Development and Application of a Simulation Environment (NEO) for Integrating Empirical and Computational Investigations of System-Level Complexity

Clemente Izurieta[1], Geoffrey Poole[2], Robert A. Payn[2], Isaac Griffith[1], Ryan Nix[1], Ashley Helton[3], Emily Bernhardt[3], Amy J. Burgin[4]

[1] Department of Computer Science, Montana State University
[2] Department of Land Resources and Environmental Sciences, Montana State University
[3] Department of Biology, Duke University
[4] School of Natural Resources, University of Nebraska-Lincoln

{clemente.izurieta, gpoole, rpayn}@montana.edu, {isaac.griffith, ryan.nix}@msu.montana.edu, {amh72, ebernhar}@duke.edu, aburgin2@unl.edu

***Abstract*—Network Exchange Objects (NEO) is a new software framework designed to facilitate development of complex natural or built distributed system models, where the system model is represented as a graph, through which *currencies* (e.g., coding information) flux. This paper introduces "system-level hypothesis (SLH) testing" as a form of computational thinking that will drive integration of computational and empirical sciences to promote efficient, self-correcting inquiry into the operations and behavior of complex systems. To demonstrate NEO, we examine the problem of maximizing the productivity of a software development organization by measuring growth in the total lines of code (LOC) contributed by developers. We develop a software framework (NEO) that allows rapid creation of model variants representing alternative SLHs. NEO is designed to investigate systems we describe as "complex adaptive hierarchical networks" (CAHNs – complex systems represented as networks that route and store multiple interactive currencies). Models built atop NEO, are organized collections of individual values (model variables) and algorithms (model logic). Modelers systematically combine algorithms to create alternative model formulations *at runtime*. Thus, NEO is a simulation framework that can be used in any domain of expertise, where systems are represented as interdependent entities that store and flux multiple currencies.**

Keywords- modeling framework; experimentation; software evolution, tools

I. INTRODUCTION

Recent research has described drivers of complexity that span natural, built, and social systems. **Network theory** explains how subtle changes in patterns of connection among system components influence system behavior [11], including the distribution, flow, and transformation of "currencies," such as energy, matter, information, capital, or genes. **Complex systems theory** explains how processes such as emergence and self-organization can result from the interaction of system components [12]. **Hierarchy theory** describes how system components observable at different levels of organization are linked and can influence one another across spatiotemporal scales [1]. And principles from disciplines as divergent as ecological economics and ecological stoichiometry reveal how the storage, flux, and transformation of energy, matter, information, capital, and other currencies are fundamental yet **interdependent measures** of system behavior [13]. Thus, these four system characteristics grounded in natural systems – patterns of connection, interactions among components, hierarchical organization, and the interdependency of system currencies – represent primary drivers of many complex adaptive systems *sensu* [7].

Based on these concepts, we introduce "complex adaptive hierarchical networks" (CAHNs) as an operational framework for studying and understanding system-level complexity. CAHNs are similar to the spatially-explicit dynamic patch hierarchies of Wu and Loucks [15], except that patches (or in the case of a CAHN, networked "cells") can represent any physical component of a system. To visualize a CAHN, we start by representing a system as a collection of cells that are observable at a particular hierarchical level and linked by "edges" to form a network. Each cell in a CAHN represents a system component and each edge represents the potential for interaction between two cells. Cells and edges are holons[1] in a hierarchy,

[1] A "holon" represents a discernable component of a system observed at a particular hierarchical level. Holons are nested hierarchically, any holon can be subdivided into additional holons at the next lower hierarchical level.

meaning cells and edges observable at any hierarchical level can be decomposed into finer-scale cells and edges at the next lower hierarchical levels. Cells can store and Edges can transport or transform multiple currencies. Although, Cells and Edges in a CAHN generally store/flux/transform the same suite of currencies they may do so using a variety of rule sets, creating functionally heterogeneous fluxes across the system. Holons (Cells and Edges) have characteristics, which can be static or dynamic. Dynamic characteristics change according to rules that may consider: 1) the current characteristics of the holon; 2) the characteristics of surrounding holons; and/or 3) the relative abundance of and/or interdependencies among currencies. Additionally, various processes may create or destroy cells or edges, representing the evolution of network topology within the system.

