

Is Traditional System Engineering Right for Engineering System-of-Systems?

Bret Michael

Naval Postgraduate School
Monterey, California



Disclaimer



- ◆ The views and conclusions in this talk are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.



Traditional system engineering

- ◆ Based on the premise that engineers can build complete system if provided a complete set of requirements
 - Assume that operational concepts are completely known during requirements elicitation
- ◆ Focus is on
 - Engineering single systems with fixed boundaries
 - Hardware integration first, with software addressed afterwards



Development “truths”

- ◆ Increasingly, software concerns are overtaking hardware concerns in system engineering, with software
 - Determining the degree of achieved success in the fielded system’s capabilities
 - Consuming the majority of costs for system development
 - Changing most often in the acquisition lifecycle of a system to meet the changing needs of the customer



Information-centric competition

- ◆ Trend toward hooking up systems into a system-of-systems (SoS) order to compete
 - An amalgamation of legacy systems and developing systems that provide an enhanced capability greater than that of any of the individual systems within the system-of-systems
- ◆ Green fields development is oftentimes not practical or discouraged



Engineering systems-of-systems

- ◆ Systems-of-systems are open systems
 - Operational concepts are not completely known during requirements elicitation—there is a lack of stability
 - No fixed system boundaries
 - Individual systems are operationally and managerially independent
- ◆ Emergent behavior
 - Each legacy system exhibits individual and unique emergent behavior
 - A system-of-systems will exhibit emergent behavior that is not predictable by the study of the independent systems in isolation of one another



Typical approach used today

- ◆ System engineering of SoS characterized by
 - Strict specification of standards
 - Interconnectivity schemes to exchange messages that follow prescribed protocol standards
 - Interconnecting multiple processors with a defined communications medium



Implications

- ◆ Does not focus on shaping the behavior of the SoS
- ◆ Leaves interpretation of the messages and the resultant actions to the individual acquirers of the various systems
 - Protocol standards oftentimes include limited behavior rules
 - Protocol standards do not include requirements for such tasks as handling runtime faults other than error-checking of transmitted messages
 - ◆ This task is left to the developers of the individual systems in the SoS



“Thou shalt” mentality

- ◆ The tendency in specifications is to
 - Document the “thou shalts” of specific system functions
 - Design to the “thou shalts” with modifications to accommodate the development
 - Field a system that little resembles the collective “thou shalts” and contains limited user utility



System behavior

- ◆ Capturing the desired SoS behavior in the traditional natural language documents is a complex issue given that the legacy systems in the SoS have a combination of existing known and unknown behaviors
 - Typically, the SoS specification is reduced to a table of information exchange requirements (IERs) that define the messaging that passes from one system to another



Lack of predictable behavior

- ◆ Because each system in the SoS develops the implementation of the interconnectivity standard independent of the other systems, there is no guarantee that all the implementations will result in consistent behavior by the system-of-systems
- ◆ Not acceptable for mission- or safety-critical systems



Communication disconnect

- ◆ Traditional system engineering state-of-the-practice assumes complete and precise understanding of the system under development
 - Integrators of the independent systems hold different assumptions and beliefs about the SoSs than that of the developers of the individual systems



System salvage

- ◆ Use of tightly coupled and lowly cohesive communications shackles to hobble together systems-of-systems
 - Represents a bend-fold-spindle-and-mutilate approach to composition
 - Encourages system developers to focus on modifying individual systems with limited deliberation and consideration for the system as a whole



Patch-and-pray mentality

- ◆ With each new failure, system engineers attempt to tighten up the protocol standard
 - The SoS cannot be made to exhibit predictable, dependable behavior by increasing the level of detail in the interconnectivity standards
 - The end-state is a collection of systems that have a high degree of coupling with a realized protocol standard that only serves to significantly increase the SoS software complexity



System unravelings

- ◆ System-software critical interactions increase as the complexity of highly interconnected systems increases—combinatorial explosion
 - System unravelings seem to have an intelligence of their own as they
 - ◆ Expose hidden connections
 - ◆ Neutralize redundancies
 - ◆ Exploit chance circumstances for which no system engineer might plan
 - Unforeseeable combination can cause cascading failures within the SoS



My position

- ◆ Need a change in thinking about how to conduct system engineering to support the design, specification, and construction of systems-of-systems
- ◆ Too much development risk to stay the current course
 - Majority of high-profile system-of-systems developments have been spectacular failures



