Complex 2004

Proceedings of the 7th Asia-Pacific
Conference on Complex Systems
Cairns Convention Centre, Cairns, Australia
6-10th December 2004

Software engineering for artificial life, complex systems, and agent-based distillation

James Watson[1], Hussein A. Abbass[2], Chris Lokan[3], and Peter Lindsay[1]
[1]*School of Information Technology and Electrical Engineering,*
University of Queensland, QLD 4072, Australia.
Email: {j.watson, Peter.Lindsay}@itee.uq.edu.au
[2]*Artificial Life and Adaptive Robotics Laboratory, School of Information Technology and Electrical Engineering, University of New South Wales, Australian Defence Force Academy,*
Canberra, ACT 2600, Australia.
Email: h.abbass@adfa.edu.au.
[3]*School of Information Technology and Electrical Engineering, University of New South Wales, Australian Defence Force Academy, Canberra, ACT 2600, Australia.*
Email: c.lokan@adfa.edu.au

Abstract

Research in *artificial complexity* (i.e., artificial life, complex systems, and agent-based distillation), largely depends on the use of computer software. The reliability of the obtained results is directly related to two key assumptions: (1) that the software represents the problem as intended by the developer; and (2) that the software is bug-free. However, developing software for artificial complexity offers new challenges to software engineering, and it becomes imperative to discuss the software engineering problems arising in artificial complexity that are different from traditional computer programming. The first aim of this paper is to identify the possible challenges confronted by developers of systems involving artificial complexity. The second is to offer preliminary suggestions on how to incorporate existing software engineering techniques into artificial complexity's simulation development.

1. Introduction

Research in *Artificial Life* (ALife), *complex systems science and engineering* (CSSE), and *agent-based distillation* (ABD), is yielding solutions to real life problems where traditional approaches seem to fail. In this paper, we will refer to the previous three areas from herein as *artificial complexity*.

Artificial complexity research is typified by its bottom-up approach to modelling systems, in which the interactions between components can be as important as the components themselves. The main principles here are: (1) there are often simple rules underlying complex systems; and (2) the complexity of a system is not only due to the complexity of the components but also to the interaction between the system's components.

Although a large number of models and problem solving techniques exist in the literature of complex problem solving, traditional methods typically look at a system from a top-down

perspective, where a problem is solved by iteratively decomposing it into smaller components until each component can be solved using an existing problem solving method. This top-down approach suffers dramatically from ignoring the interaction between the sub-components. For example, consider the collapse of the Advanced Research Projects Agency (ARPA) network in 1980. As stated by (Neumann, 1995):

> The collapse of the network resulted from an unforeseen interaction among three different problems: (1) a hardware failure resulted in bits being dropped in memory; (2) a redundant single-error-detecting code was used for transmission, but not for storage; and (3) the garbage-collection algorithm for removing old messages was not resistant to the simultaneous existence of one message with several different time stamps. This particular combination of circumstances had not arisen previously.

Writing computer programs for artificial complexity research introduces software engineering challenges different from traditional programs. This paper will discuss the differences and similarities between traditional systems and artificial complexity systems. The unique challenges facing developers of artificial complexity models are described; in particular, the difficulty in determining the reliability and correctness of such systems. Finally, preliminary suggestions for applying techniques from the maturing field of software engineering to the development of artificial complexity models are offered.

In Section 2, the modelling process of artificial complexity systems and their characteristics are presented, followed with the software engineering elements for verification and validation in Section 3. Section 4 discusses the role of software engineering in artificial complexity, while Section 5 offers preliminary suggestions for the use of software engineering techniques in such studies. In Section 6, the problems arising from artificial complexity software that represent a challenge to software engineering are discussed, and conclusions are drawn in Section 7.

2. Traditional systems vs artificial complexity systems

Research in artificial complexity largely depends on the use of simulation techniques. Different software environments have been created to establish suitable simulation environments for researchers in artificial complexity.

Webster's Ninth Collegiate Dictionary defines simulation as "the imitative representation of the functioning of one system or process by means of the functioning of another". Simulation does not necessarily need to be a computer program; it can be done by hand or in any other means. However, the use of computer programs makes it feasible to develop complex simulations and to run each simulation as many times as required.

