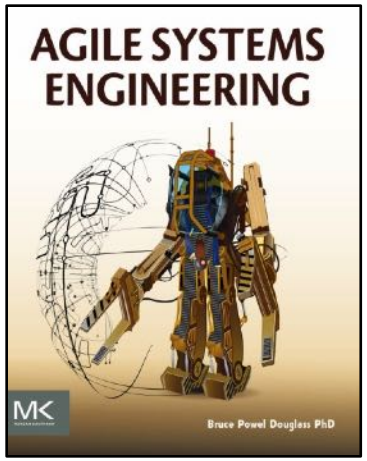
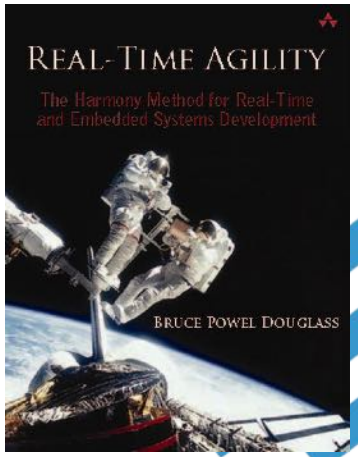
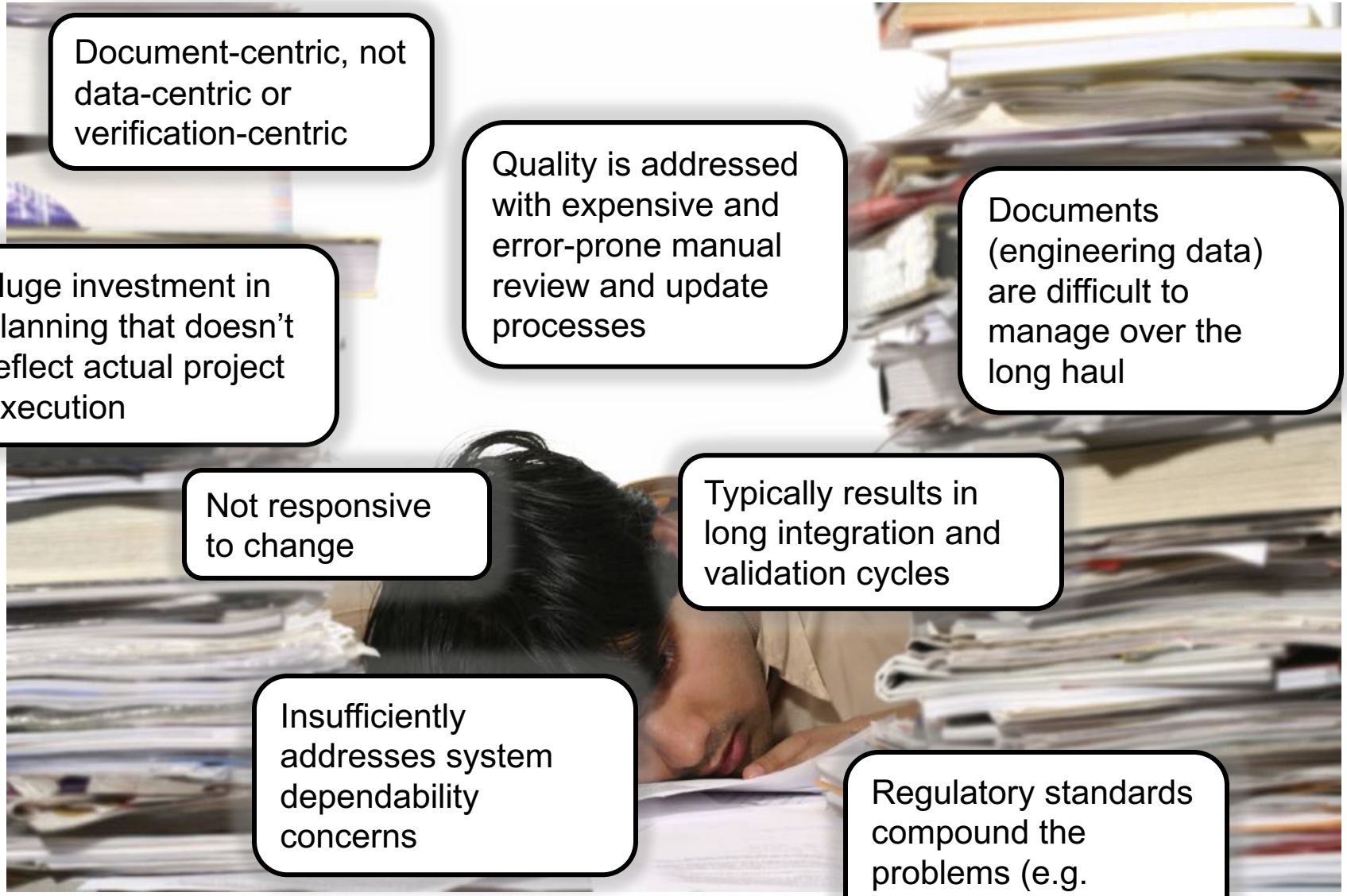
Agile Model-Based Systems Engineering (aMBSE)

Bruce Powel Douglass, Ph.D.
Chief Evangelist, Global Technology Ambassador
IBM IoT
Bruce.Douglass@us.ibm.com
Twitter: @BruceDouglass
IBM: www-01.ibm.com/software/rational/leadership/thought/BruceDouglass.html





State of the Practice for Systems Development



- Document-centric, not data-centric or verification-centric
- Quality is addressed with expensive and error-prone manual review and update processes
- Documents (engineering data) are difficult to manage over the long haul
- Huge investment in planning that doesn't reflect actual project execution
- Not responsive to change
- Typically results in long integration and validation cycles
- Insufficiently addresses system dependability concerns
- Regulatory standards compound the problems (e.g. DO0178, IEC 62304)

Key Concepts for Agility

Improve quality through *continuous feedback*

- **Verification**
  - Formal analysis
  - Review
  - Testing via execution or simulation
- **Validation:** Customer feedback (*meet the need*)
  - Correctness
  - Appropriateness
  - Usability
- **Dependability**: Analysis of safety, reliability, & security

Primarily build *executable things*
Verify them continuously
This means MODELS

Efficiency through

- Concentrate on high-value tasks
- Avoid rework
- Paying attention to how you're doing against goals
  - Project retrospective
  - Risk management

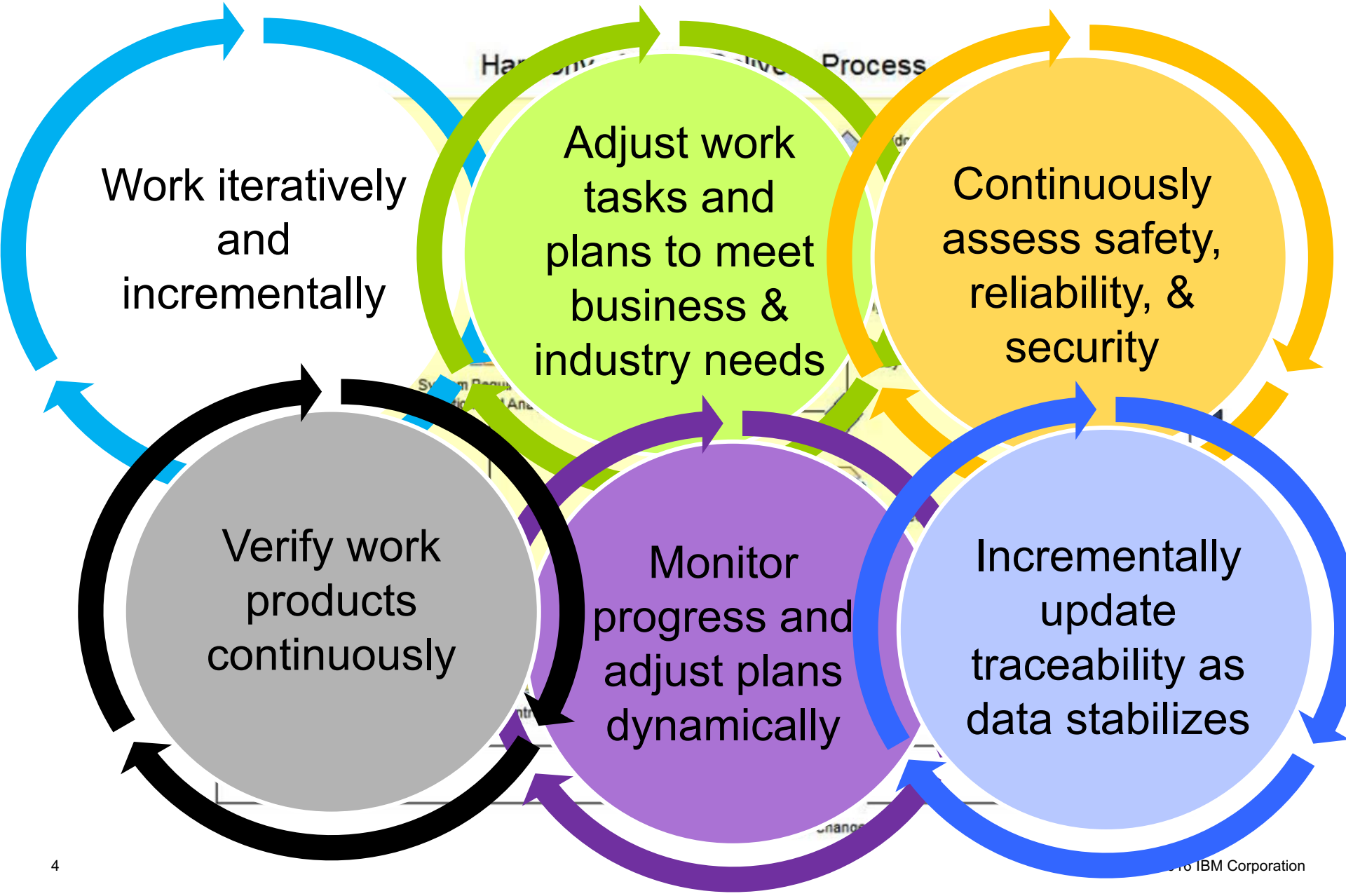
Active and continuous risk mitigation
Monitor project success

Planning

- Don't plan beyond the fidelity of the information you have
- Plan enough but not more than that
- Adjust plans based on "truth on the ground" (metrics)

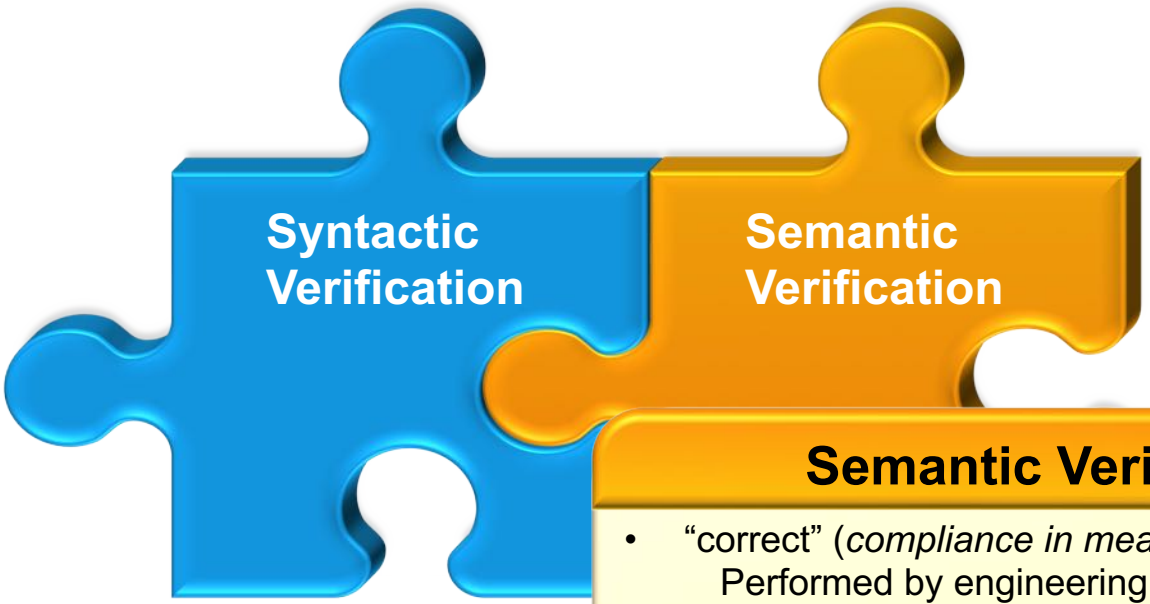
Dynamic planning
Responsive to Change

What does “agile” mean for Systems Engineering?



Computable models are essential for Agile MBSE

What do we mean by "verification"?



Semantic Verification

- "correct" (*compliance in meaning*)
  Performed by engineering personnel

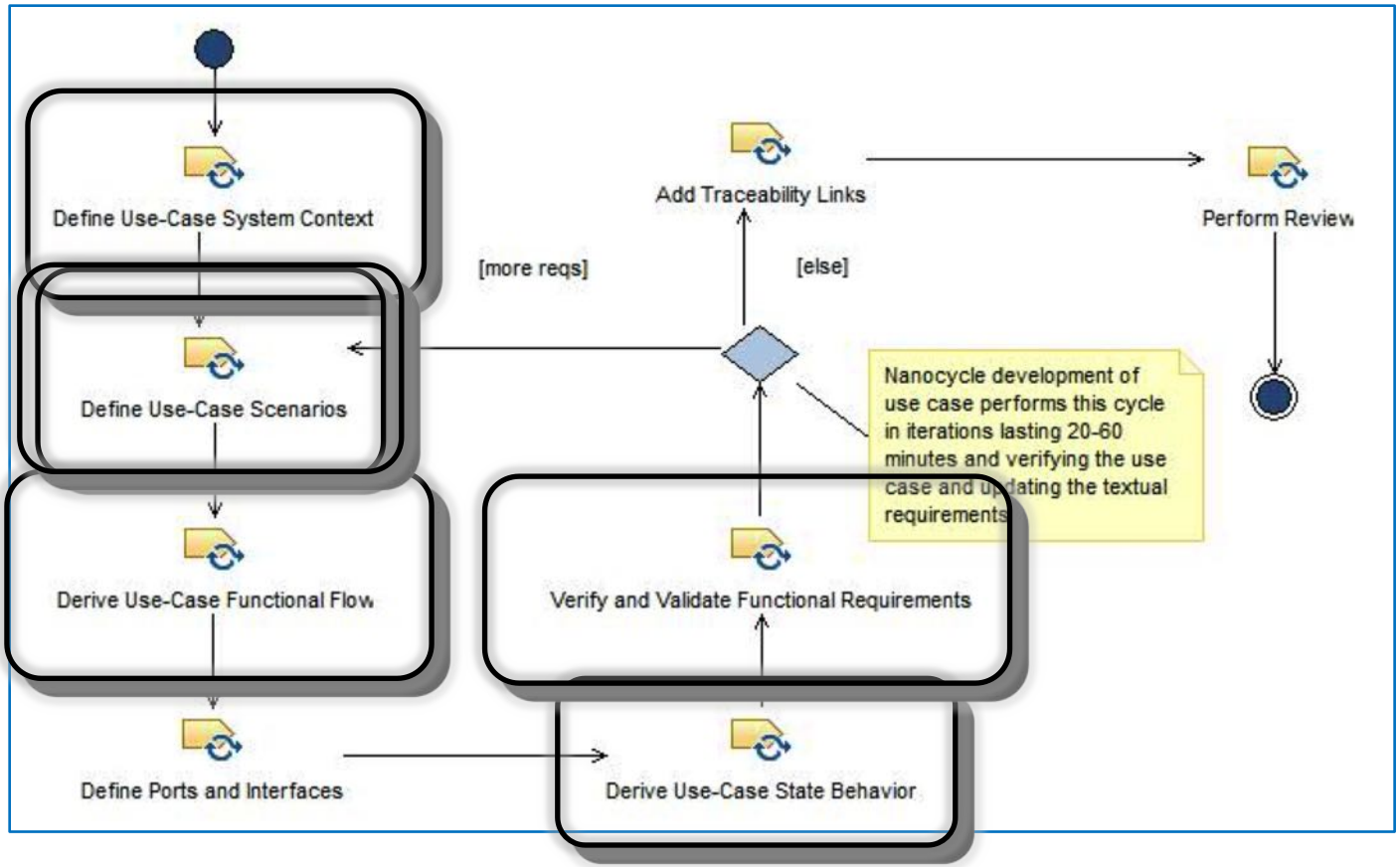
Three basic techniques

- **Testing** – requires executability of work products, impossible to fully verify
- **Formal methods** – strongest but hard to do and subject to invariant violation
- **Semantic review** (subject matter expert & peer) – most common, weakest means

Syntactic Verification

- "well-formed" (*compliance in form*)
  Performed by quality assurance personnel
- **Audits** – work tasks are performed as per plan and guidelines
- **Syntactic review** – work products conform to standard for organization, structure and format

Scenario Driven Use Case Construction / Verification



Making it Agile

Loop

Loop

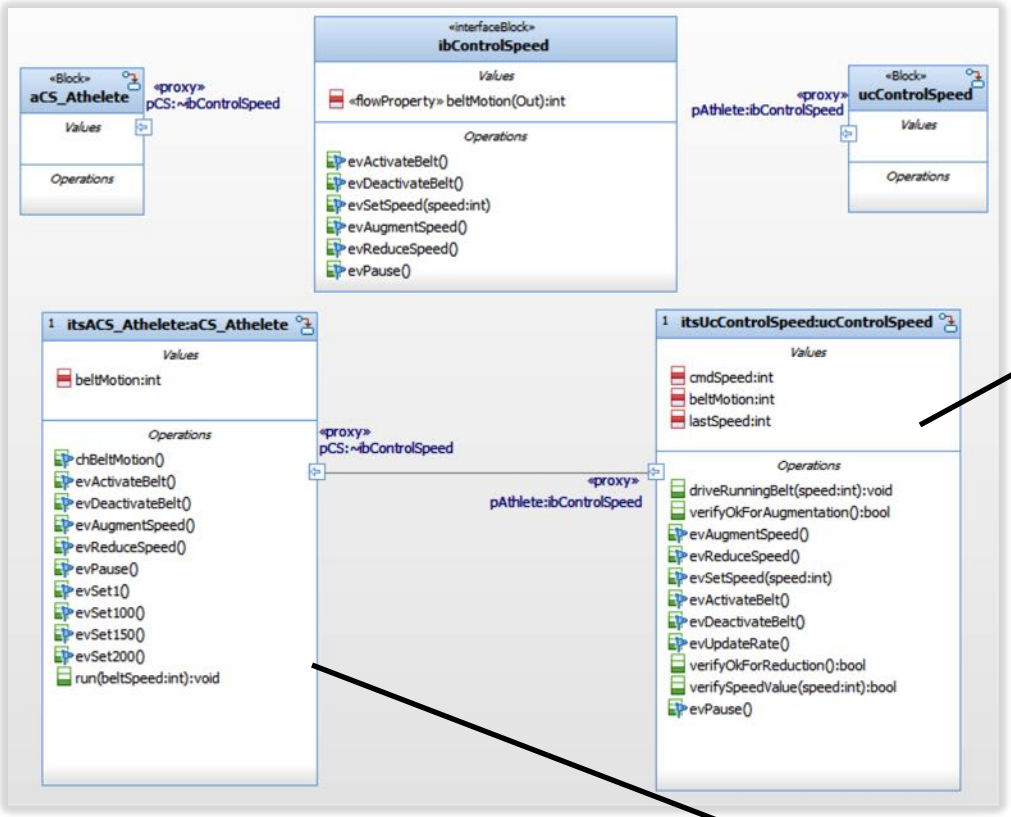
< 1 hr

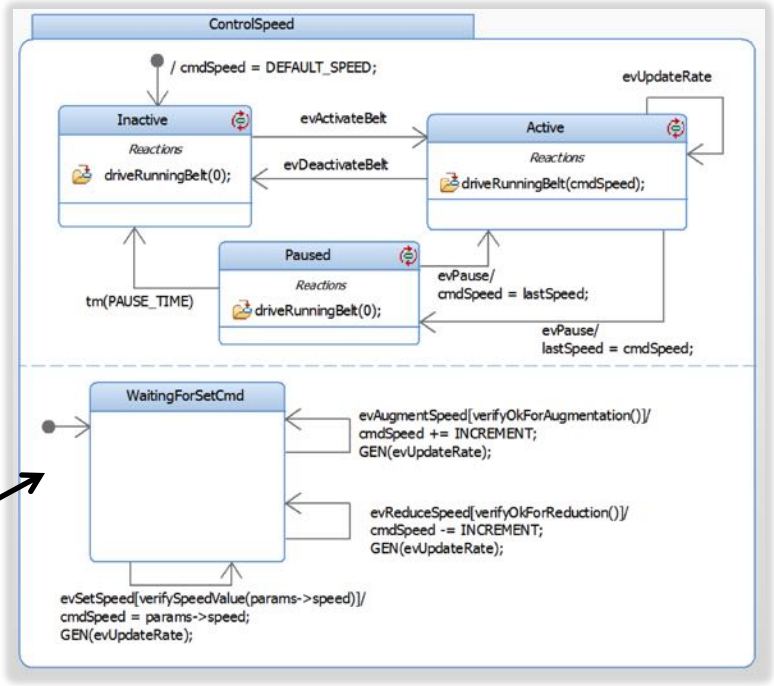
Conceptualize requirement aspect
Incrementally augment model
Verify