Shared understanding

- ◆ Need to foster among stakeholders a shared understanding of the design of the SoS, by having the system engineers
 - Define the problem space that the SoS will address, including restrictions imposed upon the SoS by the operating environment
 - Identify the elements within the SoS
 - Describe the operational concepts (i.e., business rules) with respect to the user of the SoS



System-of-systems architecture

- ◆ Think in terms of the
 - Controlling software
 - Information transport network
 - Contract interfaces
 - Individual systems



Controlling software

- ◆ Manage the activities in the system-of-systems
- ◆ Direct work to independent systems through contract interfaces
- ◆ Monitor and execute the safety and security policies for the system-of-systems
- ◆ Manage the persistent data storage of the system-of-systems
- ◆ Manage the level of system complexity through component-based engineering



Information transport network

- ◆ Manage the activities and behavior in the system-of-systems network
- ◆ Transparent to independent systems
- ◆ Support contract interfaces



Contract interfaces

- ◆ Interfaces defined by provided and consumed services
 - No independent systems has knowledge of other independent systems
 - Minimal interfaces
- ◆ Apply design-by-contract principles
 - Contracts specified as assertions at the interfaces
- ◆ Interfaces grouped by common services to reduce number of interface types



Individual systems

- ◆ Provide and consume services through contract interfaces
- ◆ State behavior isolated from state behaviors of other independent systems
- ◆ No knowledge of other independent systems



Case Study: BMDS

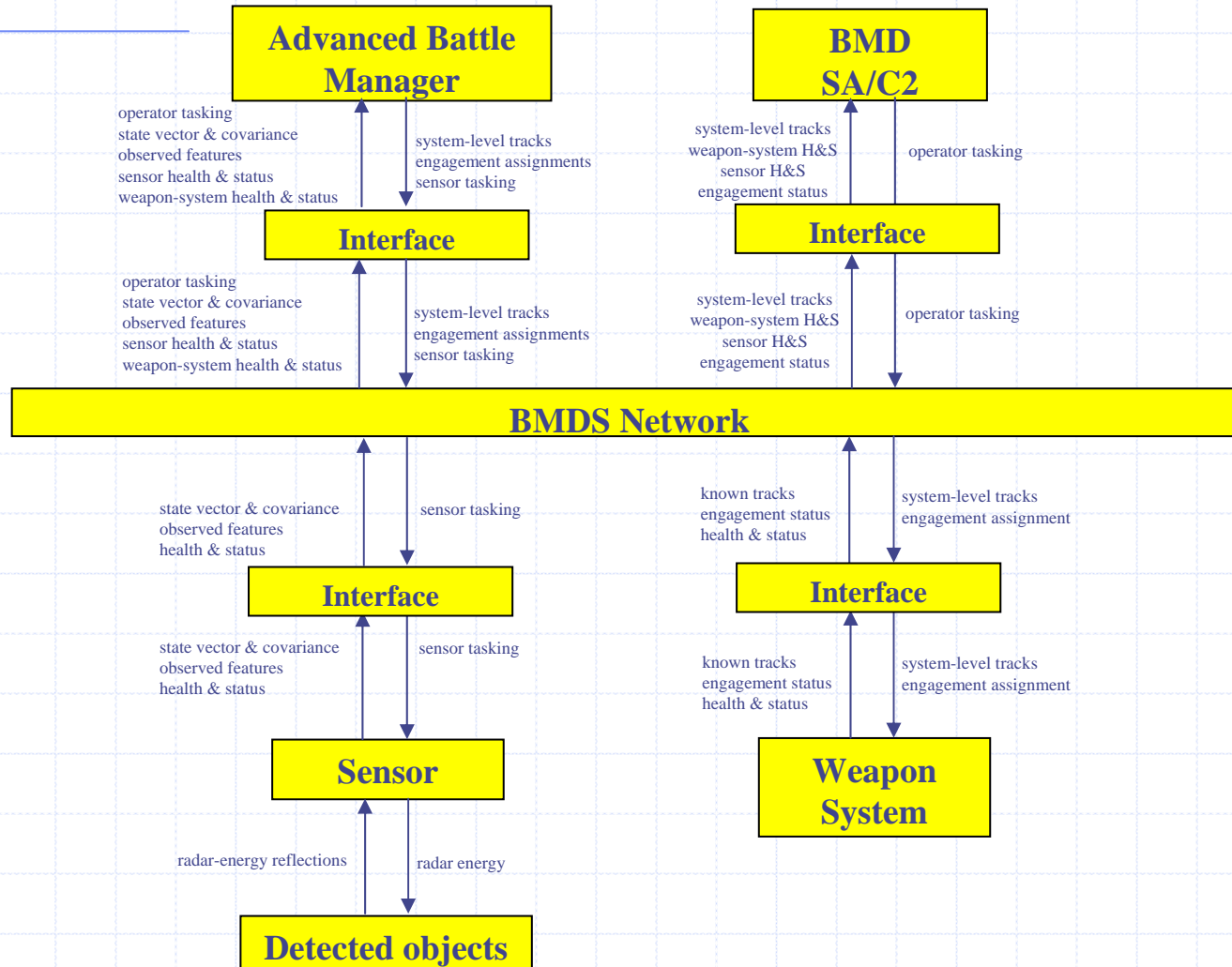
◆ Ballistic missile defense system (BMDS)