We present an object oriented software framework, NEO; which enables rapid construction of computer models that simulate CAHN behavior. The design and implementation of NEO compartmentalizes model complexity, and facilitates rapid development, simplified management, and rigorous comparison of alternative model formulations. In combination with empirical methods, NEO allows scientists and engineers to investigate: 1) patterns of connection, 2) interactions among system components, 3) hierarchical organization, and 4) the interdependency of system currencies as simultaneous drivers of complex system behavior. NEO establishes an underlying mechanistic process necessary to execute repeatable experiments, reject refutable hypotheses and enable complex software management decisions. To illustrate NEO, we compare growth patterns of source code when subjected to differences in organizational topologies. NEO allowed the rapid development of alternate SLHs to predict code growth. We find that the model results are consistent with an expected empirical relation system for how source code grows.

II. The NEO Framework

The NEO framework facilitates development of CAHN simulation models and facilitates manipulation of complexity in the interactions of currencies, allowing analyses of alternative SLH's about the behavior of a given CAHN. NEO is, simultaneously, a means of building models of complex systems, and a means of testing and refining System Level Hypotheses (SLHs). As such, NEO represents a fundamental departure from existing modeling environments (e.g., Stella, Matlab, Swarm, Repast, Netlogo, etc.). Specifically, NEO allows a modeler to: 1) compartmentalize complexity in models by writing, debugging, and managing individual algorithms that represent the dynamics controlling a single characteristic of a complex system; 2) organize these algorithms into hierarchal groups that describe the "behavior" of individual components (i.e., how individual system components store, flux, or transform system currencies); 3) combine and recombine these behaviors to create different "types" (classes) of system components that can flux, store, or transform multiple interactive currencies; 4) describe the arrangement and connections among typed cells and links to represent the structure of a complex system of interest; 5) create model variants (competing SLHs) by strategically adding, removing, replacing, or refining algorithms, behaviors, currencies, or cells/edges in the model; and 6) maintain, manage, and execute model variants to test competing SLHs. NEO is based on a fundamental abstraction of a simulation model which views a model as simply organized collections of *Values* and *Algorithms*. *Values* can be either static (akin to model parameters) or dynamic (akin to state variables) during the course of model execution. Each dynamic *Value* has an associated *Algorithm* used to update the *Value* during model execution. Under this paradigm, a NEO model is run by executing each of the *Algorithms* once during every model iteration. To determine the order of *Algorithm* execution, NEO sorts the *Algorithms* based on their interdependencies.

III. Modelling Scenarios

In an extensive and systematic review of studies related to estimation of software development effort, Jorgensen [10] finds no substantial evidence in favor of model estimation techniques over expert estimation. Further, Jorgensen finds situations that suggest the importance of calibrating estimation models to specific organizations. To illustrate NEO we present five example scenarios that illustrate SLH testing by contrasting the productivity levels of an organization (measured by counting the total LOC produced) based on the topology of the organization. NEO is specifically designed to allow fine-grained calibration of models by facilitating how domain knowledge is imparted.

In section 3A we describe the equations that govern the simulation of source code growth on a development branch. We describe how individual and groups of programmers, as well as the topology of an organization can be conceptualized as a NEO network and how LOC can be thought of as a currency moving through the network. In section 3B we describe variations made to the topology of an organization and to the experience levels of individual programmers. In section 3.C we analyze the observed growth curves.

A. Simulating Development of a Code Base

In a coding flow model a single programmer (Figure 1a), a group of programmers (Figure 1b), and a development branch are represented as Cells (Nodes). The edge connecting the cells represents the flow of information between developer(s) and the development branch. In this case, information is made up of lines of code. The pink boxes represent static parameter *Values* that the model uses. The black (diamond) boxes represent *Algorithms* that calculate the value of the dependent variables (in the pink boxes) during each time step (iteration) of the model. We posit that change in size of the code base to which one or more programmers contributes can be modeled by assuming that each programmer has a maximum potential rate at which LOC are generated. Thus, the "currency" of the model (i.e., what flows and/or is accumulated within the model) is LOC.