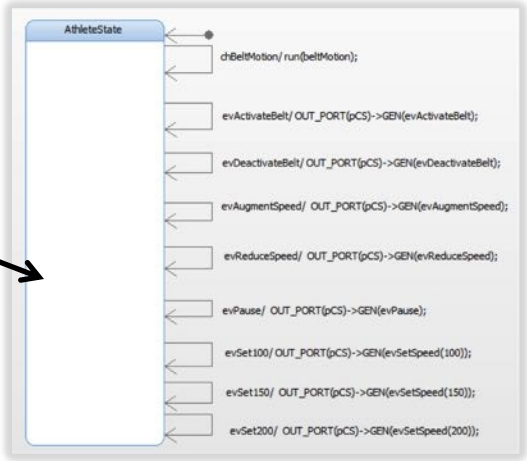
Repeat until all requirements added

Repeat for all use cases

High Fidelity Models are verifiable







… and may be built and verified incrementally, enabling agility

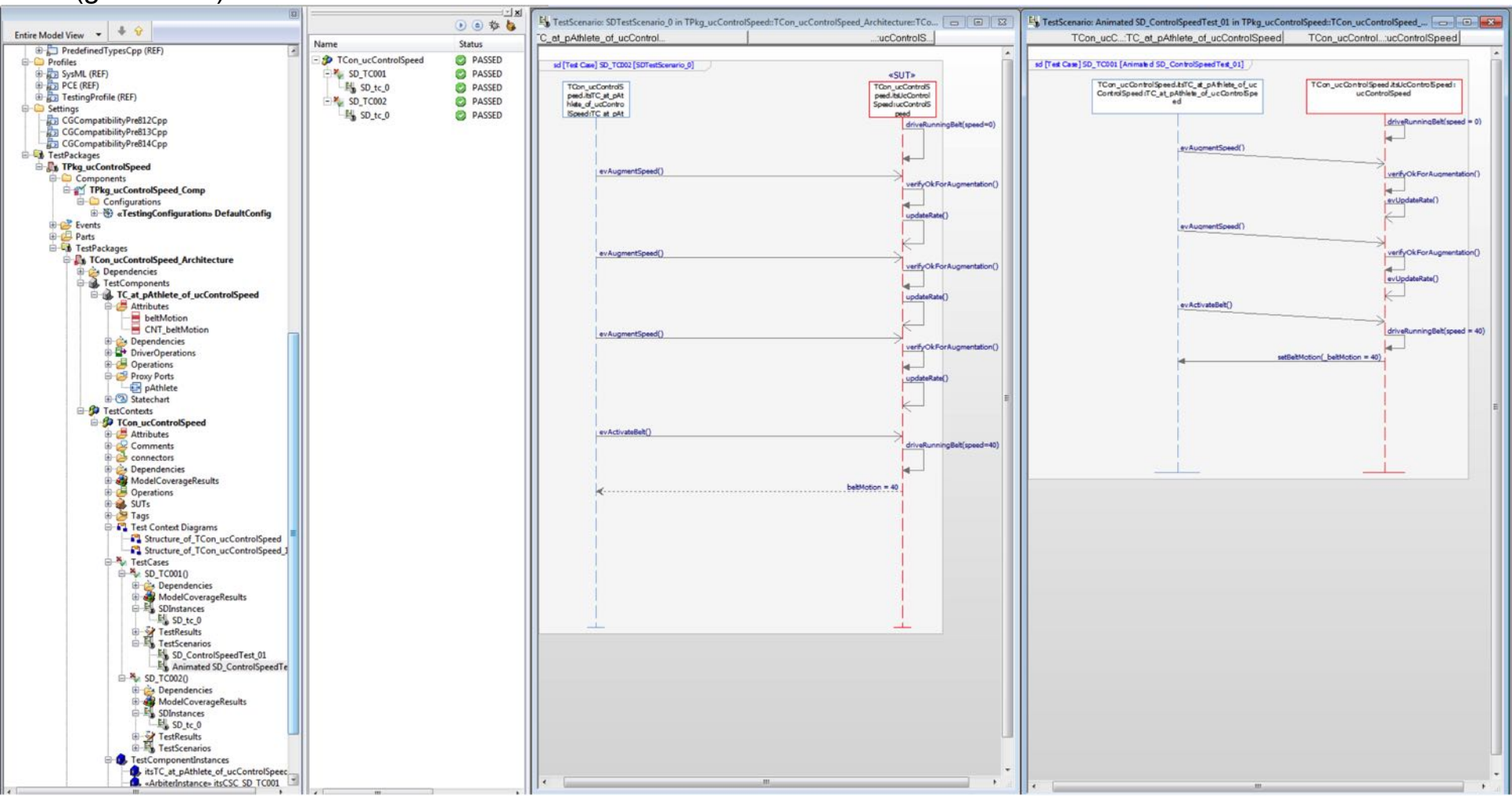
Verification may be manually done or via Test Conductor

Test Architecture (generated)

Test outcome

Test case definition

Test case test case result



Test Conductor implements the OMG UML Testing Profile standard

Determine the degree of verification with model coverage analysis

TestCase Coverage Result

TestCase: SD_TC001

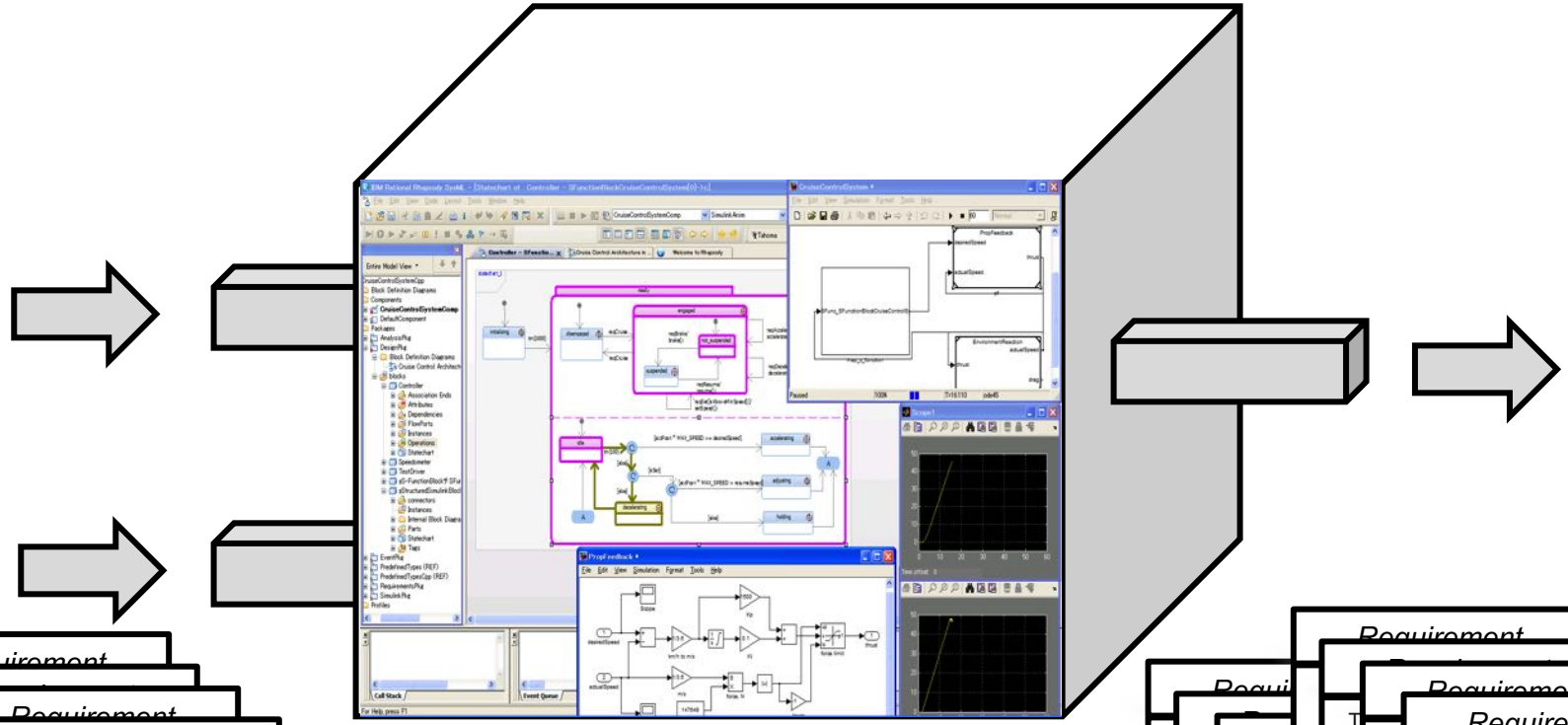
Environment Information	
Test executed on machine:	P8050Z6-27298
Test executed by user:	Bruce Douglass
Used operating system version:	Windows 7
Used Rhapsody version:	8.1.4, build 9641846
Used TestConductor version:	2.6.4, build 4386
Generation Date:	Wednesday, April 13, 2016 15:18:58

Tested Project	
Project:	SpeedDemonTreadmill
Active Code Generation Component:	TPkg_ucControlSpeed_Comp
Active Code Generation Configuration:	DefaultConfig
TestPackage:	TPkg_ucControlSpeed::TCon_ucControlSpeed_Architecture
TestContext:	TCon_ucControlSpeed
TestCase:	SD_TC001
Covered Classes:	ucControlSpeed (highlight | goto)

– Detailed Coverage Summary of ucControlSpeed (15/29)		☐ hide covered
– Operations		Go to: class top
covered	driveRunningBelt	Operation
covered	verifyOkForAugmentation	Operation
not covered	verifyOkForReduction	Operation
not covered	verifySpeedValue	Operation
– EventReceptions		Go to: class top
covered	evAugmentSpeed	EventReception
not covered	evReduceSpeed	EventReception
not covered	evSetSpeed	EventReception
covered	evActivateBelt	EventReception
not covered	evDeactivateBelt	EventReception
covered	evUpdateRate	EventReception
not covered	evPause	EventReception
– StateChart:	statechart_2 of ClassucControlSpeed	Go to: class top
covered	ROOT.ControlSpeed	Composite State
covered	ROOT.ControlSpeed.state_6	Composite State
covered	ROOT.ControlSpeed.state_6.Active	State
covered	ROOT.ControlSpeed.state_6.Inactive	State
not covered	ROOT.ControlSpeed.state_6.Paused	State
not covered	10	Transition
covered	3	Transition
not covered	4	Transition
covered	5	Default Transition
not covered	7	Transition
not covered	8	Transition
not covered	9	Transition
covered	ROOT.ControlSpeed.state_7	Composite State
covered	ROOT.ControlSpeed.state_7.WaitingForSetCmd	State
covered	0	Transition
not covered	1	Transition
not covered	2	Transition
covered	6	Default Transition

Functional Analysis via Executable Requirements?

- A functional requirement is a *specification of an input-output control or data transformation or flow*
- A quality of service requirements is a specification of how well a control or data transformation is achieved



Requirement

The system shall be "real good" and "very fast" and "generally awesome"

Poor requirements

Much better requirements

Requirement

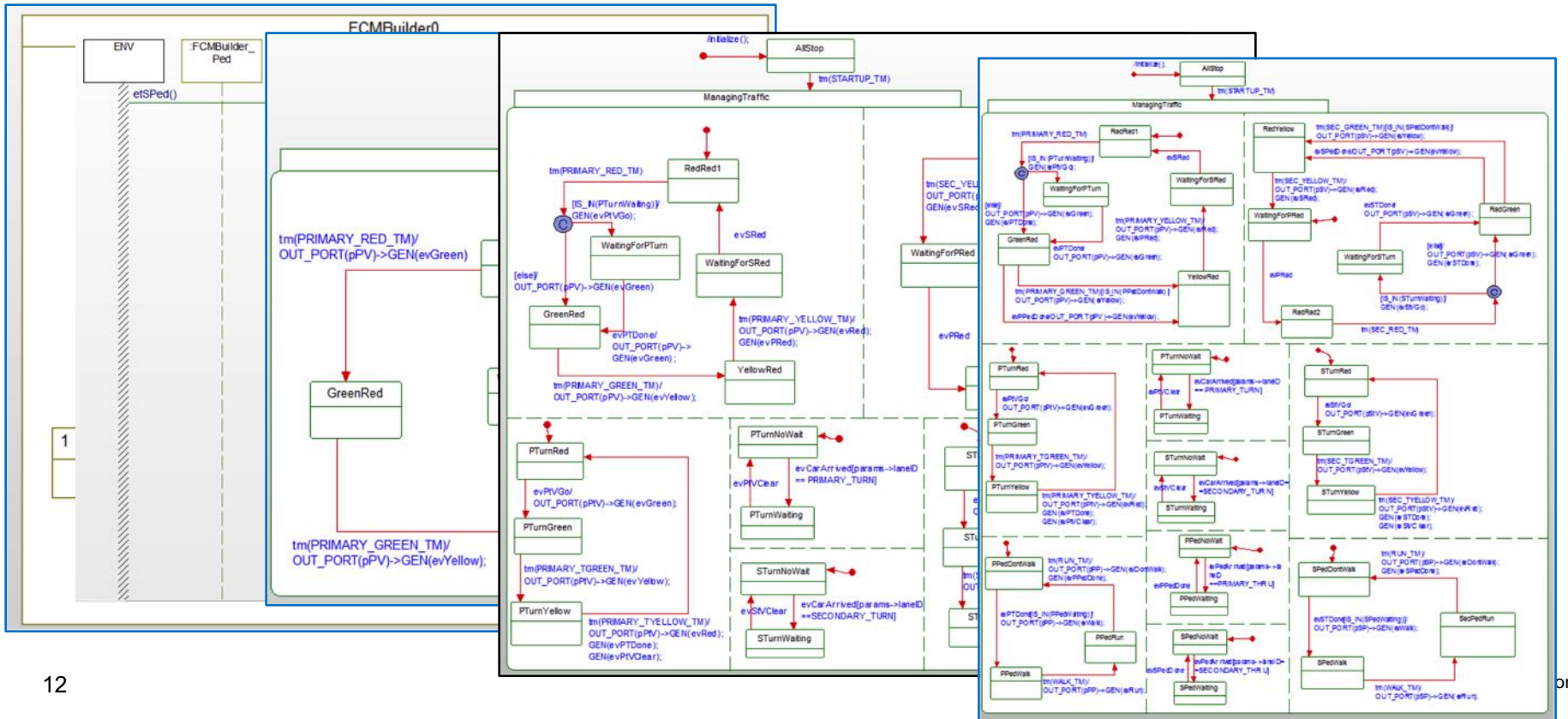
The system shall perform error correct and detection during initialization and operation.

Control surfa
updated every 10
2 ms with an accuracy of
.5 cm and a latency of no
more than 1 ms

Cyb
detected
onset and re
security measures
including incident reports.

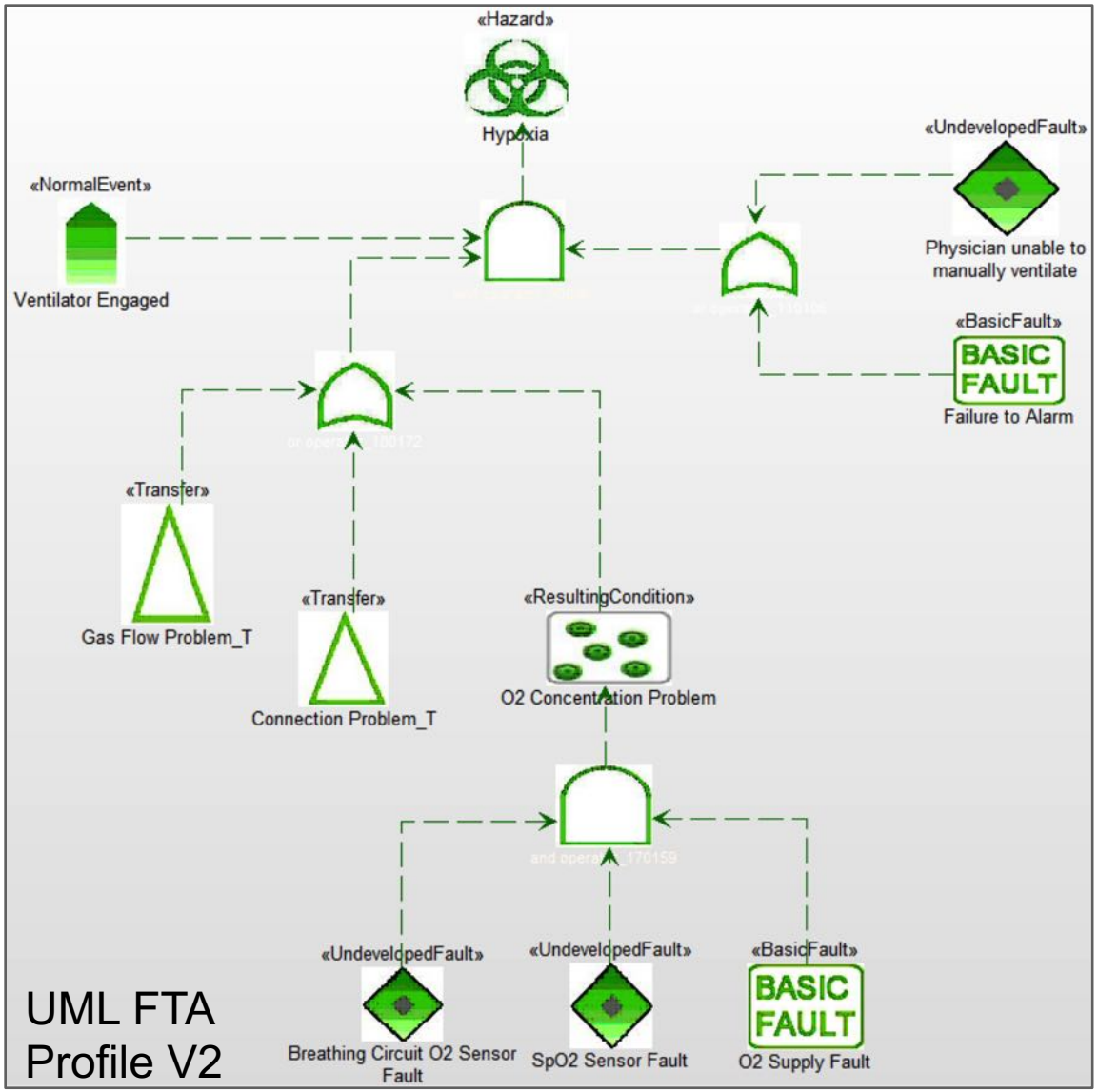
Test-Driven Development for MBSE Work Products

- The principle behind TDD is to develop and apply test cases as you develop a system to demonstrate that it is correct
  - This is done in parallel with the system development and *not* ex post facto
  - This is about *defect avoidance* not so much *defect identification and repair*
- TDD applies to the development of complex system use case, architecture and design models

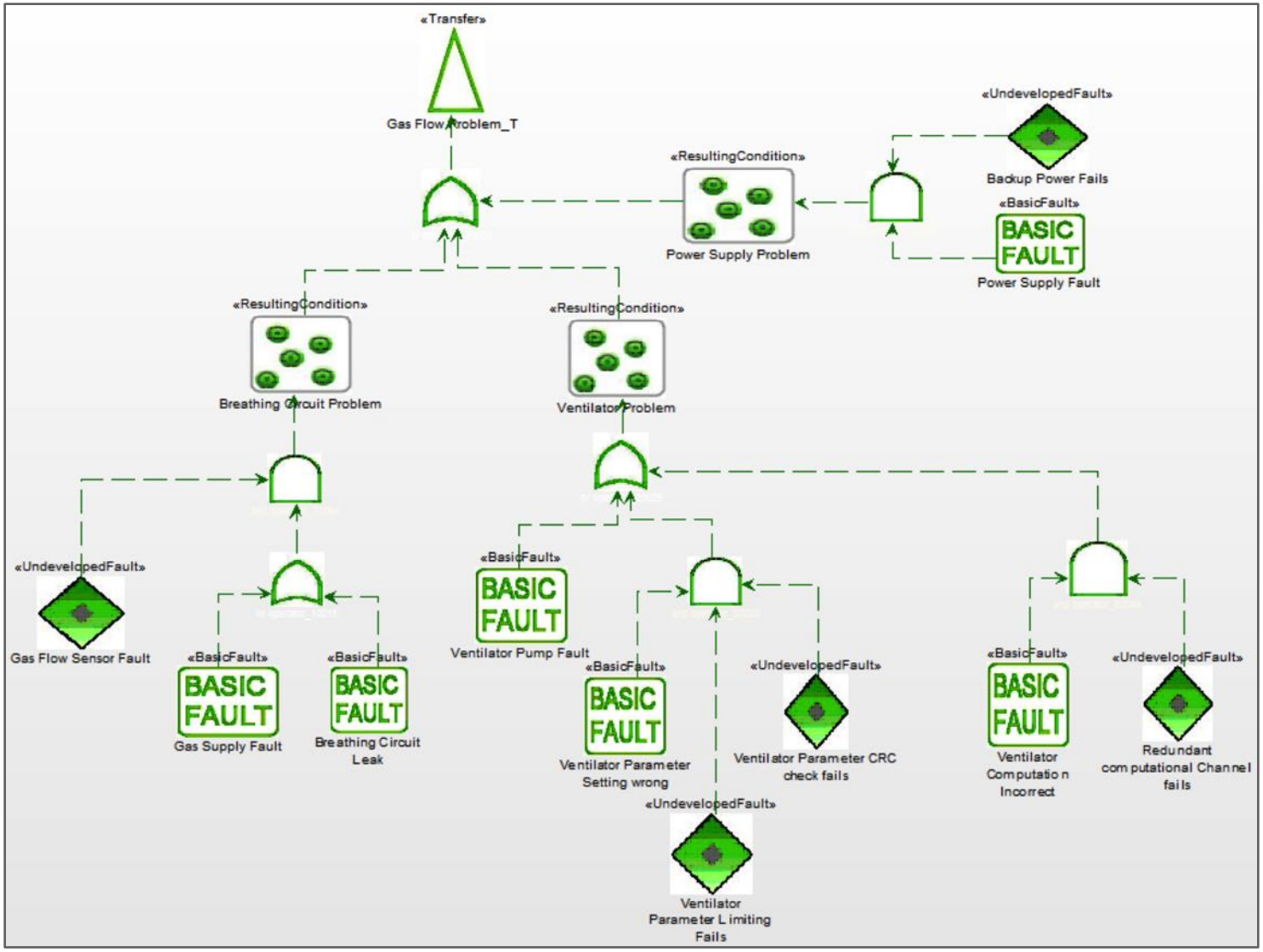


Integrated Safety and Reliability Analysis

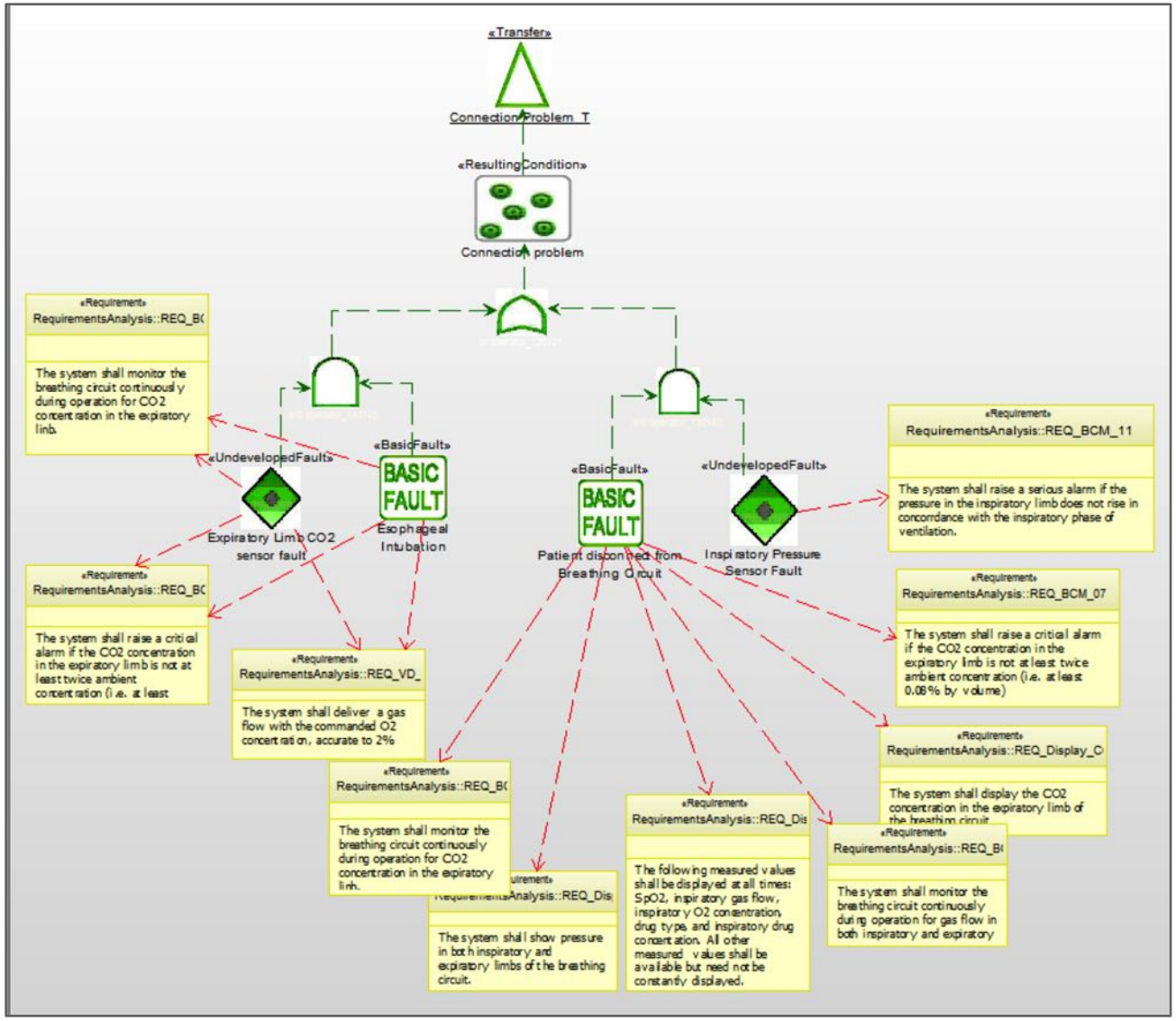
- Fault Tree Analysis (FTA) connects *hazards* with logical combinations of events, conditions, errors, and faults
- Allows you to identify
  - Effects of combinations of conditions and events on safety
  - Safety measures
  - Safety requirements
  - Impacts of architectural, technological, and design choices on safety



