INTERNATIONAL DESIGN CONFERENCE - DESIGN 2014
Dubrovnik - Croatia, May 19 - 22, 2014.

DESIGN 2014

CONCEPTUAL BLIND SPOTS IN COMPLEX SYSTEM ENGINEERING PROJECTS – A COMPUTATIONAL MODEL

R. C. Thomas and J. S. Gero

Keywords: agent-based modelling, design cognition, trade space, design dependencies

1. Introduction

Within the setting of complex technologies and organization networks, the goal of this research is to explore how conceptual blind spots emerge from the interaction of misaligned design organization cultures. Our interest is in the social and cognitive mechanisms involved and possible interventions that would reduce the odds of the emergence of blind spots that could lead to serious errors, accidents, or missed objectives. In this initial stage of research the goal is to demonstrate the viability of Agent-based Modelling (ABM) to study system engineering in multi-team R&D projects, and specifically how incongruous value systems and conceptual schemata cause blind spots to emerge and persist, increasing the likelihood of a program missing it's performance goals.

In complex systems, it is common for components to have non-trivial influences on system and subsystem behaviours and performance variables. Influences can be intended or unintended, and also direct or indirect. One of the major goals of system engineering is to anticipate these influences early in the program lifecycle or to discover them and help manage them as the program nears completion. To "manage" these influences and interdependencies means making trade-off decisions in the multi-dimensional space of performance ("trade space") and also trying to limit the disruptive effects of design changes.

Our hypothesis is that conceptual blind spots emerge when the value systems and conceptual schemata of collaborating teams are incongruous – i.e. work at cross-purposes to each other – primarily in low-visibility domains that are far from the main priorities of each organization [Masys 2012]. Furthermore, we hypothesize that these incongruities persist and the blind spots remain undiscovered because there is a short-term benefit to each organization, namely preserving and reinforcing its core value system and conceptual schemata [Hedberg 1981]. As a consequence, blind spots discovered late in the program life cycle lead to cascading design changes, which lead to missed schedules, missed performance goals, and/or cost over-runs. Furthermore, the undesirable effects of blind spots increase as the program nears completion, and this manifests in the form of organizational pressure to meet specific goals at the expense of others. For example, pressure to reduce weight while facing schedule pressure can lead designers to increase unit cost by switching to a lighter but more expensive material – a "quick fix" compared to a time-consuming component redesign.

This is a difficult research problem to investigate because it involves the interplay between the technical aspects of design and the social aspects of design teams. Case study methods have been used to study particular designs and teams in depth. However, it is not feasible to perform case studies on a large number of cases or to control conditions to study alternative interventions.

Methods for analysis of system interdependencies have been in use for several decades [Steward 1981], [Browning 2001]. However, system engineering methods applied to complexity management

generally focus on specific development tasks (e.g., identification of conflicting requirements) or particular design objectives (e.g., product modularization) [Lindemann et al. 2009], and they do not directly model cognitive and social factors that can give rise to blind spots [Browning 2001], [Maurer et al. 2006].
A new field has developed that focuses on modelling social behaviour called "computational social science". Computational social science models social interactions and simulates the resulting social behavior through the use of computational agents rather than equation-based methods [Gilbert and Doran 1994], [Gilbert and Conte 1995], [Epstein and Axtell 1996], [Casti 1999], [Castelfranchi 2001], [Macy and Willer 2002], [Epstein 2007], [Miller and Page 2007].
Computational agents are encapsulated computer programs that respond and behave autonomously [Jennings and Wooldridge 1998], [Ferber 1999], [Weiss 2000], [Wooldridge 2002]. Agent-based Modelling (ABM) is used since it allows explicit modelling of technological and sociological dynamics over a broad range of settings and conditions. It is even possible to perform controlled experiments to evaluate the effectiveness of alternative interventions. The main challenge for ABM for this purpose is to design the ABM to be rich enough to model the essence of the phenomena while keeping it simple enough to understand and to provide credible research results. The main claim of this paper is that the ABM we have developed achieves these goals.
The paper is organized as follows. Section 2 describes the modelling and simulation methods used in this research, focusing on conceptual issues. Section 3 describes the setting for our simulation – a space launch system – and the relevant characteristics of this setting, such as performance dimensions, subsystems, dependence between subsystems, the operational and cognitive meaning of blind spots, processes to discover blind spots, and the influence of performance pressure in the context of the program lifecycle. Section 4 describes the interface for the simulation software. Section 5 presents exemplary results with the goal of demonstrating the viability of this approach to research. The paper closes with a discussion of the contributions and significance of this research and directions for future work.

