The Whole Is More Than the Sum of Its Parts:

Understanding and Managing Emergent Behavior in Complex Systems

Dean M. Morris, Software Engineering Consultant
Kevin MacG. Adams, Ph.D., NCSOSE

Abstract. How does the project or maintenance manager control the unknown? The unknown in this case is the negative or positive behaviors or properties that emerge from a complex software system. The application of systems theory to software is becoming increasingly important as systems become more complex. Looking at a complex software system through the lens of systems science can give the manager the insight needed to understand and control negative or enhance positive emergent behaviors. This article provides an overview of some of the key terms and concepts of systems theory, complexity, and emergence. Both the positive and negative effects of emergent behavior on software systems are considered. Additionally, some speculative and new methodologies for managing undesirable emergent behavior are explored.

Introduction

Imagine a hypothetical scenario where you were contracted to manage the development and maintenance of a new multi-server, web-based software system for the Defense Finance and Accounting Service. The system was tested vigorously and passed all acceptance tests. After several months of operation, the system users noticed increasing lag in the system until it finally locked up. Tests of the individual components of the system indicated there were no problems. It only displayed the lock-up problem when the entire system was online and operating. So, what happened? Why did the system exhibit this negative behavior only after being in operation for some time?

A real-world variation of this scenario at another organization was presented by Mogul [1]. After much troubleshooting by the maintenance team, it was diagnosed that the database server load balancer was set incorrectly. As data was being received, routed, and stored in the databases, the databases' response time increased. The unexpected effect was the system load balancer interpreted the increased database delays as a failure. After the timing expectations of the load balancer were lowered (i.e. the expected response time from the servers was increased), the system functioned well.

In hindsight, it was concluded that the behavior of the load balancer was totally unexpected as it only manifested itself when the entire system was operating. This type of unexpected or emergent behavior in a complex system is one of the key concepts of systems theory [2]. As software systems have become much more prevalent in government, commercial, and peoples' daily lives, so too has the complexity of the systems that support them. The knowledge and application of systems theory and methodologies is becoming increasingly important for the management and maintenance of today's complex software systems.

A Brief Primer of Systems Theory and Emergence

Systems theory [3] provides the underlying theoretical foundation for understanding systems, and as such, serves as the foundation for the purposeful engineering for all complex systems. Knowledge of the systems theory axioms is essential for the modern software project or maintenance manager. While understanding all the basic concepts of systems theory is important in software systems, the concept of emergence or emergent behavior (both positive and negative) is paramount. By its nature, a software system usually only exhibits emergent behavior(s) after the system has been accepted and transitioned to the software maintenance phase. Thus, the thrust of this article is toward managing the maintenance of complex software systems with the potential to exhibit emergent behaviors.

Systems Theory

Systems theory is a unified system of propositions, linked with the aim of achieving an understanding of systems, while invoking improved explanatory power and predictive ability. It is precisely this group of propositions that enables thinking and action with respect to systems [3]. A theory does not have a single proposition that defines it, but is a population of propositions (a model) that provides a skeletal structure for the explanation of real-world phenomena. The relationship between theory and its propositions is not a direct relationship. It is indirect, through the intermediary of the axioms, where the links in the theory represent the correspondence through similarity to the empirical, real-world system. Figure 1 depicts these relationships.

