Using Architectural Properties to Model System-Wide Graceful Degradation

Charles Shelton    Philip Koopman

Workshop on Architecting Dependable Systems
International Conference on Software Engineering
May 25, 2002
Scalable Graceful Degradation

- **Graceful degradation can increase system dependability**
  - Individual component and subsystem failures reduce functionality but do not cause a system failure
  - Non-critical features shed while critical features are preserved

- **Current practice for specifying graceful degradation:** [Herlihy91]
  - Specify a “relaxation lattice” of system constraints
    - Constraints are relaxed as failures occur
  - Lattice is exponentially complex with number of constraints
  - Must specify a specific system response for each lattice point

- **IDEA:** Exploit system decomposition into subsystems and components
  - **Goal:** Create a more scalable model for graceful degradation
Focus: Dependable Embedded Systems

- **Embedded systems are increasingly software driven**
  - Complex software systems necessary to implement more features and functionality
  - “Smart” sensors and actuators encourage more distributed/networked systems

- **But, they have high dependability requirements…**
  - System failures have high consequences (loss of life, money)
  - Software patch or upgrade is often impractical

- **… and are extremely cost sensitive**
  - System-wide replication for dependability is cost prohibitive

How can we get scalable, graceful degradation for them?
Utility and Graceful Degradation

◆ **Utility** - measure of system’s usefulness
  - Different for each problem domain
  - Incorporate functionality, reliability, performance, etc.

◆ **System Utility is a function of component utilities**
  - Maximum Utility – All system components working
  - Some Utility – degraded system operation
  - Zero Utility – System failure

◆ **Graceful Degradation goal:**
  - Component failures proportionately reduce system utility
  - Ideally, each functional subsystem retains residual functionality
    - Previous work assumed whole-subsystem failures, not component failures
Key Is Handling System Configurations

◆ **Focus on software component configurations**
  - Assume individual software components either working or failed
  - $2^n$ possible configurations of $n$ software components
    - Each configuration can be represented as a string of $n$ bits

◆ **For sufficient graceful degradation we want:**
  - Many valid configurations
  - System bit string values with low Hamming distance have small differences in utility

◆ **Previous work considered the subsystem level**
  - What if instead we looked at fine grain component level?
  - “$n$” goes from a handful to perhaps hundreds
  - Analysis complexity is $O(2^n)$ – are we crazy?
Example Elevator System Architecture

Elevator Architecture
September 6, 2000
Philip Koopman
18-540 / CMU ECE

RED TEXT: Environmental Model
NUMBER: Replication
- Network Connection
- Electromechanical Connection
- Passenger Action

SYSTEM COMPONENTS
- Sensor
- Actuator
- Complex Behavior
- System State

OBJECT

Data Network

CAR
- CarCall[i]
- MAXFLOORS
- CarButton Control[f]
- CarLiftControl[d]
- Lantern Control[d]
- CarPosition Indicator
- CarLight[f]
- CarDoor Control[a]

PASSENGER
- Passenger[p]

SAFETY
- Emergency Brake
- DriveControl

DRIVE
- Drive

FLOOR
- MAXFLOORS
- HallCall[f, d]
- HallButton Control[f, d]
- HallLight[f, d]

Doors
- DoorPosition[i]
- DoorClosed[i]
- DoorOpen[i]
- DoorReversal[i]
- DoorControl[i]
- DoorMotor[i]

- MAXFLOORS
- MINFLOORS
Utility Analysis

- Utility analysis for all system configurations is $O(2^n)$
  - *But*, we are only interested in valid configurations
    - Correct sensors and actuators present for desired functionality

- Software architecture constrains valid configurations
  - Component and interface definitions
  - Organization of components into subsystems
  - Dependencies between subsystems

- Develop system model for scalable analysis
  - System data flow graph derived from interfaces among components
  - *Feature subsets* defined from subgraphs of components
Feature Subsets

 prominently

 A feature subset is a subset of components that outputs a set of system variables

 - Defined by component output interfaces
 - A feature subset with $k << n$ components has $2^k$ configurations
 - Each feature subset defines (potentially overlapping) subsystems

 Allow hierarchical definition of subsystems

 - Feature subsets can contain other feature subsets as components

 Identify critical and non-critical feature subsets

 - System utility is zero when critical feature subset’s utility is zero

 Finding valid system configurations made easier:

 - Only determine valid configurations for each critical feature subset
 - Exploit fact that many systems have decoupled subsystems
Drive Control Feature Subsets

- Door Control Feature Subset
- Drive Speed Sensor
- Car Position Sensor
- Door Closed Sensor

Components that output Door Control data

Components that output Desired Floor data

Desired Floor Feature Subset

AtFloor Sensors (one per floor)

VirtualAtFloor (one per floor)

Drive Control Feature Subset

Drive Control

Software Component

Sensor

Actuator

Data Flow

Feature Subset

Drive Motor Actuator
Our system model for graceful degradation enables scalable system analysis

- Use feature subset definitions to simplify configuration analysis
  - Only consider subsets of each configuration bit string relevant to a feature subset
  - If average feature subset has \( k << n \) components, analysis reduces from \( O(2^n) \) to \( O(n/k * 2^k) \)

- Can determine all valid configurations without examining every possible component configuration
  - Encapsulate graceful degradation analysis within each subsystem

Model provides structured view of system-wide graceful degradation

- Identify system properties that improve graceful degradation
- Identify critical subsystems that require extra redundancy
- Basis to compare graceful degradation of similar configurations