To formally define simulations in artificial complexity, let us denote the system or the phenomena that we are trying to model by Λ_s. To simulate Λ_s, one needs to model it. A model can be seen as an abstraction Γ defined as follows:

$$\Gamma_1 : \Lambda_s \rightarrow \Lambda_m \quad (1)$$

where Λ_m is a model of Λ_s. If we think of Λ_s and Λ_m as two languages, the first is the language of the actual system we are trying to model, and the second is the language used to represent the system, one can define abstraction as a formal mapping between these two languages (Giunchiglia and Walsh, 1992).

Simulation does not work on Λ_s; it does not even see Λ_s. Instead, simulation works on Λ_m. One can see the simulation program as a translation of the model into software, formally,

$$\Gamma_2 : \Lambda_m \rightarrow \Lambda_p \tag{2}$$

where Λ_p is the simulation software.

2.1 The issue of correctness

Once a model for an artificial complexity simulation is developed and programmed, a natural question arises: is the model correct? Here, the software engineering concepts of validation and verification come into play.

Software Engineering (SE) is "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" [1]. *Software validation* is "the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements" [1]. *Software verification* is "(1) the process of evaluating a system component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase, and (2) formal proof of program correctness" [1].

In other words, we can define validation in artificial complexity as whether or not Λ_s is equivalent to Λ_m; that is,

$$\Lambda_s \equiv \Lambda_m \tag{3}$$

With equivalence we mean that Λ_m captured the important aspects to represent Λ_s. However, as previously mentioned, Λ_m is an *abstraction* of Λ_s, so direct equivalence is generally not a useful metric. This is a dilemma in practically all modelling fields, where the modeller is faced with the problem of defining which aspects of Λ_s to include/exclude from the model. Computational modelling of natural phenomena (such as crop growth) often involves constant validation of the model against real-world data, with the model behaviour used to direct further real-world experiments and refine hypotheses. We may *estimate* that a model/hypothesis is a correct abstraction of a real system if its behaviour conforms to real-world observations. In the domain of artificial complexity, modellers face greater validation challenges due to very large system sizes, a lack of data, and the absence of readily testable real-world systems.

In traditional systems, this is not the case. For example, an engineer would almost fully understand a logical circuit that she designed (whether or not it is feasible to validate a model of this logical circuit in a reasonable time is a different question). The theories and tools for understanding and verifying logical circuits exist. In artificial complexity, many of these tools are still in a very early stage, which makes verification and validation difficult.

An interesting question is, if verification is a formal proof, does this proof exist for simulation? In traditional *operations research* (OR), the answer is yes. Take an example from petri-nets; we can easily build a proof to verify the simulation. We can even do simulation by hand. Another example is if we are simulating a production line, once more proofs can be constructed. The definition of simulation introduced earlier by the Webster's Ninth Collegiate Dictionary is consistent with the OR view to simulation. OR sees simulation as a process which imitates the system. Simulation here can be done by hand through observing the actual system. The only reasons we do it in software are (1) it may not be economical to do it on the actual system for time or cost factors, and (2) it may not be technically possible to do it on the actual system, where the actual system does not yet exist (Gdaellenbach, 1994).

[1] IEEE Std 610.12-1990

To be able to simulate a system by hand or simulate a system design, the assumption is that we understand the system's structure. We would have neither designed the system nor simulated it by hand without understanding its structure. In artificial complexity, however, one objective can be to find out how the structure emerges from the interaction between the system's components. In other words, the structure is not pre-determined in artificial complexity. Consider the contrast between a traditional simulation and an artificial complexity simulation within the domain of social networks. A traditional simulation would assume a network structure to simulate the characteristics of interactions occurring within the network. It may even go beyond a fixed structure and define rules of how the structure changes. The network's structure in artificial complexity may not be defined; the agents self-organize and the network structure emerges through the course of the simulation.

In artificial complexity, it is very difficult to see how to validate the model. Should we validate the model against the actual phenomena that we have not understood ourselves as yet; or should we validate the model against our understanding of what the problem is? For example, assume that we wish to simulate the walking behaviour of a special species of army ants. If a specific behaviour emerged from the simulation, should we compare this behaviour against those behaviours occurring in nature or should we compare this behaviour against what we expected it to be? If the behaviour is different from what we expected it to be, it can be because either there is a bug in the software or because our understanding of the problem is still incomplete.