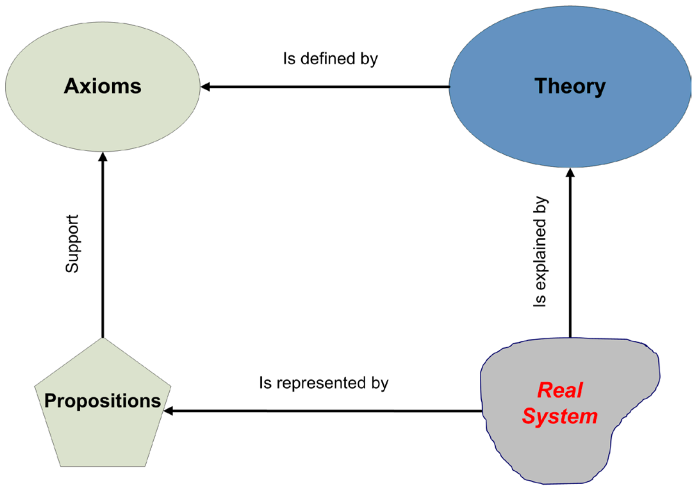

Figure 1: Propositions, Axioms, Theory and the Real World System

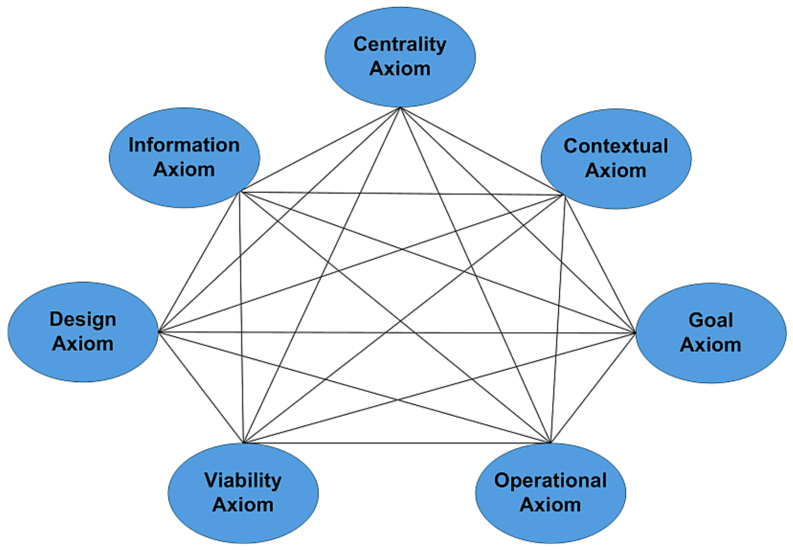

Figure 2: Axioms of Systems Theory

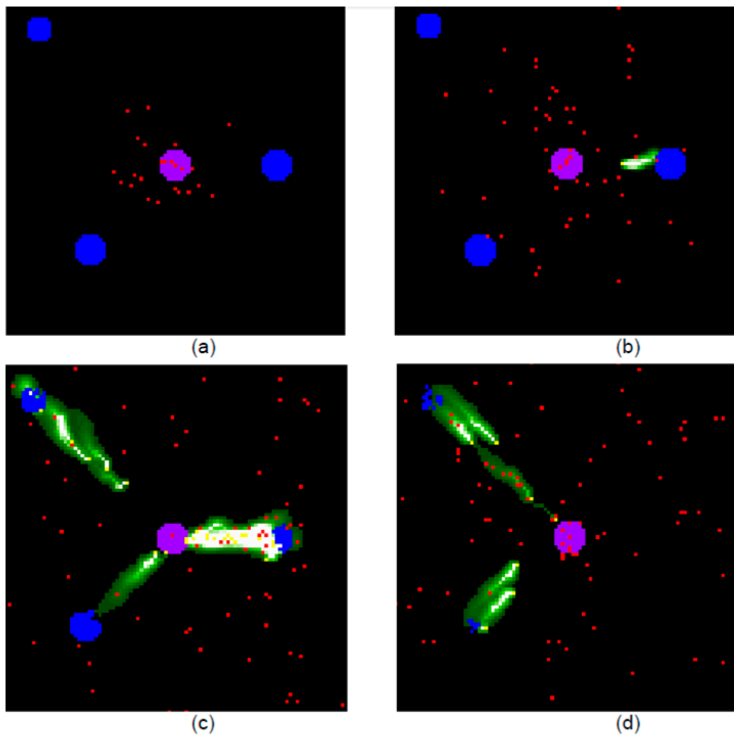

Figure 3: Snapshots from an Ant Colony Simulation [7]

Systems theory provides explanations for real world systems. The explanations increase our understanding and provide improved levels of explanatory power and interpretation for the real world systems we encounter. Our view of systems theory is a model of linked axioms that are represented through similarity to the real system [4]. Figure 2 is a model of the axioms of systems theory. The axioms presented in Figure 2 are called the theorems of the system or theory [5] and are the select set of propositions, presumed true by systems theory, from which all other propositions in systems theory are deducible.

The axioms of systems theory [6] are as follows:

• The Centrality Axiom states that central to all systems are two pairs of propositions; emergence and hierarchy, and communication and control.

• The Contextual Axiom states that system meaning is informed by the circumstances and factors that surround the system. The contextual axiom's propositions are those which give meaning to the system by providing guidance that enable an investigator to understand the set of external circumstances or factors that enable or constrain a particular system.

• The Goal Axiom states that systems achieve specific goals through purposeful behavior using pathways and means. The goal axiom's propositions address the pathways and means for implementing systems that are capable of achieving a specific purpose.

• The Operational Axiom states that systems must be addressed in situ, where the system is exhibiting purposeful behavior. The operational axiom's propositions provide guidance to those that must address the system in situ, where the system is functioning to produce behavior and performance.

• The Viability Axiom states that key parameters in a system must be controlled to ensure continued existence. The viability axiom addresses how to design a system so that changes in the operational environment may be detected and affected to ensure continued existence.

