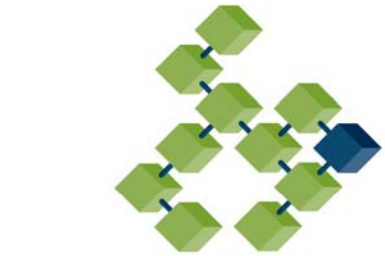

ARC CENTRE FOR
COMPLEX SYSTEMS

Technical Report

ACCS-TR-04-02

Practical Software Engineering Techniques for Regulatory Models in Biology

James Watson, Jim Hanan and Janet Wiles

December 2004

ARC Centre for Complex Systems
School of ITEE, The University of Queensland
St Lucia Qld 4072 Australia
T +61 7 3365 1003
F +61 7 3365 1533
E admin@accs.edu.au
W www.accs.edu.au

Practical Software Engineering Techniques for Regulatory Models in Biology

James Watson, Jim Hanan and Janet Wiles
ARC Centre for Complex Systems
The University of Queensland

Email: jwatson@itee.uq.edu.au

Executive Summary

Findings

This report has focused on verification and validation techniques that are suited to GRN research methods and are of immediate, practical benefit to the computational modelers of the ACCS community. Through an online survey and focused interviews, this project identified the following key characteristics of GRN modeling:

1. **Small team size:** Simulations are implemented and used by a very small number of people, often a single developer is the single user.

2. **Transient systems:** Software models are developed to help understand a particular research question, and are often redesigned or even discarded as the question / understanding changes.

3. **Rapidly evolving specifications:** Simulations are developed and used in an iterative situation of rapid specification change, with specifications often changing as quickly as the software is run.

4. **Non-linear:** The precise behaviour of the complete simulation is unknown before runtime, with emergent interactions of system components providing the observations of interest.

Recommendations

The rapidly evolving requirements coupled with small team sizes (points 1-3 above) conspire to make many popular software engineering techniques impractical. The following recommendations distil the most useful techniques for such modeling:

A Maintenance of static components: Although the specifications of such software systems undergo rapid change, many components remain

the same, and their expected behaviour is precisely known. Existing techniques that aid development, testing, and maintenance at the component level should be immediately incorporated into current modeling processes. These are:

1. *Design notations:* Notations that aid in the design of components and their relationships should be used as both a convenient means of design documentation and a method to help eliminate design inconsistencies. See Section 4.1.1.
2. *Version control:* Version control software, which provides a convenient method of storing different versions of files, should be used to provide documentation of historical changes and prevent data loss. See Section 4.1.2.
3. *Unit testing:* Unit testing, or the testing of individual software components, should be employed to ensure the components behave as specified under a range of conditions. See Section 4.1.3.
4. *Automatic document generation:* The automatic document generation capabilities of many modern programming languages should be used where available. See Section 4.1.4.

B Manually tracking component interactions: For simulations with a *small* number of critical components (e.g., simulations of biological networks with five genes), methods to manually track object interactions should be employed to understand expected runtime behaviour. See Section 4.2.

C Understanding emergent interactions through visualization: Visualizations that show the relationship between micro level interactions and macro level behaviours provide some of the most powerful methods for understanding the system-wide dynamics of these GRN models. Visualization can aid understanding of micro and macro level interactions at three levels (see Section 4.3):

1. *Structure (or network) of interactions.* Examples include network diagrams and distribution plots.
2. *Dynamics of interactions.* Examples include gene expression diagrams and state space diagrams.
3. *Function (or behaviour) of interactions.* Examples include L-systems and Niklas phenotypes.

Opportunities for ACCS outreach are tutorials, workshops, and online resources illustrating the benefits of these recommended techniques, targeting biological modelers.

Contents

Acknowledgments

The authors would like to thank the many people who gave a significant amount of time in discussing their modeling process. In particular, thanks must go to Dr David Thornby, who provided invaluable insight into the computational modeling process from a botanist's perspective. Also to Elizabeth Dunn, who likewise provided both the details of her model and a biologist's view of the process. Thanks must also go to Nic Geard, Kai Willadsen, Jennifer Hallinan, Daniel Bradley, John Hawkins, Stefan Maetschke, Mikael Bodén, Marcus Gallagher, Naveen Kumar, Bo Yuan and Jian Xiong Wang for participating in a general discussion of modeling issues and/or feedback. Finally, thanks goes to all the anonymous survey respondents who generously shared the details of their modeling processes and various perspectives of software engineering. The project was funded by the ARC Centre for Complex Systems.

Section 1

Introduction

> "The intricacy and variety of biological signalling networks often defy analyses based on intuition. System properties are often dependent on subtle timing relations and competition between negative and positive regulators. Given the wealth of biochemical data, a computational analysis is well suited to handling both the complexity of multiple signalling interactions and the fine quantitative details. .....models such as these should not be considered as definitive descriptions of networks within the cell, but rather as one approach that allows us to understand the capabilities of complex systems and devise experiments to test these capabilities." (Bhaller and Iyengar, 1999, page 386).

Recent advances in biology have offered unprecedented insights into the details of phenomena such as gene regulation. This ever-increasing body of knowledge has prompted calls for the integration of this detailed information into a more holistic understanding in order for progress to continue (Galis, 2003). For this to occur, disciplines that operate at different levels of abstraction (e.g., the molecular details of specific regulating genes versus theory of Boolean networks) must pool their knowledge through collaborative ventures.

These recent biological advances have coincided with a significant rise in available computing power. Theoretical experiments and computational simulations that were once impossible to study in a feasible amount of time can now be utilized in most scientific disciplines, providing a framework through which new understanding and collaborative ventures can emerge. The field of Complex Systems, which studies the macro-level behaviours which emerge from systems composed of well-understood, interacting components, has particularly benefited from this readily available computing power. Investigating gene regulation as networks of emergent interactions is one such avenue of

scientific endeavour.

Analyses of potential wide-scale effects in genetic regulatory network (GRN) models address what dynamics, in principle, are possible in large systems of interconnected switches (Bornholdt, 2001). Such theoretical studies, which are at present practically impossible in experimental science, have become increasingly important as evidence gathers of the relatively small number of genes that make complex organisms, and the significance of their interactions. Rather than replacing experimental studies, models that allow analysis of the emergent properties of biological phenomena can provide the pre-experimental means to drive experiments, the ability to look inside a running model, the combination of smaller-scale models to observe hypothetical global behaviour, and demonstration that a process is *sufficient* to generate an observed phenomena (Johnson et al., 2004). An overview of the various approaches to modeling genetic regulatory networks in this context (in the form of an ACCS technical report) can be found at:

`http://www.itee.uq.edu.au/~nic/_accs-grn/index.html`

"It is unlikely that humans will ever write software with zero defects." (Aho, 2004, page 1333). A significant proportion of these models are implemented as computer software, whose emergent behaviour makes verification and validation of correctness difficult. By relying on software-generated results, the success of scientific computing rests heavily on a solid foundation of good software design and development practices. Developing confidence in such software can be difficult, since the specific behaviour of the system is often not known beforehand. The increasing use of computer software in research has come with an increasing concern with the quality of simulation software.

There are many techniques and much formal literature discussing the use of software simulations and the appropriate reporting of their results (see, for example, Barzel (1992); Sargent (1988)). The application of software engineering knowledge to large systems exhibiting emergent behaviour forms an area of active research. However, many software models developed within the academic community (such as the GRN models within the ACCS) are often done by single researchers working on a very specific problem. The needs of developers of scientific software at this scale have not been adequately addressed. Typical software engineering methodologies are geared towards large projects which generally include teams of developers, testers, managers, and users. Many of the issues addressed by these methodologies do not apply to researchers within the academic community. Furthermore, many practising academic modelers have not been exposed to current programming tools and techniques that can increase confidence in software developed.

This project seeks to address this issue by identifying how software is used in the research activities of a sample of current academic modelers, and by providing a collection of tools and techniques that address the needs of these modelers. While many advanced techniques exist to increase confidence in simulation software, this project focuses on tools and methods that can be quickly and easily incorporated into a research program without assuming a computer science background.

Section 2 provides an overview of software engineering in computational modeling. Section 3 summarizes the results of surveys and interviews aimed at eliciting the software development requirements of practising modelers, while Section 4 proposes a collection of tools and techniques that address these requirements. Finally, concluding remarks are offered in Section 5.

Section 2

Software Engineering and Computational Modeling

Software implementations are not the same as mathematical formalisms. It may surprise some non-computer scientists that programming languages are often not as well defined as mathematics (Hatton, 1997). Source code that does one thing on one computing platform may do something entirely (or often worse, subtly) different on another.

The goal of verification and validation is to gain credibility, or confidence, in a model, and techniques to do this have been developed (Carson, 2002). Since models are the same as a hypothesis, they cannot be absolutely validated, only invalidated (see Guergachi and Patry (2003)). Sargent (1988) describes the steps of the model validation process and a set of validation techniques, which can be used subjectively or objectively (where objectively means using some type of statistical test or procedure). In addition, there is a wealth of literature focusing on the use of software engineering techniques in the development of emergent, agent-based systems (Weiß, 2001; Zambonelli and Omicini, 2004).

Multi-agent systems have proven useful in the simulation of biological networks. Khan et al. (2003) describe a molecular species modeled as an individual agent with hierarchical task network structures to represent self- and externally-initiated reactions. Johnson et al. (2004) point to software engineering techniques that have been found to be useful in modeling intracellular processes using object-oriented programming methods, which include object-oriented decomposition, UML as a notation system, class hierarchies, software patterns and component testing.