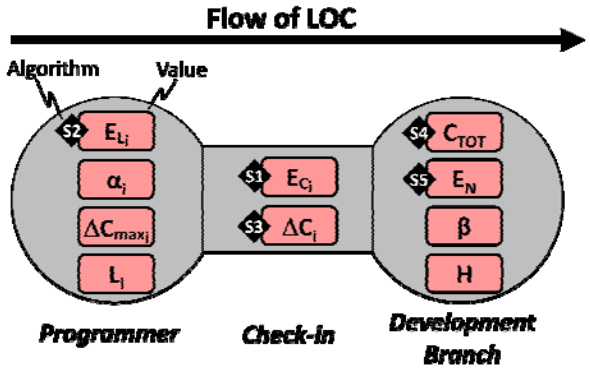

Figure 1a. Simple coding flow growth model to predict LOC contributed by a single programmer.

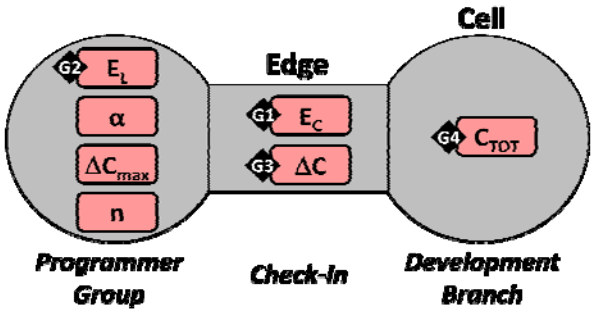

Figure 2b. Simple coding flow growth model to predict LOC contributed by a group of programmers.

As the code base increases in size, potential LOC produced by each programmer will not be realized, but instead, will be reduced by some efficiency factor (coding penalty) because coding becomes more complicated as the size of the code base increases:

$$E_{C_{t_i}} = \Delta C_{max_i}/(LOC_{t-\Delta t} + 1)^2 \quad \text{(S1)}$$
$$E_{C_t} = \Delta C_{max}/(LOC_{t-\Delta t} + 1)^2 \quad \text{(G1)}$$

where E_C is an efficiency reduction associated with the increasing complexity [14] of the code base, t is the current model time step, ΔC_{max} is the maximum potential lines of code generated by a programmer, LOC is the lines of code in the code base, and Δt is the model time step. Further, as more programmers are added to a project, we posit that each programmer will have to spend more time communicating with fellow programmers and therefore will spend less time generating code.

$$E_{L_i} = e^{-\alpha_i \cdot L_i} \quad \text{(S2)}$$
$$E_L = e^{-\alpha \cdot (n-1)} \quad \text{(G2)}$$

E_L is an efficiency reduction due to time spent communicating with other programmers, α is a constant, L_i is the number of communication links associated with a single programmer i, and n is the number of programmers contributing to the code base. An additional efficiency reduction is expressed in equation S5 for individual programmers:

$$E_N = e^{-\beta \cdot H} \quad \text{(S5)}$$

E_N represents the penalty associated with the network topology, where β is a constant and H is the maximum number of hops required for a given programmer i to communicate with programmer j in a given topology.
Thus, the lines of code generated will be the sum of the potential lines of code for each programmer, multiplied by the efficiency factors from the prior equations:

$$\Delta C_{t_i} = \Delta t \cdot \Delta C_{max_i} \cdot E_{L_i} \cdot E_{C_{t_i}} \cdot E_N \quad \text{(S3)}$$
$$\Delta C_t = \Delta t \cdot \Delta C_{max} \cdot E_L \cdot E_{C_t} \quad \text{(G3)}$$

Finally, LOC accumulate within the development branch:

$$C_{tot_t} = C_{tot_{t-\Delta t}} + \sum_{i=1}^{n} \Delta C_{t_i} \quad \text{(S4)}$$
$$C_{tot_t} = C_{tot_{t-\Delta t}} + n \cdot \Delta C_t \quad \text{(G4)}$$

where C_{tot} is the total number of LOC in the code base.

This simulation model provides a simple example that describes how the total lines of code contributed by a developer or a group of developers grow. In the degenerative case of a single developer, when the state variable L_i (S2) is set to 0, no efficiency penalty can be incurred and the value of E_{Li} (= 1) remains constant throughout the simulation. This is similar to a programmer group of size n = 1.

