UML/MDA Reality Check: Heterogeneous Architecture Styles

Nicolas F. Rouquette
Principal Software Architect
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA, USA
+1 (818) 354-9600
mailto:Nicolas.Rouquette@jpl.nasa.gov
Outline

- Evolving Standards & Pragmatic Reality
- Programming & Architecting // Software & System engineering
- Unusual characteristics of space missions
- Differences of Engineering & Culture; Bridging Gaps
- Managing interactions
- The Mission Data System
- Heterogeneous Architecture Styles
- Issues of Trust
- State, State Analysis & Software Architecture
- Frameworks, Genericity, Polymorphism & Ensuing Tradeoffs
- Ports @ the intersection of structure, behavior & coordination
- Component Architecture & Transformation
- Transformation Challenges
Evolving Standards & Pragmatic Reality

- **Evolving Standards:**
  - Today's Bleeding Edge

- **Pragmatic Reality:**
  - Still difficult to evaluate tools for:
    - Consistency across diagrams
    - A coherent subset of UML

- **Advances with UML:**
  - 2.0 has a component model!

- **Limits on Software Architecture**
  - Component, Connector, Port, Link

- **Advances with MDA:**
  - PIM, PSM
  - Transformations as Models

- **Limits on Transformations:**
  - Code-level Templates
    - PSM => Code
  - Query /Construction Rules
    - PIM => PSM
    - PSM => PSM

Architects

Programmers

UML/MDA playground
Programming ...

- **Motto**
  - "Trust the source, not the doc!" => "The truth is in the code!"

- **Pragmatic Programmer = Greatest synergy & support yet!**
  - Great advances in many related fields of programming:
    - Technology, Methodology & Tools
    - Generics, Separation Of Concerns, Patterns, Idioms
    - Refactoring, Template Code Generation

- **MDA's Approach to Scalability & Complexity**
  - UML/PSM profile: model is a 1-1 reflection of the code
  - Model / Code Synchronization & Associativity

- **Programming is the Dominant Paradigm**
• Facts
  • “The model may be stale!” => “The model is reference doc!”

• Pragmatic Architect = First, pick your biases…
  • Plenty of opinions on what architecture is & should be
    • Platform standards (COM, J2EE, etc.)
    • Architecture Styles (3-tier, client/server, pipe/filter)
    • Architecture Definition Languages (Arch, xADL, Acme)

• MDA’s Approach to Scalability & Complexity
  • Hierarchy of PIM / PSM models
  • Which abstractions to use?

• Architecting is the Dominant Buzzword
Programmers & Architects: ...sharing successes & failures

- Informal perspective from anthropology:
  - Software engineers share a common gene pool
    - Architecture mismatch => early mistakes
    - Programming bugs => latent flaws

- Extinction!

- Outlook for software engineering species
  - Significant evolution but still affected with the same old diseases
  - Bleak future for commercial success & long-term survival
  - Species comparison: viruses have greater success at lower cost

- A brighter perspective:
  - UML/MDA:
    - Bridging the gap between programmers & architects
  - JPL follows a similar path in the Mission Data System project
    - Bridging the gap between software & system engineering
Unusual Characteristics of Space Missions

- Infrequent, scheduled communication
  - The ‘network’ is not continuously available

- Distance, Time, Bandwidth & Navigation
  - Multiple clocks: ~10 hour round-trip light-time delay to Pluto
  - Physics limit bandwidth: ~300 bits/sec from Pluto
  - Communication: point antenna at where Earth will be *when the signal arrives*, not at where it is now

- Special Relativity and Time Dilation
  - Though spacecraft velocity is a tiny fraction of lightspeed, navigation must take relativistic effects into account

- Tight Coupling and Resources
  - In a resource-limited system, ‘everything affects everything’.
  - Managing interactions is fundamental to good design.

- Verification & Validation
  - Scenario-based testing will continue to be important, but combinatorics demand additional approaches such as model checking
The principle risk to success is **miscommunication**
   - What systems engineers want can be **hard to express**
   - What software engineers build can be **hard to understand**
Differences in Engineering Perspectives

Systems engineering is **outward** looking

- Mission scenarios
- Functional decomposition
- Performance requirements
- Resource allocations
- Command and telemetry
- Operational constraints
- Control laws
- Failure Modes & Test

Software engineering is **inward** looking

Languages & Operating Systems
Scheduling
Patterns & idioms
Data representation, serialization...
Separation of concerns
Exception Safety
Programmers & Architects: Different Cultures to Reconcile

- Two issues
  - Reconciling Engineering Differences
  - Strengthening Architecture Semantics

- Reconciling Engineering Differences => Communication
  - Different levels of abstraction may lead to architecture mismatch
  - UML/MDA:
    - Architecture is distorted or lost somewhere between PIM / PSM
  - JPL's Mission Data System (MDS) project:
    - State is a central, unifying concern

- Strengthening Architecture Semantics => Interactions
  - Two architecture styles: State Analysis & Software Architecture
  - Accepting architecture style heterogeneity
    - Avoid representational conundrums & loopholes
    - Focus on semantic bridges among heterogeneous styles
Disciplined "State Analysis" process

A shared set of architectural elements bridges the gap between engineering cultures.