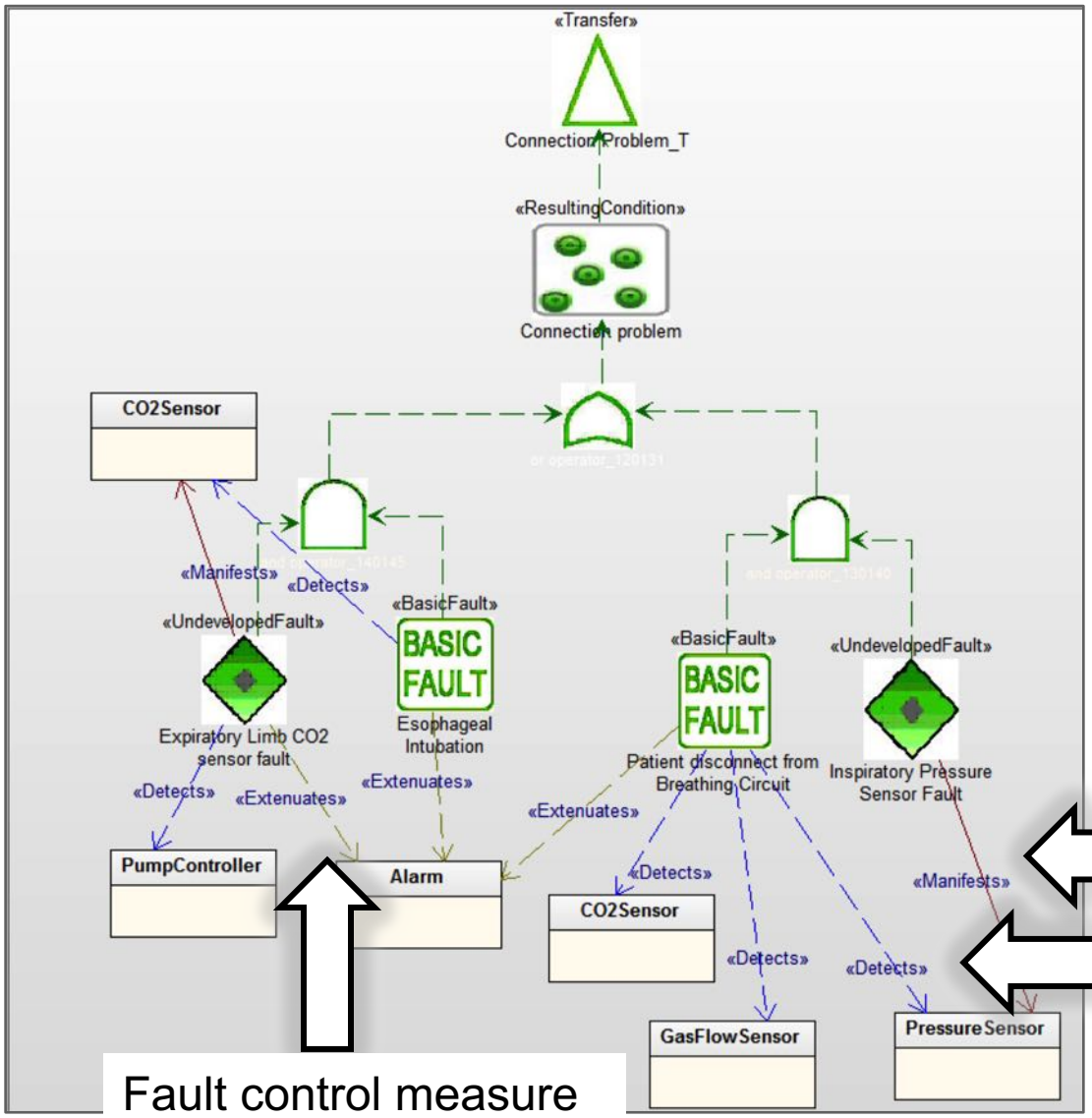
Integrated Safety and Reliability Analysis



Mapping Requirements to Fault Tree Analysis



Mapping Design to Fault Tree Analysis

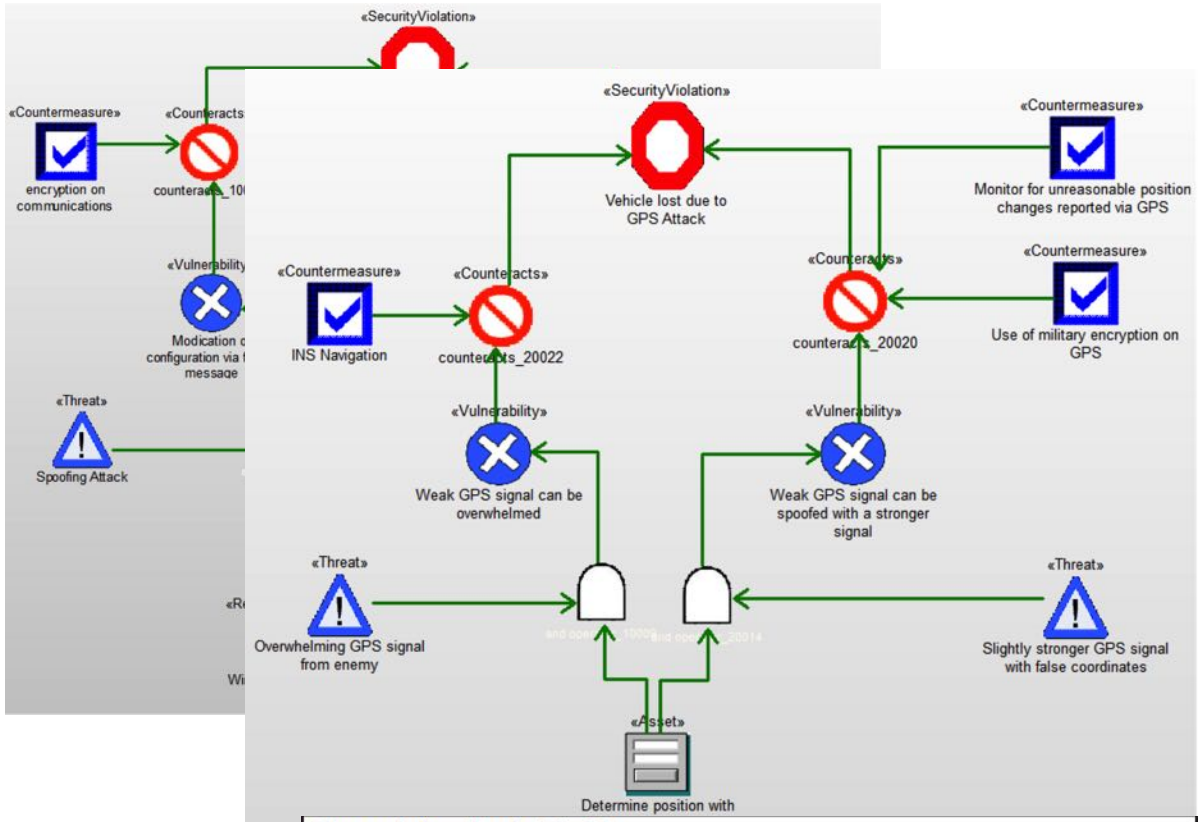


Fault isolation

Fault detection

Model-Based Threat Analysis

- Security Analysis Diagram (SAD) is like a Fault Tree Analysis (FTA) but for security, rather than safety
  - It looks for the logical relation between assets, vulnerabilities, attacks, and security violations
  - Permits reasoning about security
    - What kind?
    - How much?
    - Risk assessments
    - Cost of security penetration
    - Adequacy of countermeasures
    - Who has access to assets



Threat Analysis Table

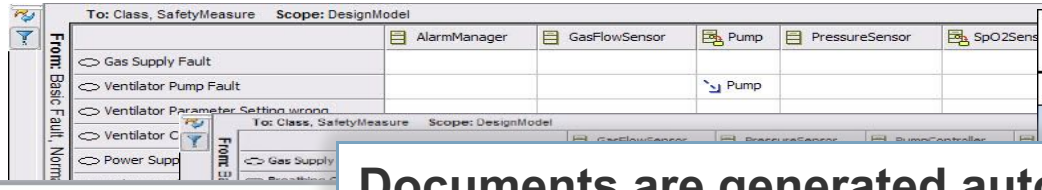
Asset value is the value of the asset to be protected (1=very low, 10=very high).									
Likelihood is the probability of the attack (1=very low, 10=certain).									
Reproducibility refers to how easy it is to reproduce the attack (for example, does it depend on timing or other circumstances?) (1=hard, 10 = very easy).									
Exploitability refers to how easy it is to launch the attack (1=very easy, 10=very hard).									
Breadth is the a measure of the extent of the attack. How widespread is it or how many systems are affected? (1=few, 10=very many).									
Discoverability is how easy is it for outsiders to find out about and exploit the vulnerability (1=very easy, 10=very hard).									
Threat Priority is the product of the above values and is used to prioritize the threats for countermeasures.									
			These are in the range of 1 -10						
Asset	**Vulnerability**	**Threat Vector**	**Asset Value**	**Likelihood of attack**	**Reproduci-bility**	**Exploitability**	**Breadth**	**Discover-ability**	**Threat Priority**	**Countermeasure**
Patient Demo-graphic Data	Access via Ethernet	Input validation weak	4	7	9	4	1	9	9072	Internal encryption
	Access via USB	Auto-execution of USB SW	4	7	9	3	1	9	6804	Internal encryption
	Access via packet snooping	Messages sent in plain text	4	7	9	5	1	5	10080	Message encryption

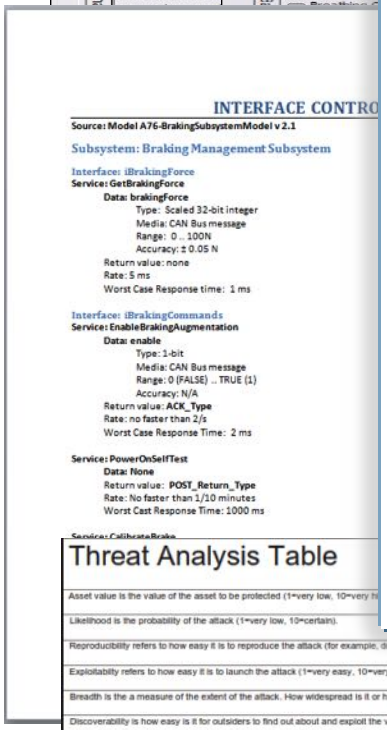
Auto-generation of summary documentation from models

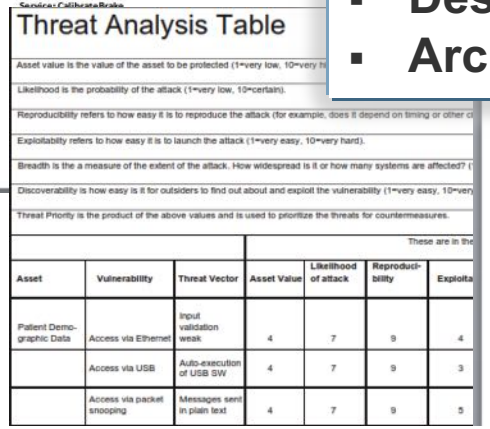
Documents are generated automatically from engineering work in models

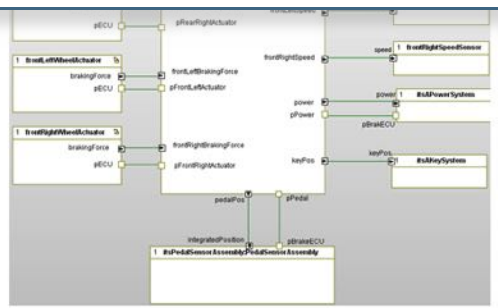
Typical auto-generated documentation includes

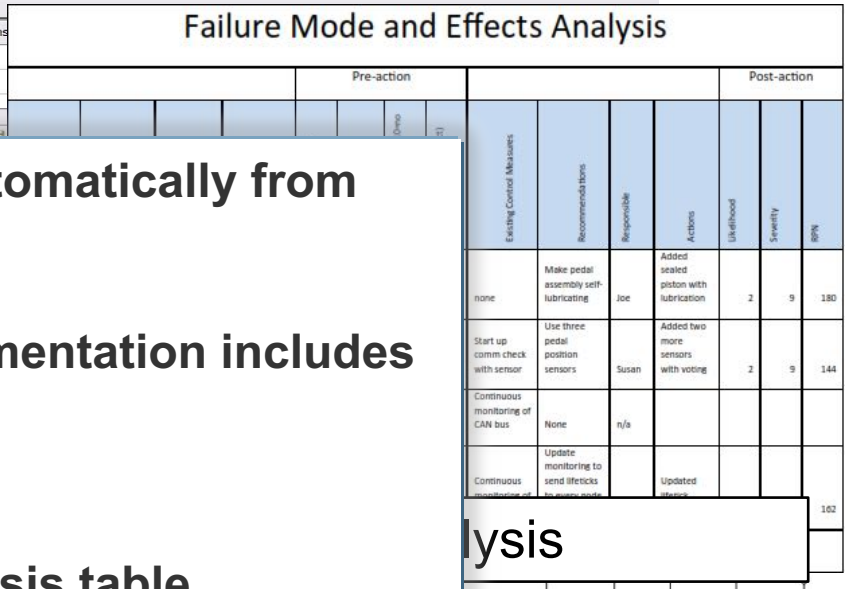
- **Traceability matrix**
- **Hazard Analysis**
- **FMEA / FMECA**
- **Cyberphysical threat analysis table**
- **Interface Control Document**
- **Design Description**
- **Architecture Notebook**

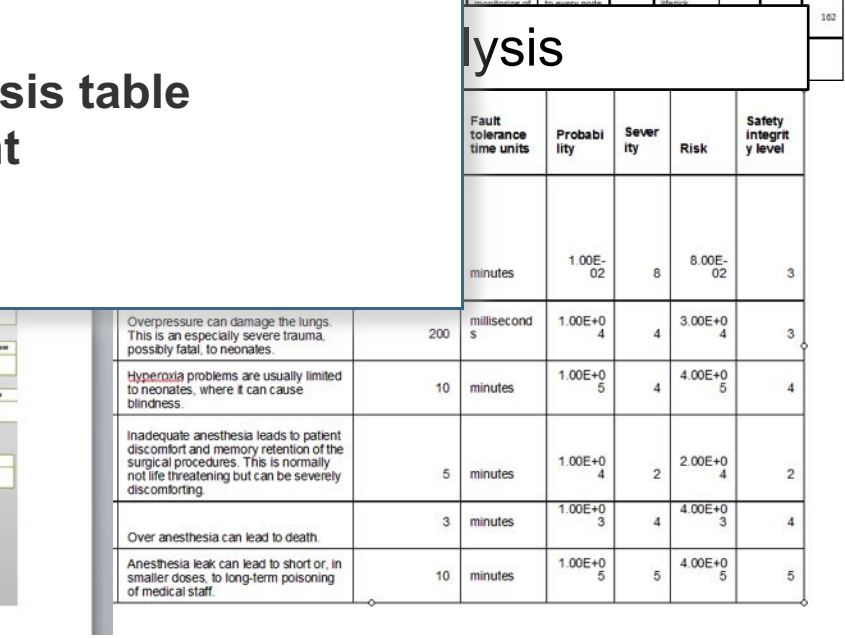












Harmony Agile MBSE Delivery Process

Rational. Method Composer

Search this Site:

Team (IBM)

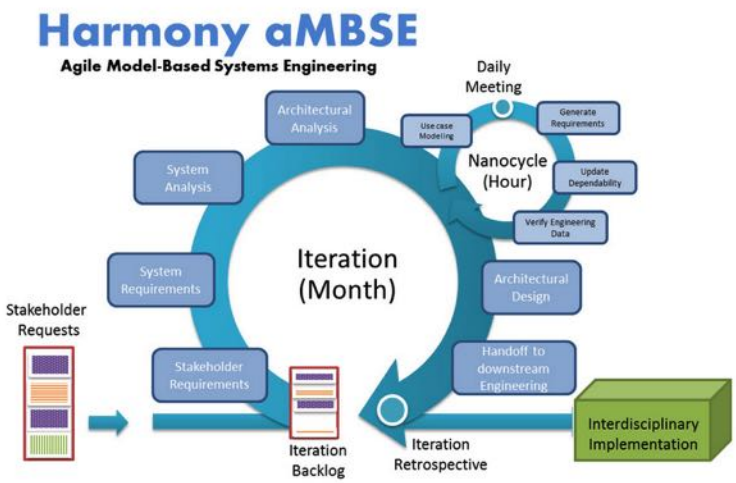
- Welcome to the Rational Harmony Agile Model-Based Systems Engineering
- Getting Started
- Delivery Processes
- Practices
- Roles Sets
- Tasks
- Work Products
- Guidance
- Tools
- Release Info

Welcome to the Rational Harmony Agile Model-Based Systems Engineering

Welcome to the Rational Harmony Agile Model-Based Systems Engineering

The Rational Harmony Agile Model-Based Systems Engineering (aMBSE) process is a delivery process for the development of systems engineering data and work product using both model-based systems techniques with UML and SysML but is at the same time agile and incorporates agile practices for improved quality and engineering efficiency.

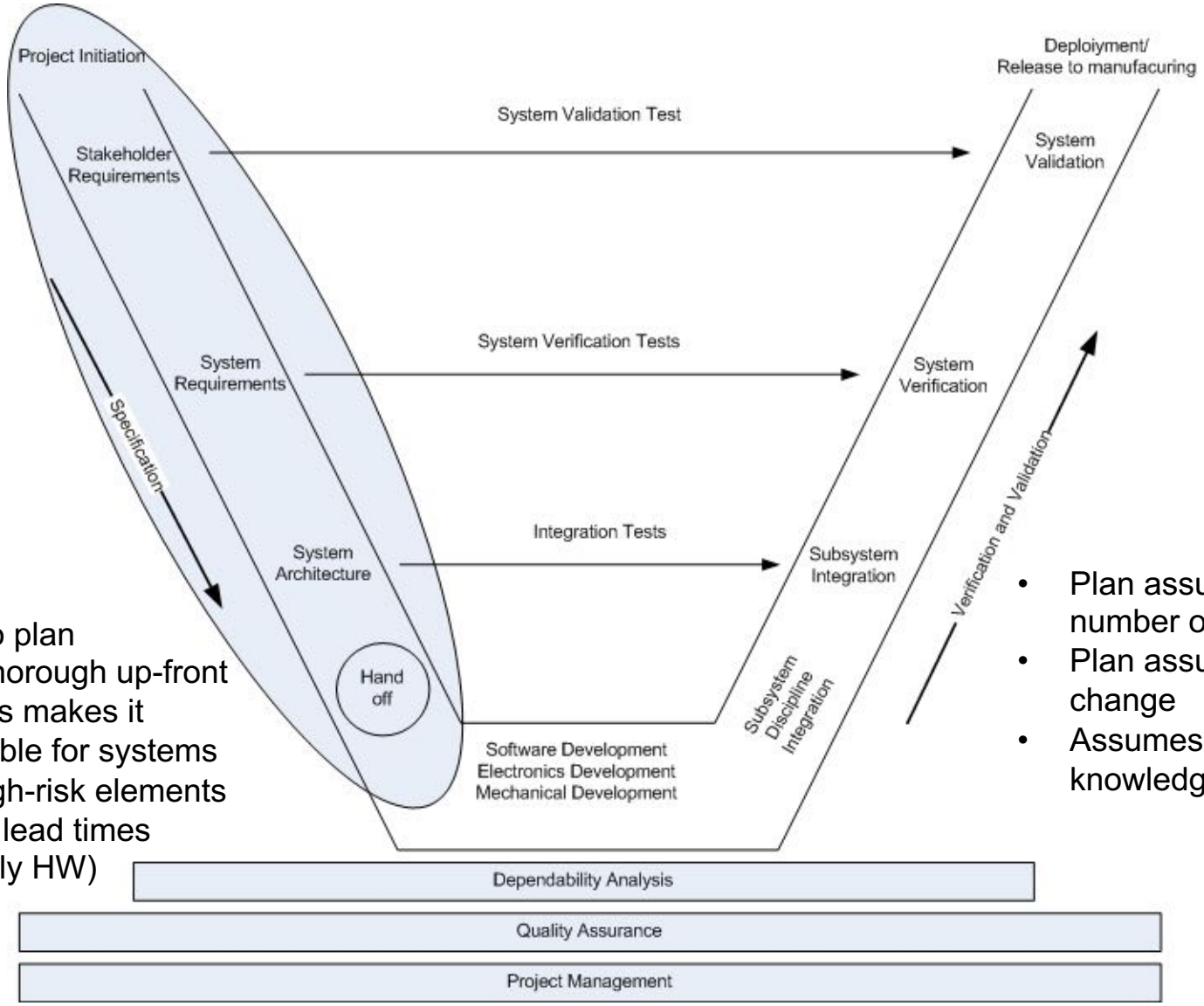
Main Description



With the initial release of the UML in 1995, systems engineers had a standard language in which they could express requirements, architectures, designs, and other kinds of engineering data. However, there was widespread belief that the Unified Modeling Language (UML) itself was too "software oriented" for general use in systems engineering which led to the development and release of the Systems Modeling Language (SysML). UML and SysML provide a number of key advantages for the development of system engineering data:

- Precision of engineering data
- Data consistency across work products and engineering activities
- A common source for engineering truth
- Improved visualization and comprehension of engineering data
- Ease of integration of disparate engineering data
- Improved management and maintenance of engineering data

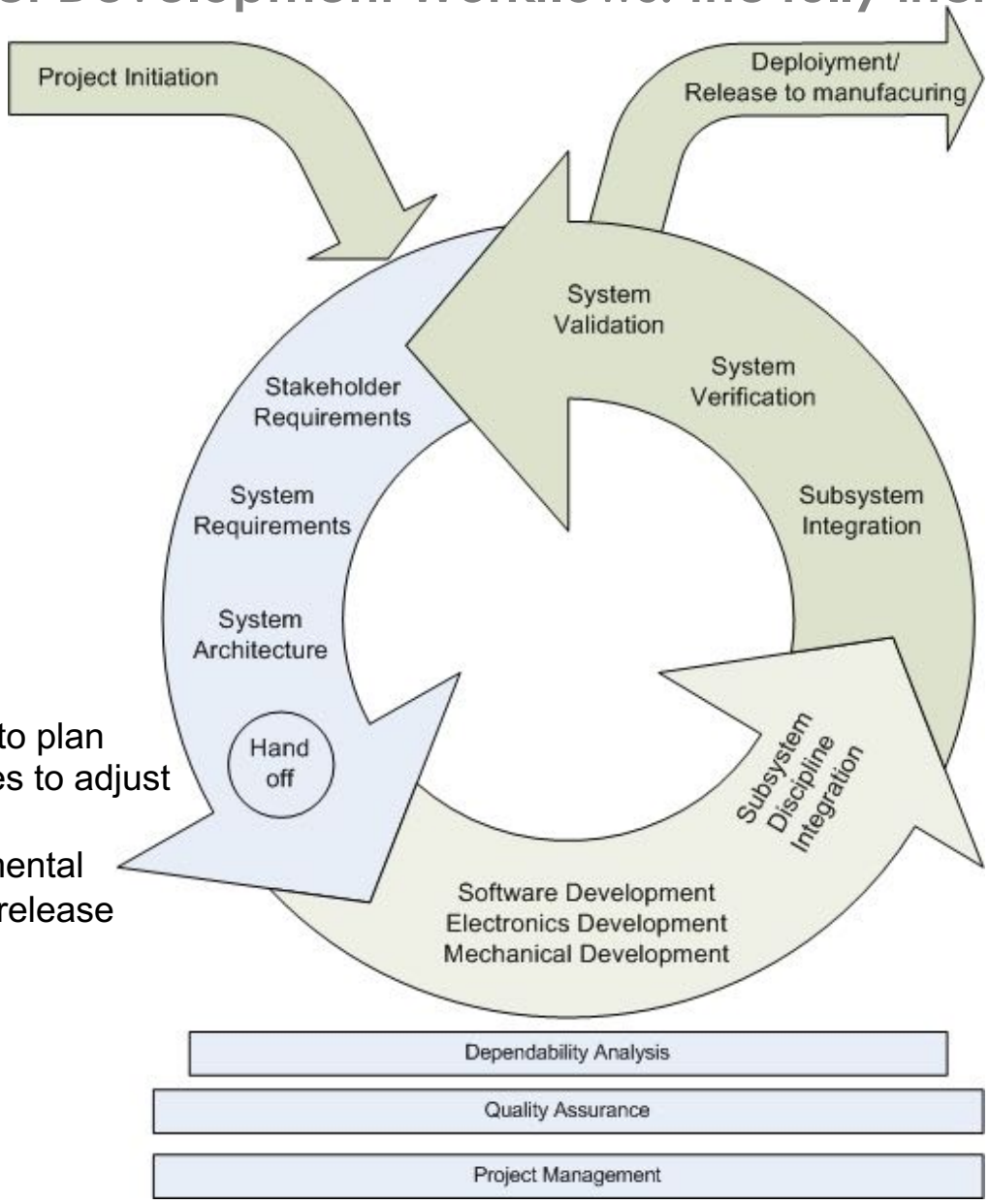
Overall Product Development Workflows: the standard V



- Easy to plan
- More thorough up-front analysis makes it applicable for systems with high-risk elements of long lead times (typically HW)

- Plan assumes small number of minor defects
- Plan assumes little or no change
- Assumes “infinite knowledge”

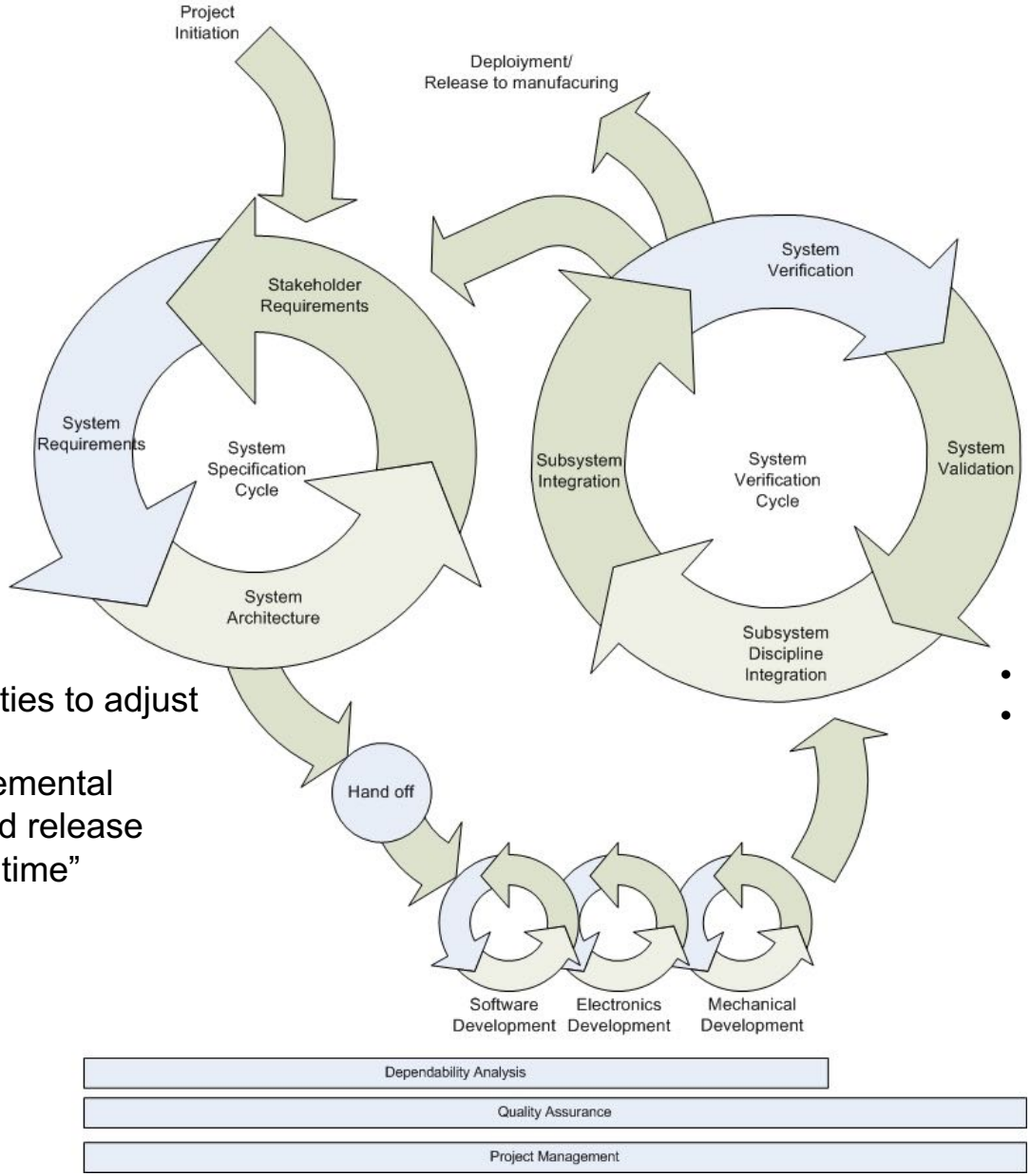
Overall Product Development Workflows: the fully incremental



- Relatively easy to plan
- Has opportunities to adjust plans
- Supports incremental verification and release

- Has “dead time” because different people are working on different activities
- May be inappropriate for high-risk items of long lead time (typically hardware)

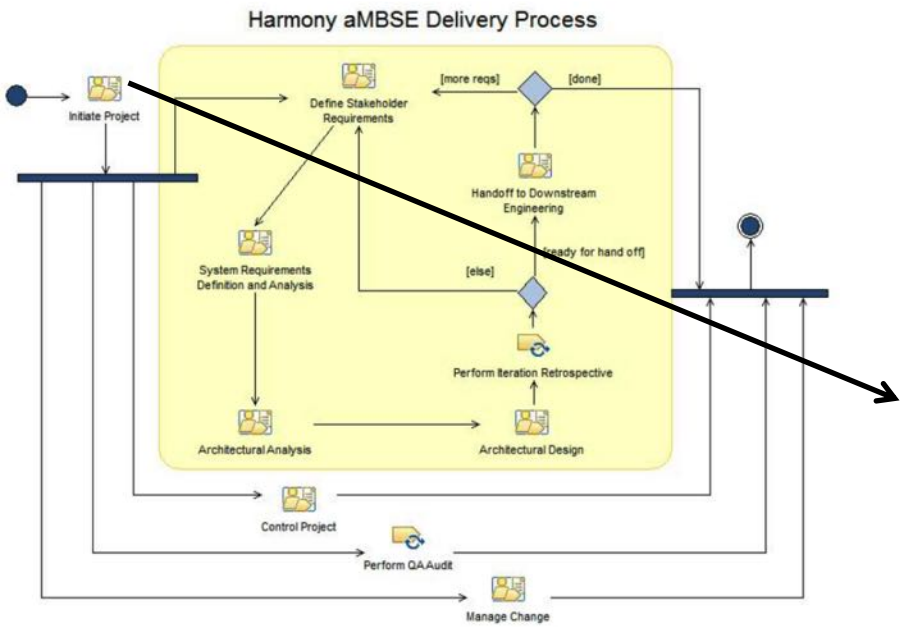
Overall Product Development Workflows: the hybrid agile V

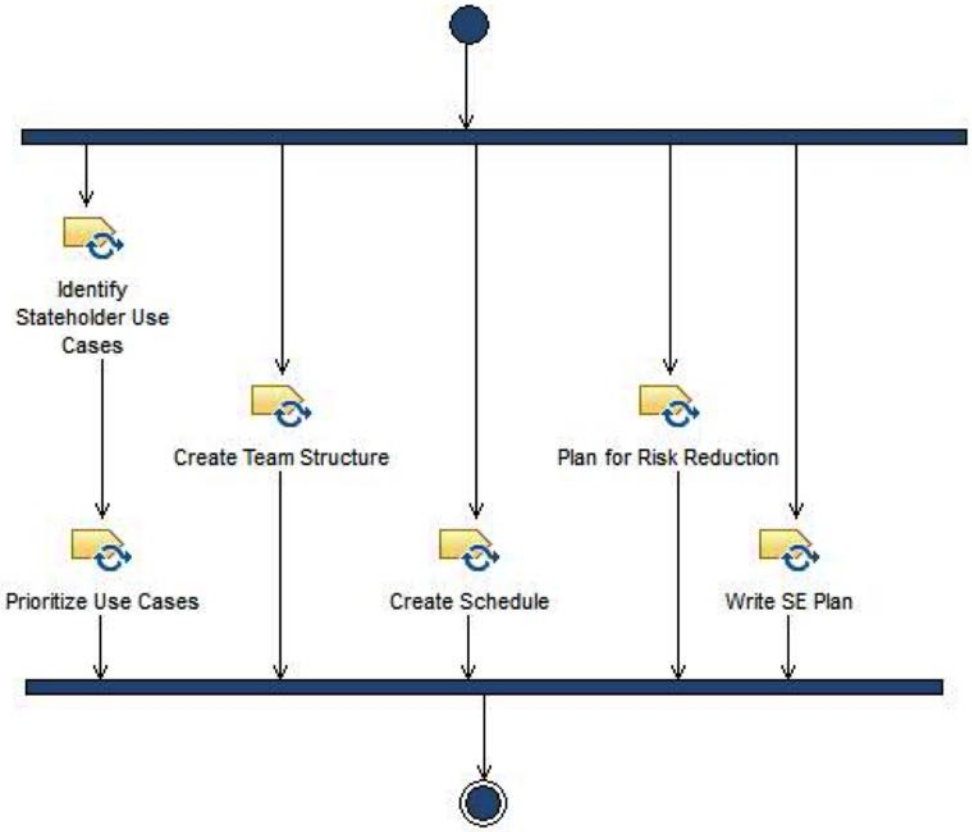


- Has opportunities to adjust plans
- Supports incremental verification and release
- Has no “dead time”

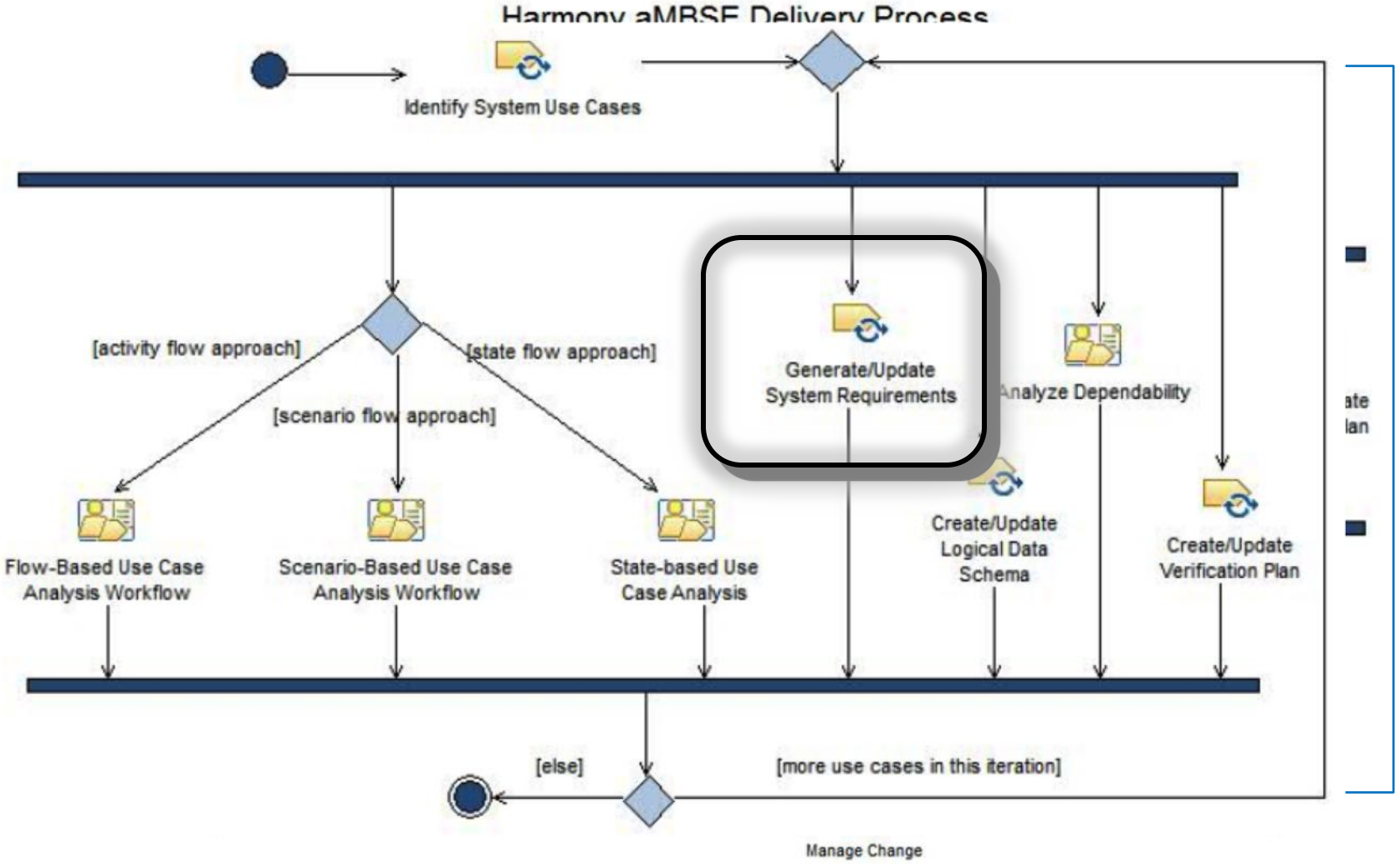
- Complex to plan
- Requires work to keep work activities coordinated