While these studies provide insights into the software development requirements of some biological models, it is unclear how well they apply to the specific modeling practises typical of researchers within the ACCS com-

munity. In order to determine these software engineering requirements, we undertook three case studies and a general survey of practising modelers, with a focus on those using emergent computational simulations to understand regulatory systems.

Section 3

Case Studies

3.1 Current Modeling Practices

Two initial approaches were taken to understand the software development background and practises of current complex systems modelers. The first was a survey of the modeling community. The second was a discussion held with researchers and students in the Complex and Intelligent Systems research group at the University of Queensland.

3.1.1 General Survey

Anonymous survey responses were solicited from members of the ARC Centre for Complex Systems, the Festival of Doubt group (an eclectic collection of researchers loosely based at the University of Queensland; see `http://festivalofdoubt.uq.edu.au/`), the University of Queensland's Computational Biology group, and the Connectionists mailing list (which includes an international assortment of researchers involved in neural computation). The survey is included as Appendix A.

The survey revealed the varied background of researchers using computational modeling (from chemical engineering to human factors), and the wide range of computational tools and methodologies currently in use. Of the nineteen respondents, seventeen reported using software simulations, with models including machine learning algorithms, evolutionary computation, L-systems, networked systems, multi-agent systems, real-time finite element modeling and Monte Carlo simulations (among others). The majority (82%) of respondents had at least a Bachelor's degree in computer science.

The variance in types of software simulations reported was mirrored in the range of hardware (Figure 3.1), operating systems (Figure 3.2), development environments and programming languages (Figure 3.3) used by the

respondents. These results indicate that recommendations for appropriate software engineering techniques should not focus on any particular programming platform or development environment.

Print statements were used in debugging by 82% of those surveyed, while 65% utilized language debuggers. The use of version control is summarized in Figure 3.4. Respondent's awareness of a collection of common software engineering methods, and their deployment in their own software development programs, is summarized in Figure 3.5. Just under a third of respondents reported using a formal software development process, but most followed a self-imposed simulation procedure (e.g., documentation of design, informal specification of software functionality, implementation and testing of components, checks against analytic solutions or published results, repeat).

Interestingly, the majority of respondents reported a development team size of one (65%), with no team sizes greater than four mentioned. User base sizes were also quite small, with six respondents reporting a user base of one (generally themselves), and the same number reporting a user base of two or three people. No user bases greater than six were reported.

An interesting perception was elicited from this survey. 65% of respondents thought that an increased use of formal processes would result in increased simulation quality, but only 29% thought that formal processes would warrant the extra time they would require. This stems from the general feeling that an increased use of software engineering techniques would not change the results of simulations, but would instead result in improved source code (as in a better simulation architecture, more reusable and efficient code, shorter development times, etc.). Some respondents expressed a concern that software engineering techniques were only suited to larger-scale models, or that they may not be appropriate in open-ended discovery situations. However, interest was expressed in learning how such methods could be applied to simulation scenarios. The time respondents would be willing to spend learning appropriate software engineering techniques is summarized in Figure 3.6.

3.1.2 General Discussion

To obtain a more in-depth overview of the general modeling practices of computational researchers, a discussion was held with members of the Complex and Intelligent Systems group at the University of Queensland. Members of this group fell into two camps: primarily Matlab users, and users of general purpose programming languages. Models were generally reported as a collection of components. The point was raised that the use of different frameworks shift the focus of effort onto different modeling aspects.

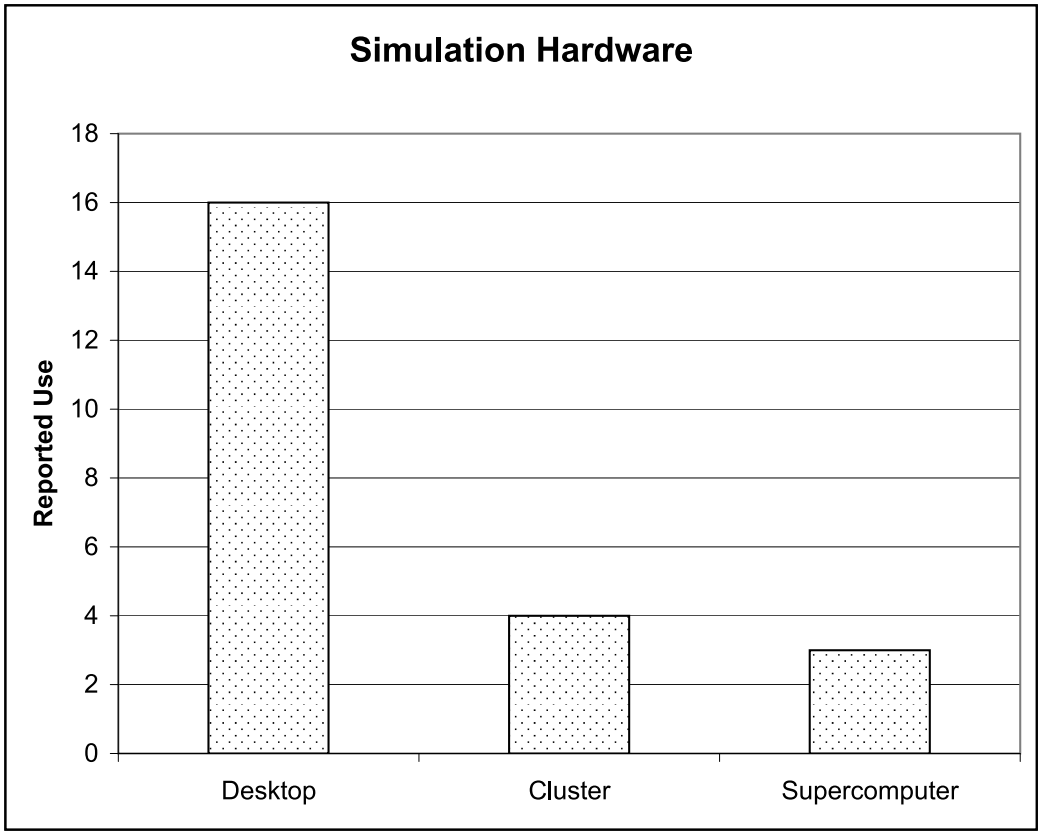

Figure 3.1: Hardware used to run the simulations reported by survey respondents.

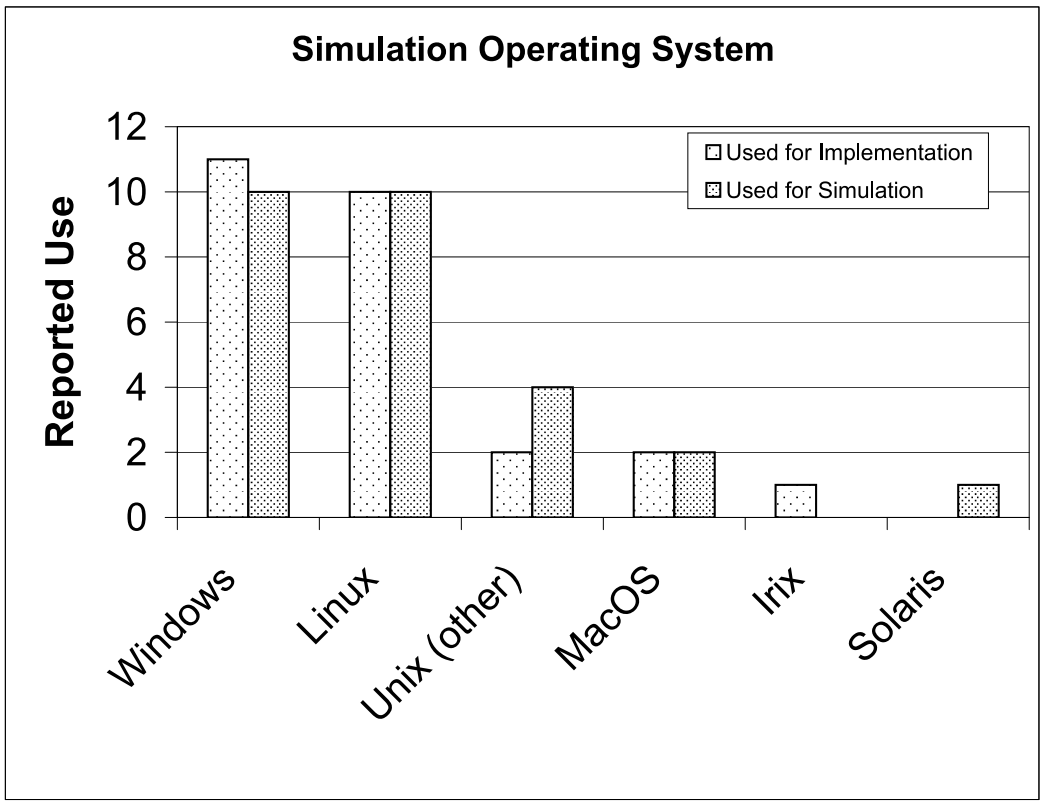

Figure 3.2: Operating systems used to implement and run the simulations reported by survey respondents.