In the programmer group example (Figure 1b), the model is constrained by assumptions that apply equally to a *clique* of programmers. Thus, there exists a high penalty for the number of communication links, but a low penalty for the number of hops required to reach another programmer (i.e., H=1). In order to relax these assumptions, NEO facilitates the calibration of models by allowing each programmer (agent) to have distinct parameter values and therefore increase estimation accuracy. Domain experts now have the ability to adjust individual parameters. Values chosen for the constants α (eq[s] S2 and G2), which affects the communication links efficiency, and β (eq. S5), which affects the number of hops necessary to reach a programmer are 0.1 and 0.3 respectively. Figures 2a and 2b depict the efficiency loss as the number of communication links and hops increase.

B. Variations of Organizational Topology

Variations in organizational topology are significant when modeling productivity. Clearly, management's goals are to maximize effort by making educated decisions when assigning responsibilities to individual programmers. For example, studies by Cain and McCrindle [5] show that organizational structure (or lack of) can be reflected in the code base, and that the number of programmers in a team is not necessarily a good indicator of an organization's productivity. The first observation was based on Conway's Law [6]: *"the structure of the system mirrors the structure of the organization that designed it,"* and the second observation is based on Brook's Law [4]: *"adding more*

programmers to a late project makes it later." The scenarios shown in Figure 3 where chosen to illustrate the changing topologies of an organization.

The dark circles represent experienced programmers with a development potential (ΔC_{max}) of 500 LOC per week, and the light circles represent novice programmers with a development potential of 250 LOC per week. Diagrams A and B implement the programmer group model depicted in Figure 1b which is constrained by assumptions that apply equally to all programmers in the group. Thus, you cannot differentiate between individual programmers. Models C through E relax these assumptions. The models depicted in C and D represent typical hierarchical groupings that represent two development teams. In C, we create a communication link between the two experienced programmers, whereas in D, the same communications link is delegated to novice programmers.

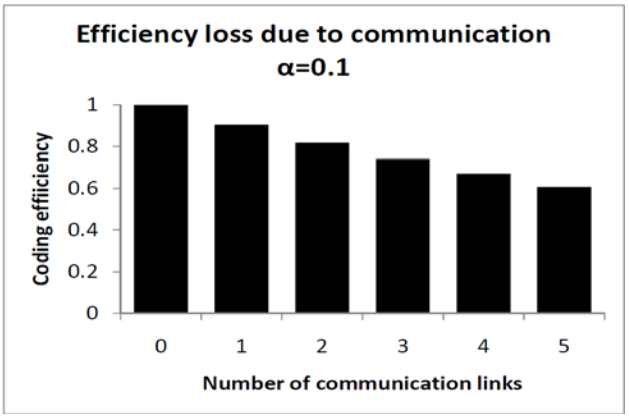

Figure 2a. Efficiency loss due to communication.

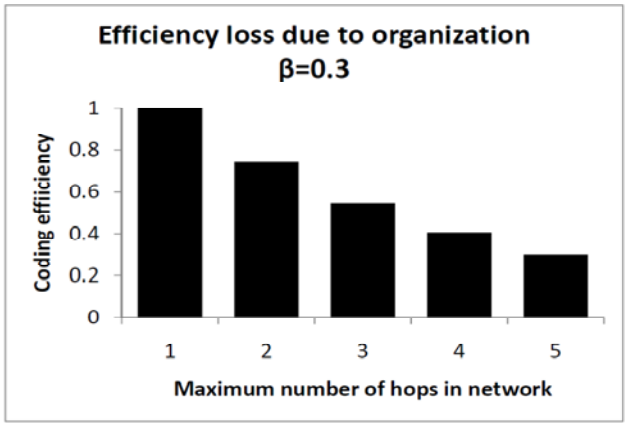

Figure 2b. Efficiency loss due to number of hops required to reach a programmer.