What do you want to achieve?
Move rover to rock.

What's the state to be controlled?
Rover position relative to rock.

How do you know what that is?
Measure position relative to stereo camera.

What does the stereo camera measure?
Distance to terrain features, light level, camera power, health...

How do you control light level?
Wait until the sun is up.

Where is sun relative to horizon?
...

Software architecture process & frameworks

Goal
Network

State Variable
Knowledge

Measurement
Package

Measurement Model
Models

State Model

Etc.
Managing Interactions
In A Unified Approach

- Interactions make software difficult & complex to understand
  - Elements that work separately often fail to work together
  - Combinatorics of interaction are staggering: not easy to get right
  - This is a major source of unreliability

- There are two approaches to this in JPL’s Mission Data System:

State-Based Architecture
- Handles interactions among elements of the system under control
- Outward looking
- Addresses systems engineering issues

Component-Based Architecture
- Handles interactions among elements of the system software
- Inward looking
- Addresses software engineering issues
The Mission Data System Vision

A unified architecture for flight, ground, and test systems that enables missions requiring reliable, advanced software

- Build a **highly reusable core software system** for a wide variety of space mission applications
- Promote modern, synergistic **processes** for **systems and software engineering**
- Establish an improved development life cycle for **more reliable mission software**
- **Reduce development cycle time and cost**
- Satisfy complex mission requirements (e.g., robust *in situ* exploration) and **reduce operations cost** with increased autonomy
Relevance To UML/MDA

- Focus on two concepts:
  - **UML**
  - **Ports** at a 3-way intersection of:
    - Structure, Behavior & Coordination
  - **MDA**
  - Semantic Definition of **Transformations** for:
    - Generation, Optimization, Evaluation, etc.

= 

- **Architecture-Centric Engineering**
  - Embracing & Unifying Heterogeneous Architecture Styles
    - No such thing as “the” architecture
    - Plurality of concerns & abstractions

- Foundations For Reliability
  - “Trust the models!”
  - “Trust the transformations!”

\[ \Rightarrow \text{“Trust the code!”} \]
Why Heterogeneous Architecture Styles?

• Isn't one architecture enough?
  • Conventional wisdom = N views / 1 (all-inclusive) model
  • Problems with Inclusive Approach
    • Irrelevant modeling details for a specific view
    • Model may be internally inconsistent
      – Changes in one view may affect another view
    • Fragile Architecture Modeling

• Embracing Heterogeneity Leads to Parsimony!
  • Each model founded on internal coherence:
    • Only include relevant concerns
    • Choose level of abstraction for conciseness & expressiveness
  • Problems with Heterogeneous Approach
    • Models may be incoherent relative to one another
Heterogeneity from Architecture Parsimony

- **2 Strategies for Choosing an Architecture Foundation**
  - Emphasize Completeness  
    - Kitchen sink of specification & description  
    - Push for executable semantics may result in undecidability  
    - Question that ought to be decidable: compatibility with architecture family  
  - Emphasize Parsimony  
    - Workable core in UML 2.0: Components, Interfaces, Ports, Connectors  
    - Q: Is it a reasonable core?  
    - A. requires probing implications of "UML connector = a link" bias

- **Constructing Heterogeneous Architecture Styles**
  - Desirable Principles:
    - Independent extensibility, Substitutability, Composability, …
  - Weaving in & out between UML 2.0 & software architecture research
    - xArch (Univ. of Calif. Irvine & Carnegie Mellon U.) & many others  
    - ACME (Carnegie Mellon U.)  
    - UML 2.x could become a "neutral, level playing field" for architectures
Issues of Trust

• Key foundations already there in UML / MDA
  • Minimal Core Structure & UML 2.0 Components
  • Heterogeneity & Multiple PIM & PSM models (> 2)
  • Coherency & Model / Model Transformation Bridges

