On Emergent Misbehavior

John Rushby
With help from Hermann Kopetz

Computer Science Laboratory
SRI International
Menlo Park CA USA

Emergence

- We build systems from components, but systems have properties not possessed by their individual components
- Emergence is the idea that complex systems may posses properties that are different in kind than those of their components: described by different languages
  - e.g., velocities of atoms vs. temperature of gas
  - e.g., neuron activity in the brain vs. thoughts in the mind
- Weak emergence: you can compute the emergent properties from those of components (but only by simulation)
  - Complicated vs. complex systems
- Strong emergence: not so—interactions at emergent level propagate back to the components (downward causation)
  - E.g., flock flowing around an obstruction: motion looks random to individual responding to actions of neighbors

Emergent Misbehavior

- There's good emergence and bad
- In particular, complex systems can have failures not predicted from their components, interactions, or design
- Emergent or just unexpected?
- Probably the latter, but in sufficiently complicated contexts that it may be useful to consider these failures as different in kind than the usual ones
- My speculation is that weak emergence explains most
- But maybe some are due to downward causation
- In any case, a possibly useful new way to look at failures

Examples

- Jeff Mogul's paper:
  - Mostly OS and network examples concerning performance and fairness degradation rather than outright failure
  - e.g., router synchronization
  - Note that these properties are expressed in the language of the emergent system, not the components
- Feature interaction in telephone systems
- 1993 shootdown of US helicopters by US planes in Iraq
- West/East coast phone and power blackouts
- Massive freeway pileups

Even "Correct" Systems Can Exhibit Emergent Misbehavior

- We have components with verified properties, we put them together in a design for which we require properties P, Q, R, etc. and we verify those, but the system fails in operation... how?

- There's a property S we didn't think about
  - Maybe because it needs to be expressed in the language of the emergent system, not in the language of the components
  - If we'd tried to verify it, we'd have found the failure
  - But it's hard to anticipate all the things we care about in a complicated system

- Call these unanticipated requirements

Even “Correct” Systems Can Exhibit Emergent Misbehavior (ctd.)

- We verified that interactions of components A and B deliver property P and that A and C deliver Q, taking care of failures appropriately
- But there’s an interaction we didn’t think about
    - We didn’t anticipate that some behaviors of C (e.g., failures) could affect the interactions of A and B, hence P is violated even though A and B are behaving correctly (and so is C, wrt. the property Q)
- Call these unanticipated interactions
  (or overlooked assumptions)

Causes of Emergent Misbehavior

- I think they all come down to ignorance
- There is no accurate description of an emergent system simpler than the system itself
- All our analysis and verification are with respect to abstractions and simplifications, hence we are ignorant about the full set of behaviors
- More particularly, we may be ignorant about
  - The complete set of requirements we will care about in the composed system
  - The complete set of behaviors of each component
  - The complete set of interactions among the components

How to Eliminate or Control Emergent Misbehavior

- Identify and reduce ignorance
- Eliminate or control unanticipated behaviors and interactions
  - i.e., deal with the manifestations of ignorance
- Engineer resilience
  - i.e., adapt to the consequences of ignorance

Identify and Reduce Ignorance

Vinerbi, Bondavalli, and Lollini propose tracing ignorance as part of requirements engineering

- Qualitatively quantify it (e.g., low, medium, high)
- Have rules how it propagates though AND and OR etc.
- If it gets too large, consider replacing a source of high ignorance (e.g., COTS, or another system) by a better-understood and more limited component

Identify and Reduce Ignorance (ctd.)

- We have to try and think of everything
- This is what hazard analysis is about in safety-critical systems
- There are systematic ways to go about it (e.g., HAZOP)
- But I think it needs to be put on a more formal footing
  - And that automated support is needed
- There are some promising avenues for doing this
  - e.g., model checking very abstract designs
  - Using SMT solvers for infinite bounded model checking with uninterpreted functions

Identify and Reduce Ignorance (ctd. 2)

- Black and Koopman observe that safety goals are often emergent to the system components
- e.g., the concept (no) "collision" might feature in the top-level safety goal for an autonomous automobile
- But "collision" has no meaning for the brake, steering, and acceleration components
- That's why FAA certifies only complete airplanes and engines
- They suggest identifying local goals for each component whose conjunction is equivalent to the system safety goal, recognizing that some unknown additional element X may be needed (because of emergence) to complete the equivalence
- An objective is then to minimize X
- Closely related to hazard analysis, in my view

Eliminate Unanticipated Behaviors and Interactions

- Behaviors and interactions due to superfluous functionality
  - e.g., use of a COTS component where only a subset of its capabilities is required
  - Or functions with many options where only some should be used

  These can be eliminated by wrapping or partial evaluation

- Interactions that use **un**anticipated pathways
  - E.g., A writes into B's memory
  - Or tramples on its bus transmissions
  - Or monopolizes the CPU

  These can be eliminated by strong partitioning of resources

Control Unanticipated Behaviors and Interactions

- Unanticipated behaviors on known interaction pathways
  - e.g., unclean failures
  - Local malfunction

  These can be controlled by strong monitoring
  - Monitor component behavior against system requirements; shutdown on failure
  - Monitor assumptions; treat source component (or self?) as failed when violated

Engineer for Resilience

- Our diagnosis is very similar to Perrow's Normal Accidents
- In his terms, we aim to reduce interactive complexity and tight coupling
- One way to do both is to increase the autonomy of components
    - i.e., they function as goal-directed agents
    - e.g., substitute runtime synthesis for design-time analysis (both use formal methods, but in different ways)
- But then may be more difficult to design the overall system
    - Actions of intelligent components frustrate system goals
    - e.g., pilot actions on AF 447
- Overall system should become adaptive or autonomic
  Using AI and machine learning

Summary

- Reductionist approaches to system design and understanding may no longer be suitable
  - Systems built from incompletely understood components, and other systems
  - System goals far removed from component functions
- Widespread emergent misbehavior seems inevitable
  - In some cases, can attempt to reduce emergence and restore validity of reductionism
  - In other cases, should embrace emergence and aim for adaptation and resilience
- In no cases will it be business as usual
- Datum: safety critical code size in aircraft and spacecraft doubles every two years (Holzmann)