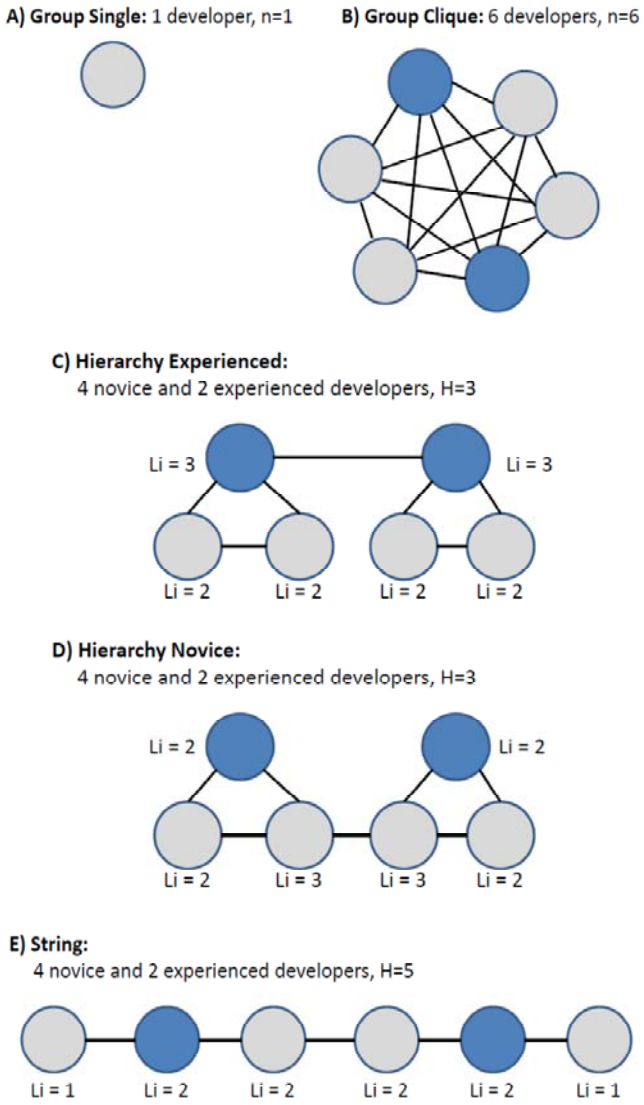

Figure 3. Group and individual developer topologies.

We observe that the maximum number of hops between two developers in these models is given by H=3. Unlike the group models depicted in A and B (graph cliques), developers are unconstrained by global assumptions. In this model this is exemplified by the varying number of communication links (L_i) that each programmer has. Finally, the model depicted in E is meant to illustrate a poor organizational structure. In this case H=5.

C. Scenario Results

We ran the simulation over a period of 520 weeks. Results are depicted in Figure 4.

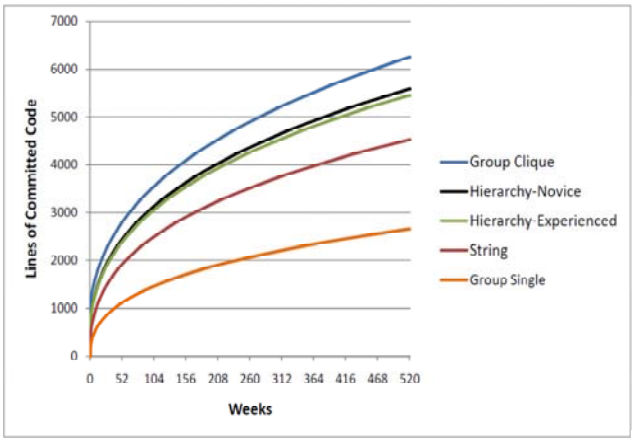

Figure 4. Growth simulation over 520 weeks.

Whilst these models are sophomoric, they clearly show the flexibility of NEO. In models A and B, we observe that while a group of programmers produces more code than a single programmer, the larger group is clearly constrained by the number of communication links that are introduced. Models C and D were created to contrast what happens to the productivity of a team when their experienced programmers are the main foci of contact versus how the productivity is affected when novice programmers are put in charge of the communication. The group that delegated communications to the novice programmers (D) is more productive. Unsurprisingly, the least productive group was E.