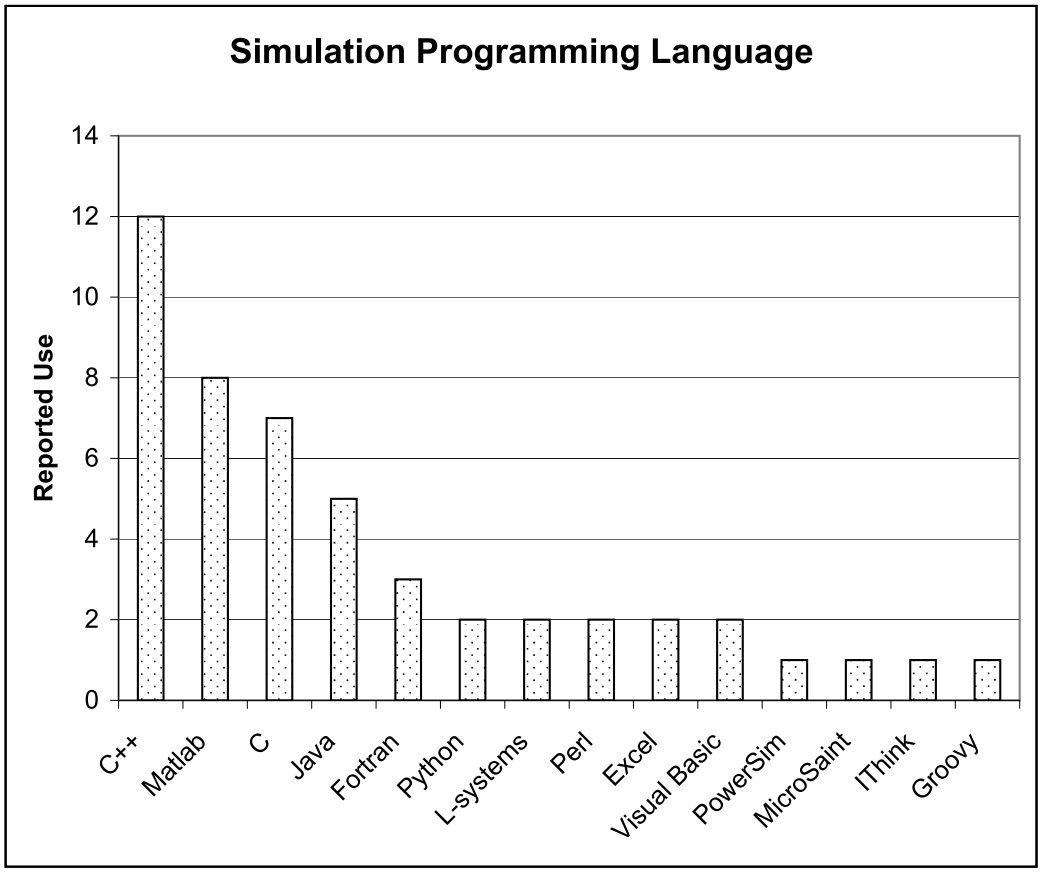

Figure 3.3: Programming languages used to implement the simulations reported by survey respondents.

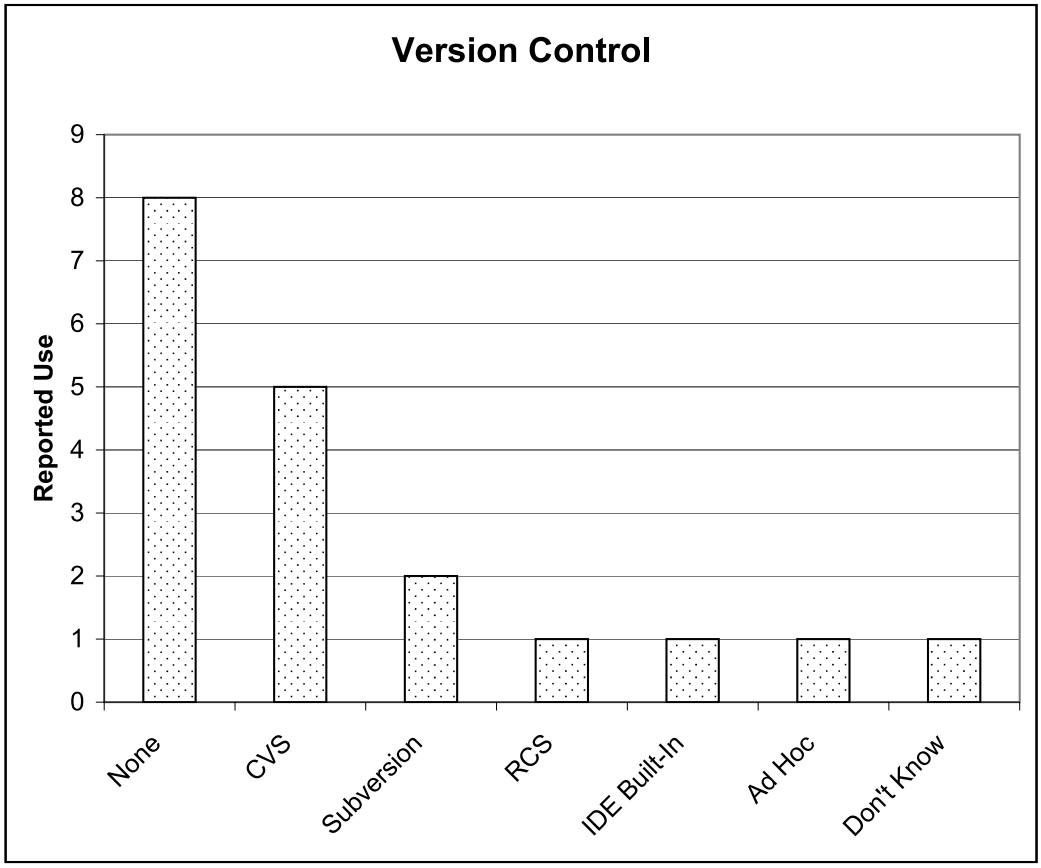

Figure 3.4: The use of version control reported by survey respondents.

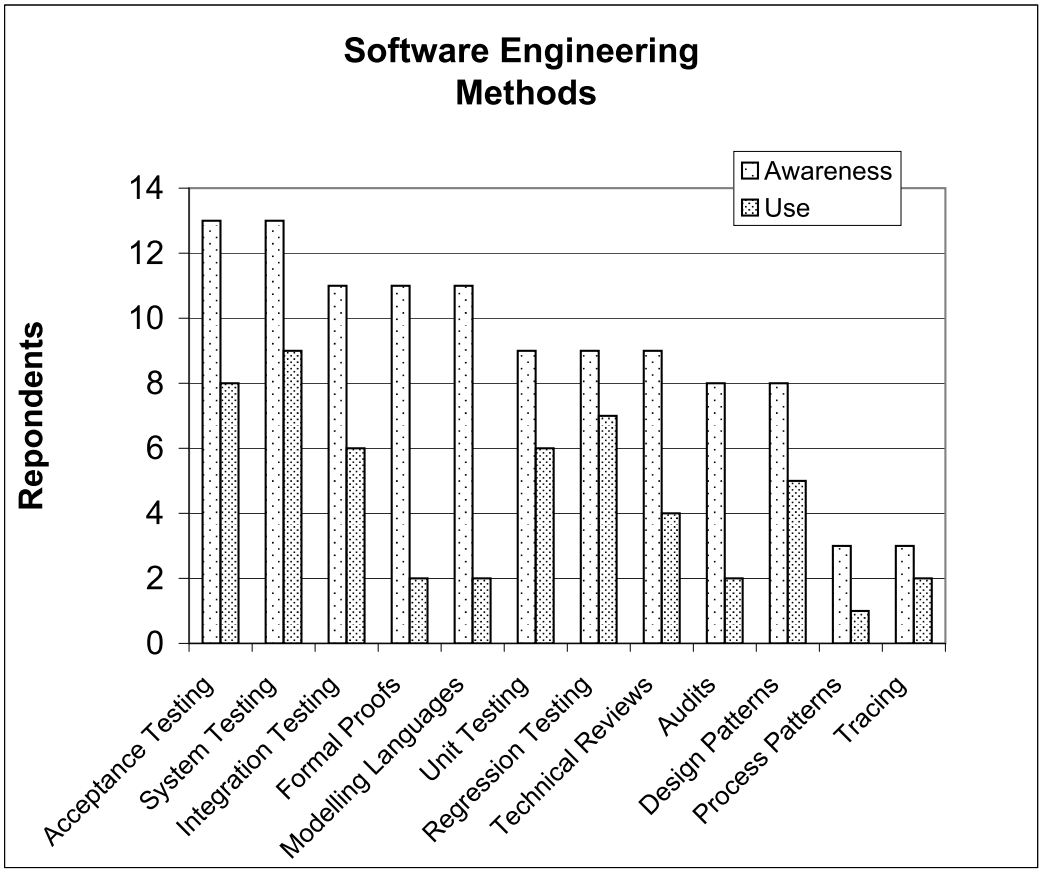

Figure 3.5: Awareness and deployment of common software engineering methods reported by survey respondents.

