MPSOC 2002

Distributed Embedded System Architecture

Philip Koopman
koopman@cmu.edu
July 12, 2002

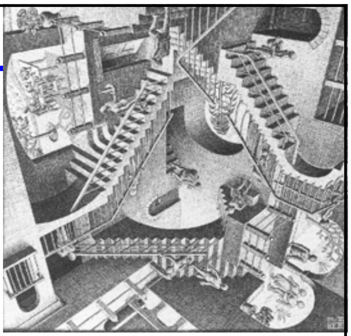
Carnegie Mellon

Institute
for Complex
Engineered
Systems

Electrical & Computer
ENGINEERING

My Perspective On (Distributed) Embedded Systems

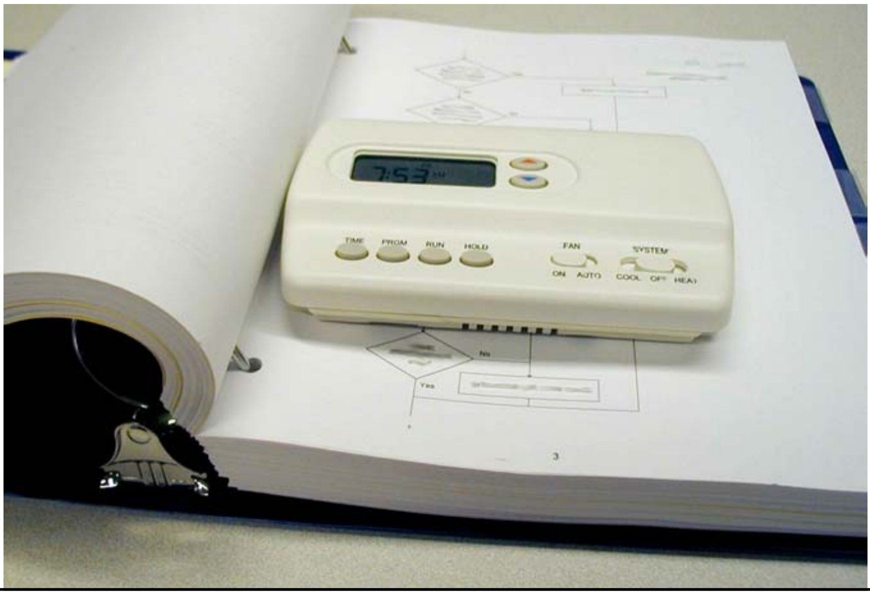
Preview



- **Embedded System Architecture =**
  - Hardware + Software + Communication +Control + other stuff
  - **Each architecture is a view into the system**
  - **Overlapping views have some degree of compatibility**
- **Make it easier for system to meet requirements**
  - Concentrate on essential system characteristics
  - Help mere mortals see the big picture(s)

Myth: "Small" Embedded Systems Are Trivial

- **Only "toy" versions are trivial; real world is complex**



What's Inside an Embedded "System"?

- **"Features"**
  - High-level system functionality
  - Mostly mapped to software…
- **Software**
  - Computation
    - Control loops
    - Finite state machines
  - Communication
    - Intra-node communication via calls
    - Inter-node communication via messages
- **Hardware**
  - Nodes + Networks + Interfaces
- **Must meet non-functional requirements (real-time, 'ilities including profitability)**

What's an Architecture?

- **Loosely: an architecture is how all the pieces fit together**
- **Architecture definitions:**
  - **System architecture:**
    The structure – in terms of components, connections, and constraints – of a product, process, or element. [Rechtin96]
  - **Software architecture:**
    The structure or structures of the system, which comprise components, their externally-visible behavior, and the relationships among them [Bass97]
- **Informally: Boxes and Arrows**
  - Boxes: objects/subsystems/…
  - Arrows: interfaces

My Definition Of An Architecture

- **An *architecture* is an organized collection of components that describes:**
  - both behaviors and interactions
    - » (boxes & arrows)
  - with respect to a specific abstraction approach and
    - » (rule for when to create a set of subsystem boxes)
  - subject to a set of *goals+constraints*
    - » (rules to evaluate how good the architecture is)
  - An *implementation* uses a specific mechanism to create a behavior and and interface for a component (it's an instantiation of an architecture)
- **One person's component is another person's system**
  - An implementation can have multiple components, each with its own architecture
  - This definition recurses

Interfaces / Specifications

- **Functional properties**
  - What exactly does each system module/subsystem do?
  - (But, not exactly how it does it – thus, implementation is encapsulated)
- **Control properties**
  - Which signal (message, variable, physical pin) does what?
- **Temporal properties**
  - Timing constraints on interface, including ordering restrictions
- **Data properties**
  - What do the data values look like?
  - Often in the form of a message dictionary, with map of data fields for each message
- **The big question – how do you know where to insert the interfaces?**
  - How do you know what decomposition steps to perform?

