Model-Based Software Design and Adaptation

Hassan Gomaa
and
Mohamed Hussein
Department of Information and Software Engineering
George Mason University
Fairfax, VA
USA

hgomaag@gmu.edu

SEAMS 07
Minneapolis, May 2007
Dynamic Software Adaptation in Safety Critical Systems

- Safety Critical Systems
  - Highly available, Time critical
- Examples: air traffic control systems, spacecraft, automotive and aircraft control systems

- Challenge
  - Evolve the configuration of software application at run-time
  - Application must be operational during dynamic reconfiguration
Approach

- Most software product line research aimed at deriving different family members from
  - Product line architecture + implementation
  - At Configuration Time
  - NOT at Run Time

- Research approach
  - Model all configurations of safety critical system as product line members
  - Dynamically change from one family member to a different family member at Run Time
  - Develop Software Reconfiguration Patterns
Related Work

- Dynamic Reconfiguration Environments
  - Dynamically change a software configuration to a new configuration
    - Conic / Regis (Imperial College)
    - C2 (UC Irvine)

- Software Design and Architectural Patterns
  - Systematic reuse concepts for the design of applications
    - Gamma et al, Buschmann et al

- Reusable and Configurable Product Line Architectures
  - Product line design methods and tools (GMU EDLC/KBSEE)
    - KOALA (Philips)
    - FAST (Lucent)
    - PULSE/KOBRA (Fraunhofer)
Evolutionary Product Line Life Cycle
- Build, then Deploy

Product Line Requirements and Analysis Models, Product Line Architecture, Reusable Components

Product Line Requirements

Product Line Engineering

Product Line Reuse Library

Target System Requirements

Application Engineering

Executable Target System

Unsatisfied Requirements, Errors, Adaptations

Copyright 2007 H. Gomaa
Software Architectural Patterns

- Software Architectural Patterns [Buschmann, Shaw]
  - Recurring architectures used in various software applications
- Goal: Design Software Architecture from
  - Software Architectural Patterns
- Architectural Structure Patterns
  - Address structure of major subsystems
- Architectural Communication Patterns
  - Reusable interaction sequences between components
Architectural Structure Patterns for Software Product Lines

- Layered patterns *very important for evolution*
  - Layers of Abstraction
  - Kernel
- Client/Server patterns
  - Basic Client/Server
  - Client/Broker/Server
  - Client/Agent/Server
- Control Patterns *very important in RT Design*
  - Centralized Control
  - Distributed Control
  - Hierarchical Control
Distributed Control Pattern

- Several control components
- Control is distributed among components
- Each component controls part of system
  - Receives sensor input from input components
  - Executes state machine
  - Controls external environment via output components
  - Communicates with other control components to provide overall control
Hierarchical Control Pattern

- Hierarchical Controller
  - Provides high level control
  - Sends commands to lower level control components
Architectural Communication Patterns for Software Product Lines

- Asynchronous communication patterns
- Synchronous communication patterns

- **Very important for evolutionary design:**

- Broker Communication Patterns
  - Broker forwarding
  - Broker handle
  - Discovery

- Group Communication Patterns
  - Broadcast
  - Subscription/notification
Subscription/Notification Pattern

- Client subscribes to join group
  - Receives messages sent to all members of group
Software Reconfiguration Patterns

- Concept
  - Design software architecture based on software architectural patterns
  - For every software architectural pattern
    - Design a software reconfiguration pattern

- Reconfiguration Pattern
  - Specifies how a set of components cooperate to change the system configuration to a new configuration

- Characteristics of Reconfiguration Pattern
  - Reconfiguration state machine model
    - Component transitions to a state where it can be removed and replaced
  - Reconfiguration Collaboration model
    - Component interactions to change configuration
Reconfiguration State Machine Model

- Basic Model is based on Kramer/Magee
  - Component transitions to a state where it can be reconfigured
    - **Active State**: Component is operational
    - **Passive State**: Component
      - Is not participating in a transaction that it initiated
      - Still participating in other transactions
    - **Quiescent State**: Component
      - Idle
      - Not participating in any transactions
      - Ready to be removed from configuration
Reconfiguration State Machine

Generalized Reconfiguration Statechart

Activate

Passivate
[Processing Transaction]

Transaction Ended

Passivate [Idle]

Quiescent

Active

Reactivate
Software Reconfiguration Patterns

- Approach
  - Develop reconfiguration patterns for well-known software architectural patterns
  - Build software product line architectures using
    - Software architectural patterns
    - Software reconfiguration patterns
- Software Reconfiguration Patterns developed
  - Master-Slave pattern
  - Centralized Control pattern
  - Client / Server pattern
  - Decentralized Control pattern
Software Reconfiguration Patterns

- Master-Slave pattern
  - Master component can be replaced after receiving responses from all Slave components
  - Slave components can be replaced after Master is quiescent

- Centralized Control pattern
  - Removing or replacing any component in the system requires the Central Controller to be quiescent

- Client / Server pattern
  - Client can be added or removed after completing a transaction
  - Server can be removed or replaced after completing current transaction(s)

- Decentralized Control pattern
Decentralized Control Reconfiguration Pattern

- Decentralized Control components communicate with each other
  - Components must notify each other if going quiescent
  - Component can cease to communicate with neighbor but can continue with other processing
Reconfiguration State Machine

Generalized Reconfiguration Statechart

Activate

Passivate [Idle]

Passivate [Processing Transaction]

Transaction Ended

Quiescent

Active

Reactivate
Component Reconfiguration State Machine

- New states to assist components in the reconfiguration process
- Needed for more complex component interactions
  - **Passivating State**: Component
    - Is disengaging itself from transactions
      - It is participating in
      - It has initiated
    - Is not initiating any new transactions
  - **Waiting For Acknowledgement State**: Component
    - Component has sent notification message (s) to disengage itself
      - To interconnected components
Reconfiguration State Machine

Generalized Reconfiguration Statechart

Activate

Passivate [Processing Transaction]

Transaction Ended [All Neighbors Passive]

Passive

Passive Ack From All Neighbors

Quiescent

Transaction Ended [At Least One Neighbor Active]

Transition Started

Passivate [Waiting For Neighbor Response]

Transaction Aborted

Waiting For Acknowledgement

Reactivate
Dynamic Software Reconfiguration Framework

- Manages reconfiguration process
  - Different configurations are members of product line
Dynamic Software Reconfiguration

- Change Management Model
  - Reconfiguration steps to switch from old to new configurations
  - Drive components to
    - Full quiescence, e.g., to unlink and remove
    - Partial quiescence, e.g., to unlink
  - Change configuration commands
    - E.g., to replace component:
      - Quiesce, unlink, remove old component, insert new component, relink, restart
- Prototype developed using Rational Rose RT
Example of Dynamic Software Reconfiguration

- Reconfigurable factory automation SPL architecture
  - Uses: Master-Slave, Client / Server, & Decentralized Control patterns
Conclusions

- **Goal**
  - Dynamically change from one SPL family member to a different family member at Run Time

- **Research Approach**
  - Software Reconfiguration Patterns
  - Dynamic Software Reconfiguration framework
    - Rose RT environment

- **Future work**
  - Develop additional reconfiguration patterns
  - Investigate issues of pattern interaction
  - Investigate performance issues in dynamic reconfiguration
  - Investigate unplanned or unexpected dynamic recovery and reconfiguration issues