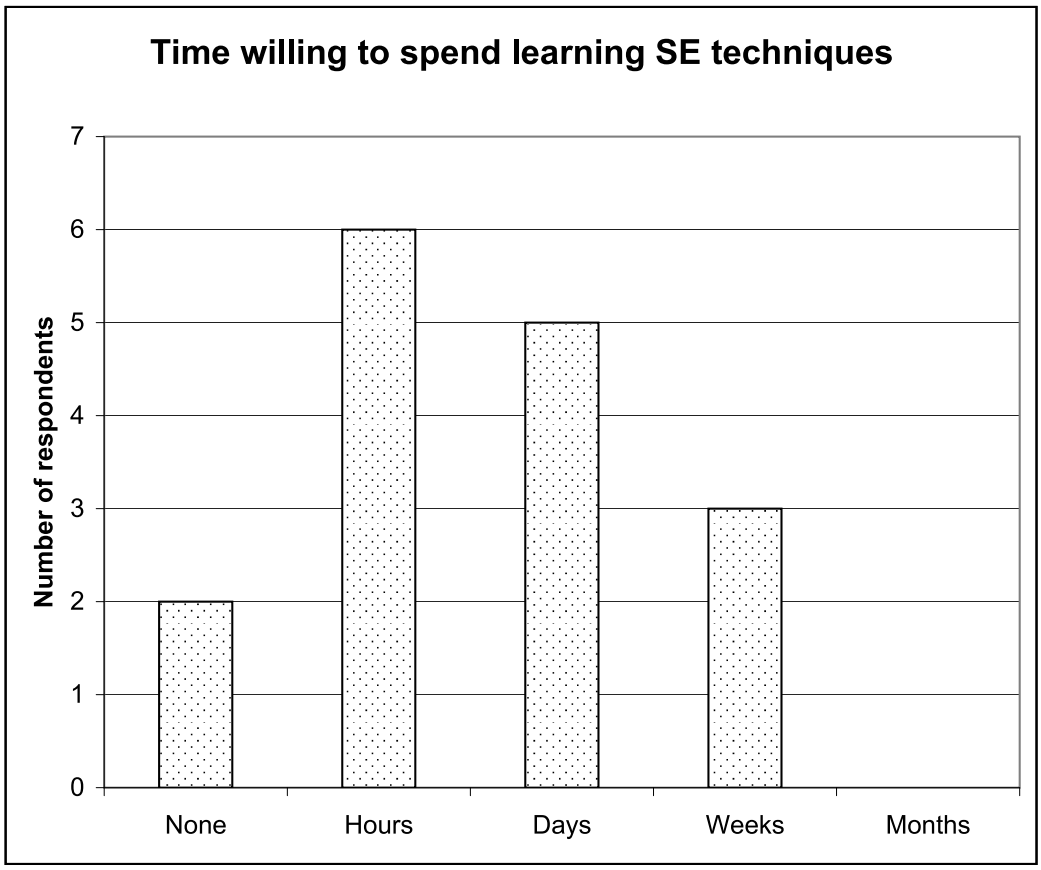

Figure 3.6: Amount of time survey respondents would be willing to spend learning appropriate software engineering techniques. Note that of the two respondents who reported no time, one already teaches in the area, and the other would not learn the techniques but would direct programmers to appropriate methods.

Simulations implemented using general purpose programming languages were generally in the order of thousands of lines of code, with the software programmed in a bottom-up fashion (i.e., lower-level components implemented, then integrated into higher-level modules, with this process eventually resulting in a complete simulation). Many of the software components developed were intended for reuse in other models and/or scenarios.

The group was asked to recommend techniques that would help increase confidence in software correctness, with an emphasis on practical methods for use in the research situations of modelers in the group. The suggested techniques were:

- Unit tests
- Design by contract
- Code sharing
- Code review

Interestingly, even within this closely related group of researchers, terms such as 'model' meant different things to different people, highlighting the need to carefully define what is being addressed before useful discussions can be achieved.

3.2 Case Study 1: The Artificial Genome

A particular genetic regulatory network model which has proven popular in macro-level studies of GRN dynamics is the Artificial Genome (Reil, 1999). The model is summarized in Figure 3.7.

The Artificial Genome (AG) has proven to be quite versatile. Not only has it been used to analyse potential network effects of sequence-level mutations (Watson et al., 2004), it has been extended to include the regulation of small RNA molecules (Geard and Wiles, 2003), to incorporate finer-grained inhibition and asynchronous updating (Hallinan and Wiles, 2004a,b), added enhancer and inhibitor sites, and 'proteins' that attach and detach from the sequence (Banzhaf, 2003, 2004). Willadsen and Wiles (2003) discussed how variations in the connectivity and degree of inhibition of AG networks affect the behaviour of the networks. The AG has also been used as the basis of GRN models used in an evolutionary context. Bongard and Pfeifer (2001); Bongard (2002) utilize an artificial regulatory network similar to Reil's model to grow agents that are evaluated for fitness in a virtual environment. The

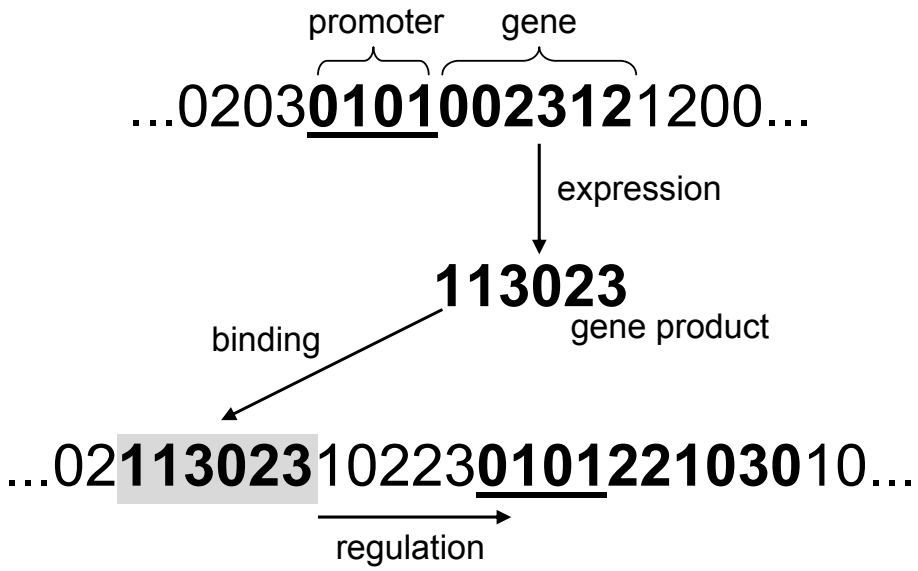

Figure 3.7: The Artificial Genome model. Sequences consisting of the 4 base values 0,1,2,3 are searched for the promoter sequence '0101'. A fixed number of values after the promoter sequence are defined as genes. A gene is expressed by incrementing each element by 1, modulo the number of bases (4). If the gene product binds to a matching sequence in the genome, its gene regulates the next gene downstream of the binding site.

AG has been used to integrate a genetic sequence with a developmental phenotype (Watson et al., 2003), and as the basis for a GRN model that controls a group of robots, with the controller evolved with a genetic algorithm (Taylor, 2004).

3.2.1 Artificial Genome Software Development

Two University of Queensland postgraduate students who use models based on the Artificial Genome were interviewed to determine their modeling process and the software development requirements of their models. Both models were implemented using the C++ programming language and the Boost libraries. This choice was made due to the range of portable software libraries available for basic functionality (such as random number generation), speed of code execution, the versatility of the language, and the background training of the modelers. The primary development environment consisted of the general Unix build tools (e.g., text editor, g++ compiler, etc.). One of the models incorporated the Python language for visualization, the other used Perl to process raw simulation output. Both models were in the order of one thousand lines of source code. The models were comprised of an object-oriented hierarchy of components, implemented by a single developer. In one model, unit testing (using the Boost unit testing libraries) was used. Results were written to standard output, and visualizations (e.g., of expression patterns: see Figures 3.8 and 3.9; and of network weights: see Figure 3.10) were used to monitor software behaviour. While the simulation code changed over time, many of the lower-level components remained the same.

3.3 Case Study 2: Compensatory Growth

The use of a computational model to study the morphological aspects of plant growth in response to damage formed the second case study of biological modeling. The simulation was used to investigate the phenomenon of compensation for defoliation in cotton plants by modeling the interactions between various plant components and the environment (Thornby et al., 2003).

3.3.1 Software Development Requirements

This model was implemented entirely in the L-studio front-end to CPFG (Prusinkiewicz et al., 2000), primarily by a single developer (a botanist). It is comprised of a set of components, implemented in approximately five hundred lines of code, and was developed in a bottom-up style of first implementing

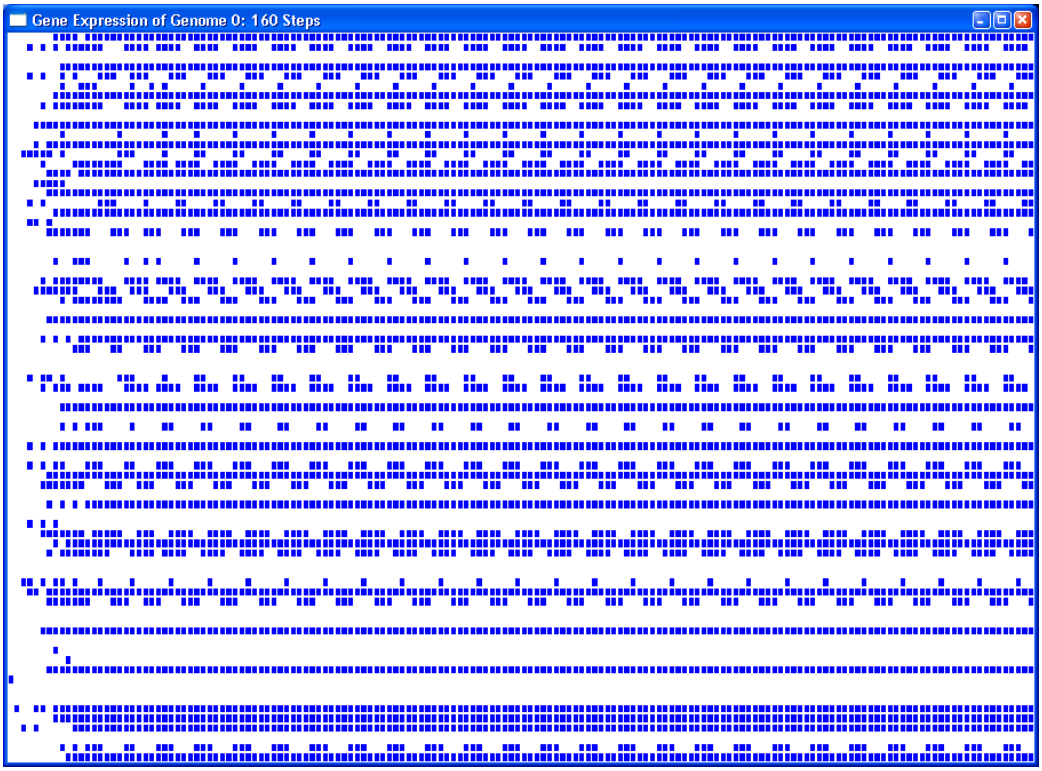

Figure 3.8: Example expression pattern of an Artificial Genome network. Time moves from left to right, with each gene identified in the sequence taking a position along the y-axis. At each time-step, activated genes are represented as coloured boxes. This visualization allows the interactions between genes to be viewed over time.