Embedded System Architectures

- **Primary Architectures (almost always used)**
  - Hardware architecture (CPU, memory, network, I/O)
  - Software architecture (software components, data repositories, message dictionary, external interfaces)
  - Communication architecture (message flows, message formats)
  - Control architecture (hierarchy of control algorithms; emergent system behavior)
- **Secondary Architectures (used when needed)**
  - Human interface
  - Component coordination & timing framework
  - Safety/security
  - Validation/verification/testing
  - Maintenance/upgrade
  - Fault management/graceful degradation
  - …

System Architecture/Partitioning

- **Partition to meet constraints of:**
  - All necessary functionality provided
  - Computation power per node
  - Memory space per node
  - Bandwidth/real-time abilities of network
  - Hardware/Software tradeoffs can help with optimization
  - Legacy issues
- **Traditional approach: hardware first**
  - Gradually moving to HW/SW co-specification/co-design
- **Alternatives are possible**
  - Functionality first / product family-based design
  - At each level of system, use an “appropriate” decomposition strategy
  - Create architectural views, then perform fusion/allocation

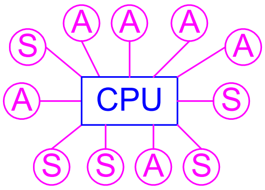
Architectural Patterns

- **General known approaches can apply to new systems**
  - Sometimes presented as “pattern catalogs”
  - Gives guidance to reduce need for create-from-scratch approaches
- **Following slides are some examples**
  - A real catalog would have detailed textual descriptions too
  - This is a very small sampling of patterns; there are many ways to do things!
    - The idea is to demonstrate the different flavors of architectural views

Hardware Patterns

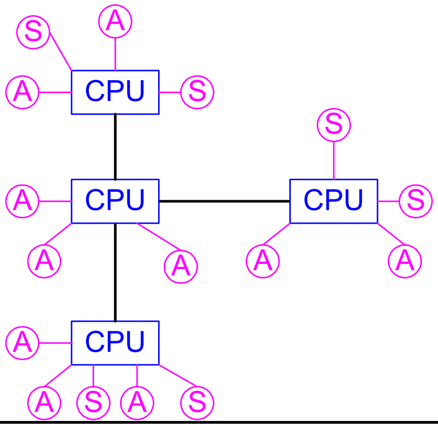
- **Centralized System**
  - Abstraction principle: all in one big pile
  - Single CPU for all sensors/actuators

  - Pro: efficient use of CPU & Memory
  - Con: difficult to expand



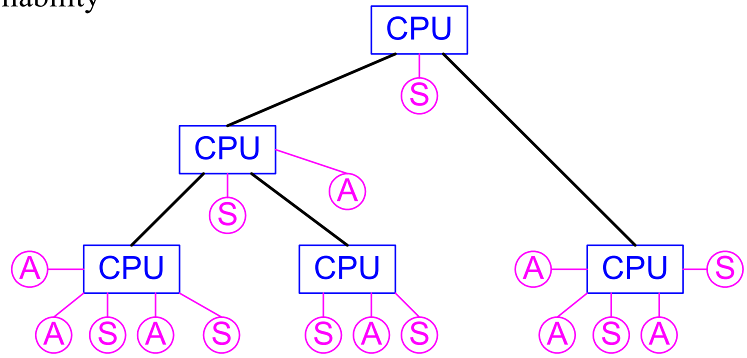
Hardware Patterns

- **Ad Hoc**
  - Abstraction principle: paste extra boxes on as system evolves
  - Pro: easy way to tack on patches in evolving system
  - Con: inefficient mapping of most architectural approaches



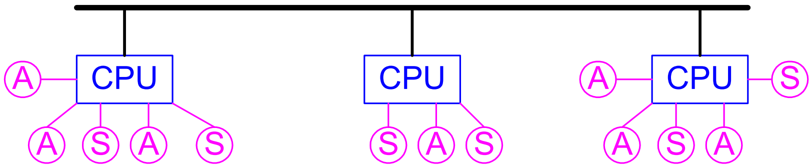
Hardware Patterns

- **Hierarchical**
  - Abstraction principle: “big” nodes at top; “little” nodes & most I/O at bottom
  - Pro: easy mapping to hierarchical control
  - Con: top/root node forms bottleneck for communications & reliability



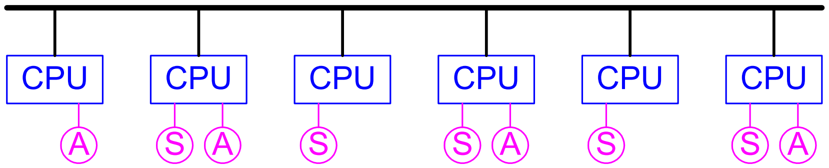
Hardware Patterns

- **Federated/Decentralized Networked System**
  - Abstraction principle: multiple boxes all on one network as peers
  - Several sensors/actuators/servo loops per CPU
    - Often sensor/actuator/CPU pairing done by 3-D geometric regions
    - Design approach is often add CPUs as you need more I/O connections
  - Pro: benefits of being distributed with lower CPU packaging costs
  - Con: can have poor mapping to control architecture



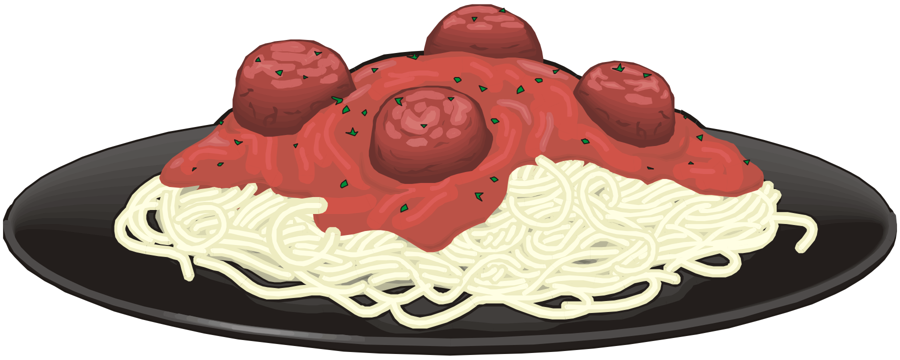
Hardware Patterns

- **Highly Distributed Networked System**
  - Abstraction principle:
    One sensor, actuator, or servo pair per CPU, on a network
  - Bus interconnect
    - Bus hierarchy may be needed to overcome bandwidth limits
  - Pro: doesn't predispose system to any other architectures
    - Good for an idealized MEMS system
  - Con: bus can be a bottleneck