The problems arising in this case can be summarized in the following points:

- If a simulation is replicating the behaviour displayed by the actual system, does this mean that the model is a true model of the system? Take a simple example from the literature of the philosophy of artificial intelligence known as the Chinese room (Searle, 1980). Imagine a computer that stores all possible translations from Chinese to English and vice-versa. Imagine that a bilingual (Chinese/English) human exists. If we test both the computer and the human on a set of sentences, they will perform exactly the same. But this does not mean that the mechanisms used by the computer can be used to explain the mechanisms used by the human even though on the global level, both systems behave similarly. Also, this does not imply that one can use the mechanisms used by the computer to understand the mechanisms used by the human in translating the sentence. To address this issue, we must carefully consider the behaviour of the model at the level of abstraction of interest. Consequently, methods to validate the model are often *ad hoc*.

- If the simulation is replicating the behaviour displayed by the actual system, and we can verify the model is a true model of the system, does this mean that the software is a correct encoding of the model? For example, let us revisit the example of army ants. Assume that the experiments generated a behaviour which replicates a behaviour in nature. This is not sufficient to conclude that the program is correct because this behaviour may have been generated by luck. Therefore, one cannot simply depend on the behaviour of the program alone to determine whether or not the program is correct.

- A key component of artificial complexity systems is emergent behaviour. A characteristic of emergent phenomena in a simulated complex system is that this phenomena is not explainable in terms of the system's components (Rasmussen et al., 2002). Nolfi (1998) states:

  > Global properties are emergent in the sense that, even if they result from nothing else but local interactions among the elements, they cannot be predicted

or inferred from a knowledge of the elements or of the rules by which the elements locally interact, given the high nonlinearity of these interactions.

Because of this nonlinearity, the final state of a system can be reached through different paths. Tracing the cause for why a specific phenomenon emerged, and hence validating the model, is therefore a challenging task.

Overall, simulation of artificial complexity is not as straightforward as simulating a traditional system. In artificial complexity, we are not usually interested in obvious answers. Interesting answers, however, raise a challenging question: what if program errors cause interesting answers to emerge? Moreover, what defines a semantic error in the code if this error resulted in an interesting emerging behaviour? Can we use the errors to refine our model?!

A key feature in artificial complexity software is the emergent behaviour of its constituent parts. Thus, it is impossible to know *a priori* the global, high-level behaviour of the system. On the other hand, the global functionality of traditional systems is understood at the time of implementation. The tools and techniques used to develop, verify and validate traditional systems are subsequently based on the assumption that there exists a complete specification of software behaviour (how often this specification changes is a separate issue). For the science of artificial complexity to benefit from the experience of the software engineering community, its members must be aware of the techniques available, and in what contexts they are appropriately applied.

3. Traditional software engineering elements for verification and validation

Traditionally, before producing software, a set of software specifications are defined. The purpose of *software verification and validation* (SVV) then becomes to check that the software functions in a consistent manner with respect to these specifications.

Software project management is a function which organizes the SVV process. In software project management, activities and their required resources are identified, and a time-plan is created to schedule each of these activities. In software engineering, four main activities can be identified for SVV; these are:

Reviews: A review is defined as "a process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval" [2]. Two formal methods for review are common in practice: technical reviews and audits.

The main objective of technical reviews is to provide evidence that a specific review item, such as a procedure, conforms to the specifications and is consistent with the standards and procedures (and additionally, that any required changes have been made properly). In short, technical reviews ensure that the progress made in the project is consistent with the original plans.

Technical reviews may be conducted as either inspections or walk-throughs. An inspection is undertaken by giving the inspector an item to be reviewed and a checklist of things to confirm that they have been done correctly. In walk-through, the author of the item to be reviewed explains ("walks-through") the item to the reviewers. The aim of both

[2] IEEE Std 610.12-1990

types of review is to detect defects in the review item, and perhaps to identify possible repair mechanisms. Inspections tend to be more successful than walk-throughs at finding defects.

An audit aims at verifying that software products and processes comply with software specifications, requirements, standards, procedures, guidelines, instructions, codes and contractual and licensing requirements. The auditing process is usually undertaken by an independent team other than the software development team. This is to ensure a reliable and unbiased audit process.

Tracing: Tracing is "the act of establishing a relationship between two or more products of the development process; for example, to establish the relationship between a given requirement and the design element that implements that requirement" [2].