• Tough Challenges Ahead
  • Trust in Models
    • Expressive yet parsimonious
    • Tradeoffs & Informed Decisions
    • Compliance w.r.t. standards, architecture styles & product families
  • Trust in Transformations
    • Specifications with declarative semantics
    • Transformation systems with correctness guarantees
    • Broad applicability beyond code generation (optimization, weaving)
State Analysis:

- A uniform, methodical, and rigorous approach to...
  - Discovering, characterizing, representing, and documenting the states of a system
  - Modeling the behavior of states and relationships among them
  - Capturing the mission objectives in detailed scenarios
  - Keeping track of system constraints and operating rules
  - Describing the methods by which objectives will be achieved
  - Recording information about hardware interfaces and operation

- Original approach: UML 1.3 & code synchronization
  - Semantic clash between UML / state analysis
  - Bleeding edge of UML 1.3 => 2.0 evolution

- Second approach: State Analysis Architecture Style
  - Architecture Style Defined by Common Framework Elements
  - Not a domain-specific architecture (state is a universal concern)
  - Projects need specific extensions (domains, mission, systems, etc..)
  - xADL 2.0 & ACME for software architecture
Gradual discovery process, prompted by a **standard** set of **questions**

- The answer to each question is a piece of the model

- Each answer prompts additional questions, and so on

- Model unfolds a step at a time in terms of **common framework elements** until all the relevant pieces are identified
Standard Questions: Common Framework Elements:

What do you want to achieve?  
*Move rover to rock*

What's the state to be controlled?  
*Rover position relative to rock*

What evidence is there for that state?  
*IMU, wheel rotations, sun sensor, stereo camera*

What does the stereo camera measure?  
*Distance to terrain features, light level, camera power (ON/OFF), camera health*

How do you raise the light level?  
*Wait until the sun is up*

Where is sun relative to horizon?  
*Etc.*
State is Central

- Mission software: monitor and control a system to meet **intent**s
- MDS manages all essential aspects of this function via **state**

..about the system & environment

System behavior

Operator intent & constraints = goals on system state & knowledge

Mediates external interactions with the system
Model Bridge between State Analysis & Software Architecture

Iterative Refinement Process involving:
- Tradeoffs
- Choices
- Refactoring

- State
- Controllers
- Estimators
- Timelines
- Timepoint
- Resources
- Constraints
- Goals
- ...

Elements of State Analysis

Cross-Cutting Concepts & Concerns

- Concepts
- Relations
- Patterns

Elements of Software Architecture

- Components
- Connectors
- Interfaces
- Ports
- Links
- ...

- Concepts
- Cross-Cutting Concepts & Concerns
- Relations
- Patterns
Frameworks & Domain-Specific Adaptation (PIM)

- Frameworks have generic interfaces
  - Interaction protocols => methods (with typed signatures)
  - Separate domain-specificity => generic or parametric data types

1. Naïve OO approach
   - We quickly loose type checking
   - Can a "temperature" state variable be updated with a "velocity" function?

2. Parametric design
   - Preserve type information
   - Brittle if parameters change

3. Generic design with traits & policies
   - Framework remains unchanged
   - Adaptations define specific traits & policies to specialize frameworks
Polymorphism

- **Pros:** Avoids code bloat
- **Cons:** Loss of type information with polymorphic base types
- **Cons:** Difficult to check that a “temperature” estimator is updating a “voltage” state variable

**Parametric design**

- **Cons:** Code bloat for large number of template specializations
- **Cons:** Few people familiar with this technique
- **Cons:** Need binding elements to show parametric specialization

**Generic design with traits & policies**

- **Cons:** Same as parametric design
- **Cons:** Even fewer people are familiar with trait & policy-based design
Generic Software Interfaces: Domain Neutrality in PSM!

- Genericity & specialization concepts can also apply to PSM
- E.g., PSM/C++
- Traits & policies map to template types in C++

```cpp
void update(Traits::StateFunction f);

template< class Traits >
struct update {
  virtual void update(Traits::StateFunction t)=0;
  ...
};

struct myTraits {
  typedef StateFunction StateFunction;
};

template<>
struct update<myTraits>;
```
**Brief Overview of Software Architecture in the Mission Data System**

- Foundations in xADL 2.0 core structure & types schema
  - JPL's extension of xADL designed in fall 2000

- Generics with traits & policies:
  - Interfaces
    - Traits used for parametric types
    - Policies used for coordination behavior (see taxonomy later)
  - Components & Connectors
    - No restrictions on use & abuse of generics

- Experience
  - OO polymorphism is a double edge sword of simplicity
  - Easy to hide type information
  - Possible to loose it entirely (e.g., C)
    
