Evaluating Concurrent Software Architectures using Petri Nets

Dr. Rob Pettit
The Aerospace Corp. / George Mason Univ.

Petri Nets 2009
Paris
Motivation

• Software Architecture Evaluation
  • Early and often evaluation of architecture properties
    – Interface issues
    – Concurrency issues
    – Verifying correct behavior before implementation
    – Much more cost effective to fix problems in early stages of development

• Independent assessments
  – Need tools and techniques that can be applied separately from contractor development tools
  – Must be able to relate analysis results to original architecture
Terms

- Architecture
- Modeling Language
- Behavioral Pattern
- Behavioral Analysis
Software Architecture

For our purposes, software architecture consists of:

- Software Units (Objects)
  - Active vs. Passive
  - Described by behavioral pattern
  - Connections between units
- Messages
  - Asynchronous vs. synchronous
- Operations
  - Access to shared resources
Modeling Languages

- Modeling Languages Provide Facilities for capturing architectural properties
  - Static vs. dynamic concerns
  - Concurrent threads of control
  - Message passing
  - Logic flow
  - Etc.

- Popular modeling languages include:
  - UML
  - AADL
Modeling Languages

• Basically two choices for modern concurrent software design modeling:
  – AADL - Architecture Analysis and Design Language
    • Language-based origin
    • Focus run-time architectural behavior
  – UML - Unified Modeling Language
    • Diagram-based origin
    • Focus on structural and functional aspects of software system

• Both are viable approaches
• Can even be complimentary
  – UML for platform-independent model
  – AADL for platform-specific model
Behavioral Patterns

• Behavioral patterns capture basic roles / responsibilities of a software unit
  – We will use these to further describe UML classes
• Behavioral patterns captured through UML stereotypes
• In this tutorial, behavioral patterns based on COMET method of Gomaa
• Basic patterns include:
  – Boundary
  – Control
  – Algorithm
  – Entity
Behavioral Analysis

• As opposed to static analysis, behavioral analysis looks at the dynamic, behavioral properties of a software architecture
  – Concurrent execution of objects
  – Effects of message communication choices
  – Utilization of shared resources
  – Exercising use case scenarios
  – Evaluating trade-off decisions
• Encompasses functional and non-functional aspects of architecture
Additive Approach to Analysis

• Early and often approach to architectural analysis means that we will be conducting analyses at increasing levels of detail throughout the lifecycle
• Using an additive approach, we build on earlier analyses rather than discard them
Levels of Architectural Analysis

• Architectural analysis can be performed at various levels:
  – Software system context
    • Illustration of external interfaces and input / output mapping
  – Initial (sub)system architecture
    • Basic interactions between objects in a (sub)system
  – Object-level algorithms and state machines
    • Detailed exploration of object interactions relative to their algorithmic or state processing
  – Integrated resources / platforms
    • Added fidelity through inclusion of system resources and platform specific information
Modeling Concurrent Software Architecture Designs with UML
UML Overview

• Three types of models of interest to us...
  – Use Cases
  – Structural
  – Behavioral
Use Case Models

• Capture black-box functionality of system
• Often useful for:
  – Communication between stakeholders and development team
  – Dividing work
  – Planning iterations
  – Structuring test cases
• Two main elements of a use case model:
  – Actors
    • External roles that interact with system
      – Can be human, device, system, etc.
  – Use cases
    • Black-box functions
      – Represented at a high level on the diagrams
      – Must be paired with specifications detailing the steps to satisfy use case
Use Case Notation

Temperature Control System

- Regulate Temperature
- Regulate Humidity
- Display Current Status

Operator
Structural Models

• Capture static, structural properties of the software design
• UML structural diagrams include:
  – Class diagram
    • Primary diagram for capturing properties and static relationships between classes
    • For architectural analysis, gives us insight into the roles, responsibilities, and behavioral patterns
  – Component diagram
    • Groups classes into component packages
    • Useful for capturing subsystems and their interfaces
  – Deployment diagram
    • Assigns components to physical nodes
    • Not often used in practice
UML Component Diagram
UML Deployment Diagram
Behavioral Models

• Behavioral models capture dynamic, behavioral properties
  – Between class instances (objects)
  – Within a class instance