Traceability has two types: forward and backward. Forward traceability tests completeness, where traceability matrices are constructed to ensure that each possible input will result in the correct output. Backward traceability works on the inverse problem, where it ensures that each output is associated with an input. Some of the functions included in this activity are to trace user requirements to software requirements which are then traced to component descriptions and vice versa. Traceability is a relationship between development phases, not (usually) within an algorithm. So inputs & outputs here refers to development artefact. In addition, tracing is used in testing, for example to relate different tests such as integration tests to architectural units, unit tests to the modules of the detailed design, system tests to software requirements, and acceptance tests to user requirements and vice-versa.

Formal proofs: Formal proofs are about correctness of the software. Formal proofs ensure that all possible inputs to the software will ensure correct outputs. In practice, this is an extremely expensive process. In addition, it requires a complete representation of the software requirements and designs in mathematical forms. Formal proofs are typically used only in critical domains, such as security systems and safety-critical systems. Automated theorem proving has been successful in hardware verification, and is being used industrially. But otherwise industrial use is not currently widespread.

Testing: A test is "an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" [3]. Testing works on the actual software by executing it. The objective of testing is to find bugs in the software. It therefore requires skills equivalent to, and sometimes more than, that of the software development team.

[3] IEEE Std 610.12-1990

4. Verification and validation in artificial complexity

Just as traditional software verification and validation techniques have been developed within the context of the target software's intended use (performance of a pre-determined set of tasks, see Figure 1), so too must the verification and validation methodologies used in artificial complexity software be developed in the context of their specific goals. One cannot disassociate the verification and validation of artificial complexity simulations from their use.

Stevenson (1999) wrote "Simulations evolve and are subject to long, costly developments. As the understanding of the model become better, the simulation must change to reflect this". Since the purpose of an artificial complexity simulation is to understand the system, new insights are gained when the simulation is run, and these insights often require modifications to the simulation. Hence, the time scale for changing the simulation because of a change in the system is usually shorter in artificial complexity than that found in traditional systems.

As can be seen in Section 2, artificial complexity simulations are an experimental method, iteratively refined and analysed as a tool in a larger endeavour (see Figure 2). Consequently, the notion of 'correct' and 'incorrect' artificial complexity software is less useful than thinking in terms of 'degrees of confidence' in the simulation's reflection of the system being modelled. Rather than being a watered-down version of a 'correct versus incorrect' approach, formalising and revising the degree of confidence in artificial complexity software is inherent in the modelling process. As components of the system are implemented and behave as expected, confidence in the model improves. As confidence in the model improves, discrepancies between the model's behaviour and knowledge of the system being modelled can be used as a cross-check of one's understanding of the modelled system, used to further refine the model, and ultimately to improve theoretical understanding. If such a rigorous experimental procedure is not adhered to, there is the very real danger of simulations merely exhibiting the originally desired behaviour (Kadanoff, 2004).

Before one can use artificial complexity as a tool of experimental science, the modeller must have confidence that the software system's behaviour coincides with the modeller's understanding of what the actual system is doing (i.e., that the modeller has not inadvertently introduced software bugs). Thus, many aspects of the implementation of artificial complexity software lie within the realm of the traditional software development process illustrated in Figure 1.

Rather than performing traditional verification and validation on the global behaviour of artificial complexity software, researchers can use these proven techniques on the well-understood components of their implementations, formalising a level of confidence in the software interpretation, and then using that confidence level in the context of the experimental method. As it is often only the emergent, global behaviour of the artificial complexity simulation that is not well-understood, this means that much of the software implementation can be verified and validated.

5. Applying software engineering to artificial complexity

The field of artificial complexity is still in the early stages of incorporating software engineering techniques into its modelling processes. What follows is a preliminary list of strategies intended to serve as a starting point for discussing the use of software engineering techniques in artificial complexity studies. Some strategies may prove more useful than others in any given situation, but modellers should be aware of them.

Unit tests: Unit tests aim at testing a specific module or group of modules against some detailed design. In general, it should be easy to apply this type of testing for almost any

