Lifecycle Modeling Language (LML) and Systems of Systems (SoS)

Presented to
Systems of Systems Engineering Collaborators Info Exchange
by
Steven H. Dam, Ph.D., ESEP
May 19, 2015
Overview

- Why a New Language?
- Lifecycle Modeling Language Overview
- How can LML support SoS?
Why a New Language?
Complex Systems Implications for Systems Engineering

Complexity has been identified by many as a critical problem facing system engineers.

- Larger and more complex systems (including systems of systems) development creates a need for:
  - Larger and more distributed teams
  - A clear concise way to express the system design (clear and logically consistent semantics)
  - New tools to enable collaboration across the entire lifecycle
Complexity

- With the growth of the Internet and daily changes in IT, systems have become more complex and change more rapidly than ever before.
- Cloud computing gives us new tools to deal with these larger systems.
- Systems engineering methods have not kept up with these changes.
- SE has been relegated to the beginning of the lifecycle.

From a presentation by Dr. Michael Ryschkewitsch, NASA Chief Engineer, at CSER Conference 15 April 2011
How Does SE Typically Respond to Complexity

- Focus on “architecture”
- More complex languages
- More complex procedures
- More layers of abstraction
  - “Systems of Systems”
  - “Family of Systems”
  - “Portfolio Management”
  - “Capability Views”
- Need more time and money!
More Money is a Problem

- Calls for doing more with less continue
- Need for lower labor and tool costs essential for acceptance of SE across the lifecycle

From a presentation by Dr. Michael Ryschkewitsch, NASA Chief Engineer, at CSER Conference 15 April 2011
State of Current “Language”

- In the past decade, the Unified Modeling Language (UML) and now the profile Systems Modeling Language (SysML) have dominated the discussion.

- Why?
  - Perception that software is “the problem”
  - Hence need for an “object” approach

- SysML was designed to relate systems thinking to software development, thus improving communication between systems engineers (SE) and software developers.
Why Objects Are Not the Answer

- Although SysML *may* improve the communication of design between SEs and the software developers it does not communicate well to anyone else
  - No other discipline in the lifecycle uses object oriented design and analysis extensively
  - Users in particular have little interest/acceptance of this technique
  - Software developers who have adopted Agile programming techniques want functional requirements (and resent SEs trying to write software)
  - Many software languages are hybrid object and functional
## Popular Software Languages

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Java</td>
<td>17.110%</td>
<td>-2.60%</td>
<td>Object</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>C</td>
<td>17.087%</td>
<td>+1.82%</td>
<td>Functional</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>C#</td>
<td>8.244%</td>
<td>+1.03%</td>
<td>Hybrid</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>C++</td>
<td>8.047%</td>
<td>-0.71%</td>
<td>Hybrid</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>Objective-C</td>
<td>7.737%</td>
<td>+4.22%</td>
<td>Object</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>PHP</td>
<td>5.555%</td>
<td>-1.01%</td>
<td>Hybrid</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>(Visual) Basic</td>
<td>4.369%</td>
<td>-0.34%</td>
<td>Hybrid</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>JavaScript</td>
<td>3.386%</td>
<td>+1.52%</td>
<td>Functional</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>Python</td>
<td>3.291%</td>
<td>-2.45%</td>
<td>Hybrid</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>Perl</td>
<td>2.703%</td>
<td>+0.73%</td>
<td>Hybrid</td>
</tr>
</tbody>
</table>
So What Do We Do?

- Recognize that our primary job as SEs is to communicate between all stakeholders in the lifecycle
- Be prepared to translate between all the disciplines
- Reduce complexity in our language to facilitate communication
What We Did

In preparing for the cloud computing world of SE we:

- Researched the variety of languages (ontologies) in common use (DM2, SysML, BPMN, IDEF, SREM, etc.)
- Researched the variety of representations (FFBDs, N2, Behavior Diagrams, Class Diagrams, Electrical Engineering Diagrams, etc.)
- Took the best of each of these languages and representations and distilled them down to the essential elements, relationships, attributes, and diagrams

The Result: Lifecycle Modeling Language
LIFECYCLE MODELING LANGUAGE (LML)
OVERVIEW
A language to simplify system design description for the cloud
LML combines the logical constructs with an ontology to capture information:

- SysML – mainly constructs – limited ontology
- DoDAF Metamodel 2.0 (DM2) ontology only

LML simplifies both the “constructs” and ontology to make them more complete, yet easier to use.

