult for a person considering applying such a system to come to a judgment upon its use for a different but similar problem. The hypotheses about the system are specific to particular problems, so one has to guess whether it is likely to be applicable to the new problem; and you do not know whether the system performance will extend to a new scope. Thus the reuse of such systems requires much individual experimentation.

6.2 Hypothesising about tag-based SOSS

'Tags' are features that are initially arbitrary but identifiable features of an agent that can act as a (fallible) indication of cooperative group membership, when part of a suitably evolutionary process. They allow a dynamic but persistent maintenance of cooperation across a whole population even when defection is possible, without complex mechanisms such as: contracts, reputation or kin-recognition. This can occur because cooperative groups with similar tags are continually forming and persisting for a period before being invaded by a defector (which quickly destroys the group). Tag systems, and their possible relevance to SOSS are discussed in [10].

In common with many SOSS, tag-based systems are stochastic and fallible. That is, there is always a probability that cooperative groups will not occur. Thus one could never prove from its specification that the system would work as intended. However this effect seems robust over a range of settings and implementation

variations. Thus its seems a viable hypothesis that such systems will result in significant amounts of cooperation over a reasonably wide range of settings.

David Hales has been working on such tag-based systems, work in which I have played a small part. As a result of inspecting the results of such systems, several hypotheses about the working of such systems, and hence the conditions under which cooperative groups might occur, have suggested themselves. One such condition that has been recently identified [9] is that the rate of tag mutation must be greater than the rate of defection in (or into) a cooperative group. This seems to be because it allows for new cooperative groups to form sufficiently often that there is always a significant 'population' of pure cooperative groups before the defection occurs in them. Thus although each group will inevitably be overrun with defectors, there are always enough cooperative groups in the total population to maintain the overall levels of cooperation. Thus we not only have a hypothesis about a class of systems which has been observed, but also some of the conditions under which it is thought to occur, and a mechanism by which it is though to occur. The information published might be:

- ?? $\boldsymbol{S}$ is a system whose description and/or method of production is described in sufficient detail to enable it to be made (at least with high probability), in this case one of the tag-based systems described in [10] or [9];
- ?? $\boldsymbol{H_i}$ are the hypothesis about $\boldsymbol{S}$ that encode the useful properties, for example that "the percentage of co-operators in the overall population is at least 30%";
- ?? $\boldsymbol{C_{i,j}}$ are the conditions under which each of $\boldsymbol{H_i}$ has been found to hold, for example: "the mutation probability of the tag > mutation probability of defection";
- ?? $\boldsymbol{S_{i,j}}$ is additional information accompanying each of the $\boldsymbol{C_{i,j}}$, for example: frequency of significance statistics concerning the occurrence of $\boldsymbol{H_i}$ under $\boldsymbol{C_{i,j}}$.

For SOSS that turn out to be useful, the $\boldsymbol{H_i}$, and $\{\boldsymbol{C_{i,j}}, \boldsymbol{S_{i,j}}\}$ will be added and refined, making the particular system even more useful and hence tested. Thus a 'meta-evolutionary' process will take place with the useful systems becoming selected and tested, and the unreliable and brittle systems being passed over. This is directly analogous to the scientific process as conceptualised by Popper [14].

7. Conclusion

'Engineering' and 'self-organisation' do not sit well with each other. The extent to which a system is engineered will constrain (as well as enable) what kind of self-organisation can occur. Likewise the extent to which self-organisation occurs will limit the scope for engineering since outcomes will be correspondingly undeducable. In other words, self-organisation will result in outcomes that are not (and can not be) foreseen by any designer. Thus with self-organised systems there will always be the possibility of an unwelcome surprise. These surprises will often have to be dealt with by adapting the system after its creation. If we are to do better than trial and error in such adaptation we will need to develop *explicit hypotheses* about our systems and these can only become something we can rely on, via *replicated experiment*. This paper can be seen as an exploratory step towards such an experimental method.

8. Acknowledgements

Thanks to the participants of the ABSS SIG of AgentLink II for their comments about the talk that eventually grew into this paper, especially Joanna Bryson. Thanks also to Scott Moss and David Hales for discussions on these issues (and everything else).