    // java
    // stateFunction must be
    // of type VoltageFunction
    void update(Object stateFunction);
Proxy Exchange & Binding Protocol

- Architecture prescription
  - Construct the architecture
  - Primitive prescription actions: instantiation & binding

- Instantiating a link between two ports
  - Link between component ports => proxy exchange & binding
  - Link between component & connector ports => connector defines the binding

- Primitive link operation = proxy exchange & port binding protocol
  1. Each port on either side of the link creates a proxy of itself
  2. Each port binds to the other port's proxy

1.a) \( x = \text{ctrl}.\text{notifications}.\text{create}_\text{proxy}() \);
1.b) \( y = \text{gsv}.\text{notify}.\text{create}_\text{proxy}() \);
2.a) \( \text{ctrl}.\text{notifications}.\text{bind}_\text{port}(x) \);
2.b) \( \text{gsv}.\text{notify}.\text{bind}_\text{port}(y) \);
Communication Policies Example

- **Simple Port-based communication**
  - An output (required) port (e.g., gsv.notify)
    - has a pointer 'p' to the other port (e.g., ctrl.notifications) on the link
  - An input (provided) port (e.g., ctrl.notifications)
    - has a pointer 'p' to the entity that implements the interface (e.g., ctrl)

- **Simple taxonomy of runtime port configurations**
  - Communication policy: how is the “call” coordinated? (method? Message?)
  - Exception policy: what happens if an exception occurs?
  - Interception policy: can other aspects weave into the communication?
Tradeoffs on Ports as Coordination Pointcuits

Example Taxonomy of connections mechanisms

- Exception support
  - None
  - Partial Report
  - Full Report

- Interception support
  - None
  - Partial Closure
  - Full Closure

- Synchronization mechanisms
  - Direct method call only
  - Asynchronous invocation only
  - Dual support

A profile is a set of policy combinations, e.g:
- The runtime system enforces coordination compatibility linked ports
- The generated C++ code is optimized for the policies in effect
Prescription-time Optimizations (when creating & linking elements)

- Pass-through property
  - No interceptors
  - No exception handlers
  - Direct method call

⇒ A port with pass-through property can be bypassed

- Input bypass
- Input/Output bypass
- Output bypass
**Interceptors & Exception Handlers**

- **Variations**
  - Full interceptors & exception => provides access to method arguments
  - Partial interceptors & exception => provides access to method name

- **Benefits**
  - Exception handlers: Separate of exception detection vs. exception recovery
  - Interceptors: infuse aspect-oriented style into the component/connector style
Component Architecture Brings Many Transformation Challenges

- Component Technology Concerns (adapted from Halloway)
  - Metadata: description & introspection
  - Type information: prescription (construction, changes)
  - Loader architecture: availability of metadata & type information
  - Object lifecycle management: dynamic architectures require a database
  - Exception handling: distinguish exceptions about the architecture itself & the application
  - Other concerns: threading, security, authenticity, etc...

- Current approach
  - Pick a component technology (COM, J2EE, etc...)
  - Make the application "fit" within the limitations and available technology

- Resource-constrained approach
  - Configure the component technology explicitly to fit the requirements
Transformation Challenges

- "try before you buy"
  - (quantitative / qualitative evaluation)
  - Evaluate the consequences of each tradeoff before choosing an option

- What are the expected power consumption profiles?
- Repeat the analysis if the profiles change
Transformation Challenges

- "non-interference guarantee"

- (semantic analysis of interactions)

- Are architecture boundaries (components, connectors) sufficiently solid to isolate a component A (and its realization) from changes in another component’s (e.g., C)

- Requires semantic knowledge of type & object aliasing in the target programming language (C++, Java, etc.)
Transformation Challenges

- "partial evaluation"
  - (protocol compliance)

- 1) Relevance question
   Is a specific change in A's going to affect C in some way?

- 2) Trust question
   What can I do to eliminate the influence changes in A have to C?

- Transformation (as a form of compilation) is not sufficient
Transformation Challenge

1) Transformation history

- Record the tradeoff decisions made and transformations made as a consequence of these decisions
- Models have a version history that includes:
  - Tradeoffs identified
  - Decisions made to resolve tradeoffs
  - Transformation changes made
- Transformation “play back” is informative to understand the thought processes that took place in the past.

2) Historical Explanation

- Starting model may have been simple...
- End result is often much more complex than the original
- What are the root causes of this modeling complexity?
  - Requirement creep? Naïve tradeoffs? Expensive fixes for weaknesses?