Goal: A language that works across the full lifecycle
LML Ontology* Overview

- **Taxonomy**: 
  - 12 primary element classes
  - Many types of each element class
    - Action (types = Function, Activity, Task, etc.)

- **Relationships**: almost all classes related to each other and themselves with consistent words
  - Asset performs Action/Action performed by Asset
  - Hierarchies: decomposed by/decomposes
  - Peer-to-Peer: related to/relates

*Ontology = Taxonomy + relationships among terms and concepts

** Taxonomy = Collection of standardized, defined terms or concepts
LML’s Simplified Schema

- Action
- Artifact
- Asset
  - Resource
- Characteristic
  - Measure
- Connection
  - Conduit
  - Logical
- Cost

  - Decision
  - Input/Output
  - Location
    - Physical, Orbital, Virtual
  - Risk
  - Statement
    - Requirement
  - Time

Supports capturing information throughout the lifecycle
LML Models

Documentation Entities
- Artifact
- Statement/Requirements

Primary Entities
- Action
- Input/Output

Functional Model

Physical Model

Primary Entities
- Asset/Resource
- Connection

Parametric and Program Entities
- Characteristic/Measure
- Cost
- Decision
- Location
- Risk
- Time
LML Primary Entities and Relationships for DoDAF Support
LML Relationships Provide Linkage Needed Between the Classes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- decomposed by*
- orbited by*
- related to*
Diagrams Are Needed for Every Class

- Action Diagram (Mandatory)
- Asset Diagram (Mandatory)
- Spider Diagram (Mandatory)
- Interface Diagrams
  - N2 (Assets or Actions)
- Hierarchy Diagrams
  - Automatically color coded by class
- Time Diagrams
  - Gantt Charts
  - Timeline Diagram
- Location Diagrams
  - Maps for Earth
  - Orbital charts
- Class/Block Definition Diagram
  - Data modeling
- Risk Chart
  - Standard risk/opportunity chart
- Organization Charts
  - Showing lines of communication, as well as lines of authority
- Pie/Bar/Line Charts
  - For cost and performance
- Combined Physical and Functional Diagram
No constructs – only special types of Actions – ones that enable the modeling of command and control/information assurance to capture the critical decisions in your model.
Diagram Comparison: SYSML

Figure 9. UML 2 Activity diagram corresponding to Figure 8.

LML Action Diagram Captures Functional and Data Flow
Execution Logic – Concurrency With Trigger; No Coordination Action

Trigger: Action A enabled, but must wait to execute; Asset A performs Action A

Timeline:
0 sec → y sec → y + x sec

Wait → Action A → Action B

Finish to Start (FS) between B and A
Asset Diagram (mandatory)

Block diagram general form

Block diagram using pictures
Spider Diagram (Mandatory for Traceability)

Shows entities and relationships in visual form.
LML Translation

- Two types of mapping for tailoring:
  - Map names of classes to enable other “schema” models to be used
  - Map symbols used (e.g., change from LML Logic to Electrical Engineering symbols)
  - Enable diagram translations (e.g., Action Diagram to IDEF 0)

<table>
<thead>
<tr>
<th>LML Class</th>
<th>DM2</th>
<th>SysML</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Activity</td>
<td>Activity</td>
<td></td>
</tr>
<tr>
<td>Asset</td>
<td>Performer</td>
<td>Actor</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LML Symbol</th>
<th>Electrical Engineering</th>
<th>BPMN</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Translation to DM2

The diagram illustrates the relationships between different concepts such as Activity, Capability, Resource, Performer, and others. It shows how these concepts are interconnected through relationships like "is-performable-under," "requires-ability-to-perform," "achieves-desired-effect," and others.