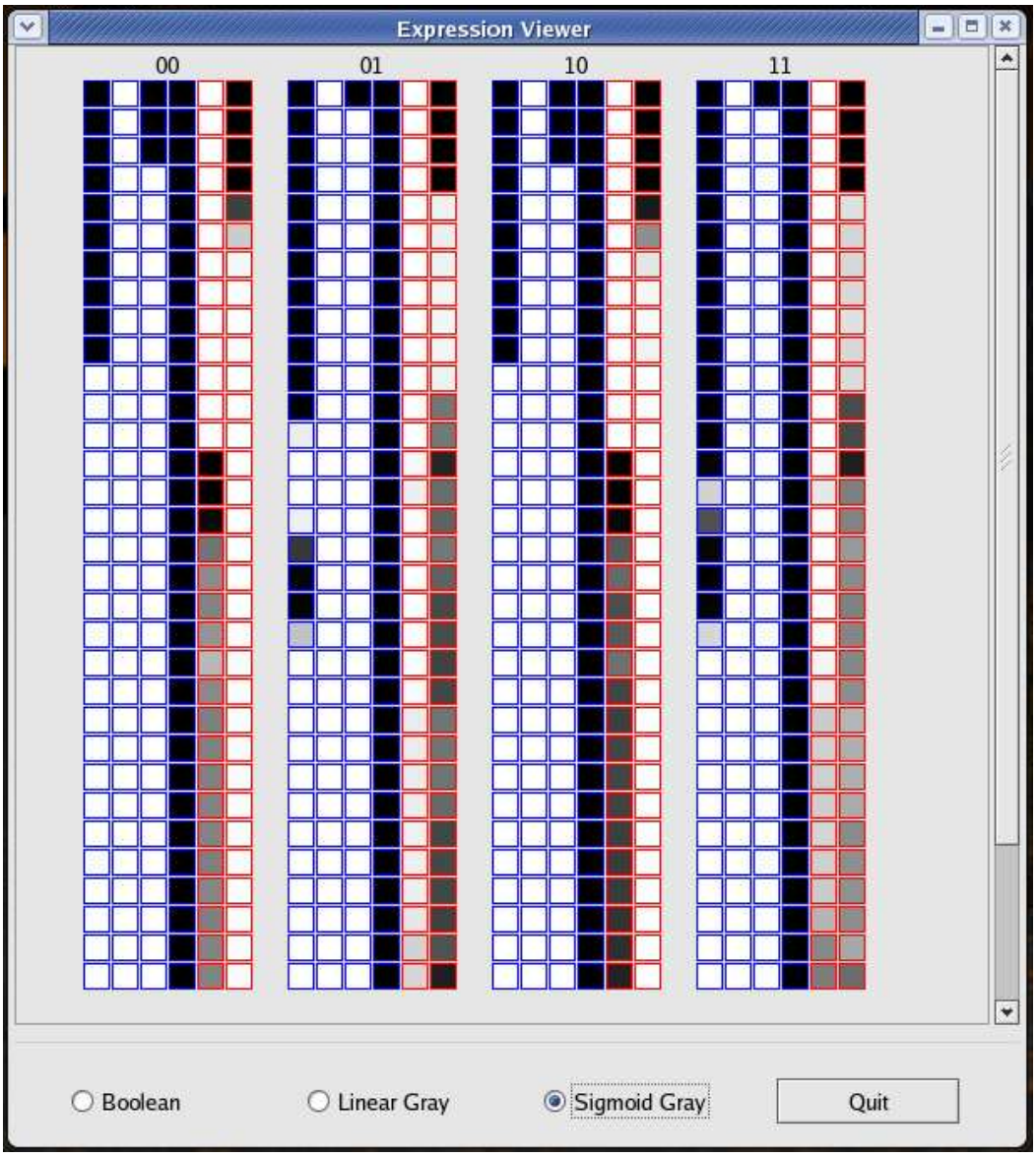

Figure 3.9: Custom expression pattern viewer (image courtesy of Nicholas Geard).

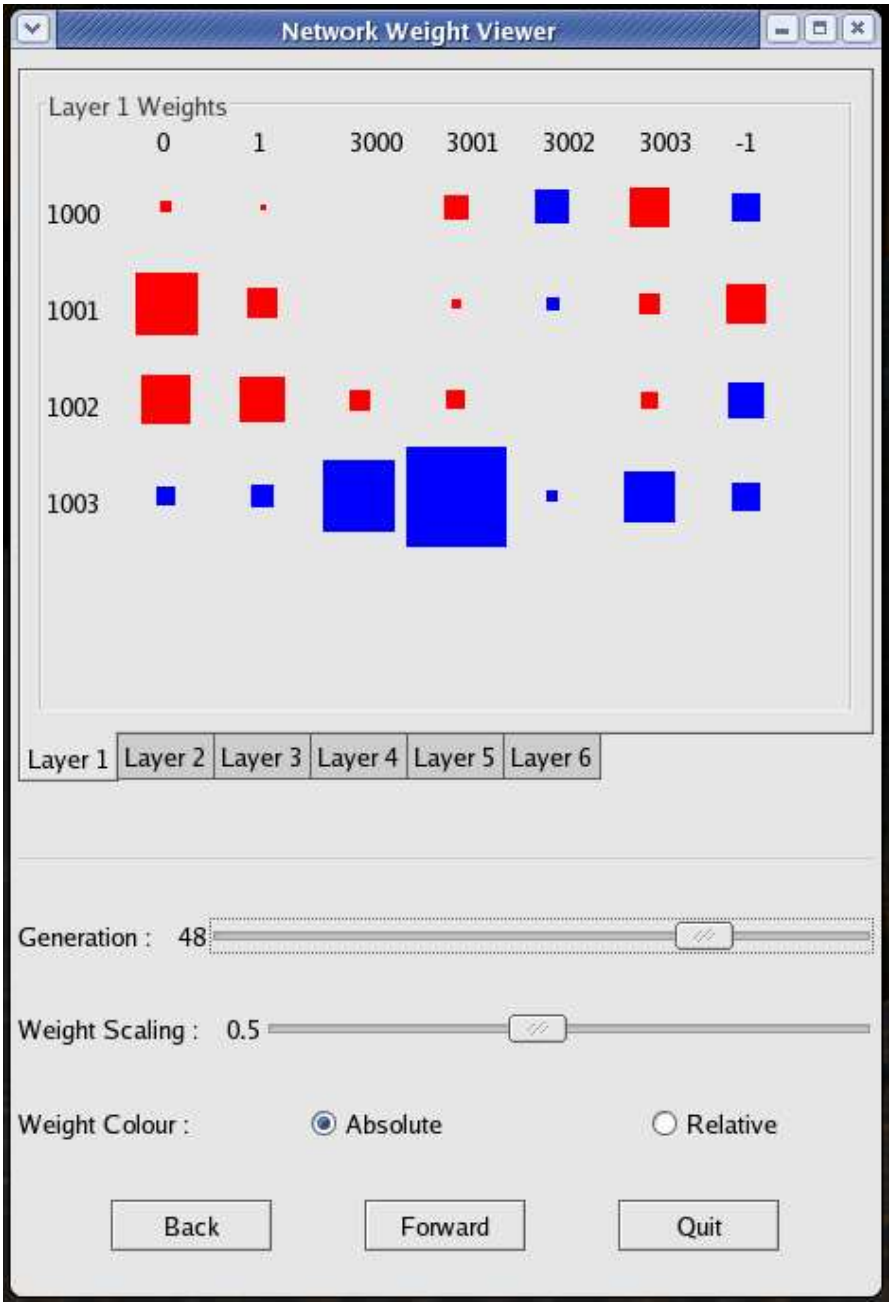

Figure 3.10: Custom network weight visualization (image courtesy of Nicholas Geard).