• UML behavioral diagrams include:
  – Interaction diagrams
    • Sequence diagram
    • Communication diagram
  – State machine diagrams
  – Activity diagrams
  – Timing diagrams
UML Sequence Diagram

ENV

«boundary»
: Thermostat

«control»
: Temperature Control

«entity»
: Temperature

«boundary»
: Cooler

changeMode(cool)

cool()
UML Communication Diagram
Case Study

Photo Courtesy NASA/JPL-Caltech
Simplified Mars Rover

- Autonomous rover with a mission to search for mineral deposits
- System includes:
  - 2 motors
    - Forward, reverse, and steering (by differential control)
  - Spectrometer
    - Detects potential mineral deposits
  - Communication link
    - Receives commands from and transmits findings to base station
  - Position sensor
    - Used for Martian navigation
  - Ultrasonic sensor
    - Used to detect terrain hazards
Development Process Framework

• Process framework in this tutorial applies a use-case driven, iterative process
  – Based on the COMET method by Gomaa
  – Compatible with the Unified Process
• Use cases outline functionality to be developed and tested
• UML models are constructed in an additive approach
  – Balanced static and dynamic models
  – Significant iteration between models
• Models are language-neutral through the detailed design of each iteration
Process Framework

- COMET:
  - Requirements
    - Use cases
  - Analysis
    - Class identification
    - Use case realization
  - Design
    - Refine class design
    - Develop concurrent task architecture
Use Case Modeling

• For software modeling, use cases form the bridge between systems requirements, software requirements, and the subsequent software design

• Actors
  – Anything “acting” on the system
  – For embedded systems, this is typically not a human

• Use Cases
  – High-level, black box functionality
  – Must be accompanied with specification describing the sequence of events needed to complete the use case
## Use Case Specification

<table>
<thead>
<tr>
<th>Use Case: UseCaseName</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: xx</td>
</tr>
<tr>
<td>Requirements Traceability:</td>
</tr>
<tr>
<td>Brief Description:</td>
</tr>
<tr>
<td>Actors:</td>
</tr>
<tr>
<td>•</td>
</tr>
<tr>
<td>Preconditions:</td>
</tr>
<tr>
<td>1. System states that must be satisfied prior to executing the use case</td>
</tr>
<tr>
<td>Main Flow:</td>
</tr>
<tr>
<td>1. Sequence of events for the main use case flow</td>
</tr>
<tr>
<td>Postconditions:</td>
</tr>
<tr>
<td>1. System states that result from the use case execution</td>
</tr>
<tr>
<td>Alternative Flows:</td>
</tr>
<tr>
<td>1. Alternative paths to the main flow</td>
</tr>
</tbody>
</table>
Case Study: Rover Use Cases

- SpectrometerDevice
- UltrasonicDevice
- CommDevice
- PosSensorDevice

Rover Subsystem

Perform Rover Mission

- Avoid Obstacles
- Transmit Report

«extend»

«include»
Static and Dynamic Modeling

• **Static modeling**
  – Identifies classes needed to satisfy use cases
  – Captures static, structural properties about classes

• **Dynamic modeling**
  – Captures behavioral properties about classes and their instances (objects)
  – Interaction diagrams capture behavior across objects
  – State machines capture behavior within objects

• **Constructed in close iteration**
  – Identify some classes; sketch their interactions; identify more classes; elaborate on details
Identifying Classes

• Classes discovered from requirements documents; use case specifications; domain experts; etc.
  – Look for problem domain abstractions
  – Think about structuring highly-cohesive, loosely-coupled units
  – Plan for change
• UML stereotypes can help to identify classes through behavioral patterns:
  – «boundary»
    • external device/system/etc. that your software must interface with
  – «control»
    • Controls behavior among a set of objects
    • Often one controller per use case
    • Often associated with a state machine
  – «algorithm»
    • Separate specialized algorithms for maintainability
  – «entity»
    • Captures persistent data
Initial Static Modeling

- **Focus on:**
  - Static structure of system
  - Problem-domain abstractions
  - Capturing system context
    - "<<boundary>>" class for each external interface

- **Class diagram:**
  - Classes
    - Stereotypes
    - Attributes
    - Roles / responsibilities
  - Relationships
    - Static relationships
      - Structural, not behavioral
Rover Context Diagram
Rover Initial Objects