• The Design Axiom states that system design is a purposeful imbalance of resources and relationships. Resources and relationships are never in balance because there are never sufficient resources to satisfy all of the relationships in a systems design. The design axiom provides guidance on how a system is planned, instantiated, and evolved in a purposive manner.

• The Information Axiom states that systems create, possess, transfer, and modify information. The information axiom provides understanding of how information affects systems.

Emergence

Central to the discussion of system theory is the centrality axiom and the principles of emergence and hierarchy. Hierarchy and emergence contribute to complexity because new and interesting properties that cannot be found in the parts emerge and add a whole new dimension to understanding [2]. The father of modern systems theory, Ludwig von Bertalanffy explains that the meaning of the somewhat mystical expression, "The whole is more than the sum of its parts," is simply that constitutive characteristics are not explainable from the characteristics of isolated parts. The characteristics of the complex, therefore, compared to those of the elements, appear as new or emergent [7].

However, Odell [8] noted that complex systems do not have to be complicated to display emergent behavior. In fact, the agents (i.e., the elements) of the system can all be homogenous, follow a simple set of rules, and still exhibit emergent properties. To illustrate the point, Odell [8] described an ant colony computer simulation where each ant agent behaved by the following rules:

1. Wander randomly.
2. If food is found, take a piece back to the colony and leave a trail of pheromones that evaporate over time; then go back to rule 1.
3. If a pheromone trail is found, follow it to the food and then go to rule 2.

Figure 3 shows the simulation display at four stages. The anthill is represented by the purple circle in the center and the three blue dots are the food piles. The ants are the small red dots and their pheromone trails are the white and green areas.

Notice that in Figure 3.a, the ants have individually begun moving away from the colony in a random way. Later in the sequence, in Figure 3.b, some ants have located the food on the right and have started marking their path back to the anthill with pheromones as they carry bits of food. By the time of Figure 3.d, the entire right food pile has been moved to the colony and they are rapidly consuming the other two piles of food.

Relating back to systems theory, while the ants individually behave by a simple rule set, collectively as a colony system, they act in a complex way. They communicate with each other via the pheromone trails, while the pheromones also serve to exert overall system control. That is, when an ant encounters a pheromone trail, it is obligated to follow it to the food, get some food, and return to the anthill marking the path with more pheromones. The resulting emergent property for the ant colony system is a full food storage area. This emergent property cannot be discerned by observing the behavior (via the simple rule set) of individual ants. Only once the ants begin interacting in an environment where there is food and a storage area (i.e. the colony) does the emergent property of food storage become evident.

Effects of Complexity and Emergence on Software Systems

This section moves systems theory into the realm of software systems. Before getting into some positive and negative implications of complexity and emergence within software systems, a more abstract view of the software and its stakeholders will be discussed in a systems theory context.

Evolution of Software

Rajlich and Bennett [9] developed a versioned, staged model for software maintenance because they believed that the more traditional models of maintenance did not accurately capture the evolutionary nature of the software lifecycle. Rajlich and Bennett's [9] maintenance model consists of the following five stages:

1. Initial development: Not maintenance yet.
2. Evolution: Significant changes may be made to a given version to meet changing user needs. The experience of the development team and an adaptable architecture are being leveraged to accommodate the major changes. This stage is iterative for the given version.
3. Servicing: As team experience and knowledge for the version is lost and the code starts to decay; only minor updates are made to the system. This stage is also iterative.
4. Phaseout: No more updates are performed as the system continues to operate.
5. Closedown: The system is retired from service.

In this model, after the initial development, the first version enters the evolution stage where significant updates are made in an iterative fashion. Even as the current version is being supported through the various stages, the development team is evolving the system to the next major version that will enter its own set of stages upon release. For an example, Rajlich and Bennett [8] pointed out that the Microsoft Corporation uses this model for the production, evolution, and support of its operating systems (e.g. Windows XP, Vista, 7, etc.).

Examining this model from a systems perspective, a few observations can be made. Independent of whether or not the actual software exhibits emergent behavior in its operation, the fact that the system is being evolved both within each version and to the next version indicates that there is a complex system involved. The system where the evolution is occurring is at least one hierarchal level up from the software system and includes human agents (i.e. stakeholders such as developers, maintainers, users, etc.) interacting with each other. Also, the environment outside of the open system may be changing (e.g. competition with rival organizations, advancements in technology, etc.), thus causing the system to adapt and evolve.