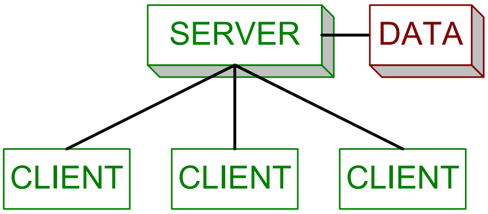
Software Patterns

- **Ad Hoc** (with “object-oriented” meatballs)



Software Patterns

- **Client/Server**
  - Abstraction principle:
    All data at a server; replicate clients to interface elsewhere
  - Pro: keeps clients small/cheap
  - Con: server is performance & reliability bottleneck

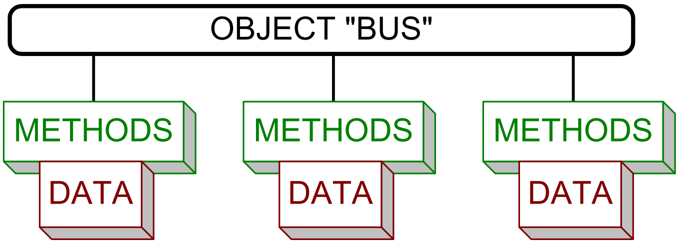


Software Patterns

Object oriented / Federated

- Abstraction principle: partition by data types, hide data behind methods
  - Note: flow of control is completely obscured

- Pro: helps with multi-vendor/mult-subsystem integration (compatible with CORBA)
- Con: can have high overhead to access data

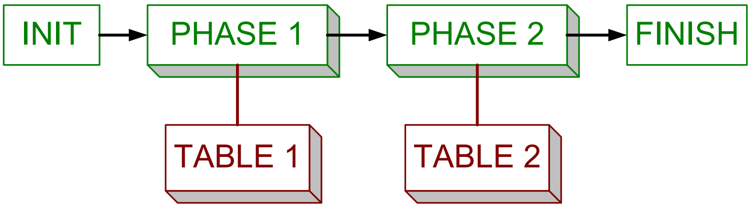


Software Patterns

Table Driven, phased, flow of control

- Abstraction principle: Partition by phases of execution, use tables to specify detailed behavior for general software modules
  - This is actually a combination of “control flow” and “table driven” patterns

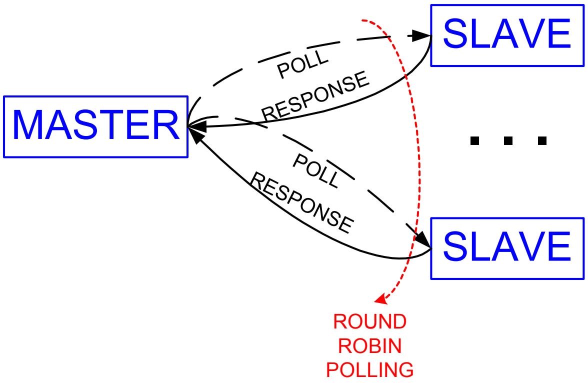
- Pro: frequently used for customizable system
- Con: flow-of-control organization is harder to get right than object oriented for many systems



Communication Patterns

- **Master/Slave**
  - Abstraction principle: master node explicitly coordinates all traffic

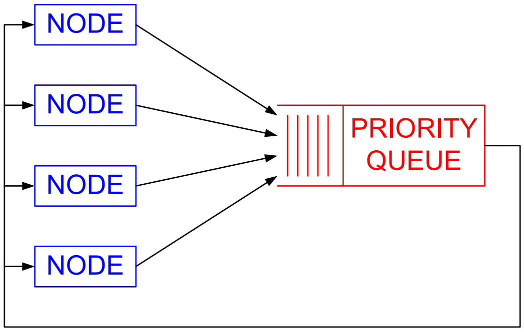
  - Pro: Very simple to implement and get right
  - Con: Coordination consumes bandwidth;
    Master is potential single point of failure



Communication Patterns

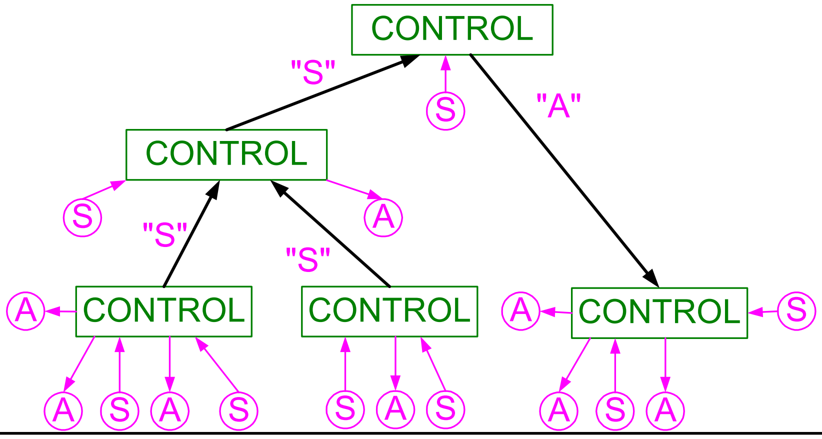
- **Global priority**
  - Abstraction principle: highest priority message delivered first
    - Does ***NOT*** require a physical node to act as a queue – fully distributed implementations are commonly used!