Initiate project

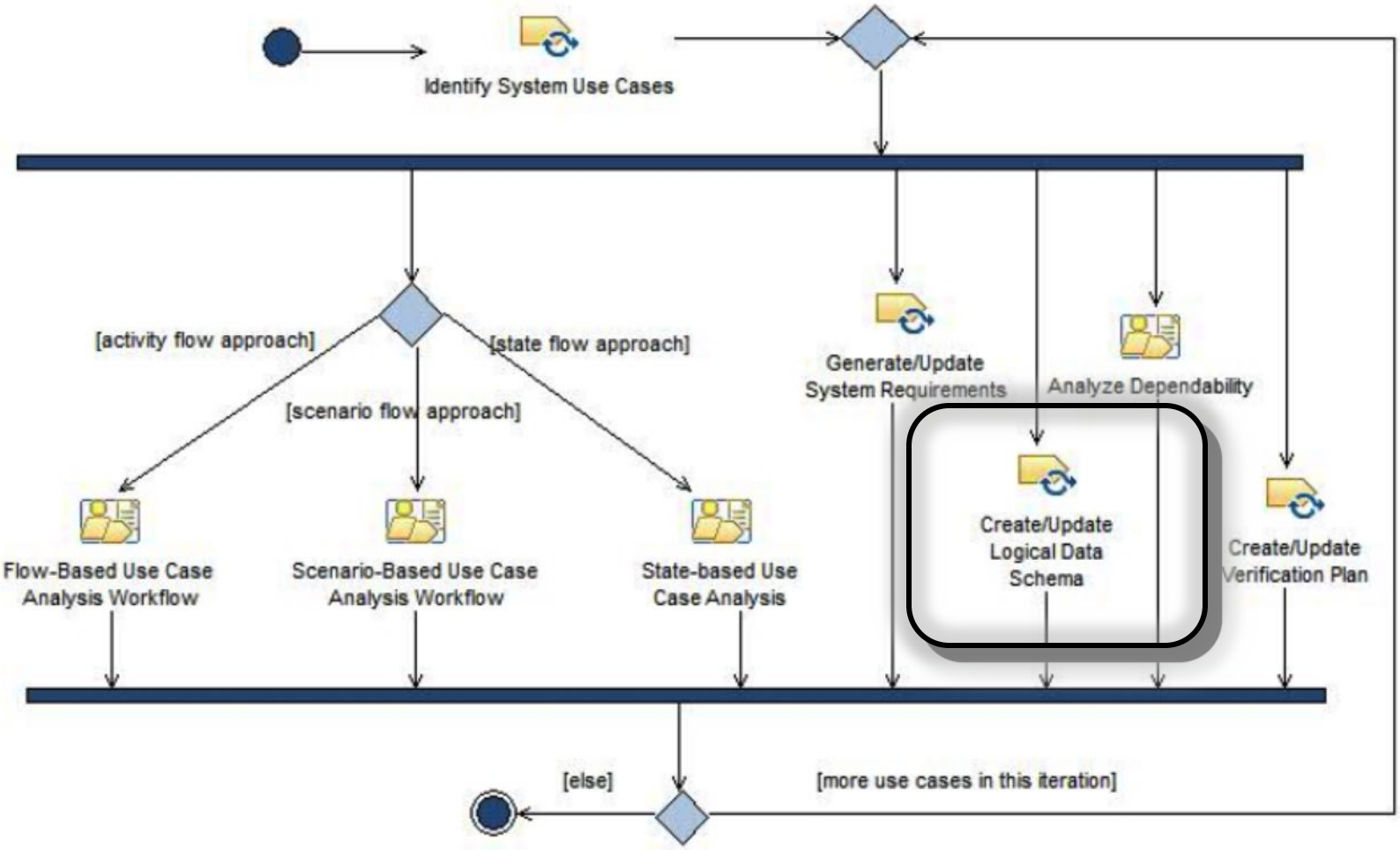




Harmony Process for Agile MBSE

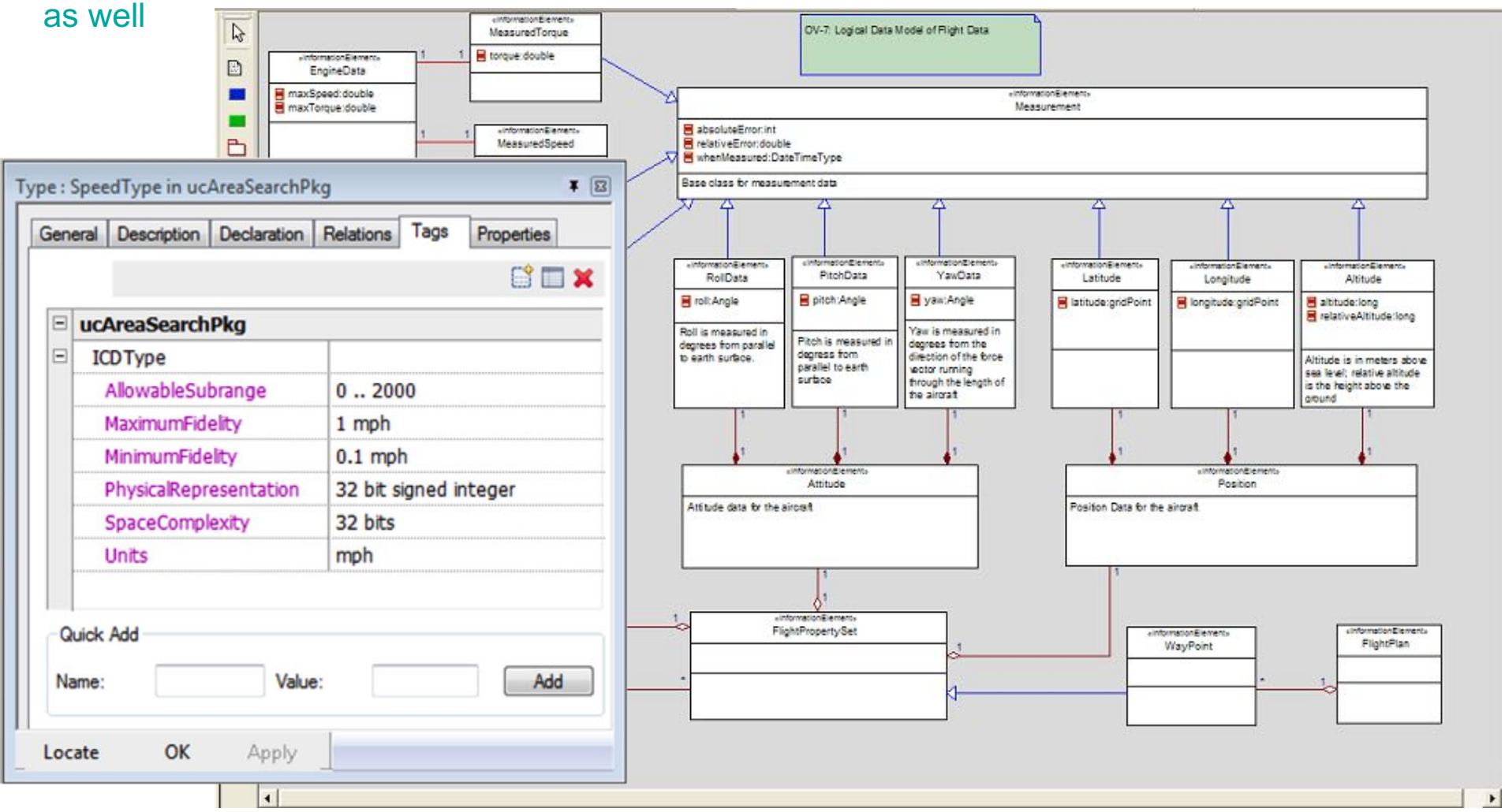


Harmony Process for Agile MBSE

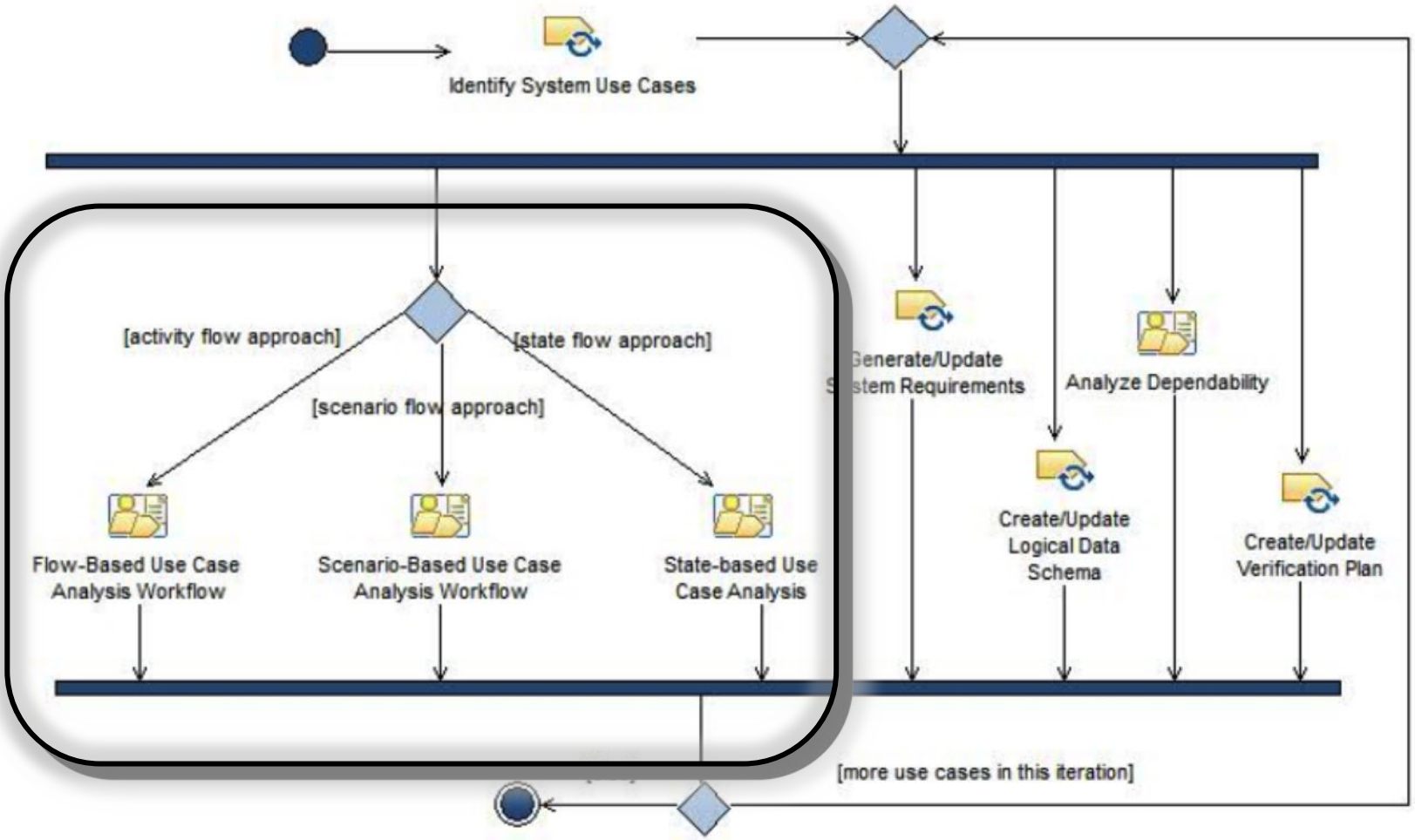


Logical Data Schema Modeling

- A logical data schema identifies the logical properties of important data elements and types and the relations among such data elements and their metadata
- Although the name is “data schema” it includes physical, materiel, and energy flows specification as well



Harmony aMBSE: System Requirements Def & Analysis

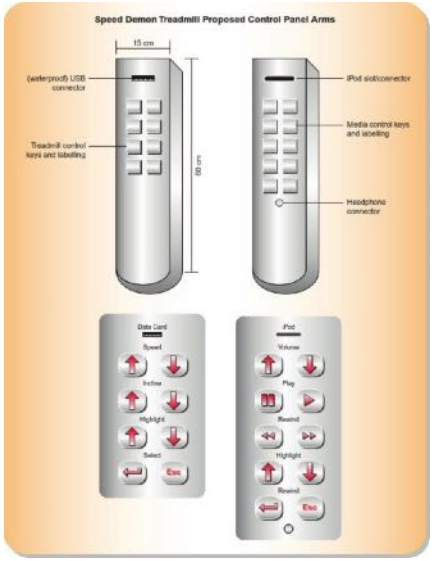


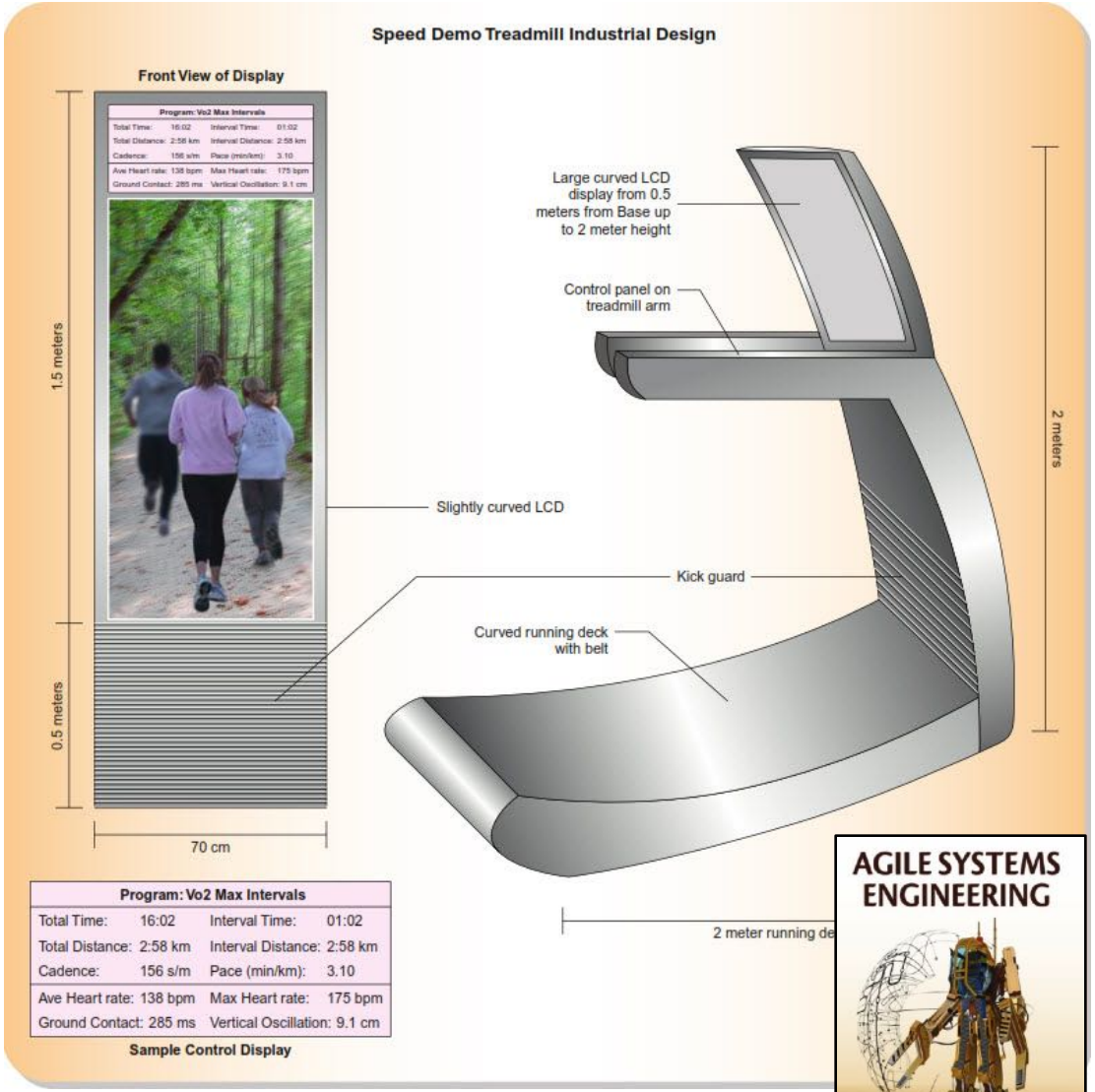
Alternative approaches to Build Executable Model of UC

Alternative 1: Scenario Based		Alternative 2: Flow Based		Alternative 3 State Based	
Create model context	*Block Diagram*	Identify functional flow large-scale view	*Activity diagram*	Create Model Context	*Block Diagram*
Identify sequences of messages between system use case and actors	*Sequence Diagram*	Derive sequences from functional flow	*Sequence diagram*	Create executable state machine	*State Diagram*
(optional) cluster sequences together as flows	*Activity Diagram*	Create model context	*Block Diagram*	Identify interfaces	*Block Diagram*
Identify interfaces	*Block Diagram*	Identify interfaces	*Block Diagram*	Derive sequences from functional flow	*Sequence diagram*
Create executable state machine	*State Diagram*	Create executable state machine	*State Diagram*	Execute State machine	*Model execution views*
Execute State machine	*Model execution views*	Execute State machine	*Model execution views*	Repeat until all requirements and sequence variants covered	
Repeat until all requirements and sequence variants covered		Repeat until all requirements and sequence variants covered			

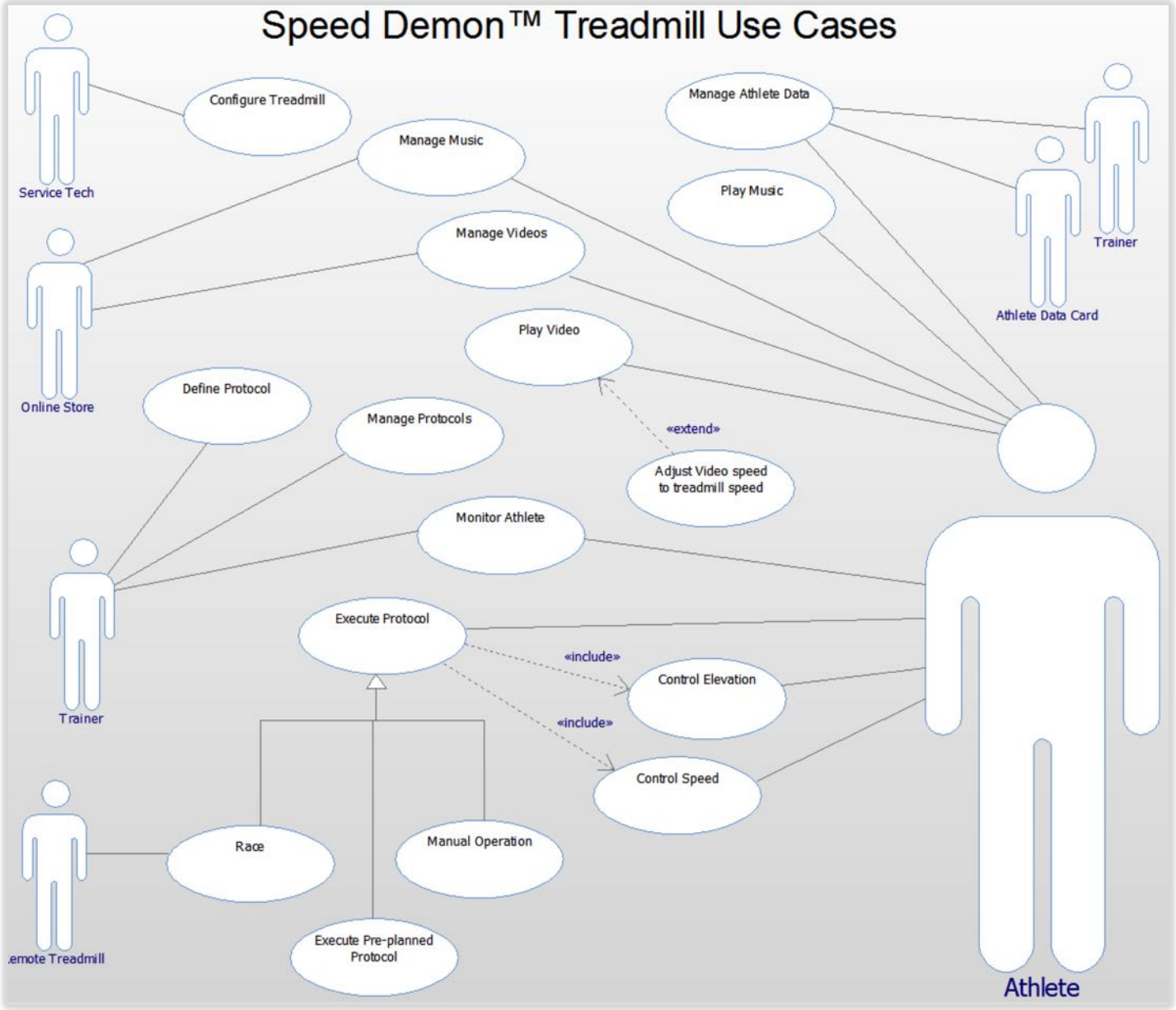
Today's Example: High Performance Treadmill: SpeedDemon™

- Biometrics
  - Heart rate
  - Speed
  - Power / VO_2 max
  - Elapsed time/interval time
  - Ground contact time
  - Vertical oscillation
  - Cadence
- Programmable protocols (workouts)
- Performance storage and analytics
- Music and video playlists
- Virtual racing (simulated or over-the-web)