2. Method

The method for modelling and simulation involves four aspects: 1) focal behaviour , 2) ontology and representation, 3) agency and agent capabilities, and 4) software implementation.

2.1 Focal behaviour

The ultimate behaviour of interest is the effect of conceptual blind spots and change propagation on the overall goals for a system and its design and development program. Therefore, we aim to model and portray those as directly as possible and abstract away as many other details of system design and system engineering as we can. In particular, we chose a very abstract way to represent and model the design process for components and system relationships. Also we do not attempt to simulate the performance of the designed artefact in any physical or informational sense. Instead, performance variables are calculated based on more abstract representations of the system design.

2.2 Ontology and system representation

The ontology of a model is the way it represents the phenomena of interest, and especially how elements are defined and related to each other. We represent a complex system design program using a Component-Behaviour-Performance ontology, Figure 1. The results of the program are characterized by "Performance Variables" related to the program itself (e.g. program cost and schedule) as well as the system that is being designed. Each Performance Variable is determined by the design choices and their interdependencies. Together the Performance Variables constitute the "trade space" of the program, meaning that agents often face trade-off decisions in their design where one Performance Variable is sacrificed (e.g. unit cost) in favour of another (e.g. reliability). The basic element of design is a "Component" which has a cost and component-level performance. Components work together to yield "Behaviours", which are changes in physical state or informational state that relate to a Performance Variable.

For simplicity, we represent each of these variables as a bounded positive integer, 0 to 100. Any given Performance Variable is determined by a weighted sum of the Behaviours that relate to that Performance Variable, with the weights given by Nature, simulated by random initialization. In this simplified world, the act of designing consists of searching for Components that can be mapped to Behaviours in such a way to yield the desired Performance. The mapping between Components and Behaviours is represented as a bi-partite graph that effectively creates dependencies (positive or negative) between Behaviours. In a matrix with a row and column for each Behaviour in sequence, these dependencies can be portrayed and analyzed as a Design Dependency Matrix [Clarkson et al. 2004]. Design Dependency Matrices have been used frequently in the research literature for both Product Design and Management Science to study modularity and interdependence in designs. [Gero and Maher 1992] present a complete dependency network model to link Components to Behaviours, and Behaviours to Performance Variables.

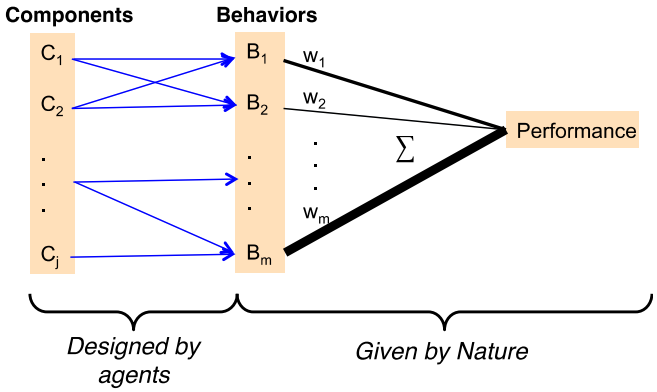


Figure 1. Component-Behaviour-Performance Ontology

2.3 Agency and agent capabilities

Agents in an Agent-based Model (ABM) are those entities that sense the environment, including other agents, and take action, possibly learning along the way. While it is common to choose to model individual people at a single level of organization, we have chosen a three-level hierarchy of agents that more closely matches the structure of design teams in a large program: 1) Programs, 2) Teams, and 3) Activities.

Programs consist of Teams that persist throughout a run. Teams consist of Activities that have a single task – to produce a design for a particular Component or to produce a mapping between Components and Behaviours. New Activities are spawned when a new or improved Component or mapping is needed by the Team. Activities have a finite life depending on the resources devoted to them and they are terminated either when they produce a design or when they run out of resources.