Not shown but implied by the IDEAS Foundation:
- Everything is 4-D and so has temporal parts, i.e., states
- Everything has parts
- Everything has subtypes

anything can have Measures
## DM2 Conceptual Model to LML Schema Mapping

<table>
<thead>
<tr>
<th>DM2 Schema Element (Conceptual)</th>
<th>LML Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>Action</td>
</tr>
<tr>
<td>Capability</td>
<td>Action with “Capability” type</td>
</tr>
<tr>
<td>Condition</td>
<td>Characteristic with “Condition” type</td>
</tr>
<tr>
<td>Information/Data</td>
<td>Input/Output</td>
</tr>
<tr>
<td>Desired Effect</td>
<td>Statement with “Desired Effect” type</td>
</tr>
<tr>
<td>Guidance</td>
<td>Statement with “Guidance” type</td>
</tr>
<tr>
<td>Measure</td>
<td>Measure</td>
</tr>
<tr>
<td>Measure Type</td>
<td>Measure Type</td>
</tr>
<tr>
<td>Location</td>
<td>Location</td>
</tr>
<tr>
<td>Project</td>
<td>Action with “Project” type</td>
</tr>
<tr>
<td>Resource</td>
<td>Asset with types for “Materiel,” “Organization,” etc.</td>
</tr>
<tr>
<td>Skill</td>
<td>Characteristic with “Skill” type</td>
</tr>
<tr>
<td>Vision</td>
<td>Statement with “Vision” type</td>
</tr>
</tbody>
</table>
How can LML support SoS?
Systems of Systems

- Definition*: “An SoS is defined as a set or arrangement of systems that results when independent and useful systems are integrated into a larger system that delivers unique capabilities”

- SoS’s can be
  - Virtual (lack central management and purpose)
  - Collaborative (voluntary interaction)
  - Acknowledged (independent, with higher level coordination)
  - Directed (integrated)

- The common denominator in all the SoS types: systems are dependent on other systems

*From Systems Engineering Guide for Systems of Systems
How do we capture and manage dependencies?

- First, we need to identify the relationship between the different systems
  - LML provides a set of relationships between Assets and Actions (Programs) that can capture this traceability
    - Assets are related to other Assets
    - Actions are related to other Actions
    - Assets perform Actions
  - Note the “related to/relates” relationships have an attribute for context
    - This attribute enables you to identify which Asset is dependent on the other
  - If you want to add another relationship between Assets and/or Actions, LML encourages extensions for specific domains (e.g., DoDAF & SysML)
What Next?

- We want to establish the relationship between the different system schedules
  - The Time entity can be used to capture specific milestones, which can then also be related to one another using the related to/relates relationship
  - Visualization suggested is a timeline chart
    - Timeline charts can be created and compared
  - LML also provides another mechanism for time – the duration and start attributes for Action entities provide a means to capture tasks and milestones as part of the program process model
- Note that the language provides more than one way to capture and express the necessary information giving the analyst some flexibility to communicate to a broad audience
Capture other program information

- LML’s ontology provides a means to capture other program information, such as Artifacts, Statement/Requirements, Input/Outputs (e.g., deliverables), Risks, Decisions, Location, and Costs
- This information can be related to each other within and between programs
- Critical information (dependencies) between the programs can also be related to each other

By capturing all the relevant program information in one place, it is easier to identify potential areas of concern and resolve them before they become problems
LML Summary

- LML provides a ontological foundation for supporting SoS SE
- LML contains the basic technical and programmatic classes needed for the lifecycle
- LML defines the Action Diagram to enable better definition of logic as functional requirements
- LML uses Physical Diagram to provide for abstraction, instances, and clones, thus simplifying physical models
- LML provides the “80% solution”
  - It can be extended to meet specific needs (e.g. adding Question and Answer classes for a survey tool that feeds information into the modeling)
For more information

- See the LML specification at www.lifecyclemodeling.org
- For implementation see www.innoslate.com
- Contact Steve Dam at www.specinnovations.com or sdam@specinno.com