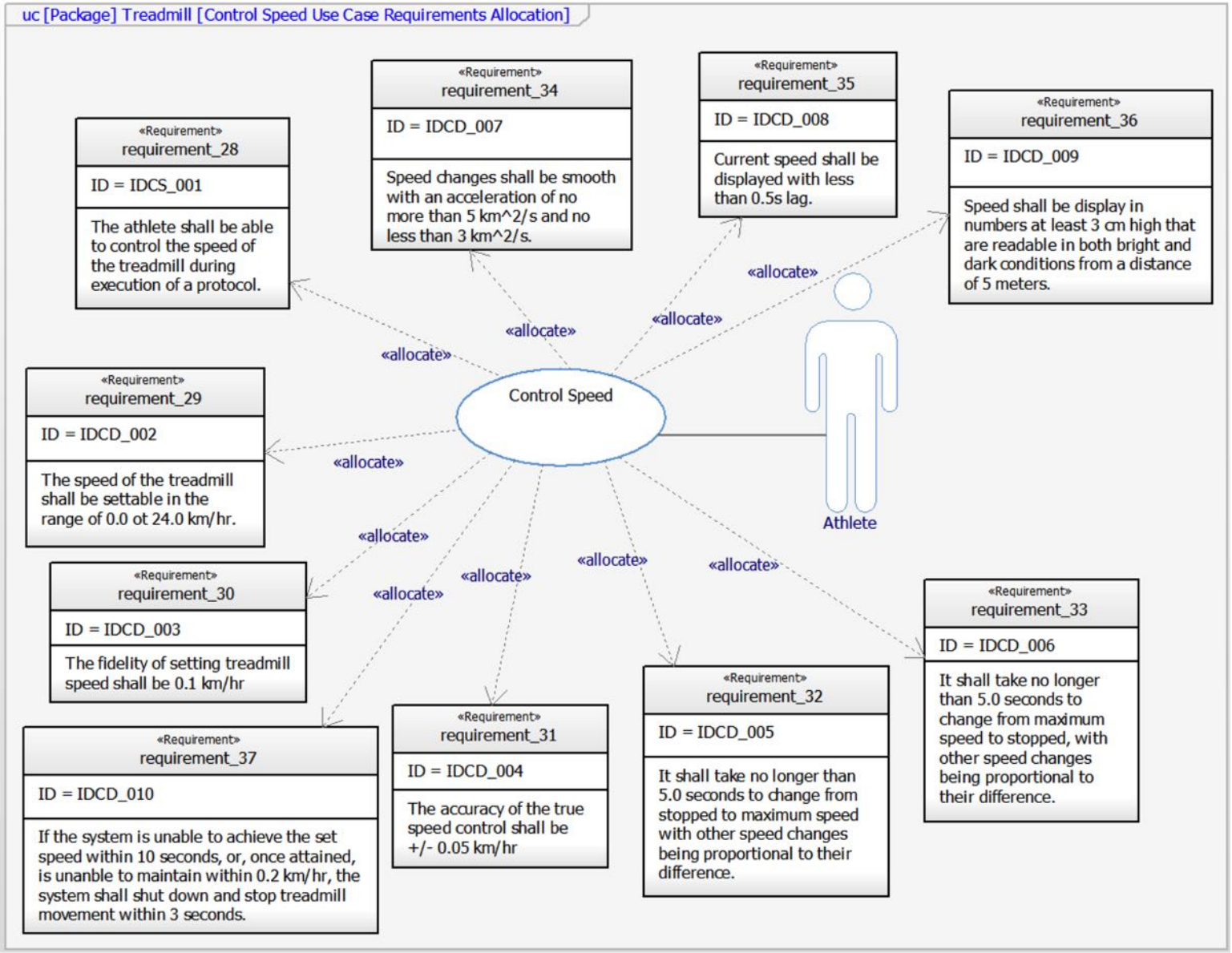




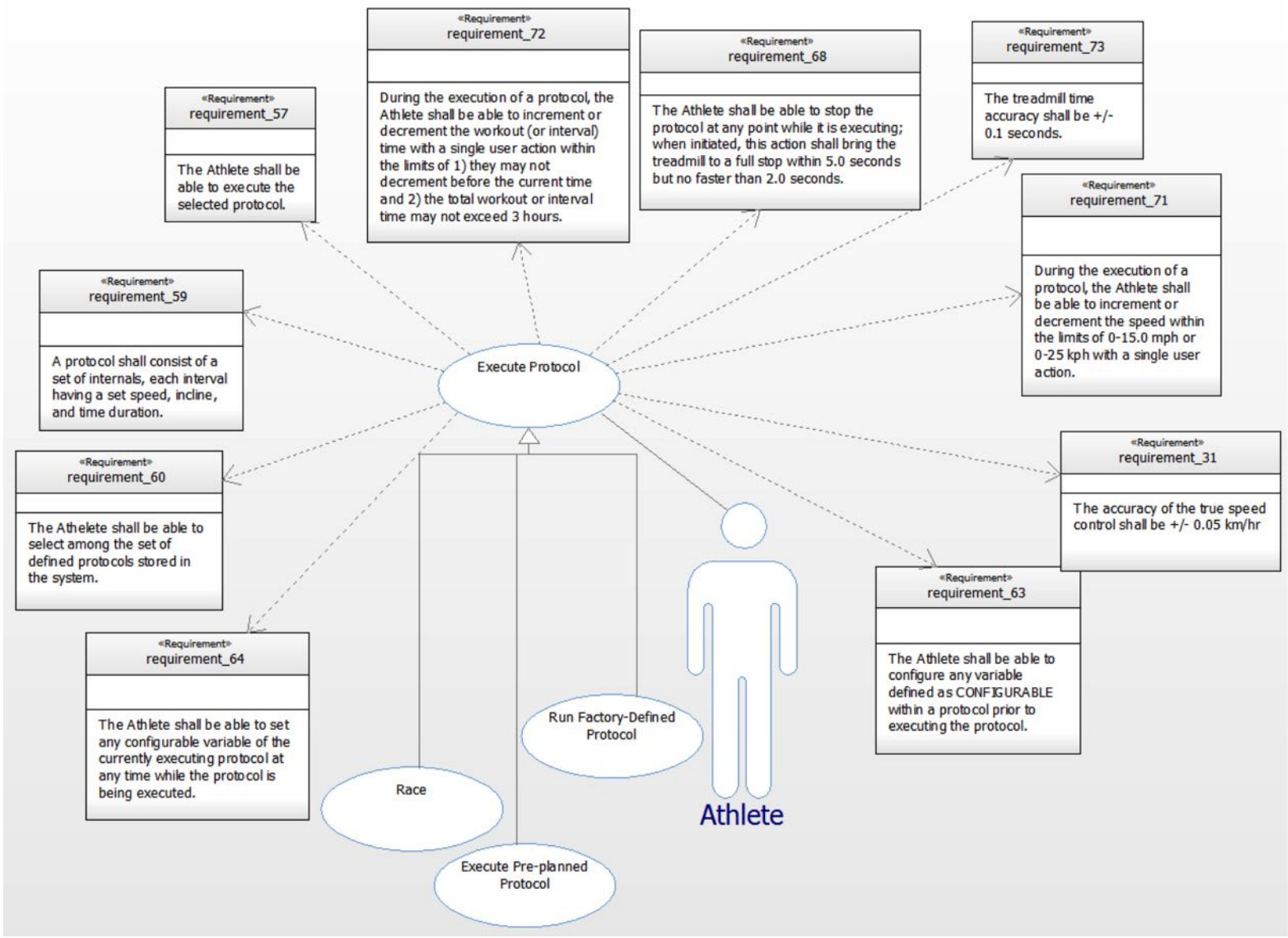
SpeedDemon Use Cases



Allocation of requirements to Control Speed Use Case



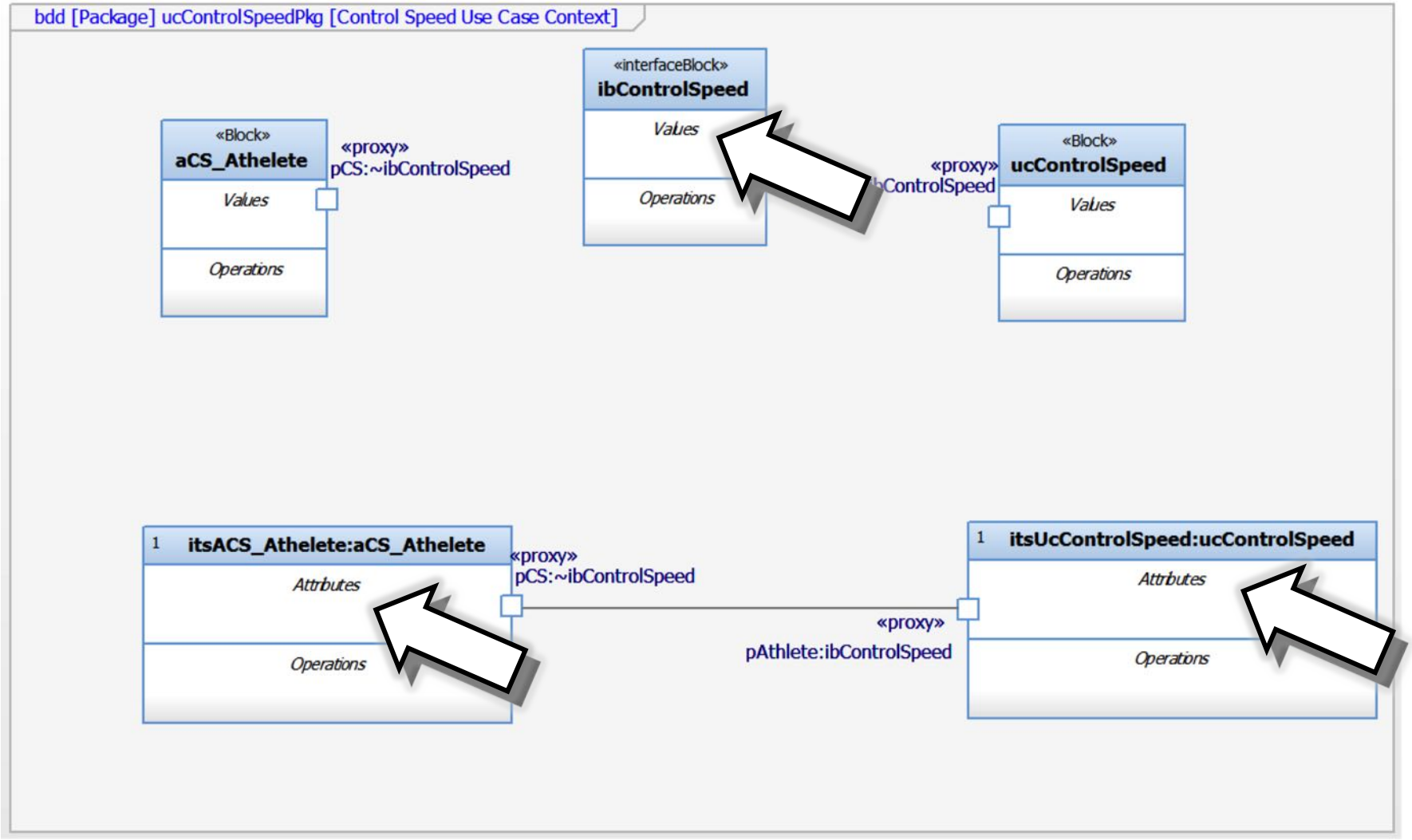
Allocate Requirements to Execute Protocol Use Case



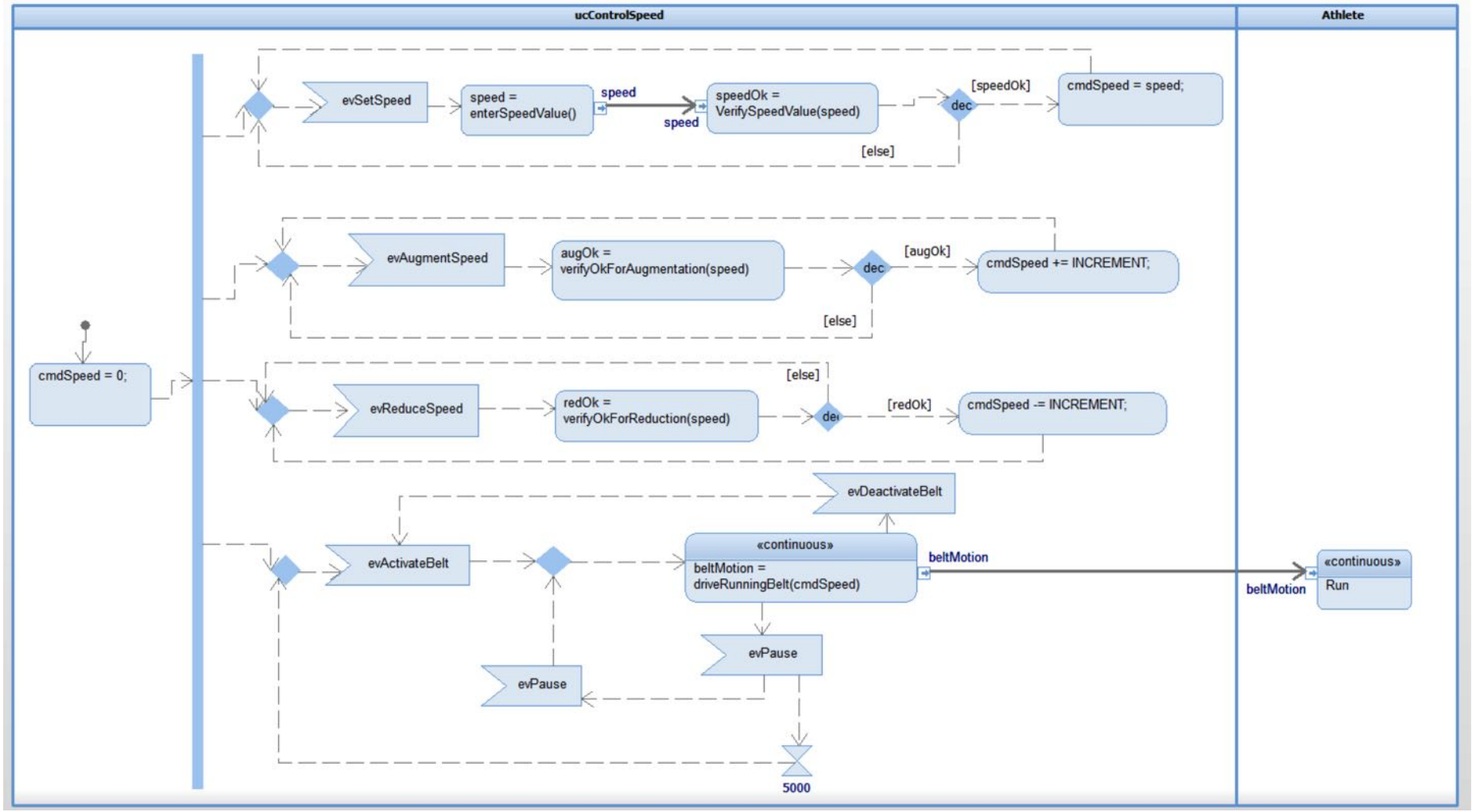
Showing allocations within a matrix (generated)

	Control Elevation	Control Speed	Monitor Athlete	Execute Protocol	Define Protocol	Manage Protocols	Play Music	Manage Music	Play Video	Manage Videos	Run Factory-Defined Protocol	Race	Adjust Video speed to treadmill speed	Configure Treadmill	Manage Athlete Data
requirement_24															
requirement_25															
requirement_26															
requirement_28		requirement_28													
requirement_29		requirement_29													
requirement_30		requirement_30													
requirement_31		requirement_31		requirement_31											
requirement_32		requirement_32													
requirement_33		requirement_33													
requirement_34		requirement_34													
requirement_35		requirement_35													
requirement_36		requirement_36													
requirement_37		requirement_37													
requirement_48															requirement_48
requirement_49															requirement_49
requirement_50															requirement_50
requirement_51															requirement_51
requirement_52															requirement_52
requirement_53															requirement_53
requirement_54															requirement_54
requirement_55															requirement_55
requirement_56															requirement_56
requirement_57				requirement_57							requirement_57				
requirement_59				requirement_59							requirement_59				
requirement_60				requirement_60							requirement_60				
requirement_61											requirement_61				
requirement_62											requirement_62				
requirement_63				requirement_63							requirement_63				
requirement_64				requirement_64							requirement_64 requirement_64_0				
requirement_65											requirement_65				
requirement_66											requirement_66				
requirement_67											requirement_67				
requirement_68				requirement_68							requirement_68				
requirement_71				requirement_71											
requirement_72				requirement_72											
requirement_73				requirement_73											

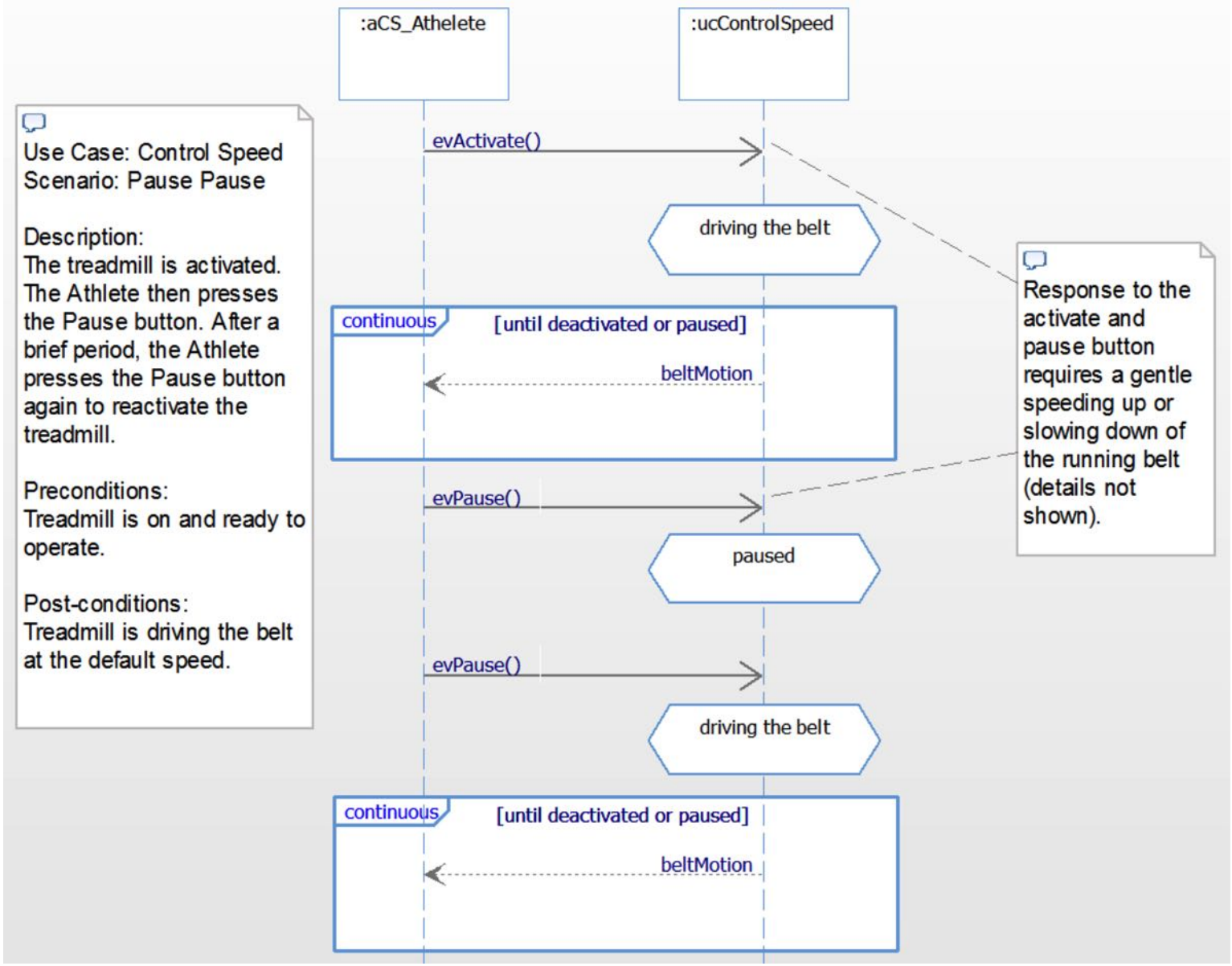
Create context for use case execution model



Creating an activity diagram to show behavioral flows

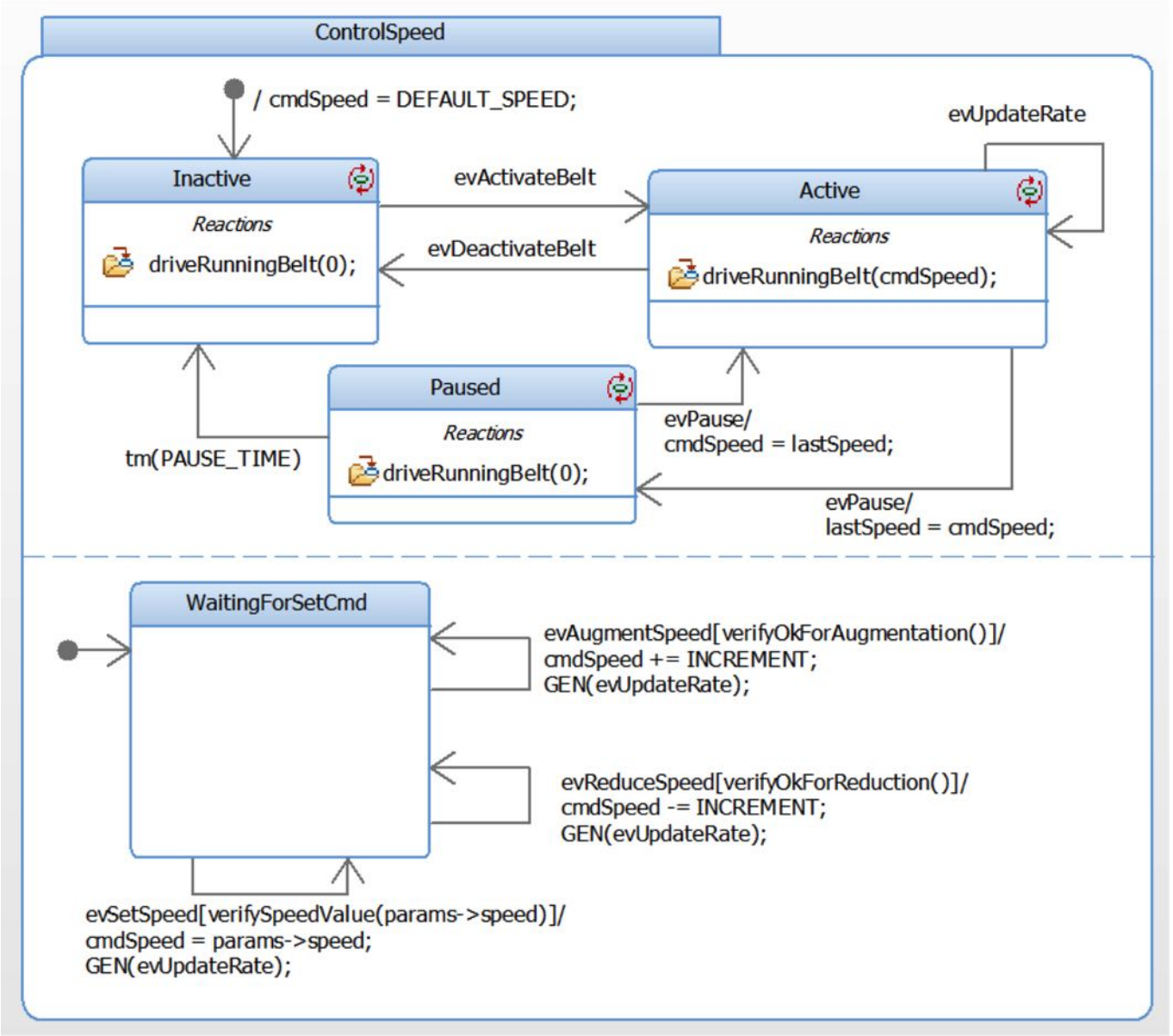


Create scenarios from activity diagram

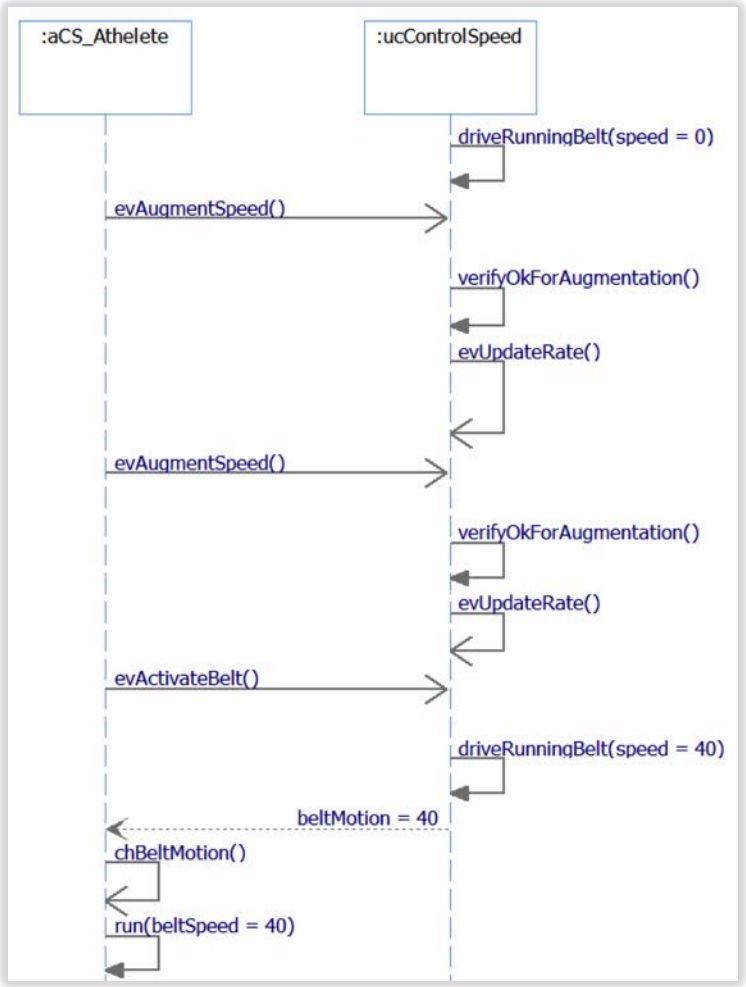


Create Behavioral Model Realizing Requirements for Use Case

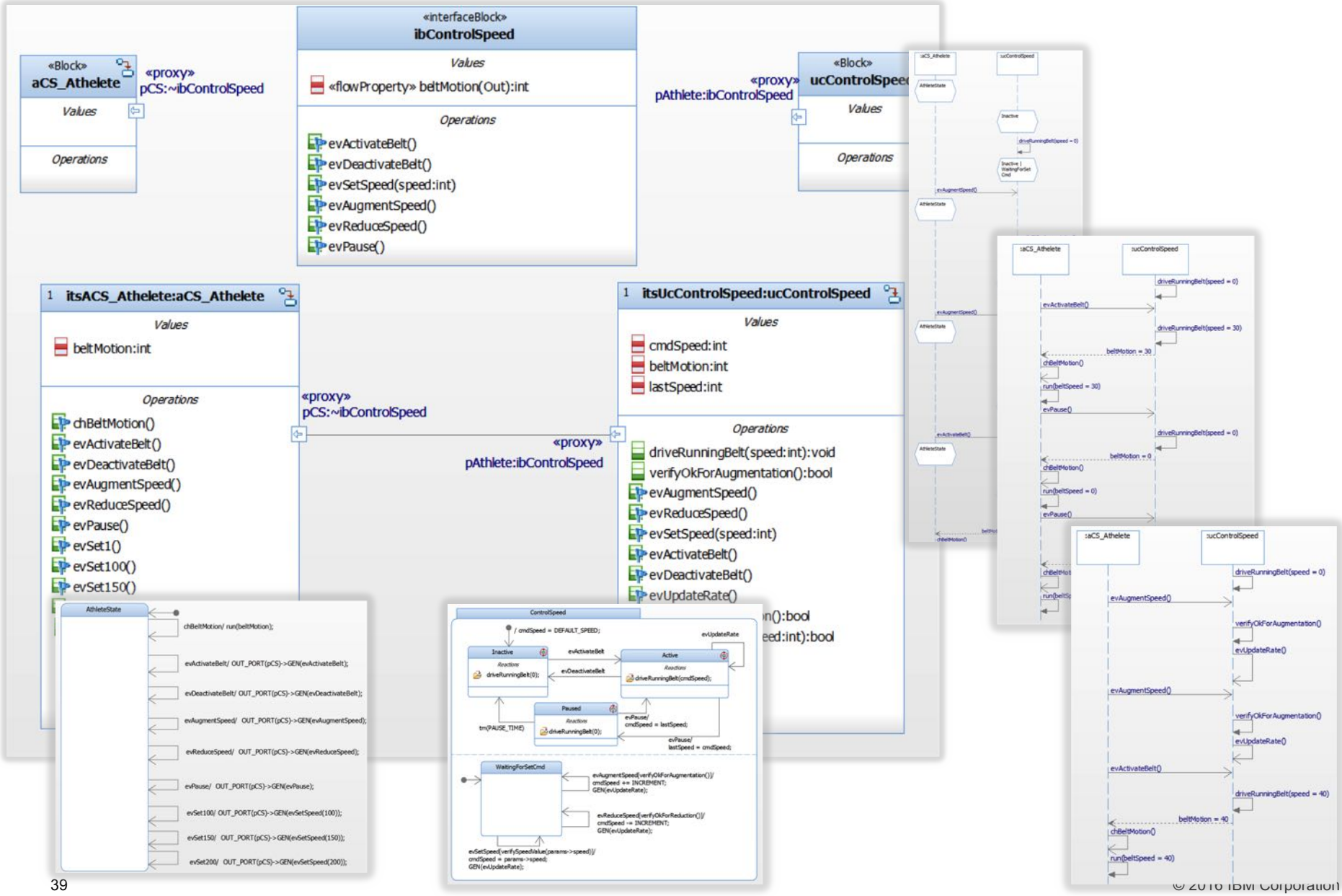
- Notes
  - This is after several Model-Verify–Elaborate iterations
  - Start with 2-4 states, and add more each nanocycle iteration
- Questions still remain
  - Can the system be commanded into OFF directly?
  - How are timing requirements verified?
  - Reqs for degraded operation are now discovered to be inadequate – **add new reqs!**