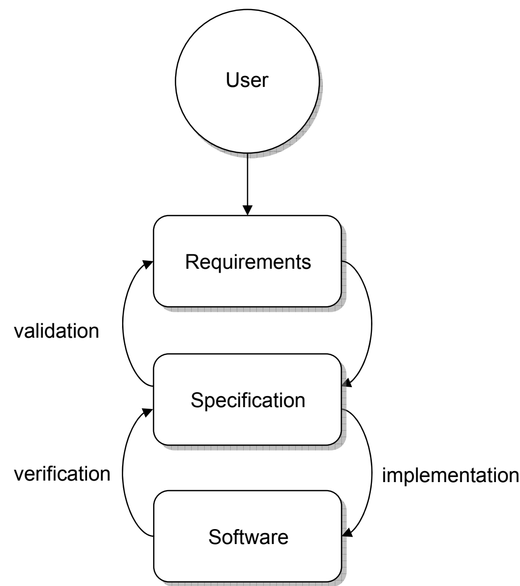

Figure 1. *Verification and validation in traditional software development.*

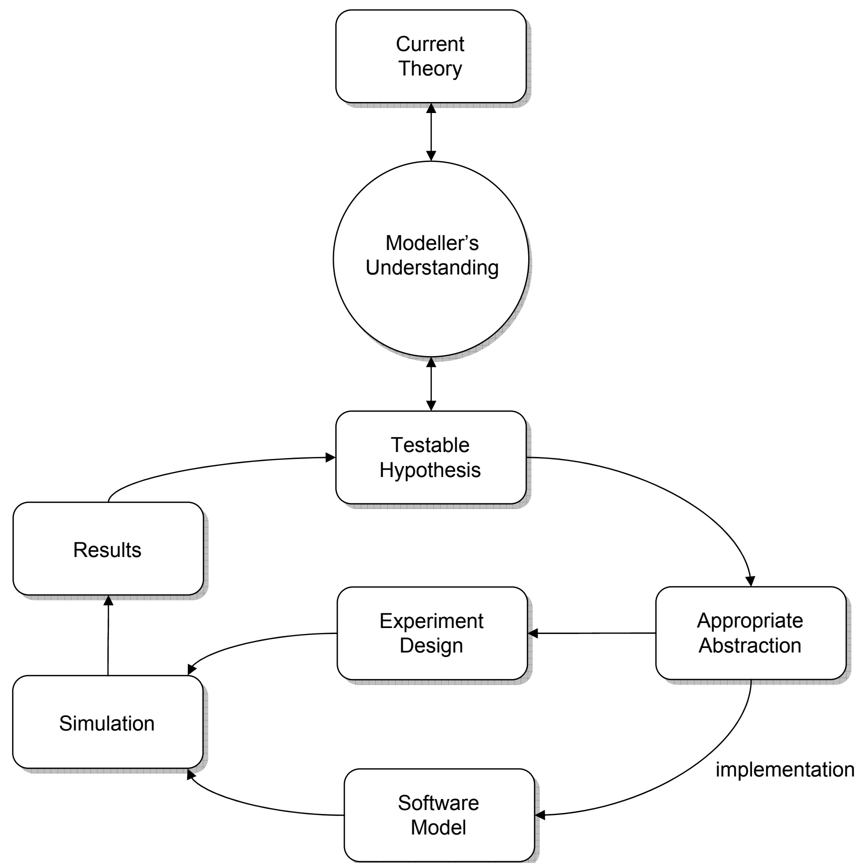

Figure 2. *The dynamic interaction between modeller and modelling software in a typical artificial complexity investigation. When contrasted with Figure 1, it can be seen that the role that software takes in this setting is much more dynamic. This has important implications for verification and validation.*

code, including that written for artificial complexity. The program components are usually well-defined and their specific goals should be known for artificial complexity applications.

Integration tests: Integration tests aim at testing an integrated subset of subsystems. Each subsystem may contain one or more units. Here, each unit is tested against its architectural design unit and all architectural design units are integrated to form the overall system. This type of testing is a challenge for some artificial complexity applications. Since we may not even know what output we should expect, it becomes difficult to define how to perform integration tests. In some artificial complexity applications, architectural design is fairly standard (such as when using neural networks). However, in other applications (such as agent swarms or self-organized social networks), the architectural design can emerge during the run. One needs to differentiate here between a straightforward simulation of a swarm of agents and a highly dynamical feedback system, where the architecture can change during the run. For example, we may not know in advance that agent A_{c1} defined by class $c1$ needs to communicate its results to agent B_{c2} defined by class $c2$. The closest example here is event-based simulations, where a piece of code is triggered by an event. In a swarm of agents, the events may not be well-defined before execution time. The events may even be defined but determining which agent will be triggered can result in a combinatorial problem if one wishes to consider all possible cases during the design.