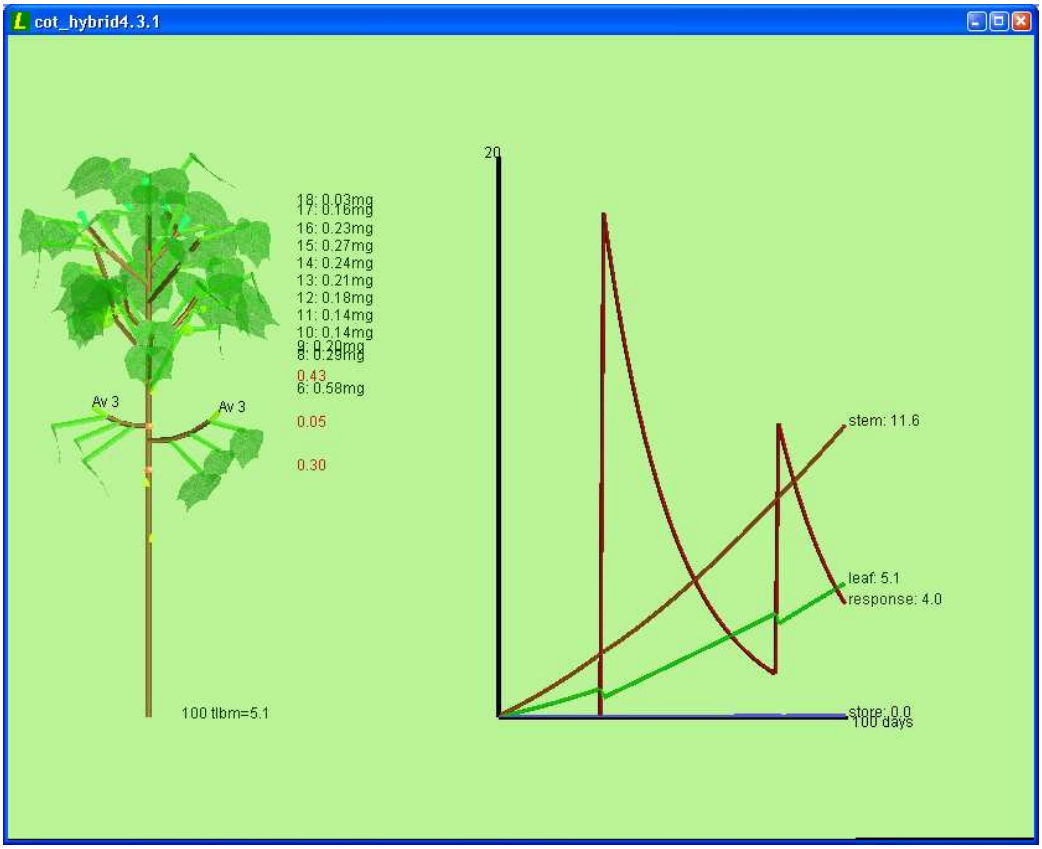

Figure 3.11: The visual output of the hybrid model used to study compensation for defoliation in cotton plants.

these components and then integrating them. This form of computational model was used due to its ability to model at an abstract level for qualitative study, and because hundreds of actual plant experiments would have taken an infeasible amount of time. In order to analyse the emergent interactions of the model, a visualization of the plant through development was implemented (see Figure 3.11). The model was designed to be constantly changed, as it formed the current version of the hypothesis.

In L-studio, there are no provisions for automated test cases or dedicated debuggers. However, no L-system bugs were identified – the issues in this style of implementation were reported to be errors in model assumptions (as opposed to errors in the implementation). To find these erroneous assumptions, laboratory experiments were required to determine what was wrong. On the other hand, the researcher was interested in using test cases, since this process would formalize the modeling process and perhaps make assump-

tions more explicit. While time is an issue, both in terms of implementation time and time to learn new techniques, he believes more rigour in the implementation of the model would be useful, and a more rigorous form of testing would improve confidence in the emergent properties of the simulations.

Many software engineering techniques are used in this project, but not in a formal manner. For example, conceptual modeling is employed, but not in a language such as UML, while versioning is practised by saving different versions of a file separately.

3.4 Case Study 3: Regulatory Network in Pea

The final case study was a computational model developed to specifically study the interactions of the regulatory network controlling branching in pea (*Pisum sativum*) (Harding, 2003). This model was developed since real-world experiments have resulted in a number of components and variables that make the integration of existing hypotheses with new ones complicated and time-consuming. Through comparisons with experimental data, the model has been successfully used to refine and improve the underlying biological hypotheses. The regulatory network hypothesis under investigation is shown in Figure 3.12, while the graphical output of the model is shown in Figure 3.13.

3.4.1 Software Development Requirements

This model was once again developed in L-studio/CPFG, based on a set of components and developed in a bottom-up fashion (i.e., initially starting with two genes, then more genes and hormones were added while comparing the simulation behaviour to experimental data). The model was refined as more experimental data was obtained, and also on the basis of simulation output. The primary developer was a biologist.

As found in Section 3.3, there are no debugging tools readily accessible to this project. The researcher expressed interest in automated and manual test cases for this model's development – if they could be integrated with the L-studio environment. A more rigorous model development was considered worthwhile, as this was thought to give greater weight to whether the model fits the data. Additionally, she considers the idea of version control as a useful technique to facilitate back-tracking and branching of ideas.

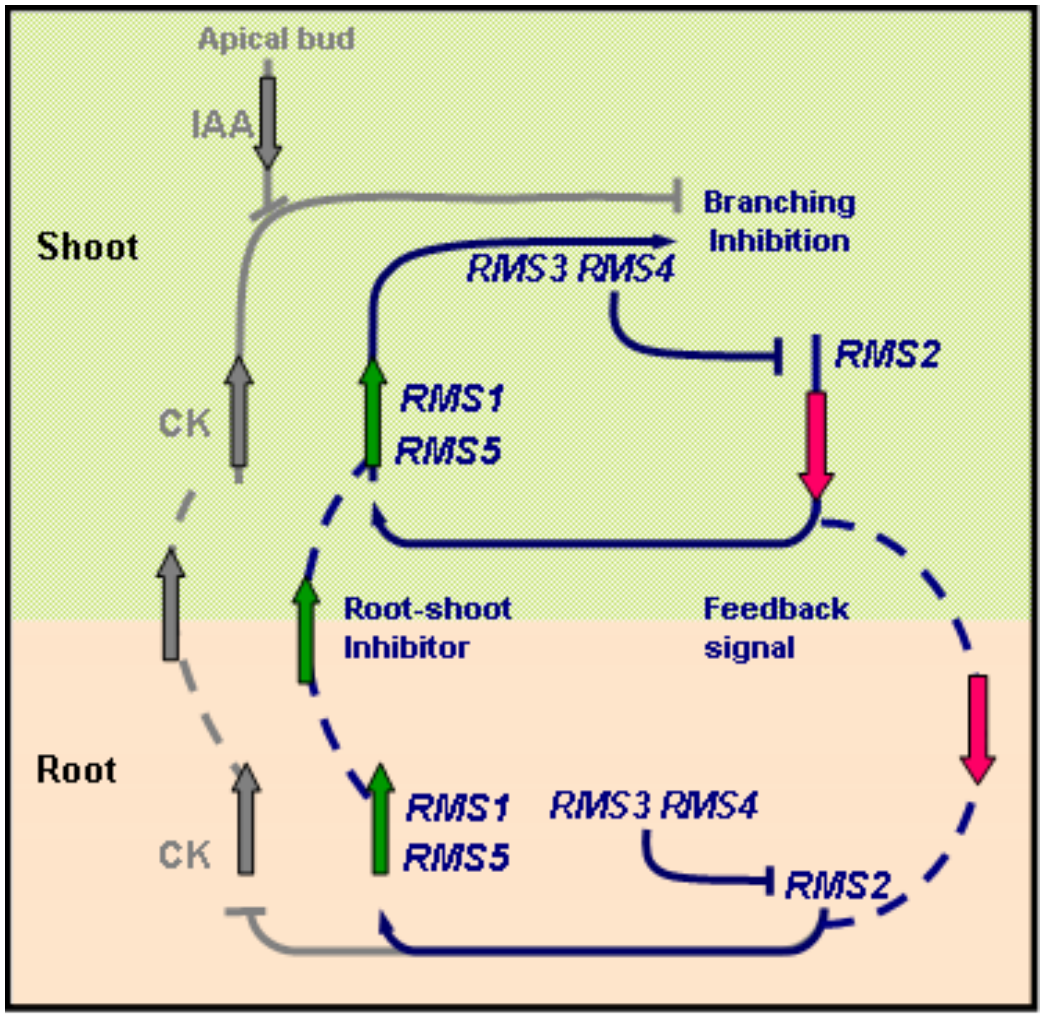

Figure 3.12: Modified multi-cycle network, taken from Harding (2003), page 14.

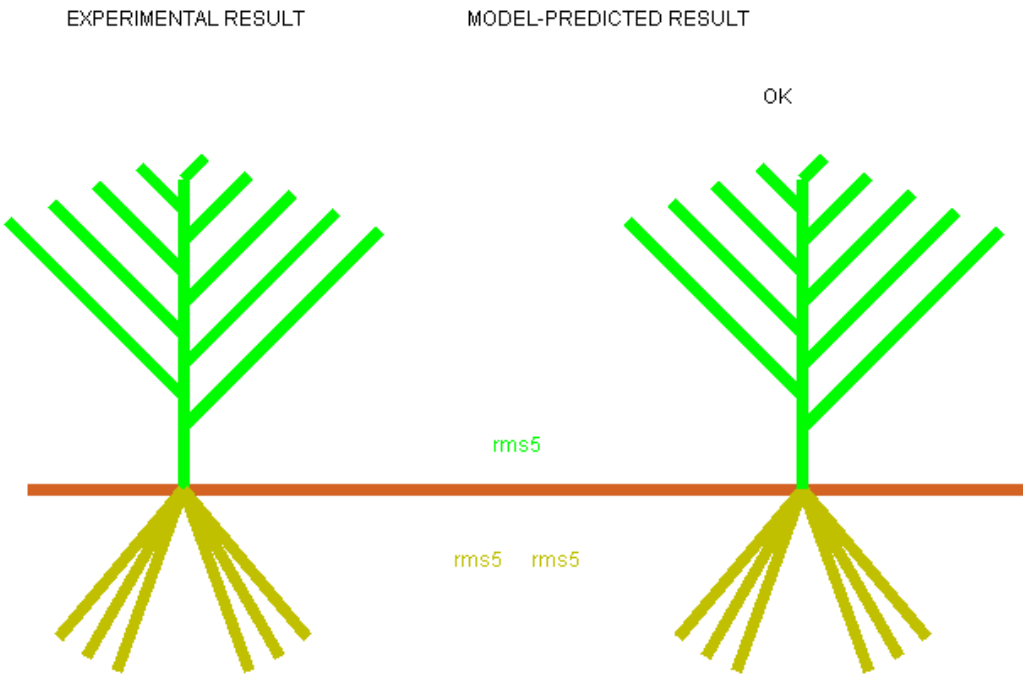

Figure 3.13: Model visualization of branching regulatory network in pea. This model runs through various genotype combinations for the root and shoot of a pea plant. The biological result is shown on the left and the model's predicted result is on the right.