IV. NEO MODEL ORGANIZATION

Although the associations among *Holons(Cells and Edges), Behaviors, Algorithms,* and *Values* may appear complicated, implementing models is straightforward (Figure 5A). A NEO model is comprised of a source code hierarchy containing a collection of *Currency Packages*, each of which contains all of the *Behaviors* that pertain to a single currency. *Behaviors* are classified according to *Cell* or *Edge*, and each *Behavior* contains a collection of *Algorithms* implemented as Java classes. Each *Algorithm* has an *initialize()* and *calculate()* method (Figure 5B), which are called by NEO during model initialization and execution. The initialize method requests references to *Values* in the simulation and returns an initial magnitude of the *Value*. The calculate method returns the magnitude of the *Value* after each model iteration, as calculated by code performing a function of other values in the model. Thus, creating an *Algorithm* involves extending a Java object called *Dynam* and implementing the abstract *initialize()* and *calculate()* methods in the subclass named for its associated *Value*. A collection of implemented *Dynams* within the code hierarchy determines a currency *Behavior* sub-package, or the total set of rules for how a currency behaves in a given cell or edge *Holon*. Hierarchically, a complete set of cell and edge *Behaviors* define the potential actions or interactions available for a given currency *Package*. This approach allows a diverse set of potential *Algorithms* for a given currency in a CAHN to be divided among relatively simple constituent bits of code. Thus, complexity of a given CAHN model arises from the complexity of the network structure (see Developing NEO Models section), not complexity in the algorithms controlling the changes in each model *Value*. It also provides a formidable tool for creating, falsifying, and refining alternate SLHs, as described next.

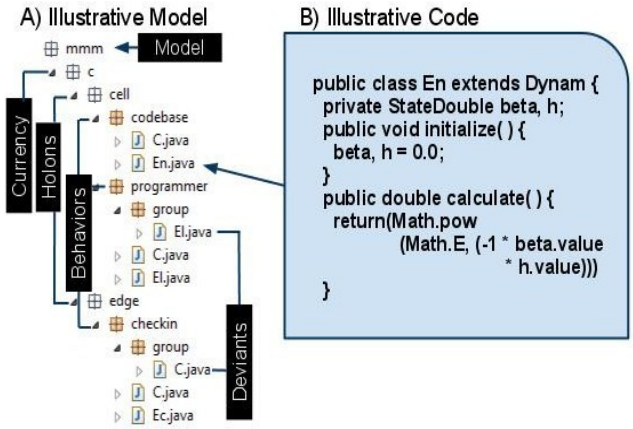

Figure 5. Hierarchy of code growth model. A) Levels of the code hierarchy are labeled by black boxes. B) Example code for implementing key algorithms. Such code is simple, but can be sophisticated and arbitrarily complex in NEO models.

SLHs can be created, maintained, and tested by managing model code at three levels of the coding hierarchy. First, alternate *Algorithms* can be inserted into the code hierarchy to incrementally alter existing *Behaviors*. For instance, the potential lines of code produced by a programmer (ΔC_{max}) are often modeled with governing equations [14]. In the NEO model, ΔC_{max} can be converted from a static parameter *Value* to a dynamic state variable simply by adding an *Algorithm* that calculates ΔC_{max} = *f(workDays, sickDays, yearsOfExperience)*. This would alter the *Behavior* to include new dependencies on the variables *workdays, sickDays,* and *yearsOfExperience*, but would require recompiling the package. To avoid the need to recompile code, the *Algorithm* for ΔC_{max} can also be added to a subfolder of the *Behavior* to create a *Deviant* (Figure 5A). Under this scenario, the user can specify either the original *Behavior* or the *Deviant*. An unlimited number of *Deviants* can exist for any *Behavior*, and *Behaviors/Deviants* are specified for *Holons* **at run time**, without recompiling. Thus, *Deviants* allow the user to maintain and execute code that represents alternate SLHs by easily altering/adding/removing model complexity at the granularity of individual algorithms. Second, alternate *Behaviors* can be inserted into the code hierarchy. Both model topology and *Holon* types are defined **at run time** and can be refined without recompiling code. A NEO input table – the "*Holon* type table" – allows users to create different *Cell* and *Edge* types by recombining *Behaviors*. Resulting *Holon* types are referenced in another input table

– the "matrix table" – which describes the desired network topology for a particular simulation and assigns a *Holon* type (and therefore, associated *Behaviors*) to each cell and edge in a network. Thus, Alternate *Behaviors* and *Holon* types provide mechanisms that allow the user to maintain and execute code that represents alternate SLHs by easily altering/adding/removing model complexity at the granularity of individual behaviors and individual holons. Third, *Currency Packages* ("c" in Figure 5A) can be enabled and disabled **at run time** to add or remove a currency from a model. By enabling a currency within a model, the currency *Package* installs additional *Values* and *Algorithms* within *Holons*, potentially converting some existing model parameters (static *Values*) into state variables (dynamic *Values*).