  - Pro: priority helps meet deadlines
  - Con: priority interferes with fairness



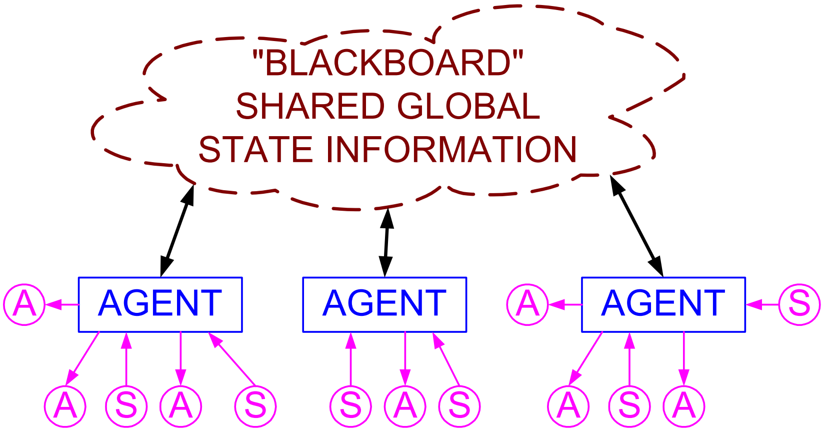
Control Patterns

- **Intelligent Hierarchical Control (IHC)**
  - Abstraction principle: nest control loops based on sensors/actuators
    - Use sub-levels as logical sensors & actuators to close a control loop
    - Each level may itself have sub-levels



Control Patterns

- **Federated Agents/“Blackboard”**
  - Abstraction principle: each object has a control agent; agents monitor and transmit global state information for coordination



Human Interface Patterns

- **State machine model**
  - E.g., digital watch with 4 buttons
  - Maps well onto statechart and other engineering design tools
  - Person has to keep track of mode information
    - This is a classic usability problem
- **Menu-driven interface**
  - "User friendly"
  - Can be frustrating for experts
- **Command line interface**
  - "User hostile"
  - Can be very efficient for expert users

Component Coordination Framework

- **Direct integration**
  - E.g., direct procedure calls & messages (e.g., sockets)
  - High efficiency; high flexibility in detailed implementation
  - Requires knowledge of all the details to integrate a component
- **"Basic" middleware**
  - E.g., CORBA, D-COM, Jini; perhaps RPC/RMI; but few services
  - Provides interface abstraction; hides differences in implementation
    - May facilitate use of COTS software components
  - Centralized point for adding fault tolerance, monitoring
  - Incurs various overheads, especially execution speed & memory size
- **Advanced middleware**
  - E.g., naming & discovery services added to middleware
  - Simplifies dynamic reconfiguration, collaboration among designs
  - Adds more complexity & overhead

Safety Patterns

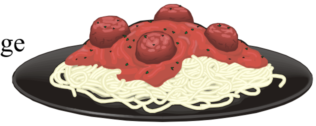
- **Automatic safety net approach**
  - Provide a distinct safety system that can ensure safety
    - E.g., emergency brake, or other emergency stop system
  - Keep safety system simple in content and interface
- **Rely on human operator to keep system safe**
  - Simple, easy way to attempt to evade liability
  - Humans can be counted upon to make mistakes
    - But, operators are great scapegoats for the accident investigation
- **Field data collection + engineering feedback**
  - Partially shows up in technical system as black box/flight recorder
- **There are non-architectural approaches as well**
  - E.g., formal verification; extensive field trials
  - The architected techniques result in a "safety box" that somehow gets mapped into other architectural views

Security Patterns

- **"Air Gap" security**
  - If there is no network connection, it is difficult to mount a network-based attack
  - Increasingly unrealistic for most systems
- **Firewall security**
  - Create a constrained interface
  - Is proven somewhat effective, but difficult to ensure there are no holes at all
  - Constrains inter-system communication, coordination & optimization
- **Encrypted communication/authentication**
  - All interfaces have encryption/authentication
  - For efficiency, often combined with firewall pattern (encryption only outside firewall trusted zones)
- **Non-architectural approaches include:**
  - Attempted security through obscurity
  - Attempted security through criminalizing reverse engineering

Validation/Verification/Certification Patterns

- **Segregate critical subsystems and recertify only those**
  - This is the current "best" approach for mixed critical/non-critical systems
- **Include access points for testing**
  - Hardware testing (e.g., boundary scan)
  - Create formalized APIs and components (e.g., use certified RTOS)
    - But it is tricky to make an API truly bulletproof
- **Non-architectural approaches:**
  - Recertify everything after every change



  - Use design rules that avoid need to certify
    - In some cases this really works
      (e.g., keep below certain wattage for RF transmissions)
    - "Certification" in that case is being sure you followed the design rules