System tests: System tests aim at comparing the software system and the software requirements. System tests can be seen as to verify that Λ_p is equivalent to Λ_m. While modellers are expected to know enough about their system to be able to check at least some of its aspects, the expected behaviour of Λ_m is usually unknown which makes this type of testing a real challenge for artificial complexity systems. Consider for example the use of agent-based distillation to simulate a conflict. Every time we run the system, different behaviours occur. Nonlinearity plays a role here, and different initial conditions can generate the same behaviour. However, as for acceptance tests (below), the behaviour of the software system is often known for particular sets of parameters, and increased confidence can be gained by replicating known behaviour within these parameter constraints.

Acceptance tests: Acceptance tests are carried out by the user to ensure that the system meets the user requirements. Acceptance tests can be seen as verifying that Λ_p is equivalent to Λ_s. It can be difficult to see how the user can globally test the software system when the simulation is being used to understand the phenomenon being modelled. However, in the general case of investigating a specific hypothesis, the global behaviour of the system should be known for a given set of parameters. Consequently, confidence in the model implementation can be increased by testing within the constraints of such parameters.

Regression testing: Last and not least, during the evolution of the software, retesting is required. A change in the software may introduce bugs that did not exist before. Indeed, unit, integration, system, and acceptance tests are needed after each change in the software. This raises the importance of a distinct type of testing called regression testing. Regression testing is "selective retesting of a system or component, to verify that modifications have not caused unintended effects, and that the system or component still complies with its specified requirements" [4]. Regression testing is needed after each software modification.

[4] IEEE Std 610.12-1990

Apart from the testing methodologies described above, other tools and techniques from the software engineering community may prove useful, in a practical sense, to the artificial complexity community. Unfortunately, rumour and unjustified biases exist that often steer the practising modeller away from some of them. Currently available techniques that can contribute towards the minimization of software development errors in artificial complexity include:

Use of existing testing frameworks: A range of free testing frameworks exist for most modern programming languages. These can simplify and standardise the implementation of testing methodologies. Examples include UnitTest (Python), NUnit (.Net framework), JUnit (Java), etc..

Use of existing library functions: When taken from a reliable source, software libraries provide functionality that is developed and tested by experts across a wide range of parameters (e.g., random number generators), eliminates a possible source of error, and standardises given functionality between simulations. In addition, modern libraries are often highly optimized for performance.

Use of higher-level languages: Where possible, the use of higher-level languages should be encouraged. Gone are the days of "it needs to be fast, I must implement it in C". For standard programming situations, modern garbage collection can be *more* efficient than manually handling memory allocation (e.g., modern garbage collectors keep allocated memory in a defragmented state, resulting in more efficient memory allocation than calls to malloc(), which must search memory for a large enough block (Schanzer, 2001)). Also, optimizing compilers and interpreters often generate more efficient machine code than the casual programmer, so efficiency is now less of an issue. More importantly, a significant proportion of software bugs are introduced in the translation step from human understanding to language implementation, rather than the programmer misunderstanding the requirements. As a trivial example, consider the following algorithm implemented in two popular programming languages:

C version:

```
int i;
for (i = 0; i < 10; i++)
{
     int j;
     for (j = 0; j < 15; i++)
          f(matrix[i][j]);
}
```

Python version:

```
for row in matrix:
     for item in row:
          f(item)
```

Note the extra overhead and the bug introduced on the 5th line of the C version. It is worth noting that it can often take a scientist far longer to find an obscure bug (if it is identified at all) than the extra computing time required by modern higher-level languages. Coupled with the fact that the sharing of source code is increasingly being encouraged by scholarly journals, the use of more human-readable languages has the potential to significantly increase the productivity and reliability of artificial complexity modellers. (As a side note,

all popular high-level languages such as Python, C# and Java allow the native implementation of resource-intensive components, and are portable across most major operating systems).

Version control: As discussed in Section 4, artificial complexity software exists as a constantly-evolving experimental tool. This can lead to multiple versions of the software. Traditional software development has also had to deal with the issue of version control; that is, how to keep track of each version of the software. This technique is not just a history of changes or a software engineering quality control requirement – it documents how our understanding of the system changes. It can provide the basis for a logical argument of how our current understanding of a scientific phenomenon emerged over time. This history can be represented as a cause and effect graph.

As an example, assume a situation where we model population growth. We first simulate the growth process using a clock, every time the clock clicks, all cells are replicated. After the first version of this software, we ran into memory problems. We then realize that the computer's finite memory can be seen as an analogue for limited resources in real life. This triggers the idea that we need a mechanism for the individuals to compete for survival based on how much memory they occupy. Once we simulate this, we find large agents get eliminated all the time. This triggers another idea of differentiating between the individuals using conflicting objectives, giving a large agent an advantage for survival.