Agency at the Program level is essentially equivalent to a Program Manager. The Program monitors the program goals for each Performance Variable relative to goals and allocates resources to Teams to achieve those goals. The Program agent also can direct teams to change focus or actions when the Program is under pressure (see Section 3.3).

Agency at a Team level is essentially equivalent to an Engineering Manager for a specific subsystem or function within a subsystem. While our simulation can support many teams per subsystem, the current implementation uses just three teams, one for each subsystem.

Most of the intelligence in our model is at the Team level. Teams spawn Activities to design new components or refine existing components to meet the Team's goals, focusing on most important components (narrowly defined), and increasing the Team's expertise in the process. Thus, Teams accumulate knowledge and experience during the course of a run that enable them to produce Component designs of increasing sophistication. If a focal variable for the Team plateaus, the Team will spawn sub-components, at the cost of increased complexity in the subsystem. If the Team is under performance pressure, the Team will try more radical component designs and mapping structures.

In addition to these core design capabilities, Teams also have the vital capability to discover design dependencies, in three ways: 1) During routine design (low cost but also low probability), 2) Explicit search (higher probability, but costly), and 3) Learn from other teams (costly, and also dependent on what other teams know).
Team preferences for each of these three discovery methods are under experimental control.
Agency at an Activity level is intentionally abstract and simplistic because our focus lies at the Team and Program levels. Activities perform a blind search through recombination of abstract "raw material", with preference given to combinations that have been successful in the past. As they do they consume resources until the goal has been reached or the resource allocation is exhausted.
An important feature of our agency model is the specification of "value systems", both at a Program level and at a Team level, both under experimental control. At a Program level, the values are expressed using goals for each Performance Variable, and a budget for both Schedule and Program Cost. At a Team level, the value system is expressed by weights given to each of the Performance Variables. Because these weights are under experimental control, it is possible to test a range of conditions from "no commonality" (i.e. there is no commonality between the weights of the teams) to "full commonality" (i.e. the weights of all teams are the same, and balanced across the performance variables). The extreme of "no commonality" corresponds to a maximum of isolation between teams, promoting gaps but also promoting self-interested, focused decision-making. The opposite extreme of "full commonality" corresponds to maximum alignment between teams but also least specialization and least focus in design decision-making.

2.4 Software implementation

NetLogo was chosen as a simulation platform. It is written in Java, has a simple command language, and a graphical user interface that is easy to design and program. The simulation was custom designed, with some inspiration from the SKIN model by [Gilbert et al. 2010].

3. Setting

The setting chosen for our research is an idealized space launch system, consisting of the functional subsystems of a rocket plus performance dimensions related to system support such as safety, reliability, and maintainability. The design program is idealized as a single program manager and a collection of independent teams working on individual subsystems.

3.1 Subsystems and performance dimensions

The three subsystems in the idealized model are 1. Control (e.g. guidance, staging, engine control, sensors, safety triggers, etc.), 2. Structure (e.g. fuel storage, payload, external skin, barriers between components and modules, interconnections, etc.) and 3. Propulsion (main rocket engine, guidance engines, etc.). These three subsystems contribute variously to ten Performance Variables (Figure 2).

	1. Control	2. Structure	3. Propulsion
Unit Cost	✔	✔	✔
Maintainability	✔		✔
Safety	✔	✔	✔
Program Cost	✔	✔	✔
Schedule	✔	✔	✔
Stability	✔		
Capacity		✔	
Reliability	✔	✔	✔
Thrust			✔
Weight		✔	✔

Figure 2. Subsystems and their contribution to Performance Variables

Each of the three subsystems have a Performance Variable that is theirs uniquely (shaded cells in Figure 2) and these can be considered their main purpose and value in the program. There are also

several Performance Variables that are influenced by two subsystems but not all three (e.g. Weight). When teams focus narrowly on their unique Performance Variable, they are more likely to achieve their goal but they are also likely to cause the program as a whole to miss goals for other Performance Variables.
In addition to dependence created by shared Performance Variables, there can also be dependencies between subsystems via the design choices for Components and their mapping to Behaviours that support the prime Performance Variables, as shown in the example in Figure 3.