- Sensors
- Shooters
- Battle manager
 - ◆ Controlling software
 - ◆ Enforces Commander's selected course of action

◆ Developed in a distributed manner with participation of many countries



External View: BMDS





Recap

- ◆ Controlling software
 - Manage the activities in the system-of-systems
 - Direct work to independent systems through contract interfaces
 - Reduce level of system complexity through component-based engineering
- ◆ Information transport network
 - Manage the activities and behavior in the system-of-systems network
 - Transparent to independent systems
- ◆ Contract interfaces
 - Service-based
 - Design-by-contract
- ◆ Individual systems
 - Provide and consume services through contract interfaces
 - State behavior isolated from state behaviors of other independent systems
- ◆ Software for systems-of-systems must be designed for reuse
 - Infrastructure for reuse should be an integral part of a component
 - All phases of the reuse process must be supported by the infrastructure



Dependability issues revisited...

- ◆ Implementing software designs is relatively easy compared to architecting distributed systems to be both dependable and reconfigurable (to adapt to new or modified requirements)
 - In the systems of today and the future, the system architecture must “outlive” the system components, supporting “plug-and-play” operation



Unpredictable environment

- ◆ The environment can be unpredictable and therefore difficult to model
 - Configuration of the system can change to adapt to changes in the environment
- ◆ Is it possible to obtain a sufficient set of system requirements from which to define the dependability properties for the system?



Example: System safety

- ◆ System safety, for instance, relies on predictability
 - There is a need to know what the system must guard against (i.e., hazards)
- ◆ However, with a lack of environment predictability, how does one handle unanticipated hazards?



Assurance of dependability

- ◆ Typically performed against a fixed known model, not a radically changing environment
 - Adaptive systems can have lots of configurations
 - ◆ Hard to characterize, because each instance of a component has a different view of the system
 - The set of things in the environment is neither closed nor stable
- ◆ Implication
 - It might be possible to create a sufficiently large closed world so that one can deal with all of the system hazards
 - ◆ Even so, there will still be a challenge to validate an upper bound on the probability of a system failure leading to a given hazard



Dependability a pipedream?

- ◆ Is this intractable from a dependability perspective: in my opinion, the answer is “no”
 - ◆ Challenge is to think in terms of integration properties:
 - Identify the emergent requirements from the collaborations (i.e., value-added)
 - ◆ Must certify that the legacy system meets constraints of the “plug-in slot”



Achieving dependability

- ◆ We advocate a departure from “business as usual” in the engineering of systems, by requiring:
 - Spartan and Draconian designs of systems
 - Distinguishing up front which system requirements are stable from those that are expected to change
 - ◆ Institutionalizing the invariant part of the principles of operation of the system
 - Take a positive approach to handling emergent properties, thinking in terms of integration
 - ◆ Define emergent requirements and ensure realization



Dependability disciplines

- ◆ *Spartan safety kernel* – liveness properties with service guarantees
 - Services needed to achieve critical requirements
 - Includes timeliness
 - No other services
- ◆ *Draconian global structure* – safety with non-interference guarantees
 - Visible dependencies much less than potential dependencies
 - Fault containment at boundaries
 - No invisible interactions



Beyond technical challenges...

- ◆ Changes in system acquisition policy
 - Need to encourage a system-of-systems philosophy in the systems engineering community, rather than continued development of monolithic throwaway systems
- ◆ Changes in engineering culture
 - Can no longer think in terms of starting from a clean slate, that is, building a system from scratch
 - ◆ Need to think in terms of reuse rather than salvage



Contact information

J. Bret Michael

bmichael@nps.edu

Tel. +1 831 656 2655