---

Maintenance/Upgrade Patterns

- **Software upgrade capability**
  - Use flash memory to deliver fixes
    - Cost vs. flexibility tradeoff
    - Upgrades can occur between IC manufacturing and product assembly
  - Causes architectural ripples to hardware, connectivity, etc.
- **Mechanically partitioned units** (e.g., socketed chips)
  - Partition design into replaceable units
    - Replace subsystems to accomplish upgrades/repairs
  - Might include replacing hardware components as a software upgrade maintenance operation
    - Can be difficult to accomplish inexpensively if each chip is highly integrated (and therefore expensive)
- **Non-architectural approaches include:**
  - Make a product disposable (no maintenance/upgrade possible)

Fault Tolerance/Degradation Patterns

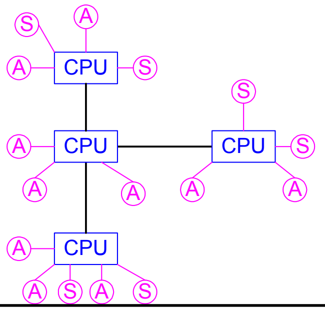
- **Replication with failover**
  - Every critical function has at least one backup
    - Active replication with hot standby failover
    - Passive replication with cold standby + transaction logs for catching up
    - Spare resource pool with reboot after reconfiguration
  - Works well if failures are random (not all software defects are random!)
  - Aggressive replication is expensive

- **Function/load shedding as replicants fail**
  - Architecturally, this shows up as a configuration or workload manager
  - Spread workload over replicated units
    - As units fail, capacity is reduced, but each unit can operate standalone if needed
  - Have configuration plans that map functions to units
    - As units fail, different mappings are used to keep key functions running

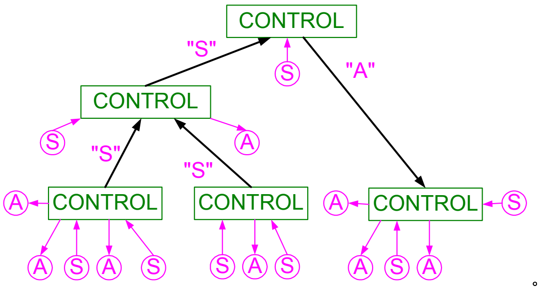
Multi-View Architectural Fusion

- **Every real system has several architectural views**
  - Differing views have to be combined to form “The Architecture”
  - This process is a generalization of allocating software modules to hardware, but can have much higher dimensionality
- **Most times you can use any architectural combination**
  - But, you/your design may suffer significantly if you pick poorly

Point-to-Point Hardware

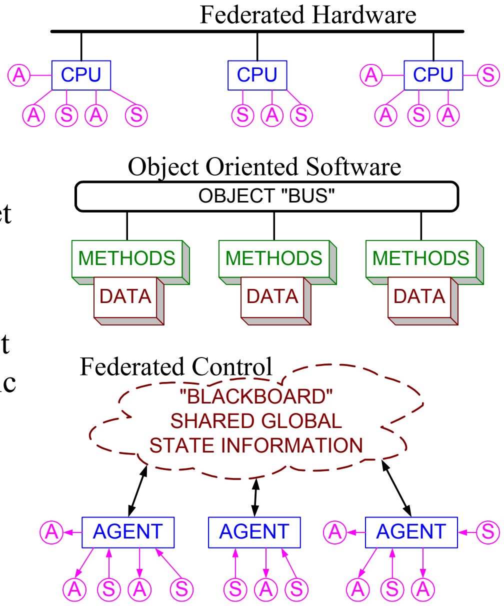


Hierarchical Control



Observations – Isomorphism

- **Some patterns are isomorphic across different architectural perspectives**
  - Often, they are used as a set
  - But, they don't *have* to be used together
  - And, more importantly, just because they are isomorphic does not mean they aren't all there as distinct concepts!



Other Observations

- **Multiple architectural approaches can be combined/nested**
  - e.g., Client Server plus object bus,
    PLUS some "objects" are implemented as distributed systems

- **There are no exactly correct answers**
  - This area is more art than science
  - Each architectural pattern tends to have tradeoffs
    - Architectural selections are not entirely independent
    - Tradeoffs can occur due to combinations of patterns

- **Businesses are systems too**
  - And they have multiple architectural views

Non-Architectural Approaches

- **Where do all those "non-architectural" approaches fit?**
  - Typically they are things that don't trace to specific boxes in any architecture
  - Sometimes they are omissions
    - e.g., "we don't have a security strategy"
  - Sometimes they trace to non-engineering business architecture boxes
    - e.g., information access architecture uses an NDA in support of "security through obscurity"
  - Sometimes they trace to a business *model*
    - e.g., "we want consumers to upgrade by throwing the old one away"
      - Thus, make products non-repairable, but cheaper than repairable ones
      - Perhaps it consumers encounter a bug, tell them their unit has worn out and they need to buy another one to replace it (one that will have newer software…)

- **Most "systems" are really "systems of systems"**
  - Some high level functions get diffused into emergent properties within components (this is a traceability problem)
  - Some high level constraints get converted into boxes within components
  - …

How To Create A Functional Architecture

Note: this is a combined view, 1-D approach to architecture

- **Functional Architecture = subsystems created by splitting "functions"**
  - Classical large system development technique
  - Seldom optimal, but most engineers can be trained to think this way
  - Historically the architecture of choice for weapon systems
  - Single, combined view of hardware + software + control, with implied federated communication architecture (1 "box" = 1 "subsystem")

- **Architectural methodology (a guide to "Functional Boxology")**
  - List primary mission goals
    - Associate secondary mission goals
  - List verbs that correspond to "marketing requirements"
    - One verb per requirement
    - Be sure that verbs are orthogonal
  - Architectural decomposition is one box per verb
    - Recurse as necessary
    - Stop recursing when each box is a design team of 4 people or fewer