9. References

1. Bak, P. (1997) *How Nature Works: The Science of Self Organized Criticality*. Oxford, Oxford University Press.
2. Baram, Y., El Yaniv, R. & Luz, K. (2004). Online Choice of Active Learning Algorithms. *Journal of Machine Learning Research*, **5**:255-291. http://www.jmlr.org/papers/v5/baram04a.html
3. de Castro, L. N. & Timmis, J. I. (2002), *Artificial Immune Systems: A New Computational Intelligence Approach*, Springer-Verlag, London, September, 357 p.
4. Cutland N.J. (1990). *Computability*. Cambridge: Cambridge University Press.
5. Edmonds, B. (2002) Simplicity is Not Truth-Indicative. CPM Report 02-99, MMU, 2002.
6. Edmonds, B. & Bryson, J. (2004) The Insufficiency of Formal Design Methods – the necessity of an experimental approach for the understanding and control of complex MAS. AAMAS04, New York, July 2004.
7. Fogel, L. J., Owens, A. J. and Walsh, M. J. (1967). *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons.
8. Gödel, K. (1931) Uber formal unentscheidbare Sätze der Principia Mathematica und verwandter System I. *Monatschefte Math. Phys*. **38**:173-198.
9. Hales, D. (2004, submitted) Change Your Tags Fast! - a necessary condition for cooperation? Submitted to the MAMABS workshop at AAMAS 2004.
10. Hales, D. and Edmonds, B. (2003) Evolving Social Rationality for MAS using "Tags", In Rosenschein, J. S., et al. (eds.) *Proc. of the 2nd Int. Conference on Autonomous Agents and Multiagent Systems*, Melbourne, July 2003 (AAMAS03), ACM Press, 497-503.
11. Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, MIT Press..
12. Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Subprograms*. Cambridge: MA, MIT Press.
13. Langdon, W. B. Terry Soule, Riccardo Poli and James A. Foster (1999) The Evolution of Size and Shape. In Lee Spector, William B. Langdon, Una-May O'Reilly and Peter J. Angeline (eds.) Advances in Genetic Programming, Volume 3. MIT Press, 163-190.
14. Popper, K. R. (1969) *Conjectures and Refutations*, London : Routledge & Kegan Paul,
15. Stanley, K.O. and Miikkulainen, R. (2004) Competitive Coevolution through Evolutionary Complexification, *Journal of Artificial Intelligence Research*, **21**:63-100. http://www.jair.org/abstracts/stanley04a.html
16. Teller, A. (1994) The Evolution of Mental Models. In Kenneth E. Kinnear, Jr. (ed.) Advances in Genetic Programming. MIT Press, 199-220.
17. Teller, A. (1996) Evolving Programmers: The Co-evolution of Intelligent Recombination Operators. In Peter J. Angeline and Kenneth E. Kinnear, Jr. (eds.) Advances in Genetic Programming , Volume 2. MIT Press, 45-68.
18. Turing, A.. M. (1936) On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc*. **42**:230-65; **43**:544-6.

19. Zambonelli, F., and Van Dyke, P. H. (2002) Signs of a Revolution in Computer Science and Software Engineering. 3rd International Workshop on Engineering Societies in the Agents World, Madrid, Spain. http://www.ai.univie.ac.at/~paolo/conf/ESAW02/

10. Appendix – an example GASP

This is an example of a simple GASP evolved to have a repetitative period of 89. First I list the plans of the agents (agent, plan number, give list, test agent, then, else). This is followed by a graph showing the cycle in 3 of the stores.

```
1, 1, [], 5, 3, 2
1, 2, [], 4, 2, 1
1, 3, [], 5, 2, 4
1, 4, [1], 3, 2, 4
1, 5, [3 4], 2, 1, 5
2, 1, [], 5, 4, 3
2, 2, [], 4, 2, 4
2, 3, [3 6 6], 1, 5, 1
2, 4, [6 5 4], 2, 2, 3
2, 5, [6 3 3], 3, 3, 2
3, 1, [], 4, 3, 1
3, 2, [6], 5, 3, 4
3, 3, [3 4 2], 3, 3, 4
3, 4, [4 4 5], 1, 3, 5
3, 5, [3 6], 1, 2, 1
4, 1, [], 1, 3, 3
4, 2, [], 1, 5, 5
4, 3, [3 3], 3, 3, 5
4, 4, [2], 1, 3, 1
4, 5, [3 2], 5, 5, 4
5, 1, [3 2 2], 5, 3, 5
5, 2, [1 6], 2, 3, 1
5, 3, [3 1 5], 2, 2, 4
5, 4, [1 2 1], 5, 1, 4
5, 5, [4 4], 4, 4, 4
```