3.5 Common Requirements

Features common to all of the models surveyed in this study can be summarized into the following four key points:

1. **Small team size:** The simulations are implemented and used by a very small number of people – often a single developer is the single user;

2. **Transient systems:** These software models are developed to help understand a particular research question, and are often redesigned or even discarded as the question / understanding changes;

3. **Rapidly evolving specifications:** The simulations are developed and used in an iterative situation of rapid specification change, with specifications often changing as quickly as the software is run;

4. **Non-linear:** The precise behaviour of the complete simulation is unknown before runtime, with emergent interactions of system components providing the observations of interest.

Section 4

Recommendations

Rapidly evolving requirements coupled with small team sizes (points 1-3 in Section 3.5) conspire to make many software engineering techniques impractical for the modelers surveyed. The following recommendations distil the most immediately useful techniques for such modeling.

4.1 Maintenance of Static Components

The regulatory models surveyed generally exhibited a component-based architecture. Although the systems on the whole changed rapidly, many of their components remain unchanged for long periods, and their expected *isolated* behaviour is known before runtime. The following techniques aid the development, testing and maintenance of such logical software units. They can be easily incorporated into any software development program, and are deemed to be of immediate, practical benefit to regulatory modeling projects.

4.1.1 Design Notations

Spending more time and effort at the design phase of a software project improves development speed and productivity (Cusumano et al., 2003). Notations that aid in the design of component functionality and their relationships (e.g., inheritance, class interactions), should be used both as a convenient means of design documentation and to help eliminate design inconsistencies. Formal notations, as opposed to *ad hoc* maps or diagrams, are preferable since they provide placeholders for required information (which help to reduce inconsistencies) and provide a formal language which aids communication with other team members, or with oneself in the future. UML (`http://www.uml.org`), provides two notations that aid component design:

Class Diagrams: which document classes and their relationships, and

Object Diagrams: which document instances instead of classes to explain complicated class relationships (such as recursive relationships).

A good introduction to these notations is provided online by Borland at `http://bdn.borland.com/article/0,1410,31863,00.html`

4.1.2 Version Control

Version control software provides a means of conveniently storing different versions of files. Changes to files are accompanied with comments, which provide a record of how the model changed. Versions can be tagged and branched, allowing easy retrieval of major revisions or milestones of the code (e.g., the code used to generate the results of a particular publication). Version control systems prevent the loss of information through overwriting, as local copies of a file are modified before being merged with the version control repository, and deleted/altered parts of a file are available by checking out previous versions. Remote repositories provide automatic remote backup of files. Version control systems also help teams larger than a single developer (or a single developer working at multiple locations) ensure that the correct version of a file is being worked on. Two of the most popular version control systems are:

CVS: `https://www.cvshome.org/`

Subversion: `http://subversion.tigris.org/`

4.1.3 Unit Testing

Unit testing is the testing of individual software components against predefined specifications. This helps improve confidence that components behave as expected under a range of conditions. Libraries are available for many popular programming languages that help automate this testing process. Such libraries include:

Boost test libraries: for C++, available at `http://www.boost.org/libs/test/doc/index.html`

UnitTest: for Python (comes as part of the standard Python library)

NUnit: for the .NET framework, available at `http://www.nunit.org/`

JUnit: for Java, available at `http://www.junit.org/`

4.1.4 Automatic Document Generation

The automatic document generation capabilities of many modern programming languages can generate both source code documentation (hyperlinked where appropriate) and often diagrams of component relationships from custom source code comments. A particular advantage of using these systems is that documentation can occur as the source code is written, making it easier to record detailed information. Document generation software includes:

Doxygen: for many languages, including C, C++, Java, Objective-C and IDL, available at `http://www.doxygen.org/`

NDoc: for the .NET framework, available at `http://ndoc.sourceforge.net/`

JavaDoc: for Java, see `http://java.sun.com/j2se/javadoc/`
(class diagrams can be achieved with third party tools such as UMLGraph, available at `http://www.spinellis.gr/sw/umlgraph/`)

4.2 Manually Tracking Component Interactions

For simulations with a *small* number of critical components, simple means of manually tracking object interactions should be employed to aid understanding of expected runtime behaviour. Two such methods are:

UML Interaction Diagrams: description available at
`http://bdn.borland.com/article/0,1410,31863,00.html`

Behaviour Trees: more information available at
`http://www.sqi.gu.edu.au/gse/papers/`

4.3 Visualization

Visualization provides a valuable way to understand emergent phenomena (Dorin, 2002). Visualizations that show the relationship between micro level interactions and macro level behaviours are some of the most powerful methods for understanding the system-wide dynamics of the models surveyed. Other methods that track the interaction between system components (such as system testing and audit trails) are generally less useful due to the emergent characteristics of the models and their rapidly changing specifications.

Good visualizations provide the researcher/developer/user a means of developing an understanding of system-wide behaviour, and provide a way of identifying unexpected phenomena and software bugs. Visualization can aid understanding of micro and macro level interactions at three levels:

Structure (or network) of interactions: Examples include network diagrams, which show the structure of interactions between network nodes, and distribution plots, such as in-degree and out-degree distributions, which illustrate larger-scale network structure. The Pajek program is useful for such network-level visualization (see `http://vlado.fmf.uni-lj.si/pub/networks/pajek/`).

Dynamics of interactions: A good example is gene expression diagrams, which show activated genes over time and provide a visualization of patterns that emerge from networks of interactions. See Reil (1999) for an example of their use.

Function (or behaviour) of interactions: Examples include L-systems (Prusinkiewicz et al. (2000)), which are particularly suited to plant modeling but can be extended to other visualization tasks, and developmental phenotypes that can be conveniently placed under selective pressure (see Niklas (1997)).

Section 5

Conclusions

Computational models are becoming an increasingly popular means to understand regulatory models in biology. While the distinction between the software development stage and the usage stage is less clear cut in scientific software than in commercial software (Johnson et al., 2004), many techniques from the software engineering field are directly applicable to the development and use of biologically-grounded regulatory models. This project has focused on existing software development techniques that provide immediate, practical benefit to modelers of such systems.

Through an online survey and focused interviews, commonalities between various regulatory models were identified. Software models were generally implemented and used by a small number of people. Development followed an object-oriented paradigm. The specifications of the model changed rapidly, and software systems were generally written to address a specific research question. The behaviour of the software was unknown before runtime.

While surveying existing modeling practises, it became clear that a language barrier exists between biological researchers and the software engineering community, which has resulted in many techniques not being employed. Terms such as 'unit testing', 'version control' and 'design notations' sounded irrelevant to the biologists surveyed, but were found to be interesting once their function was explained. Encouragingly, the modelers surveyed were generally willing to learn software engineering techniques shown to be useful.

The small team sizes, coupled with rapidly evolving requirements, conspire to make many software engineering techniques impractical. For example, formal proofs and audits are unlikely to be applied in the day to day development and usage of such systems. However, the surveyed systems exhibited a component-based architecture, with many of these components remaining unchanged for long periods. Thus, techniques that aid the development and maintenance of small logical units, such as design notations, version

control, unit testing, and automatic document generation, can all serve immediate, practical benefit to biological modelers. In addition, methods that allow the micro level tracking of component interactions, and visualizations that aid understanding of micro and macro level dynamics, are also highly useful.

Opportunities for ACCS outreach from this project include tutorials, workshops, and online resources of relevant software engineering techniques targeted at the biological modeling community.

References

Aho, A. V. (2004). Software and the future of programming languages. *Science*, 303:1331–1333.

Banzhaf, W. (2003). On the dynamics of an artificial regulatory network. *Lecture Notes in Artificial Intelligence*, 2801:217–227.

Banzhaf, W. (2004). On evolutionary design, embodiment, and artificial regulatory networks. *Lecture Notes in Artificial Intelligence: Embodied Artificial Intelligence*, 3139:284–292.

Barzel, R. (1992). *Physically-Based Modelling for Computer Graphics: A Structured Approach*. San Diego, Calif.: Academic Press.

Bhaller, U. S. and Iyengar, R. (1999). Emergent properties of networks of biological signaling pathways. *Science*, 283:381–387.

Bongard, J. C. (2002). Evolving modular genetic regulatory networks. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC2002)*, pages 1872–1877, Piscataway, NJ. IEEE Press.

Bongard, J. C. and Pfeifer, R. (2001). Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In *Proceedings of The Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 829–836, San Francisco, CA. Morgan Kaufmann Publishers.

Bornholdt, S. (2001). Modeling genetic networks and their evolution: A complex dynamical systems perspective. *Biological Chemistry*, 382(9):1289–1299.

Carson, J. S. (2002). Model verification and validation. In *Proceedings of the 2002 Winter Simulation Conference*, pages 52–58.

Cusumano, M., MacCormack, A., Kemerer, C. F., and Crandall, B. (2003). Software development worldwide: the state of the practice. *IEEE Software*, 20:28–34.

Dorin, A. (2002). The visual aspect of artificial life. In Sugisaka and Tanaka, editors, *Proceedings of the 7th International Symposium on Artificial Life and Robotics*, pages 451–455.