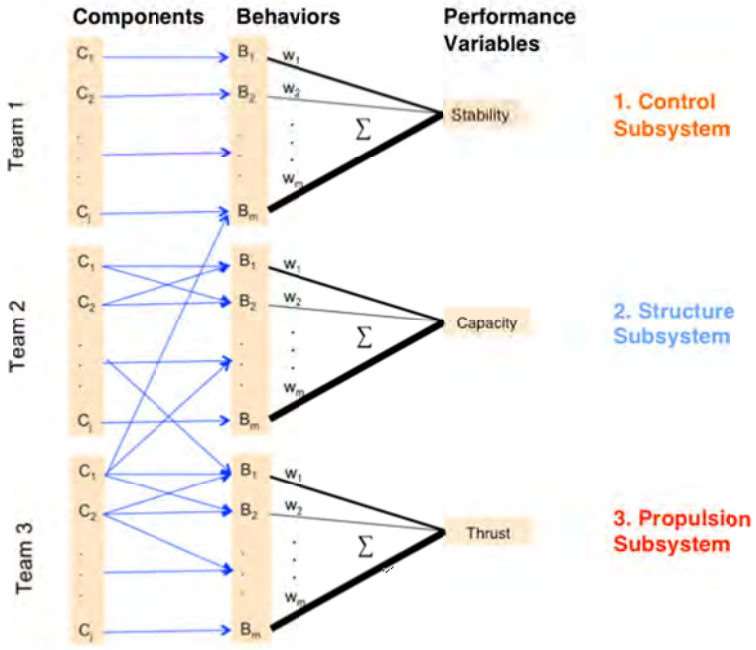


Figure 3. Dependence between subsystem due to design choices for mapping Components to Behaviours

3.2 Blind spots

The map between Components and Behaviours includes two types of relations – intended and unintended. 'Intended' relations are intentionally chosen by design agents, while 'unintended' relations are side effects or implications that are not initially known by design agents. Any unintended relation that has a major effect on a Performance Variable is defined as a 'blind spot'. They are called 'blind' because they are not factored into the design agent's decision process. The more blind spots that exist, the more the Performance Variables will deviate from the collective expectations of design agents. Blind spots can either be discovered in the course of core design tasks (with a probability controlled by experimenters) or can be discovered by explicit search. Because it can take time and resources away from the core design tasks, the search for blind spots can be seen as an exploration/exploitation trade-off.

3.3 Performance pressure and the program lifecycle

As complex programs near critical deadlines (e.g. schedule milestones or budget thresholds), organizational pressure to meet key goals can increase. Any goal that is far from its target could become the object of this pressure – something to achieve "at all costs". To model this as simply as possible, we have implemented a penalty system for specific Performance Variables depending on the type of pressure that arises. 'Pressure' is a separate variable that is a function of time remaining in the program schedule and the gap between goal and actual for a given Performance Variable (Table 1).
The table entries can be read this way (first row): "When there is pressure to reduce unit cost (left side), there is a penalty imposed which increases Schedule time consumed, and reduces Safety, Reliability and Maintainability". If pressure is felt only in one dimension (e.g. Safety) and the other

dimensions are exceeding their goals, then the penalty will have no affect on the design. The reason for this is that the Program as a whole can absorb this trade-off and still meet it's overall goals. However, pressure is felt in several dimensions, the penalties can lead to under-performance in those dimensions and therefore trigger design changes. These design changes can cascade and result in programs that fail to meet one or more goals for Performance Variables before completion.

Table 1. How performance pressures map to performance penalties

Pressure	⇒	*Penalties*
↓ Unit Cost	⇒	↑ Schedule ∧ ↓ Safety ∧ ↓ Reliability ∧ ↓ Maintainability
↓ Schedule	⇒	↓ Safety ∧ ↓ Reliability ∧ ↓ Maintainability
↑ Safety	⇒	↑ Unit Cost ∧ ↑ Weight ∧ ↑ Schedule
↑ Reliability	⇒	↑ Unit Cost ∧ ↑ Weight ∧ ↑ Schedule
↑ Maintainability	⇒	↑ Unit Cost ∧ ↑ Weight ∧ ↑ Schedule
↑ Thrust	⇒	↑ Unit Cost ∧ ↑ Weight
↑ Capacity	⇒	↑ Unit Cost ∧ ↑ Weight
↑ Stability	⇒	↑ Unit Cost ∧ ↑ Weight
↓ Weight	⇒	↑ Unit Cost

4. Simulation system interface

The interface to the simulation system is shown in Figure 4. On the left side are control sliders for goals for each of the ten Performance Variables. Next to each slider is a box that shows the actual value of that variable, given the current design. To the right is an other box that shows the value of the Pressure variable associated with each Performance Variable. (For simplicity, one Pressure variable is defined for the combination of Safety, Reliability, and Maintainability. In the centre is a Design Dependence Matrix with a row and column for each Behaviour that contributes to the prime Performance Variable in each Subsystem. There are three graphs on the right that portray the time series related to overall program performance.