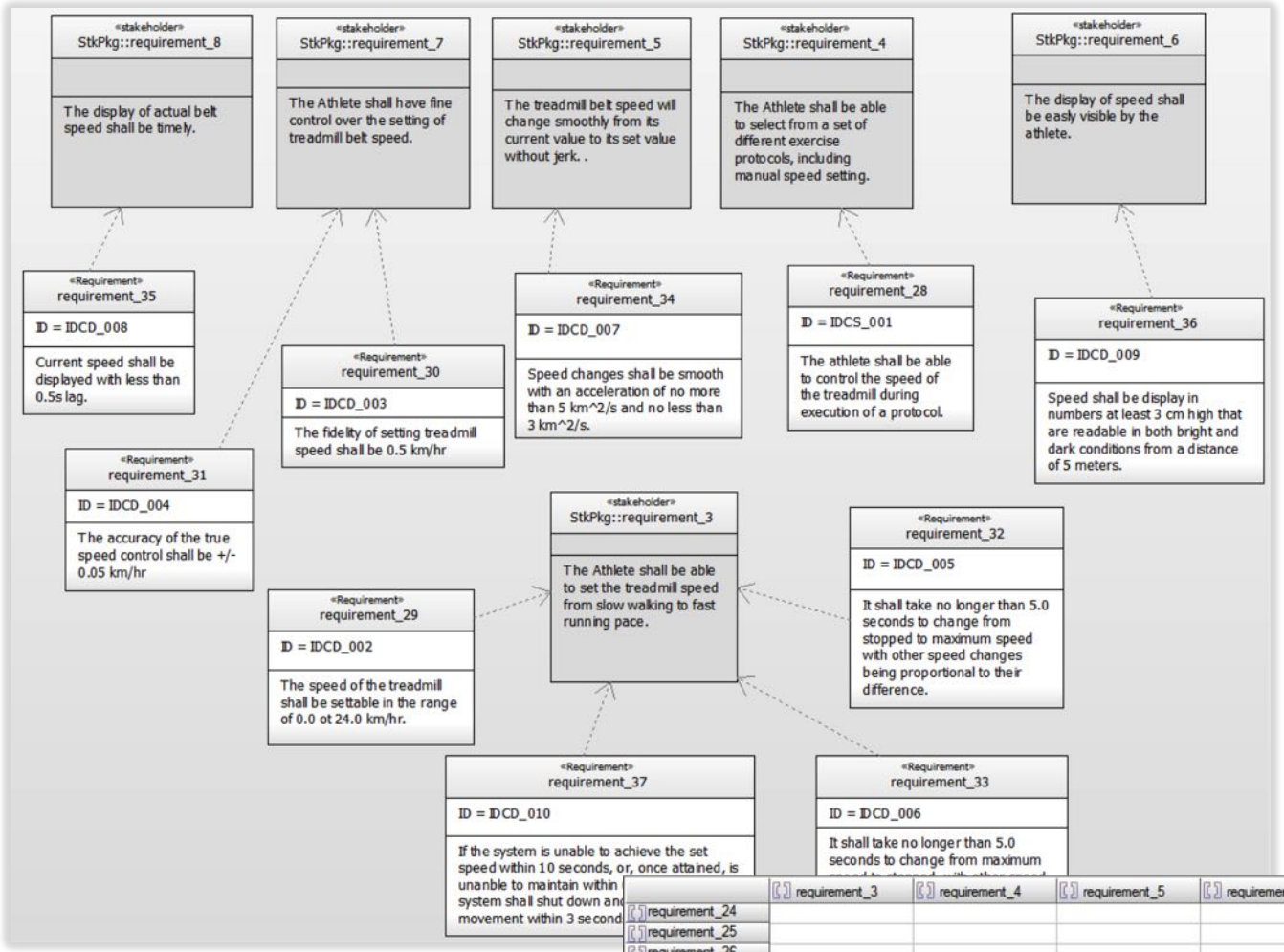
Verify Requirements Adequacy Through Execution



Identify gaps – add missing reqs – elaborate model - reexecute



Add traceability



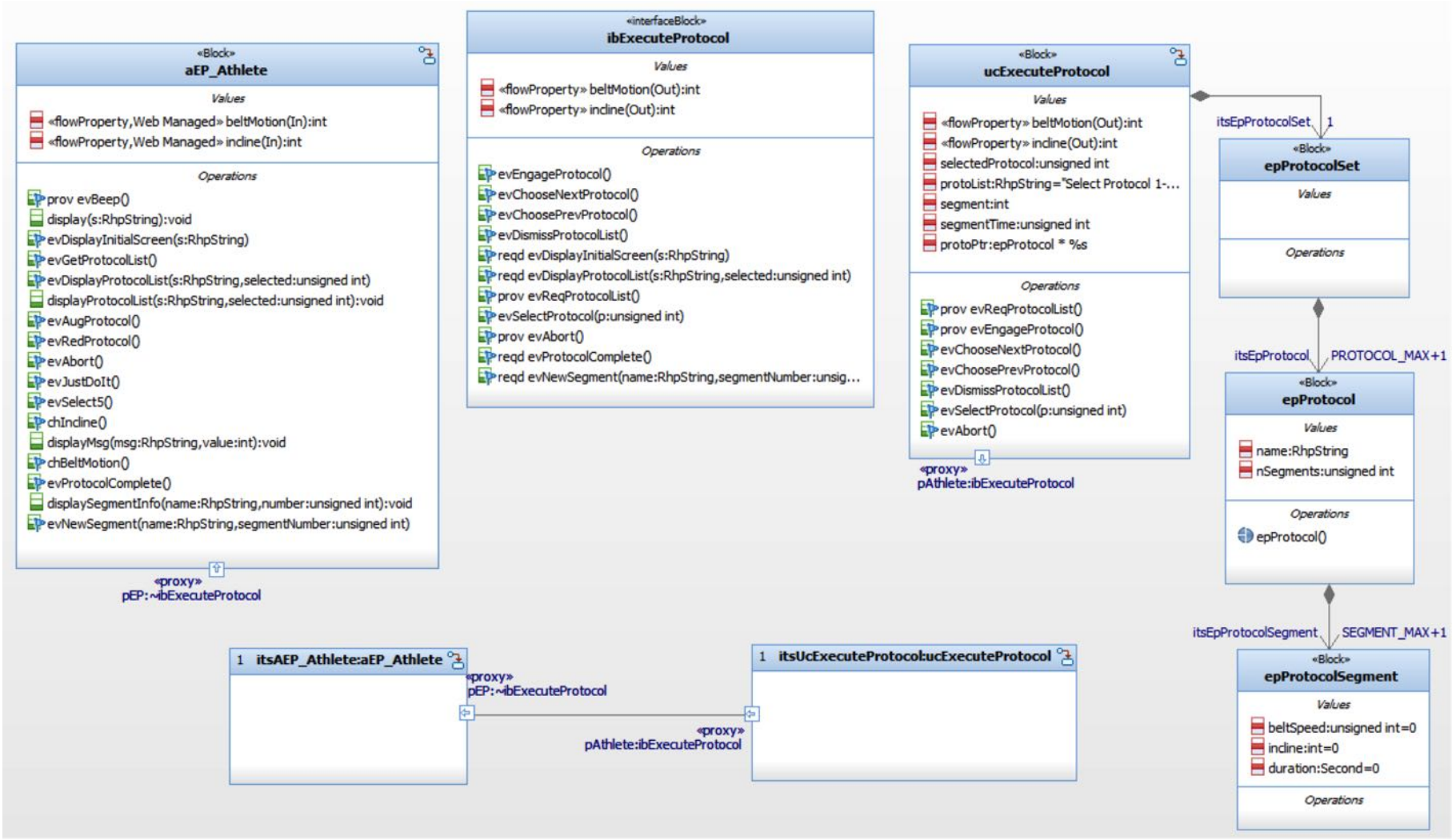
Stakeholder requirements

System requirements

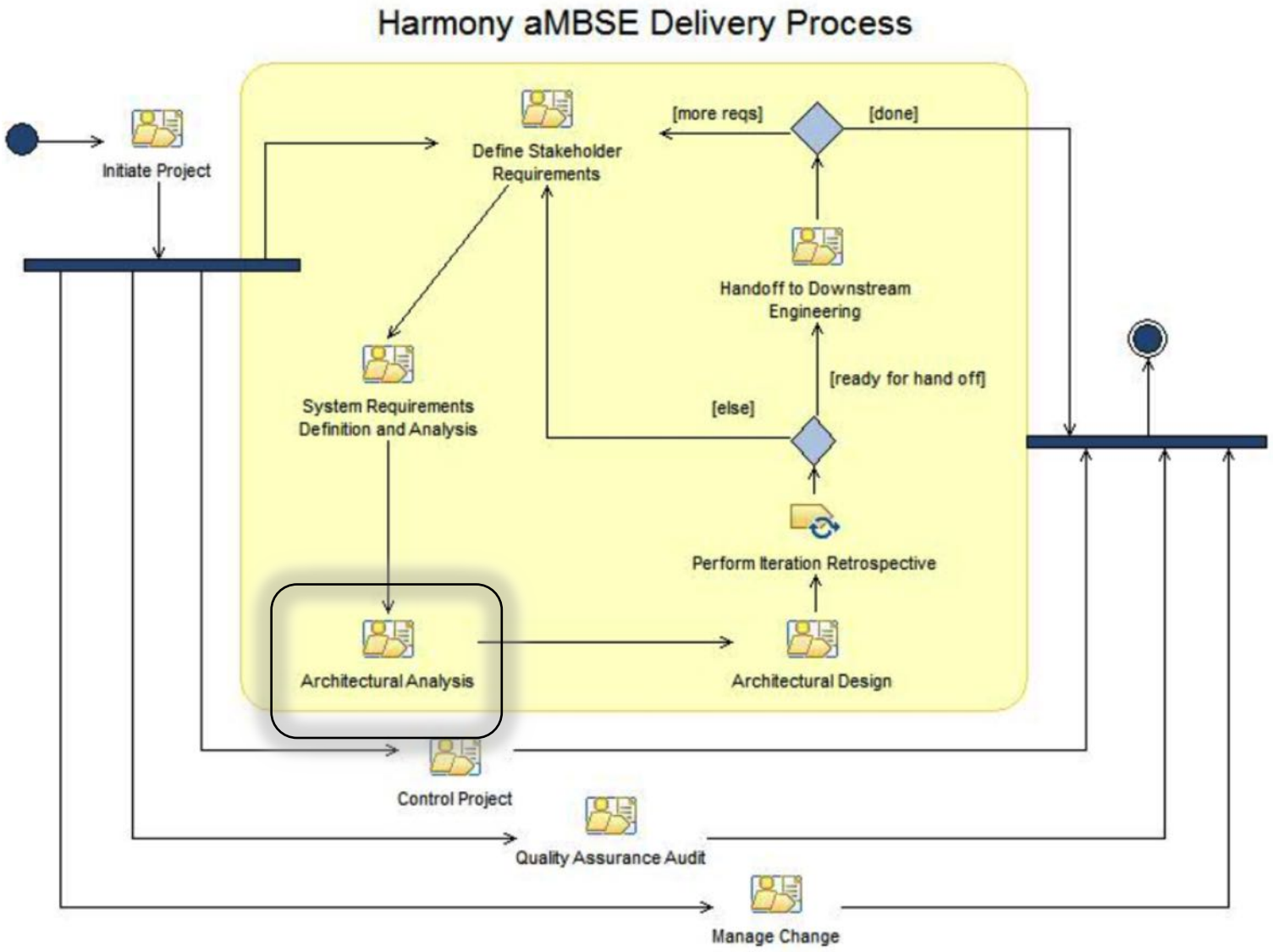
Generated trace matrix

	requirement_3	requirement_4	requirement_5	requirement_6	requirement_7	requirement_8
requirement_24						
requirement_25						
requirement_26						
requirement_28		requirement_4				
requirement_29	requirement_3					
requirement_30					requirement_7	
requirement_31					requirement_7	
requirement_32	requirement_3					
requirement_33	requirement_3					
requirement_34			requirement_5			
requirement_35						requirement_8
requirement_36				requirement_6		
requirement_37	requirement_3					

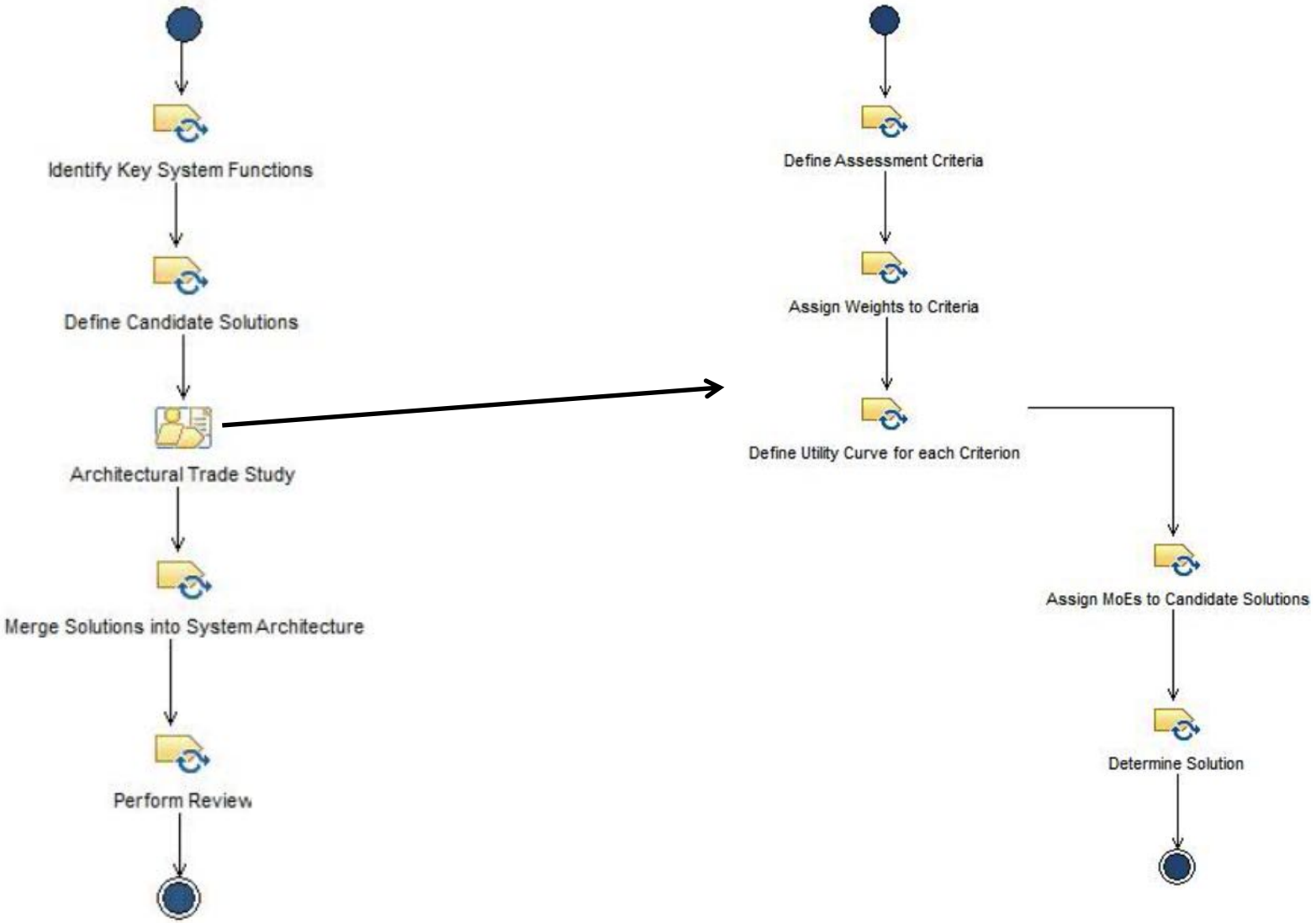
Generate Logical Interfaces



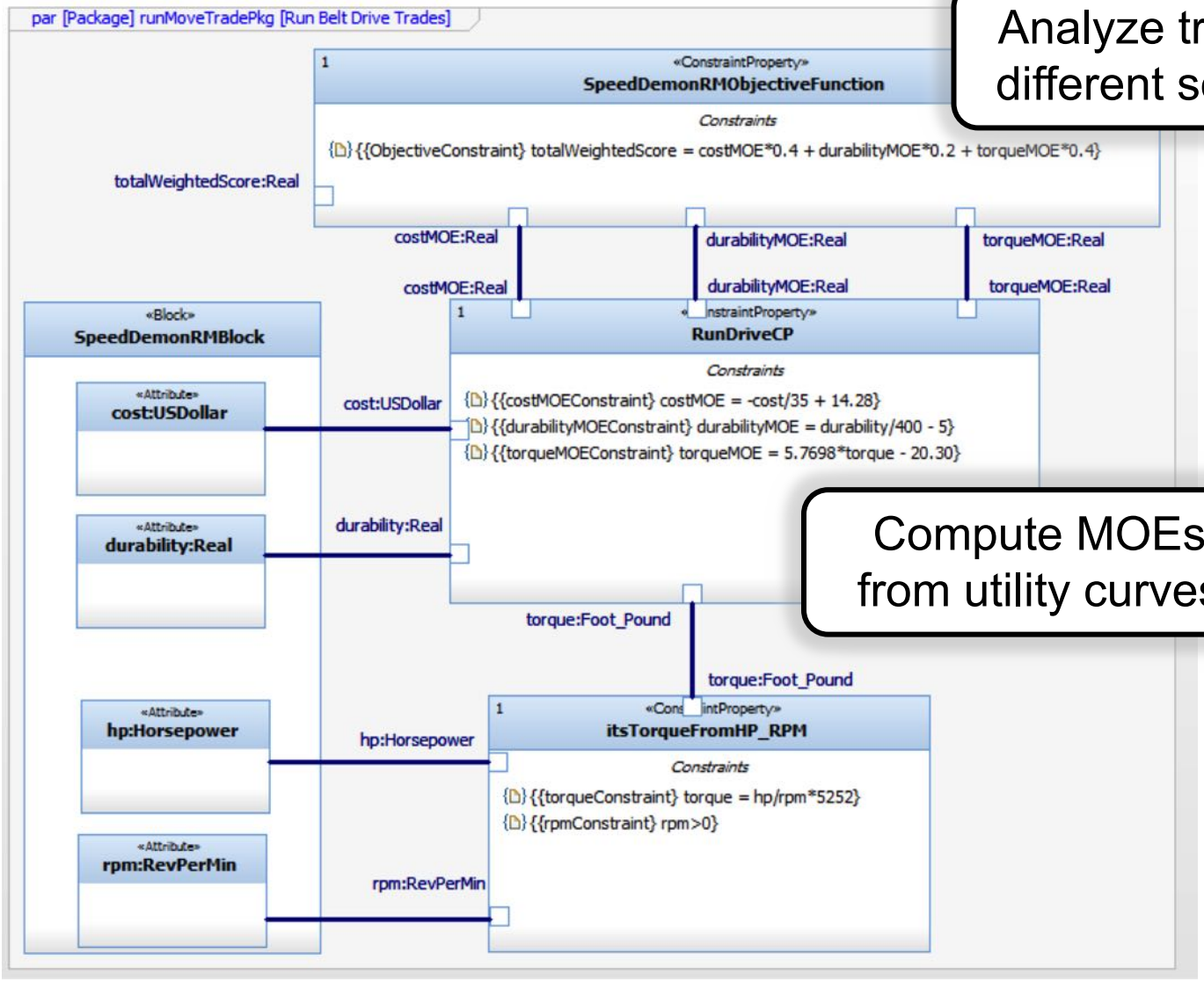
Systems Architectural Analysis



Systems Architectural Analysis



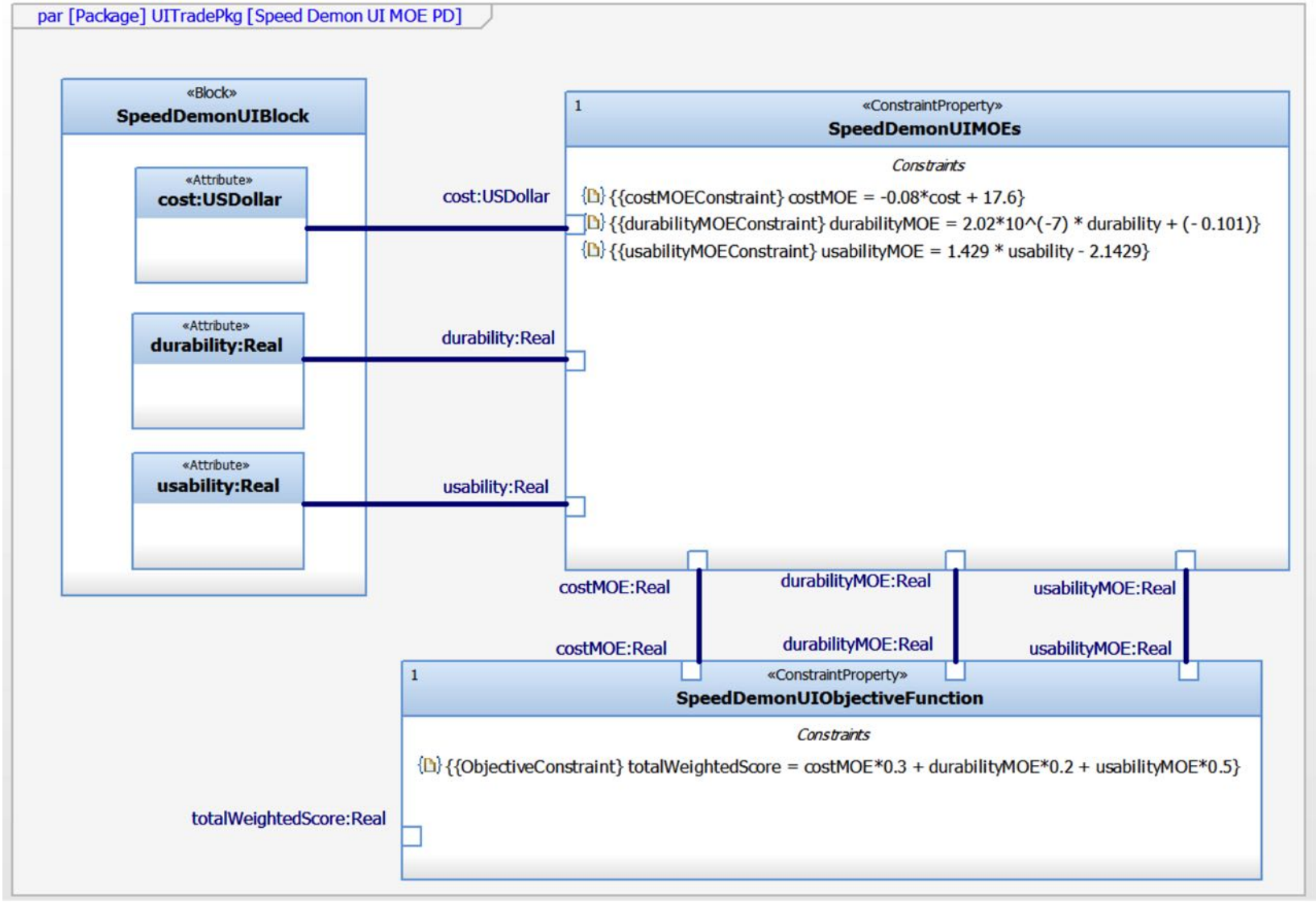
SysML Parametric Diagram for Trades for SpeedDemon Motor