Elevator Functional Architecture

Example Functional Architecture for Elevator

Primary Mission	Provide safe, timely, comfortable passage between floors.
Secondary Missions	Deliver Passengers Quickly Protect Passengers Inform Users Provide Tranquil Environment Support Customized Behavior Conform To Building Codes Support Maintenance

TOP-LEVEL FUNCTIONS

MOVE

DETERMINE PASSENGER INTENT

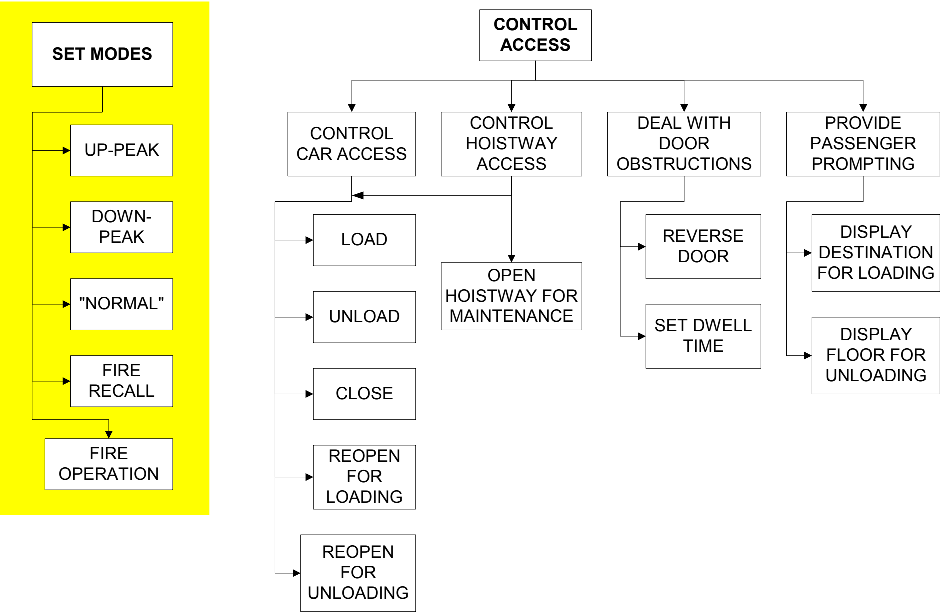
CONTROL ACCESS

DISPATCH

INFORM USERS

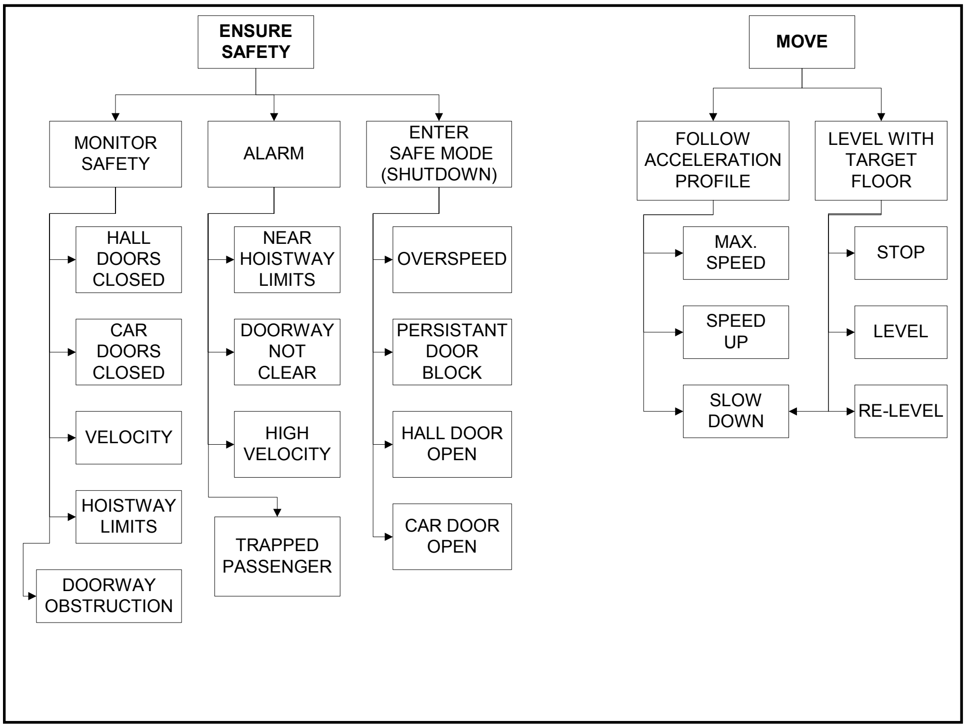
SET MODES

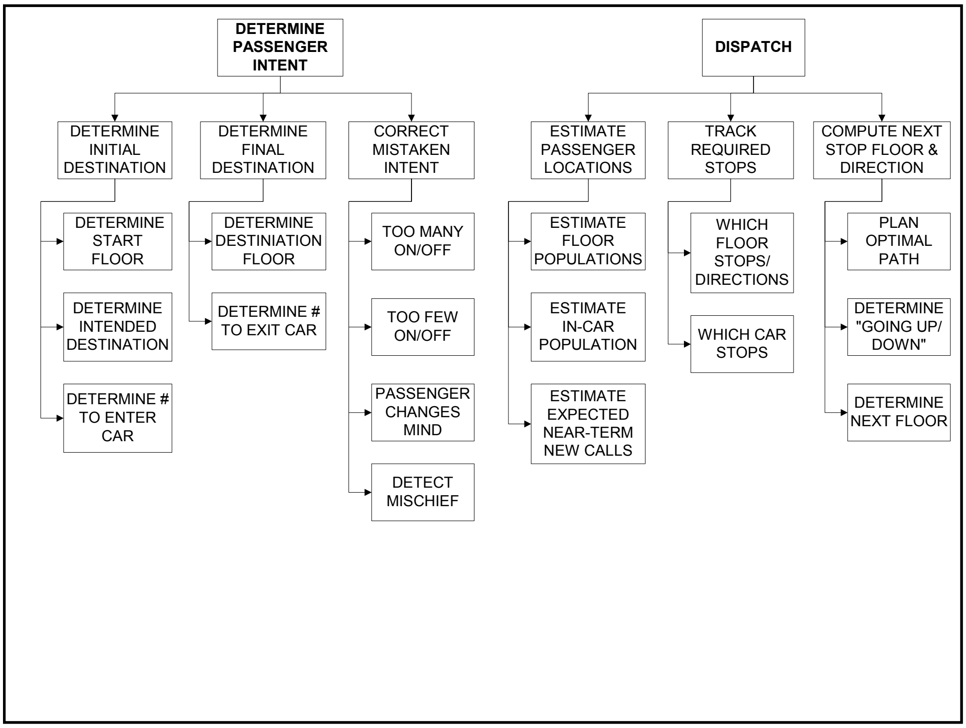
ENSURE SAFETY

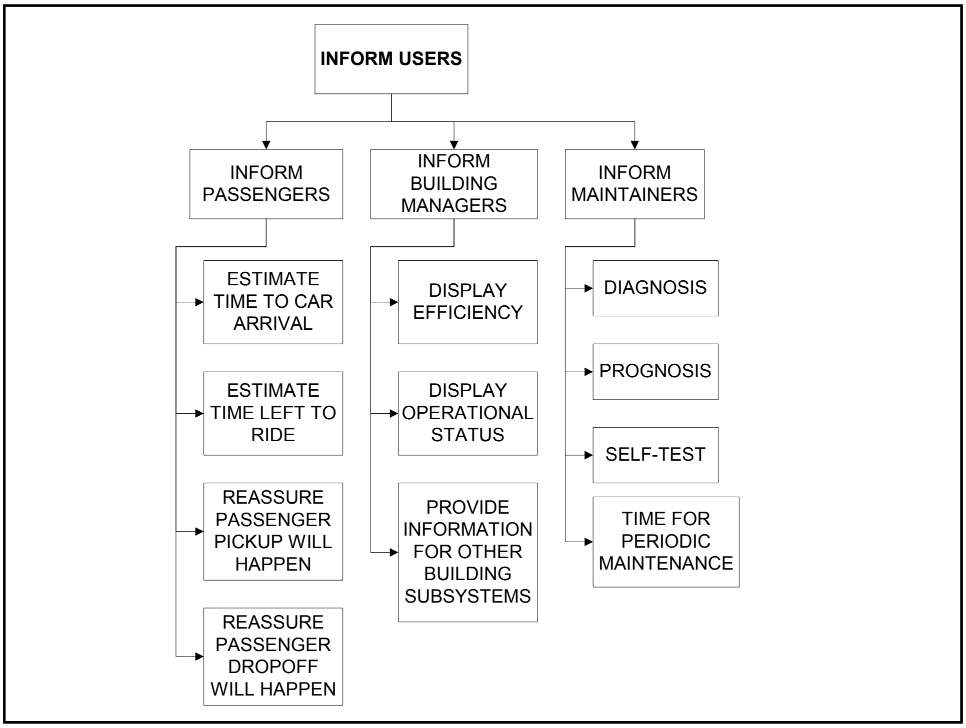


- **SET MODES**
  - UP-PEAK
  - DOWN-PEAK
  - "NORMAL"
  - FIRE RECALL
  - FIRE OPERATION

- **CONTROL ACCESS**