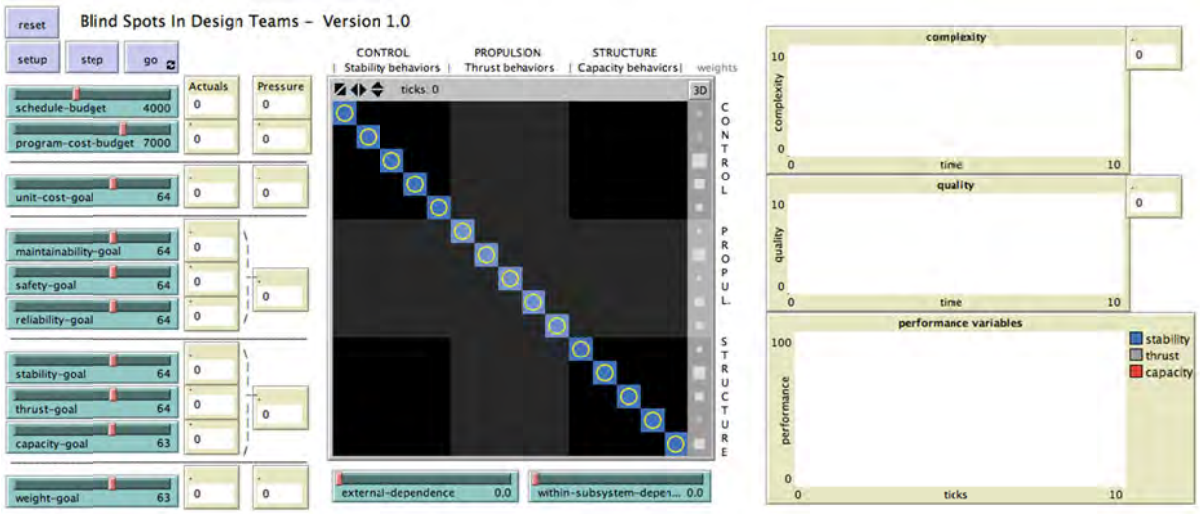


Figure 4. Simulation system interface

Figure 5 shows a Design Dependency Matrix after initialization. Three pieces of information are coded in each cell in the matrix. First is the direction of dependency between a pair of Behaviours, either positive or negative. Second, the size of the circle is proportional to the degree of dependence. (Hollow circles on the diagonal indicate trivial dependence.) Third, the shading of the cell indicates whether any dependence is known or unknown. There is a grey column on the right that indicates the

weights associated with each Behaviour (determined by Nature, simulated by random initialization), with the size of the light grey square being proportional to the weight. Blind spots only arise when the weight for the Behaviour is high, the degree of dependence is high and the dependence is unknown.