Positive Emergent Behavior

Moving back down to the software system level, positive emergent behavior represents great potential. Ideally, a developer can design a system so that desired properties emerge while undesired behaviors can be suppressed. This is a difficult task, as emergent behavior is unpredictable by nature. In one research paper, Maciaszek [10] prescribed a meta-architecture for complex software systems that was known to produce the desirable emergent property of adaptability while preventing other properties from emerging. This strategy can be applied to using design patterns to repeat positive results from previous proven systems.

In other research, Olaru, Gratie, and Florea [11] developed a data distribution scheme using a cognitive Multi-agent System (MAS). The overall concept is that simple cognitive agents that have basic goals and behaviors are connected in a network or matrix configuration. Data can be introduced into the system through any of the agents. After the data is introduced into the MAS, it is propagated throughout the system so that it is available to be read from any agent in the system. Thus, individual cognitive agents interacting on the local level produce the emergent property of distributing the data throughout the system without any central control.

Negative Emergent Behavior

Even though positive emergent behavior in software systems holds great promise for the future, the maintenance manager or developer of today will more than likely have to deal with mitigating the undesirable or negative emergent behaviors in complex software systems. How does the maintainer troubleshoot and repair a problem that does not originate in the code, but instead originates in the interactions between agents in the system? Mogul [1] proposed a research agenda to come to grips with the negative emergent property (i.e. misbehavior) problem in the software industry. Agenda items include:

1. Creating a taxonomy of emergent misbehavior.
2. Creating a taxonomy of typical causes.
3. Developing detection and diagnosis techniques.
4. Developing prediction techniques.
5. Developing amelioration techniques.
6. Developing testing techniques.

Mogul [1] provided preliminary taxonomies of emergent misbehavior and typical causes in the paper, but indicated that the other four technique categories would be much more challenging to develop and implement. Work in these categories is ongoing as is evident with the following.

Managing Negative Emergent Behavior in Software

Software maintenance is already difficult enough in regular systems, let alone in complex software systems with emergent behavior. Software project managers can benefit by keeping up with the software industry literature to gain insight into potential methods for mitigating negative emergent behavior.

Toward Self-maintaining Systems

Pertaining to the software systems of the near future, Gabriel and Goldman [12] wrote:

"Future innovations in software will need to produce systems that actively monitor their own activity and their environment, that continually perform self-testing, that catch errors and automatically recover from them, that automatically configure themselves during installation, that participate in their own development and customization, and that protect themselves from damage when patches and updates are installed. Such systems will be self-contained, including within themselves their entire source code, code for testing, and anything else needed for their evolution."

To meet these goals, Gabriel and Goldman [12] proposed a hypothetical hybrid autopoietic and allopoietic system. According to Gabriel and Goldman [12], an autopoietic system is one that is continually re-creating itself and allopoesis is the process whereby a system produces something other than the system itself. In essence, the autopoietic part of the system would concentrate on keeping the system viable via monitoring the system health and taking corrective action if a system-threatening problem developed (e.g. a negative emergent behavior). The allopoietic part of the system would operate as programs do today (i.e., perform the functions of the system).

Verifying Complex Systems Through Formal Methods

NASA's answer for dealing with undesirable emergent behavior in a complex system may lie with verification through a formal methods cocktail. Rouff, Hinchey, Truszkowski, and Rash [13] reported on research into the viability of utilizing formal methods to verify the emergent behavior of the Autonomous Nano-Technology Swarm (ANTS) mission that may be used to explore the asteroid belt.

Basically, the mission entails 1,000 two-pound autonomous space vehicles that will be transported to the edge of the asteroid belt. From there the ANTS will self-organize into exploration teams with leaders. Various instruments will be used to collect data from asteroids that will be periodically transmitted back to earth. For autonomous operation, the ANTS will need to exhibit the properties of self-configuration, self-optimization, self-healing, and self-protection. Because the ANTS mission will potentially depend on certain positive emergent behaviors to operate while not developing any negative attributes, many formal methods and techniques were considered for the verification of this intelligent swarm. After the evaluation, the research team settled on a combination of four current formal methods that they plan to integrate into one method that is best suited to verifying the behavior of intelligent swarms. It is conceivable that a similar combination of formal methods could be used to verify other complex software systems to prevent negative properties from emerging while grooming desired emergent behaviors.

Repairing Emergent Behaviors Through Runtime Feedback

Lewis and Whitehead [14] have conceded that many emergent behaviors simply cannot be detected using testing or other verification techniques. To combat this problem, they developed a system for detecting and repairing undesirable emergent behavior at runtime. The main component of the system is a runtime monitor named Mayet. The program they experimented with was a variation of the game Super Mario Brothers.