V. CONCLUSIONS

The goal of this research is to produce a robust modeling framework that delegates expertise to domain experts. Clearly, product and process technology, organizational information, and human factors do not remain constant and without the ability to control parameters at finely tuned granular levels, estimation errors increase as models fail to keep up. Rather than to try to anticipate trends in different domain areas [2], [3], we instead focus on providing parsimonious model execution that guarantees consistency and allows for sensitivity analysis among competing hypotheses. Using this research philosophy, the goal for simulation modeling is not to generate parameter sets that allow results from a single model structure to "fit the data," but rather, to contrast conditions under which alternative SLHs (formalized as models) succeed vs. fail to reproduce empirical observations about system behaviors. In doing so, researchers gain critical information to postulate which system dynamics might be missing from, superfluous to, inadequate for, or incorrect within competing SLHs. Subsequently, researchers can use reductionist experiments or reanalyze existing data to test or refine component rules of the SLH, create one or more alternative SLHs that incorporate the refined rule set(s) into a simulation model, and test the new SLH against observations. Such an approach integrates and leverages simulation science to drive discovery and generate, test, and reject SLHs most quickly. In the end, as competing SLHs are rejected and successful SLHs are refined over time, associated models become more reliable representations of the modeled system and can begin to be applied more conventionally to forecast the behavior of the studied system under conditions that have not or cannot be observed directly. We presented models that are simple yet clearly convey the principles behind NEO model building and execution. Significantly more complex models in other domains [8] are available.

ACKNOWLEDGMENT

This research is made possible by the National Science Foundation (NSF) under grant awards 1021001 and EAR-0120523.

REFERENCES

[1] Allen, T.F.H. and T.B. Starr, *Hierarchy: Perspectives for Ecological Complexity*. 1982, Chicago: University of Chicago Press.

[2] Boehm, B., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[3] Boehm, B., Horowitz, E., Abts, A., *Future Trends Implications in Software Cost Estimation Models*. CrossTalk April 2000, p. 4-8.

[4] Brooks, F.P., *The Mythical Man-Month*, Addison Wesley Longman, Reading, Mass., 1975.

[5] Cain, W.C., McCrindle, R.J., *An Investigation into the Effects of Code Coupling on Team Dynamics and Productivity*. Proceedings of the 26th IEEE International Computer Software and Applications Conference. 2002.

[6] Conway, M., *How Do Committees Invent?* Datamation,Vol.14, No. 4, Apr. 1968.

[7] Cowan, G.A., D. Pines, and D. Meltzer, eds. *Complexity: Metaphors, Models, and Reality*. 1994, Perseus Books: Reading, MA. 741.

[8] Helton, A.M., et al. Thinking outside the channel: Modeling nitrogen cycling in networked river ecosystems. Frontiers in Ecology and the Environment.

[9] Jones, C., *Software Cost Estimation in 2002*. 2002. The Journal of Defense Software Engineering, p. 4-8.

[10] Jorgensen, M., *A review of studies on expert estimation of software development effort*. 2004. The Journal of Systems and Software 70: p. 37-60

[11] Proulx, S.R., D.E.L. Promislow, and P.C. Phillips. 2005. *Network thinking in ecology and evolution.* Trends in Ecology & Evolution. 20(6): p. 345-353.

[12] Solé, R.V. and J. Bascompte, *Self organization in complex ecosystems*. 2006, Princeton, NJ: Princeton University Press. 392.

[13] Sterner, R.W., J.J. Elser, and P. Vitousek, *Ecological Stoichiometry: The Biology of Elements from Molecules to the Biosphere*. 2002, Princeton, NJ: Princeton University Press. 584.

[14] Turski, W.M., *Reference Model for Smooth Growth of Software System.* IEEE Transactions on Software Engineering, 22(8), August 1996.

[15] Wu, J. and O.L. Loucks. 1995. From balance of nature to hierarchical patch dynamics: a paradigm shift in ecology. The Quarterly Review of Biology. 70(4): p. 439-466.