- «boundary»
  - Detection
    - Spectrometer
    - UltraSensor
  - MotorA
  - MotorB
  - PosSensor
  - CommLink

- «control»
  - RoverControl

- «algorithm»
  - Navigation

- «entity»
  - Map
Starting to Construct Analyses

• Basic process of Petri net analysis:
  – Develop context model
  – Develop architecture model
  – Expand individual objects with behavioral pattern templates
  – Add details to objects
  – Add platform details
Developing CPN Context

• Represents top-level system view
• Mirrors UML context diagram
  – Places represent externals (input and output)
  – Central system transition decomposed into lower-level architectural model
• Adds test stubs per use case model
• Note: One critical aspect is deciding how much to model in one set of analyses
  – System as a whole or individual subsystems?
  – Often more appropriate to model subsystems
Rover CPN Context Model
Initial Dynamic Modeling

• Use cases are “realized” by creating interaction diagrams
  – Sequence or Communication diagrams
• Focus on basic object behavior
  – Not concerned with message implementation or concurrent behavior yet
• Inter-object modeling
  – Sequence or communication diagrams (slides 20-21)
• Intra-object modeling
  – State machines (slide 22)
• Performed in iteration with class identification and static modeling
• Consistency among models is important
Advanced Dynamic Modeling

• Decisions on concurrent tasks
  – Based on UML active objects
  – Two stage approach
    • Identification of potential tasks
    • Clustering to arrive at a final task architecture

• Decisions on message communication
  – Asynchronous
  – Synchronous
Rover Sequence Diagram
“Perform Rover Mission”
Developing CPN Architecture Model

• Expand system-level transition
• Populate architecture-level with:
  – Transition for every concurrent object (active object / task)
  – Place for message send / receive between tasks
    • May also add transition for higher-level message processing logic
  – Place for each operation on passive object
CPN Architecture for Rover
Adding Object Details

• Need to elaborate each object in the architecture
  – Apply behavioral patterns
    • Decompose each task transition into corresponding CPN behavioral pattern template
  – Add object-specific information
    • Input / output mappings
    • Execution periods
    • Data types (via colored tokens)
    • etc.
  – May add as much or as little detail as desired based on analytical goals
    • Always a trade-off between increased fidelity and increased state space
Looking into a single object...
More on CPN Templates

Let’s look at some models in CPNtools…
Adding Platform Specifics

• Basic CPN architecture model assumes nothing about underlying platform
  – Unbounded event arrival
  – Concurrency not limited by system processors
• Adding platform details gives us another level of information to strengthen the fidelity of our analyses
• In the model-driven development philosophy, platform information should come from a Platform Specific Model (PSM)
  – However, this is often not present, leaving the analyst to create themselves or just gather the information from various platform documentation
Conducting Analyses on the CPN Models

• CPN analysis can be conducted at different levels
  – Choice depends on audience; goals for analysis; stage of development
• Black-box analysis
  – Apply test stubs to context-level model and observe inputs / outputs to (sub)system
• Architecture analysis
  – Observe object interactions at architecture level
    • Attention to message flows; queue lengths; throughput; resource contention;
      ...
• Logic flow within objects
  – State-based behavior
  – Detailed look at resource utilization, dependencies, data transformation, timing, …
Example from Rover Context
Examples in CPNTools
Conclusions

• CPNs provide an excellent formalism to independently assess concurrent software architectures
• Hierarchical construction allows us to look at different aspects of architecture depending on our analytical goals
• Allows independent verification and validate of software architecture designs
  – Does not require IV&V organization to use (or own!) contractor tools
  – CPN templates promote faster construction of models and allow CPN models to more easily map back to original architecture models
    • Efficiency traded for clarity and round-trip engineering
Conclusions (cont.)