For the system to work, rules were input into Mayet so it would know what undesirable behaviors to look for (e.g. the character gets stuck in an on-screen object, jumps too high, etc.). Upon detecting an error, Mayet sends a message to the game. After the game receives the message, the game repairs the problem almost instantaneously. A major catch is that the repair routines have to be built into the game. This seems problematic because the developer has to anticipate the possible repairs that may be required while the types of behavior that are supposed to be fixed are emergent, thus difficult to predict. Regardless, the concept of repairing an emergent problem in a software system as it is operating is a step in the right direction.

Conclusion

The knowledge and application of systems theory and methodologies has become increasingly important for the management and maintenance of today's increasingly complex software systems. The promise and the problem of emergent behavior in complex software systems is a double-edged sword. Those that chose to ignore the implications of systems science and emergent behavior will be relegated to a reactionary role. Project and maintenance managers who embrace the systems science viewpoint will be much better prepared to be proactive in controlling their software systems. The government and civilian software engineering communities will need to gain a deeper understanding of how to capitalize on the synergies that positive emergent properties can provide while reliably excluding negative emergent behaviors from software systems.

ABOUT THE AUTHORS

Dean M. Morris is a software engineering consultant who recently developed software supporting research at the National Centers for System of Systems Engineering (NCSOSE). He retired from the Air Force after serving in the communications-electronics maintenance field for 21 years. Morris holds a B.S. in Computer and Information Science (with a minor in Finance) and an M.S. in Software Engineering from the University of Maryland University College.

2765 Hamilton Road
Waldorf, MD 20601
E-mail: deanmorris@ieee.org

Dr. Kevin MacG. Adams is a Principal Research Scientist at the National Centers for System of Systems Engineering (NCSOSE). Dr. Adams is a retired Navy submarine officer and information systems consultant. He was on the faculty at Old Dominion University from July 2007 until coming to NCSOSE in January 2009. Dr. Adams holds a B.S. in Ceramic Engineering from Rutgers University, an M.S. in Naval Architecture and Marine Engineering and an M.S. in Materials Engineering both from MIT, and a Ph.D. in Systems Engineering from Old Dominion University.

4111 Monarch Way, Suite 406
Norfolk, VA 23508-2563
E-mail: kmadams@odu.edu

REFERENCES

1. Mogul, Jeffrey C. “Emergent (Mis)Behavior vs. Complex Software Systems.” ACM SIGOPS Operating Systems Review 40.4 (2006): 293-304.
2. Flood, Robert, and E. Carson. Dealing with Complexity: An Introduction to the Theory and Application of Systems Science (2nd Ed.). New York: Plenum Press, 1993.
3. Adams, K. MacG. “Systems Principles: Foundation for the SoSE Methodology.” International Journal for System of Systems Engineering 2.2/3 (2011): 120-55.
4. Giere, Ronald N. Explaining Science: A Cognitive Approach. Chicago: University of Chicago Press, 1988.
5. Honderich, T., ed. The Oxford Companion to Philosophy (2nd Ed.). New York: Oxford University Press, 2005.
6. Adams, K. MacG., Hester, P. T., Meyers, T. J., Bradley, J. M. and Keating, C. B. Systems Theory as the Foundation for Understanding Systems (NCSOSE Position Paper 2012-001). Norfolk, VA: National Centers for System of Systems Engineering, 2012.
7. Bertalanffy, L. von. General System Theory: Foundations, Development, Applications (Rev. Ed.). New York: George Braziller, 1968.
8. Odell, James. “Agents and Complex Systems.” Journal of Object Technology 1.2 (2002): 35-45.
9. Rajlich, V. T., and K. H. Bennett. “A Staged Model for the Software Life Cycle.” Computer 33.7 (2000): 66-71.
10. Maciaszek, Leszek A. “Modeling and Engineering Adaptive Complex Systems.” Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling - Volume 83. 1386961: Australian Computer Society, Inc., 2007.
11. Olaru, Andrei, Cristian Gratie, and Adina Magda Florea. “Emergent Properties for Data Distribution in a Cognitive Mas.” Computer Science & Information Systems 7.3 (2010): 643-60.
12. Gabriel, Richard P., and Ron Goldman. “Conscientious Software.” ACM SIGPLAN Notes 41.10 (2006): 433-50.
13. Rouff, Christopher, et al. “Experiences Applying Formal Approaches in the Development of Swarm-Based Space Exploration Systems.” International Journal on Software Tools for Technology Transfer (STTT) 8.6 (2006): 587-603.
14. Lewis, C., and J. Whitehead. “Repairing Games at Runtime or, How We Learned to Stop Worrying and Love Emergence.” Software, IEEE 28.5 (2011): 53-59.