  - CONTROL CAR ACCESS
    - LOAD
    - UNLOAD
    - CLOSE
    - REOPEN FOR LOADING
    - REOPEN FOR UNLOADING
  - CONTROL HOISTWAY ACCESS
    - OPEN HOISTWAY FOR MAINTENANCE
  - DEAL WITH DOOR OBSTRUCTIONS
    - REVERSE DOOR
    - SET DWELL TIME
  - PROVIDE PASSENGER PROMPTING
    - DISPLAY DESTINATION FOR LOADING
    - DISPLAY FLOOR FOR UNLOADING


ENSURE SAFETY
MONITOR SAFETY
HALL DOORS CLOSED
CAR DOORS CLOSED
VELOCITY
HOISTWAY LIMITS
DOORWAY OBSTRUCTION
ALARM
NEAR HOISTWAY LIMITS
DOORWAY NOT CLEAR
HIGH VELOCITY
TRAPPED PASSENGER
ENTER SAFE MODE (SHUTDOWN)
OVERSPEED
PERSISTANT DOOR BLOCK
HALL DOOR OPEN
CAR DOOR OPEN
MOVE
FOLLOW ACCELERATION PROFILE
MAX. SPEED
SPEED UP
SLOW DOWN
LEVEL WITH TARGET FLOOR
STOP
LEVEL
RE-LEVEL