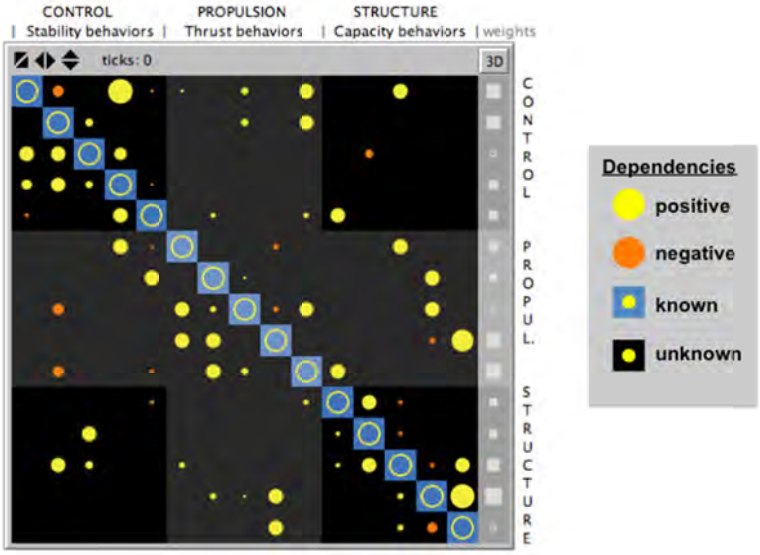
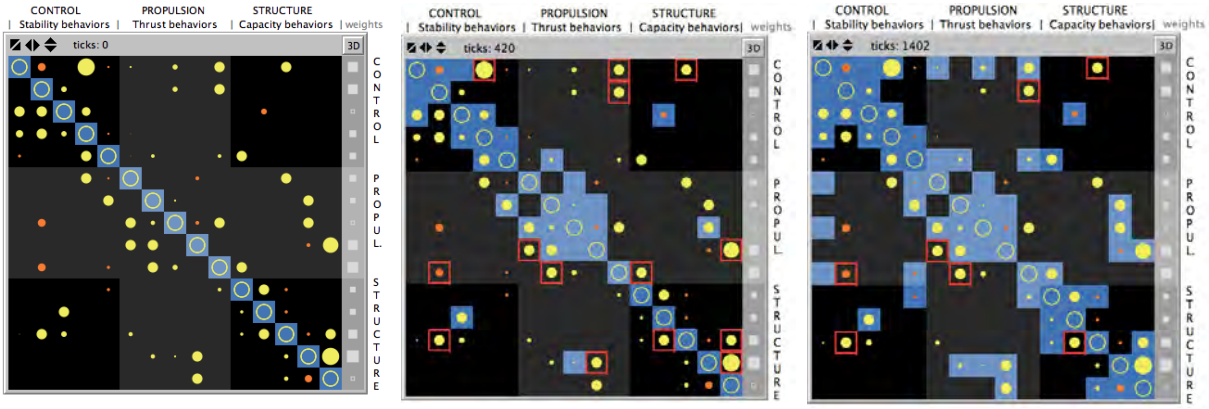


Figure 5. Design Dependency Matrix after initialization, with key on the right

Figure 6 shows the evolution of the Design Dependency Matrix during a typical run, with initial conditions of "external-dependence" = 0.2 and "internal-dependence" = 0.5. At Figure 6(a) t = 0, no dependencies are known other than the diagonal. At Figure 6(b) t = 420, many dependencies have been discovered, but these are mostly within each subsystem. Blind spots are indicated by rectangles around cells. At Figure 6(c) t = 1402, many more dependencies have been discovered but several blind spots remain.



(a) t = 0 **(b)** t = 420 **(c)** t = 1402

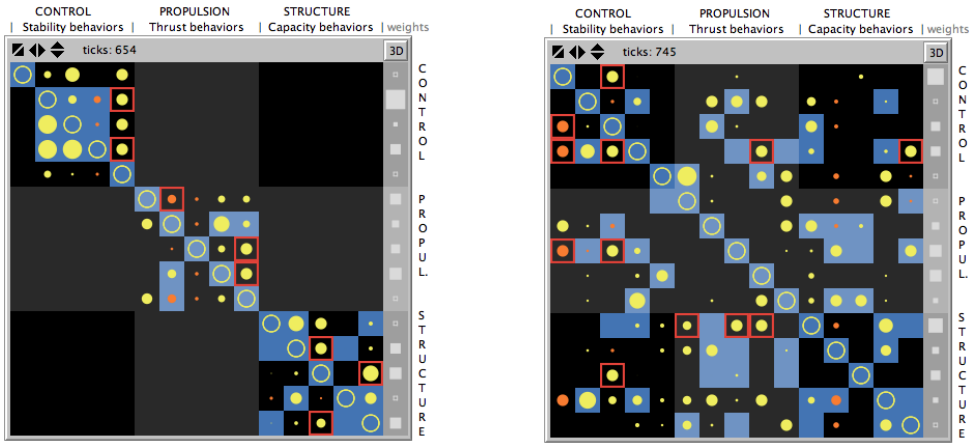
Figure 6. Discovery of dependencies and elimination of blind spots in a typical run

The degree of modularity is determined by the number of Components that map to more than one Subsystem, specifically to Behaviours within each. This is under experimental control, and is can be seen in the Design Dependency Matrix, Figure 7.