- Some initial success with automation
  - Scripts have allowed initial CPN object transitions to be created from UML objects
  - More work still needed
- Recent advances in CPNTools could provide additional visual benefits by being able to have CPN simulation update UML diagrams during execution
- Would be nice to have hierarchical places in addition to transitions
Backup
Identification of Concurrent Tasks

• Concurrent task in UML is represented as an active object
  – Executes a thread of control
• Nearly everything except an «entity» can be a task
• But...
  – Tradeoff decisions are needed to balance maintainability, complexity, and flexibility of design
  – COMET accomplishes this through a series of clustering guidelines
• Before we can cluster objects into composite tasks, we need to think about:
  – Asynchronous vs. Periodic behavior
  – Input vs. Output vs. Input/Output
  – Time-Critical vs. non Time-Critical responsibilities
Example: Temperature Control

- **Operator**
  - 1. changeMode()

- **Thermostat**
  - 4.1. fanOnly()
  - 4.2. heat()
  - 4.3. cool()
  - 4.4. Off()

- **Temperature Control**
  - 5.1. start(stop())
  - 5.2. start()
  - 5.3. start()
  - 5.4. stop()
  - 5.5. start()
  - 5.6. stop()

- **Cooler**
  - 5.1. start(stop())

- **Heater**
  - 5.2. start()

- **Fan**
  - 5.5. start()

- **Temperature Sensor**
  - 3. setCurrent()

**Entities:**
- Temperature
- Cool
- Heat
- Fan
- Heater
- Sensor

**Boundaries:**
- Input
- Output
- Asynchronous

**Actions:**
- start
- stop
- changeMode
- setDesired
- getCurrent
- setCurrent

**Periodic:**
- start(stop())
Clustering Criteria

• Clustering criteria provides guidelines for grouping objects into a single task / thread of control
  – Maximizes efficiency of concurrent solution by identifying task structures that make the most use out of concurrent execution
  – Minimizes complexity by reducing number of actual tasks

• Three primary forms of clustering are used:
  – Temporal
  – Sequential
  – Control
Temporal Clustering

- Activities activated by same event
- Activities activated on the same period (or harmonic)
- No sequential dependency
- Considerations:
  - Task priorities
  - Anticipated changes in distributed architecture
  - Related functionality of tasks
  - Different sampling rates
- Typical use: polling input devices
Example – Temporal Clustering

:Humidity

:Temperature

:TemperatureSensor

:HumiditySensor

{periodic, input, Frequency = 100ms}

{periodic, input, Frequency = 100ms}
Sequential Clustering

- Activities must (always) be executed sequentially
- Considerations:
  - Sequence terminated with candidate task that does not send inter-task message
  - Candidate tasks in the sequence that can receive inputs from other sources
  - Time critical candidate tasks
- Often seen with algorithmic classes that are tightly coupled to a «boundary» class for output
Example – Sequential Clustering

Asynchronous, output

Periodic, Frequency = 500ms

1. getCurr()
   :Humidity

2. getCurr()
   :Temperature

3. update()

4. update()
   UserDisplay

{asynchronous, output}
Control Clustering

• Control object grouped with objects it activates
• Considerations:
  – Actions triggered by transitions
    • Potentially clustered with control
  – Activities enabled or disabled by transition
    • Separate task
  – Can be combined with sequential clustering
• Generally found with control objects that provide sole control over set of output-only «boundary» objects
Example: Control Clustering

1. changeMode
2.1. setDesired
2.2. getCurrent
3. setCurrent
4.1. fanOnly
4.2. heat
4.3. cool
4.4. Off
5.1. start/stop
5.2. start
5.3. start
5.4. stop
5.5. start
5.6. stop
6. periodic, input
7. asynchronou
8. s
9. input
10. asynchronou
11. s
12. output
13. asynchronou
14. s
15. output
16. entity
17. Temperature
18. control
19. TemperatureControl
20. boundary
21. Thermostat
Modeling Clustered Tasks

- Static and dynamic models need to be updated to reflect task architecture
- Class diagrams
  - Use composition to show clustering structure
- Interaction diagrams
  - Replace low-level objects with cluster
  - Update message communication
  - Illustrated detailed cluster behavior
Class Diagram for Clustered Task
Concurrent Task Architecture
Concurrent Task Architecture – Specifying Message Communication
Inter-Object Communication

• External input to objects occurs via events
• Communication between active objects occurs through messages
  – Asynchronous (loosely-coupled) message communication
    • Use when sender can not afford to be blocked
    • Determine message ordering
      – FIFO, priority, etc.
  – Synchronous (tightly-coupled) message communication
    • Use when communication must be synchronized
    • With reply
    • Without reply
• Communication between passive objects occurs through operation calls