DETERMINE PASSENGER INTENT
DETERMINE INITIAL DESTINATION
DETERMINE START FLOOR
DETERMINE INTENDED DESTINATION
DETERMINE # TO ENTER CAR
DETERMINE FINAL DESTINATION
DETERMINE DESTINIATION FLOOR
DETERMINE # TO EXIT CAR
CORRECT MISTAKEN INTENT
TOO MANY ON/OFF
TOO FEW ON/OFF
PASSENGER CHANGES MIND
DETECT MISCHIEF
DISPATCH
ESTIMATE PASSENGER LOCATIONS
ESTIMATE FLOOR POPULATIONS
ESTIMATE IN-CAR POPULATION
ESTIMATE EXPECTED NEAR-TERM NEW CALLS
TRACK REQUIRED STOPS
WHICH FLOOR STOPS/ DIRECTIONS
WHICH CAR STOPS
COMPUTE NEXT STOP FLOOR & DIRECTION
PLAN OPTIMAL PATH
DETERMINE "GOING UP/ DOWN"
DETERMINE NEXT FLOOR



INFORM USERS

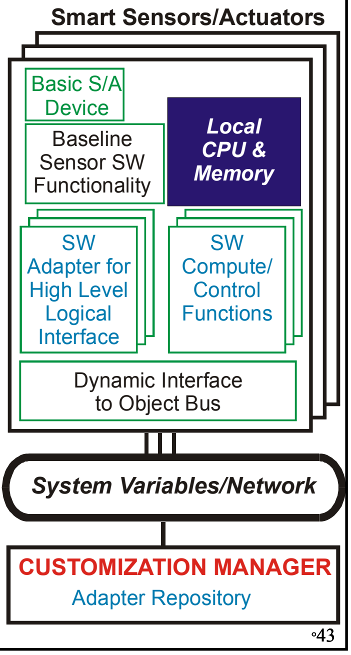
- INFORM PASSENGERS
  - ESTIMATE TIME TO CAR ARRIVAL
  - ESTIMATE TIME LEFT TO RIDE
  - REASSURE PASSENGER PICKUP WILL HAPPEN
  - REASSURE PASSENGER DROPOFF WILL HAPPEN
- INFORM BUILDING MANAGERS
  - DISPLAY EFFICIENCY
  - DISPLAY OPERATIONAL STATUS
  - PROVIDE INFORMATION FOR OTHER BUILDING SUBSYSTEMS
- INFORM MAINTAINERS
  - DIAGNOSIS
  - PROGNOSIS
  - SELF-TEST
  - TIME FOR PERIODIC MAINTENANCE

SET MODES

- UP-PEAK
- DOWN-PEAK
- "NORMAL"
- FIRE RECALL
- FIRE OPERATION

RoSES = Robust Self-configuring Embedded Systems

- **Research Context:**
  fine grain distributed embedded systems
- **Research vision:**
  Product families + auto-reconfiguration =
  - Operation with failed components
  - Automatic integration of inexact spares
  - Automatic integration of upgrades
  - Fine-grain product family capability
- **Potential Impact:**
  - Logical component interfaces + config mgr.
  - Fine-grain software component support
  - Architectures that are naturally resilient
- **What we're really learning is where all the difficult research issues are!**



Smart Sensors/Actuators

Basic S/A Device

Baseline Sensor SW Functionality

Local CPU & Memory

SW Adapter for High Level Logical Interface

SW Compute/ Control Functions

Dynamic Interface to Object Bus

System Variables/Network

CUSTOMIZATION MANAGER

Adapter Repository

Some Specification & Evaluation Research Issues

- Allocating software to available components
  - Problem: given fixed resources, how to you maximize utility?
  - What baseline set of components gives most reconfiguration flexibility?
- System specification
  - Product family architecture specification
  - Specification of utility for different features & feature sets
  - When/how to determine HW/SW/Mechanical/Business tradeoffs
- Evaluation
  - Is a system really "working" when it is partially disabled?
  - Safety/certification of component-based systems with many failure modes
- Design
  - Many real embedded systems have global modes that break design methods
    - » Do you do a distinct system design for each mode and merge?
  - Many real systems are hybrid discrete+continuous
- Implementation
  - Software runtime infrastructure (Jini was a poor fit to an embedded network)
  - Real time scheduling for distributed networked system
  - Security of embedded+enterprise combined system

Big Open Issues

- **How do we know which architecture to use and when?**
  - Can we evaluate architectures for properties such as graceful degradation in the abstract?
  - But, at least now we know that this is a decision to consider – there is more than just one possibility
- **Can system architects be trained, or must they be born?**
  - "Most really good architectures come from a single architect"
  - If functional architecture isn't the best answer, what is?
    - Or is good enough really good enough?

Review

- **System Architecture via patterns for multiple system views**
  - Multiple views for most systems are essential
    - Hardware + Software + Communication + Control + others
  - There is no "free lunch" – you probably have to choose between
    - Be constrained to a 1-D/low-D decomposition (e.g., functional architecture)

      vs.
    - Deal with allocation incompatibilities when fusing a many-D decomposition
  - Multiple architectures mean many different tradeoffs
    - System-level tradeoffs between mechanical, HW, SW, and other implementation methods are common
    - Existence of non-architectural options mean some tradeoffs happen between technical and business/non-technical system layers!
- **Functional architecture: yes, there is a multi-view recipe!**
  - But it usually produces mediocre system architectures
  - Doing better is a deep research topic