Galis, F. (2003). Book review of Gerd B. Müller and Stuart A. Newman (editors) (2003). Origination of Organismal Form. Beyond the Gene in Developmental and Evolutionary Biology. *Acta Biotheoretica*, 51(3):237–238.

Geard, N. and Wiles, J. (2003). Structure and dynamics of a gene network model incorporating small RNAs. In Sarker, R., Reynolds, R., Abbass, H., Tan, K.-C., McKay, R., Essam, D., and Gedeon, T., editors, *Proceedings of the 2003 Congress on Evolutionary Computation (CEC2003)*, pages 199–206, Piscataway, NJ. IEEE Press.

Guergachi, A. and Patry, G. (2003). Using statistical learning theory to rationalize system model identification and validation part I: mathematical foundations. *Complex Systems*, 14(1):63–90.

Hallinan, J. and Wiles, J. (2004a). Asynchronous dynamics of an artificial genetic regulatory network. In *Ninth International Conference on the Simulation and Synthesis of Living Systems (ALife9)*, Boston.

Hallinan, J. and Wiles, J. (2004b). Evolving genetic regulatory networks using an artificial genome. *Conferences in Research and Practice in Information Technology*, 29:291–296. 2nd Asia-Pacific Bioinformatics Conference (APBC2004), Dunedin, New Zealand.

Harding, E. A. (2003). Computational analysis and molecular physiology of the branching regulatory network in pea. Honours thesis, Department of Botany, University of Queensland.

Hatton, L. (1997). The T experiments: errors in scientific software. *IEEE Computational Science and Engineering*, 4(2):27–38.

Johnson, C. G., Goldman, J. P., and Gullick, W. J. (2004). Simulating complex intracellular processes using object-oriented computational modelling. *Progress in Biophysics & Molecular Biology*, 86:379–406.

Khan, S., Makkena, R., McGeary, F., Decker, K., Gillis, W., and Schmidt, C. (2003). A multi-agent system for the quantitative simulation of biological networks. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 385–392.

Niklas, K. (1997). Adaptive walks through fitness landscapes for early vascular land plants. *American Journal of Botany*, 84(1):16–25.

Prusinkiewicz, P., Hanan, J., Mech, R., and Karwowski, R. (2000). L-studio/cpfg: a software system for modeling plants. *Lecture Notes in Computer Science*, 1779:457–464.

Reil, T. (1999). Dynamics of gene expression in an artificial genome - implications for biological and artificial ontogeny. In Floreano, D., Mondada, F., and Nicoud, J., editors, *Proceedings of the 5th European Conference on Artificial Life*, pages 457–466. Springer Verlag.

Sargent, R. G. (1988). A tutorial on validation and verification of simulation models. In Abrams, M., Haigh, P., and Comfort, J., editors, *Proceedings of the 20th Conference on Winter Simulation*, pages 33–39.

Taylor, T. (2004). A genetic regulatory network-inspired real-time controller for a group of underwater robots. In Groen, F., Amato, N., Bonarini, A., Yoshida, E., and Kröse, B., editors, *Proceedings of Intelligent Autonomous Systems 8 (IAS8)*, pages 403–412, Amsterdam. IOS Press.

Thornby, D., Renton, M., and Hanan, J. (2003). Using computational plant science tools to investigate morphological aspects of compensatory growth. *Lecture Notes in Computer Science*, 2660:708–717.

Watson, J., Geard, N., and Wiles, J. (2004). Towards more biological mutation operators in gene regulation studies. *BioSystems*, 76(1-3):239–248.

Watson, J., Wiles, J., and Hanan, J. (2003). Towards more relevant evolutionary models: integrating an artificial genome with a developmental phenotype. In *Proceedings of the Australian Conference on Artificial Life (ACAL 2003)*, pages 288–298.

Weiß, G. (2001). Agent orientation in software engineering. *The Knowledge Engineering Review*, 16(4):349–373.

Willadsen, K. and Wiles, J. (2003). Dynamics of gene expression in an artificial genome. In Sarker, R., Reynolds, R., Abbass, H., Tan, K.-C., McKay, R., Essam, D., and Gedeon, T., editors, *Proceedings of the 2003 Congress on Evolutionary Computation*, pages 185–190, Piscataway, NJ. IEEE Press.

Zambonelli, F. and Omicini, A. (2004). Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9:253–283.

Index

Appendix A

General Survey

ACCS Online Survey: Tools and Techniques in Simulation Development

Thank you for taking the time to complete this survey. It is part of an ACCS project aimed at identifying tools and techniques that help increase confidence in the correctness of software simulations, with a focus on artificial life, complex systems and computational biology research.

The primary purpose of this survey is to gather information on the tools and methodologies currently used by the general research community when designing, implementing and testing software simulations. A secondary objective is to collect opinions on why software simulations and software engineering methods are/are not employed.

All responses are anonymous, but please feel free to contact me via email with more information, or with any queries about the project.

Field of work: Please briefly describe your field of research / position:

Simulation usage: Do you use software simulations in your work?

- ○ Yes (please continue to the next question)

- ○ No — If you could be convinced of the accuracy of software simulations (e.g., through validation and verification techniques), would you employ them in your research? Please briefly explain:

Thank you for participating in this survey [Submit]

CS background: What is your computer science background?

- ☐ Higher degree
- ☐ Bachelor's degree
- ☐ Technical college / TAFE
- ☐ Self-taught
- ☐ Other:

Type of simulation: Describe the type(s) of simulation that you design/develop/use (e.g., optimization, modelling of natural processes, etc.). Please include any modelling paradigms (e.g., neural networks, cellular automata, Boolean networks) that are often used.

Simulation context: Describe how your simulations fit in with your research activities (e.g., get grant, develop software, run with sensible parameters, analyse results, refine software and parameters, run again, analyse again, write report, etc.).

Team size: How many people are involved in simulation development?

- ○ 1 (just me)
- ○ Other:

User base: How many people are involved in the use of your simulations?

- ○ 1 (just me)
- ○ Other:

Implementation operating system: On what operating system(s) do you implement your simulations?

- ☐ Windows
- ☐ Linux
- ☐ Solaris
- ☐ Irix (ozone)
- ☐ Other:
- ☐ Unix (other)
- ☐ Mac OS
- ☐ Don't know

Simulation operating system: On what operating system(s) do you run your simulations?

- ☐ Windows
- ☐ Linux
- ☐ Solaris
- ☐ Irix (ozone)
- ☐ Other:
- ☐ Unix (other)
- ☐ Mac OS
- ☐ Don't know

Simulation hardware: On what hardware do you run your simulations?

☐ Desktop ☐ Supercomputer ☐ PC cluster ☐ Other:
☐ Don't know

Programming language: Which programming languages do you use to implement your simulations?

☐ Ada	☐ C	☐ C++	☐ C#	☐ Cobol
☐ D	☐ Eiffel	☐ Fortran	☐ Java	☐ Lisp
☐ L-systems	☐ Matlab	☐ Objective-C	☐ Occam	☐ Pascal
☐ Perl	☐ Python	☐ Ruby	☐ Scheme	☐ Simula
☐ Smalltalk	☐ Tcl	☐ Visual Basic		
☐ Other:				

Primary development environment: What would you consider to be your primary development environments / languages? (e.g., Matlab, C++ in Visual Studio, gcc and vi, etc.)

Debugger: What do you use to debug your software?

☐ Language debugger	☐ Print statements
☐ No debugger	☐ Postgrads
☐ I don't introduce bugs	☐ Other:

Version control: What version control software do you / your research group use?

☐ None
☐ CVS ☐ Subversion ☐ Visual SourceSafe ☐ RCS
☐ Other:

Methodology awareness: Please select all of the software engineering methodologies listed below that you are aware of.

☐ Acceptance testing	(Testing conducted by the user (as opposed to the developer(s)))
☐ Audits	(Reviews performed by a 3rd party)
☐ Design patterns	(Collections of recurring problems and their solutions in software design)
☐ Formal proofs	(Mathematical proofs ensuring all inputs produce correct outputs)
☐ Integration testing	(Testing of integrated sub-components)
☐ Modelling languages	(Notations for specification and design of software (e.g., UML))

☐ Process patterns	(Collections of recurring problems and their solutions in the software development process)
☐ Regression testing	(Selective re-testing of subcomponents as they are modified)
☐ System testing	(Testing of whole system against requirements)
☐ Technical reviews	(Checks items conform to specifications)
☐ Tracing	(Traces the relationship between development phases; e.g., between user requirements and software requirements)
☐ Unit testing	(Tests (usually automated) of specific modules)
☐ Other:	

Methodology deployment: Which do you employ in the design and development of your simulation software?

☐ Acceptance testing ☐ Audits ☐ Design patterns
☐ Formal proofs ☐ Integration testing ☐ Modelling languages
☐ Process patterns ☐ Regression testing ☐ System testing
☐ Technical Reviews ☐ Tracing ☐ Unit testing
☐ Other:

Use of formal (or informal) processes: Do you have formal or other processes you / your research group adhere to in the design and development of software simulations?

○ No
○ Yes (Please describe briefly):

Usefulness of software engineering: Do you think an increased use of software engineering methods would improve the quality of your simulation results? The quality of the final science published?

○ Yes ○ No

(Please briefly explain):

If so, do you feel that having a formalised process for the design and implementation of your simulations would result in significant enough improvements to warrant the extra time?

○ Yes ○ No

(Please briefly explain):

Time constraints: How much time would you be willing to spend on learning about and using software engineering techniques that might prove useful to your research?

○ None ○ Hours ○ Days ○ Weeks ○ Months ○ Other

(Please briefly explain):

Other comments: If there are any other comments you would like to add, including any advice on excellent modelling practices, please insert them here.

Submit