5. Exemplary results

The simulation software is currently under development and we are beginning to get some early results using limited functionality. Specifically, the results discussed here include three primary performance dimensions – one for each subsystem – plus one common dimension, Reliability. Also, the Pressure variables and Team behaviours are not yet turned on. The focus is on how Teams shift

their attention from core design tasks ("exploitation") to searching for blind spots ("exploration"), and how this affects discovery of blind spots and also how it affects their ability to achieve the Program's goals.



(a) High modularity: high dependence *within* each subsystem

(b) Low modularity: high dependence *between* subsystem and teams

Figure 7. Differences in modularity for two different runs, under experimental control

Figure 8 shows results from two different experimental treatments under two different levels system design challenge (modularity) with the same performance goals.

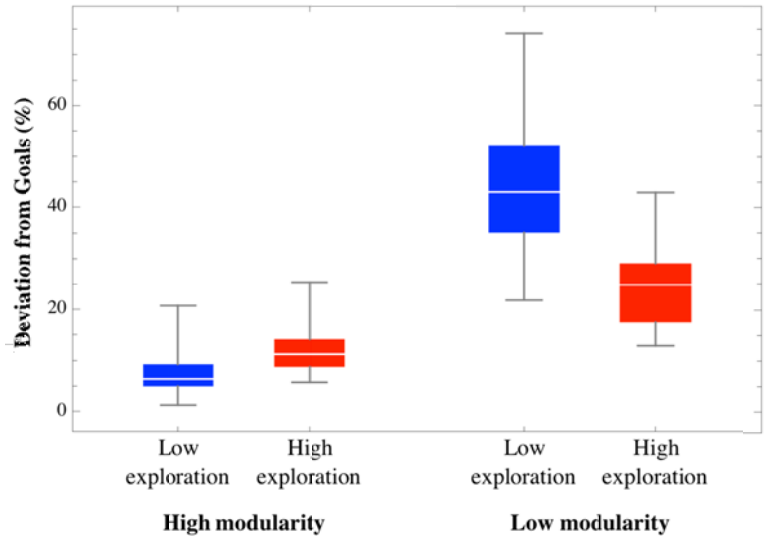


Figure 8. Results for "low exploration" vs. "high exploration" treatments in two settings with different degrees of modularity. $N = 60$ runs for each treatment and setting, with fixed time limit of 5,000 steps

All runs were performed with a fixed time limit, chosen to be a "tight schedule", meaning that it would be unlikely for the team to meet all goals within that time limit. For each run, any performance goals that are not met are measured by the percentage deviation from that goal vs. actual, and the average is recorded. A value of "0%" means all performance goals have been met.

These results show that when the system design challenge is "easy" (high modularity), the "low exploration" treatment out-performs the "high exploration" treatment. The reason is that there are fewer blind spots, these blind spots have less effect on the Team's ability to reach it's goals, and the

blind spots are more likely to be found during core design tasks. The opposite is the case with low modularity.
Though not surprising, these results demonstrate that the ABM simulation realistically models the interplay between the cognitive/social aspects and the technical aspects of the complex system design program. This prepares us for more complicated experiments and analysis.

6. Discussion

The contribution of this paper is to show the viability of agent-based modelling (ABM) to study the interplay between cognitive/social aspects and technical aspects of complex system design programs. The phenomena of blind spots in systems design arise at the interface between these two aspects, and therefore it is essential to include both in the simulation. Our ABM simulation includes models of Team values (i.e. the weights they place on the program performance dimensions) and Team focus (i.e. on core design tasks or on blind spot search), and also the structure and interdependence of the system being designed.
When the simulation is complete, we expect to be able to produce the following results:

- Identify behavioural signatures of undetected design dependencies.
- Estimate the value of early detection of blind spots vs. late detection.
- Evaluate alternative interventions for inter-team communication (e.g. formal vs. informal) and staffing (e.g. rotation) and others.
- Dock with empirical research (e.g. case studies in manufacturing industries [Clarkson et al. 2004]).