Seeing the versions of the software in the previous trivial example, one can notice that each version is reflecting some understanding of the problem, and the sequence of the versions reflect how our final model evolved and the logic behind it. Therefore, the use of versioning software (e.g., CVS, Subversion), which allow revisions to be efficiently stored and changes in simulation behaviour to be rolled back or branched off, should be encouraged.

6. Discussion

Overall, it is clear that without specifications, some activities, such as reviews, of the software engineering process cannot be applied to the system as a whole. However, standards and specifications can be developed for specific sub-components of the simulation, providing increasing confidence that the software's behaviour is an accurate reflection of the system being modelled.

Unfortunately, there are almost no specifications written and no standards have been identified in the literature. Researchers normally write their own software. Even when public-domain software is used, such as SWARM (Stefansson, 1997), specifications are yet to be taken as a serious problem, although many researchers complain about the lack of documentation and specifications. It is clear that the software engineering task becomes harder if specifications do not exist.

These specifications are needed not only for the simulation experiments developed, but also for the software environment used. For example, a system such as SWARM needs to be verified and validated against specifications. A simple fault in such a system would make the community question many of the scientific findings that are based on the use of such frameworks. This highlights the danger and significance of ignoring software engineering techniques in the artificial complexity community.

A critical issue in the development of artificial complexity software is human experts. Traditional software engineering has been an active field of research for a couple of decades. During this time, a deeper understanding of traditional systems has emerged and expertise has been

developed in such systems. Artificial complexity is a new area of research, and this paper has illustrated how the nature of its software development problems vary from traditional challenges. Therefore, more time and increased effort is needed for software engineering expertise to emerge in the artificial complexity area.

7. Conclusions

Software simulation provides the means to investigate certain phenomena in ways that would otherwise be impossible (at least within a reasonable length of time). Benefits including a controlled environment and the capacity for fine-grained analysis are just some of the many reasons artificial complexity research relies on software simulation and, consequently, software development.

This paper is an attempt to identify some of the issues surrounding artificial complexity software development, and to highlight a collection of preliminary tools and techniques to address them. We identified some of the similarities and differences between traditional systems and artificial complexity systems, and also some of the distinctions between the software engineering process and software development in the context of artificial complexity research. Our aim is to establish a framework for ensuring that the software developed in the artificial complexity field is correct and reliable.

Artificial complexity simulations are being increasingly called upon to drive experimental and theoretical studies, and are beginning to form the basis of many (e.g., political, economic) policy decisions. Yet, the software development processes of artificial complexity simulations are not nearly as mature as that for traditional software engineering. For artificial complexity simulations to be taken seriously by the other engineering and scientific fields, we must develop and adhere to rigorous standards, procedures and techniques in both the development and use of software simulation.

Acknowledgements

This study was supported by the Australian Research Council's Centre for Complex Systems.

References

Gdaellenbach, H. (1994). *Systems and decision making: a management science approach*. John Wiley and Sony.

Giunchiglia, F. and Walsh, T. (1992). A theory of abstraction. *Artificial Intelligence*, 56(2):323–390.

Kadanoff, L. (2004). Excellence in computer simulation. *Computing in Science and Engineering*, 6(2):57–67.

Neumann, P. (1995). *Computer related risks*. ACM Press, Addison-Wesley.

Nolfi, S. (1998). Evolutionary robotics: Exploiting the full power of selforganization. *Connection Science*, 10:167–183.

Rasmussen, S., Baas, N., Mayer, B., Nilsson, M., and Olesen, M. (2002). Ansatz for dynamical hierarchies. *Artificial Life*, 7:329–353.

Schanzer, E. (2001). Performance considerations for run-time technologies in the .NET framework. *MSDN Library*, August.

Searle, J. (1980). Minds, brains, and programs. *The Behavioral and Brain Sciences*, 3:417–424.

Stefansson, B. (1997). Swarm: An object oriented simulation platform applied to markets and organizations. In Angeline, P., Reynolds, R., McDonnell, J., and Eberhart, R., editors, *Evolutionary Programming VI, LNCS1213*, volume 1213. Springer-Verlag.

Stevenson, D. (1999). A critical look at quality in large scale simulations. *Computing in Science and Engineering*, pages 53–63.