For future research, we intend to conduct basic computational experiments to evaluate the effect of alternative agent rules for blended exploration/exploitation strategies. Also we would like to perform experiments to evaluate the effects of alternative interventions such as program management practices or team structures.

Acknowledgement

This research has been supported by NASA grant NNM11AA01A-SUB2012-038 and by National Science Foundation grants CMMI-1400466 and CMMI-1400466. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NASA or the NSF.

References

Browning, T. R., "Applying the design structure matrix to system decomposition and integration problems: a review and new directions", Engineering Management, IEEE Transactions on, Vol. 48, No. 3, 2001, pp. 292-306.

Castelfranchi, C., "The theory of social functions: challenges for computational social science and multi-agent learning", Cognitive Systems Research Vol. 2, No. 1, 2001, pp. 5-38.

Casti J., "The computer as laboratory: Toward a theory of complex adaptive systems", Complexity Vol. 4, No. 5, 1999, pp. 12-14.

Clarkson, P. J., Simons, C., Eckert, C., "Predicting change propagation in complex design", Journal of Mechanical Design, Vol. 126, No. 788, 2004.

Epstein, J. M., "Generative Social Science: Studies in Agent-Based Computational Modeling", Princeton University Press, 2007.

Epstein, J. M., Axtell, R., "Growing Artificial Societies: Social Science from the Bottom Up", Brookings Institute Press, 1996.

Ferber, J., "An Introduction to Multiagent Systems", Addison-Wesley, 1999.

Gero, J. S., Maher, M. L., "Mutation and analogy to support creativity in computer-aided design", in G. N. Schmitt (ed.) "CAAD Futures '91", Vieweg, Wiesbaden, 1992, pp. 261-270.

Gilbert, G. N., Conte, R., "Artificial Societies: The Computer Simulation of Social Life", UCL Press, 1995.

Gilbert, G. N., Doran, J., "Simulating Societies: The Computer Simulation of Social Phenomena", UCL Press, 1994.

Gilbert, N., Ahrweiler, P., Pyka, A., "The SKIN (Simulating Knowledge Dynamics in Innovation Networks) model", Working Paper, University of Surrey, University College Dublin and University of Hohenheim, 2010.

Hedberg, B., "How organizations learn and unlearn", In P. C. Nystrom & W. H. Starbuck, eds. "Handbook of Organizational Design: Volume 1: Adapting Organizations to their Environments". Oxford University Press, USA, 1981.

Jennings, N., Wooldridge, M. J., "Agent Technology: Foundations, Applications, and Markets", Springer, Berlin/Heidelberg, 1998.

Lindemann, U., Maurer, M., Braun, T., "The procedure of structural complexity management", in "Structural Complexity Management", Springer, Berlin/Heidelberg, 2009, pp. 61-66.

Macy, M. W., Willer, R., "From factors to actors: Computational sociology and agent-based modeling", Annual Review of Sociology Vol. 28, 2002, pp. 143-166.

Masys, A. J., "Black swans to grey swans: revealing the uncertainty", Disaster Prevention and Management, Vol. 21, No. 3, 2012, pp. 320–335.

Maurer, M., Pulm, U., Ballestrem, F., Clarkson, J., Lindemann, U., "The Subjective Aspects of Design Structure Matrices: Analysis of Comprehension and Application and Means to Overcome Differences", In "ASME 8th Biennial Conference on Engineering Systems Design and Analysis", American Society of Mechanical Engineers, 2006, pp. 869-878.

Miller, H. J., Page, E. S. "Complex Adaptive Systems: An Introduction to Computational Models of Social Life", Princeton University Press. Princeton, 2007.

Steward, D. V., "The design structure system: a method for managing the design of complex systems", Engineering Management, IEEE Transactions on, Vol. 28 No. 3, 1981, pp. 71-74.

Weiss, G., "Multiagent Systems", MIT Press, 2000.

Wooldridge, M., "An Introduction to Multi-Agent Systems", Wiley, 2002.

Russell C. Thomas, PhD Student
George Mason University, Krasnow Institute of Advanced Study, Department of Computational Social Science
4400 University Drive, MS 6B2, Fairfax, VA, 22030, USA
(703) 993-9298
Email: russell.thomas@meritology.com
URL: http://www.css.gmu.edu/node/8?q=node/104