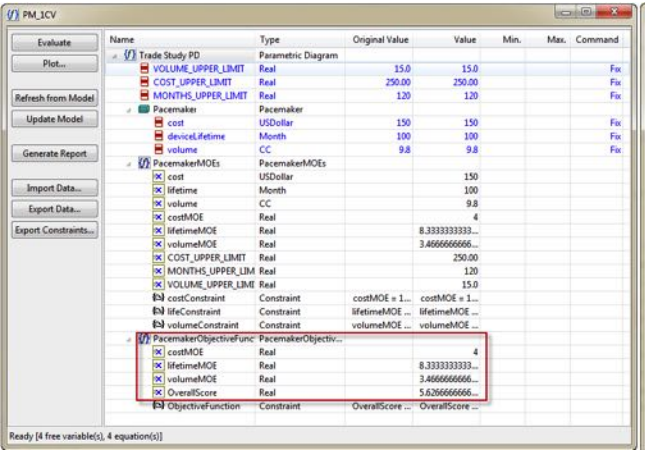
Analyze trades of different solutions

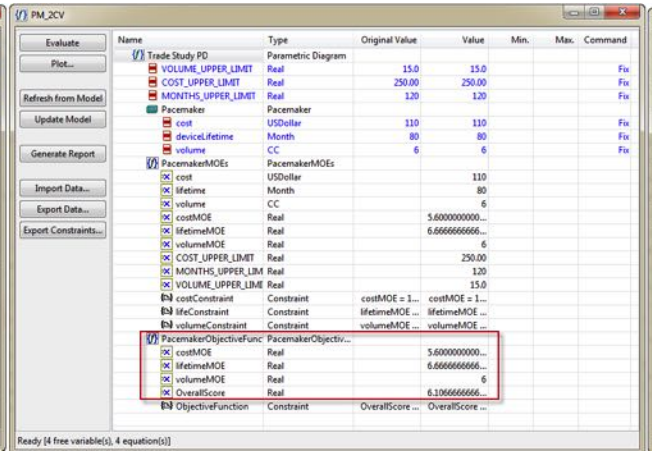
Compute MOEs from utility curves

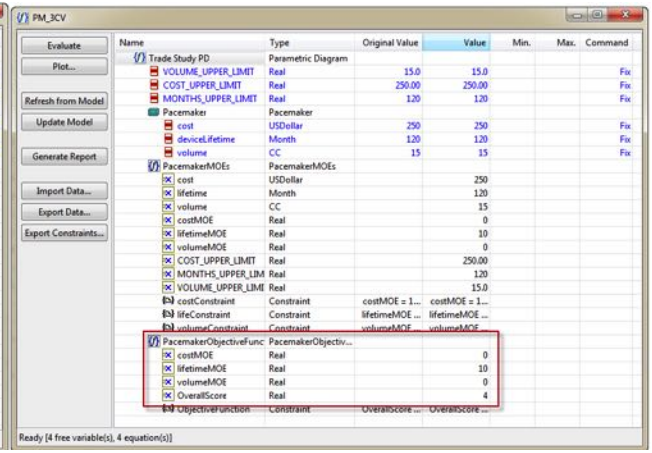
SysML Parametric Diagram for Trades for UI system



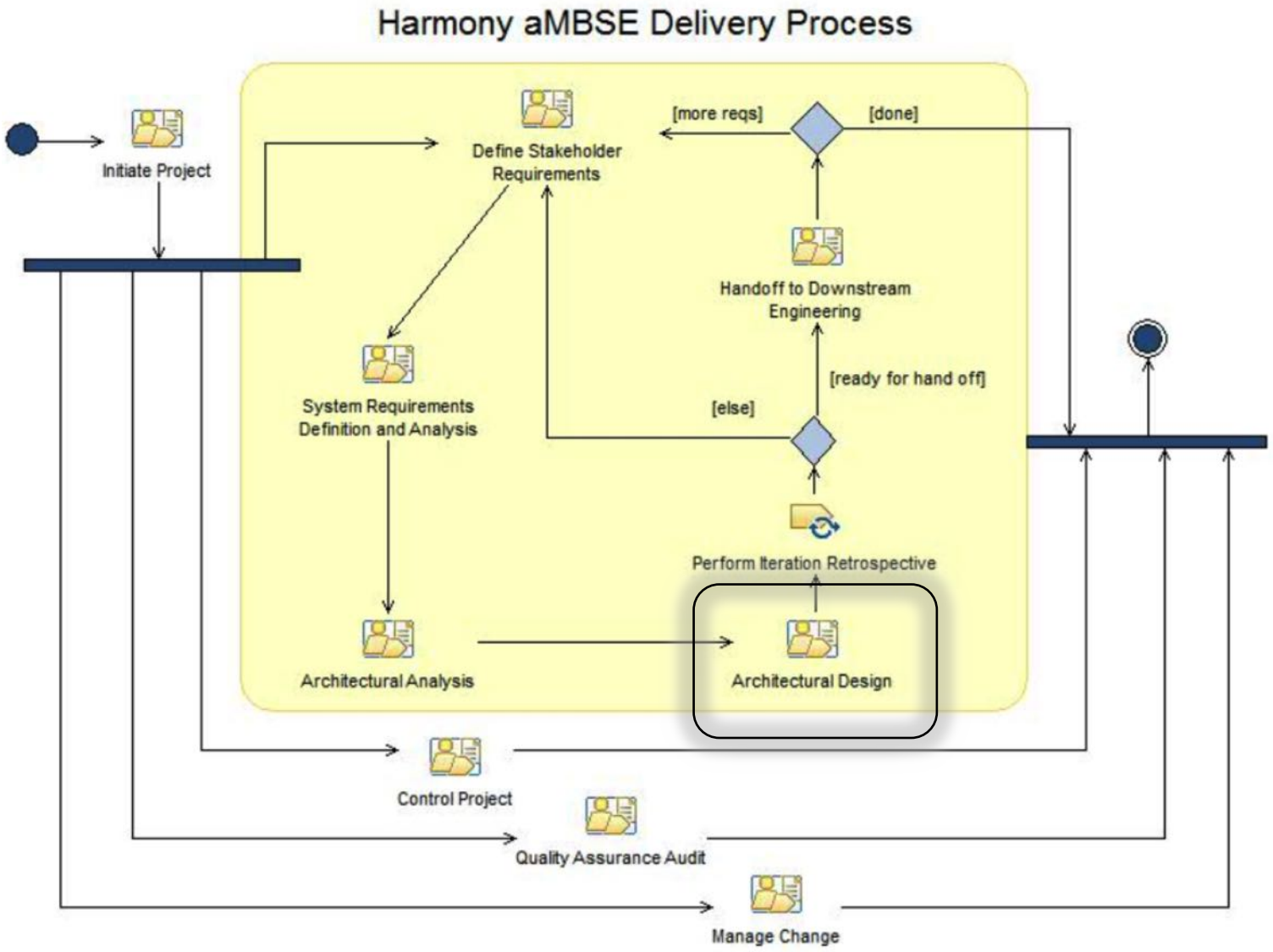
Outputs of the trade analysis



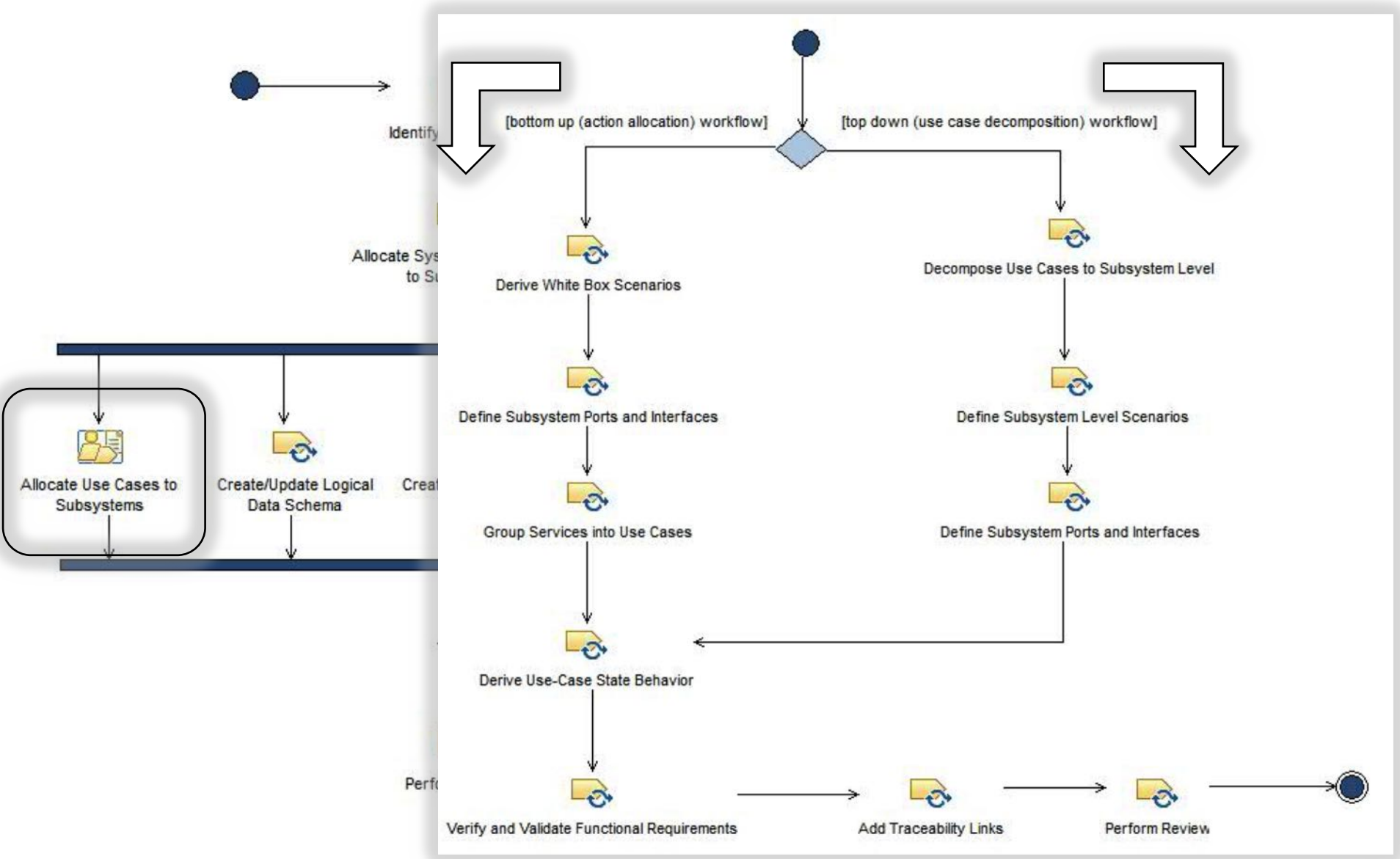




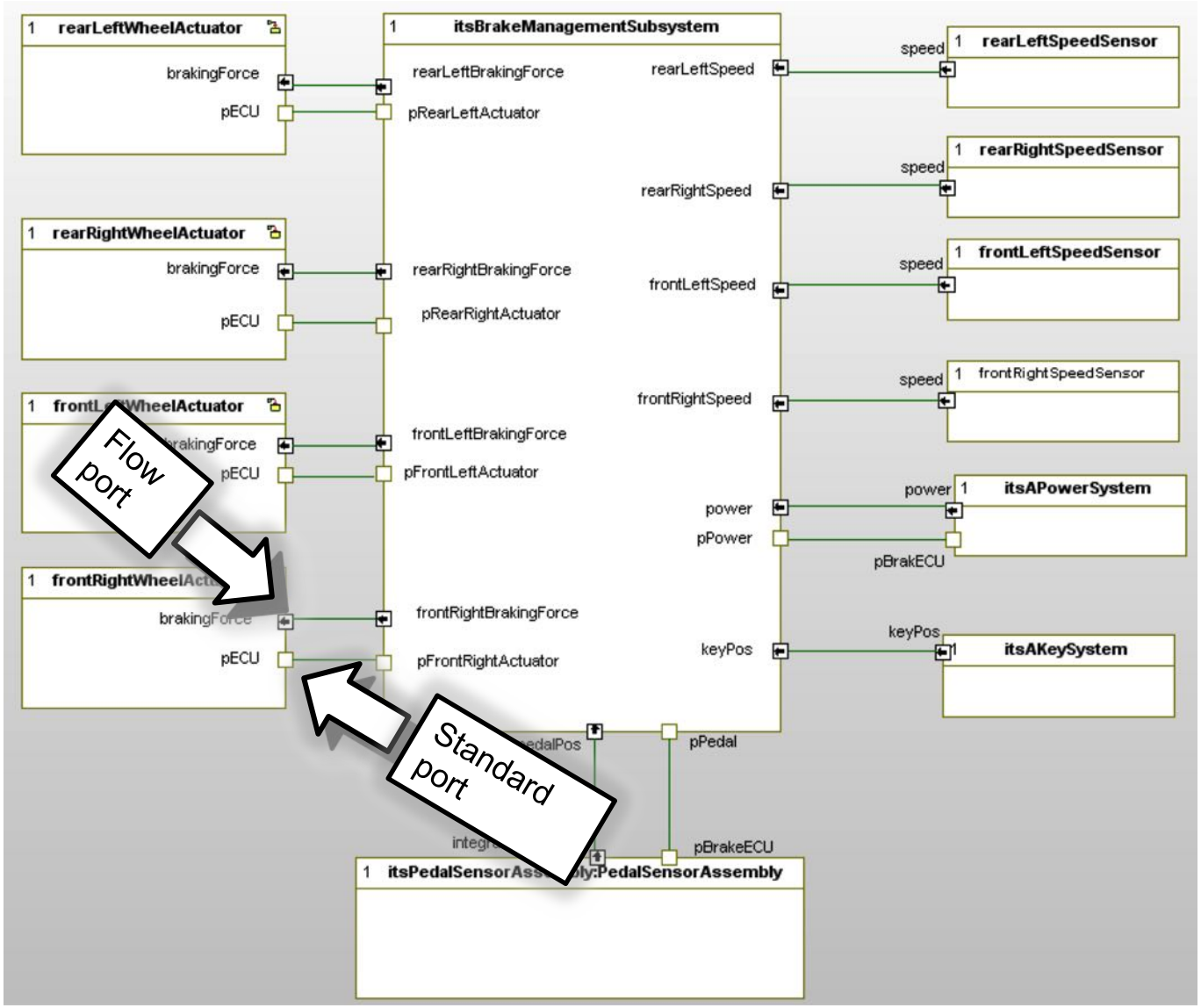
Architectural Design



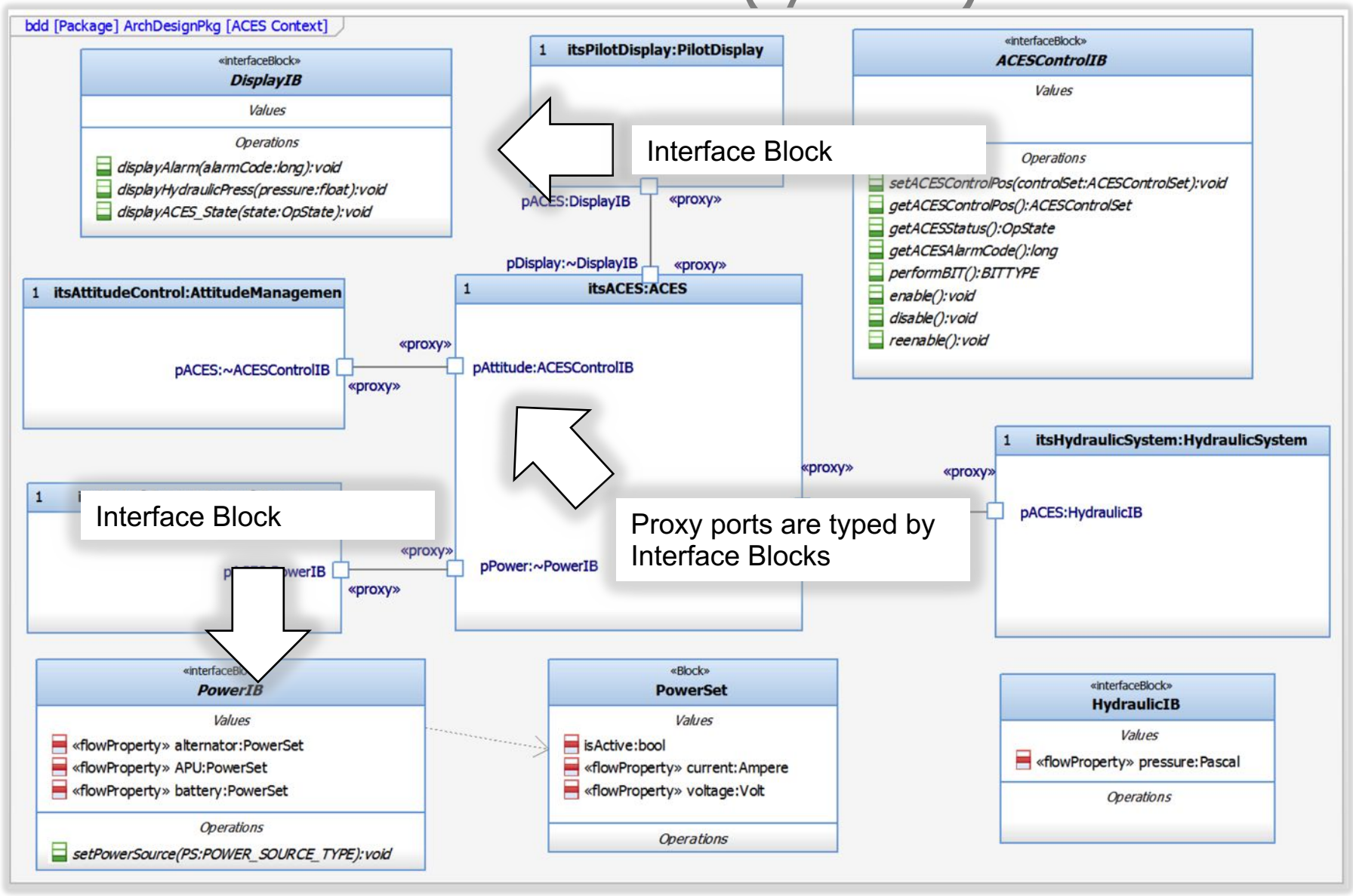
Activity: Allocate Use Cases to Subsystems



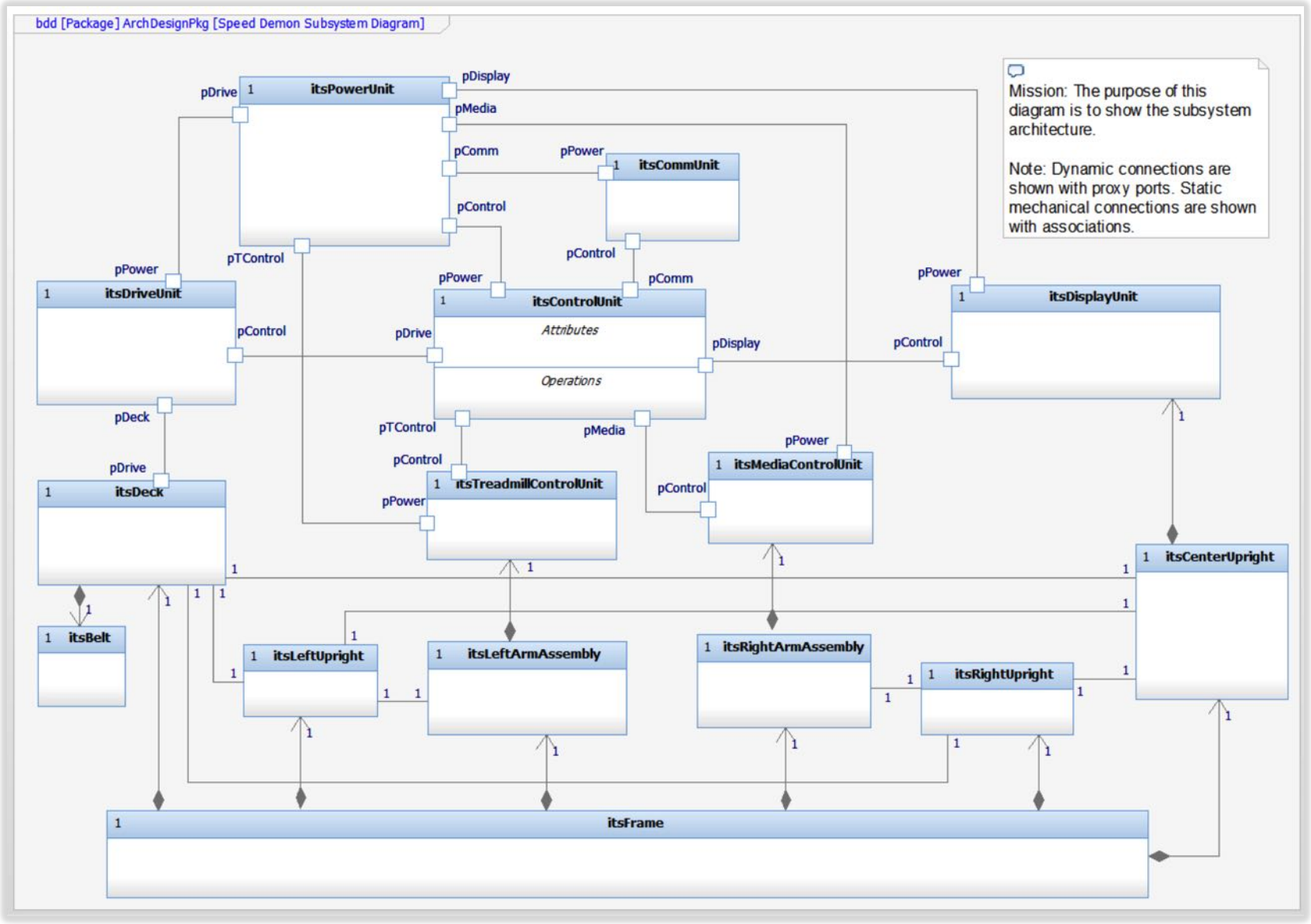
Architecture View for Control Flow (UML & SysML 1.2)



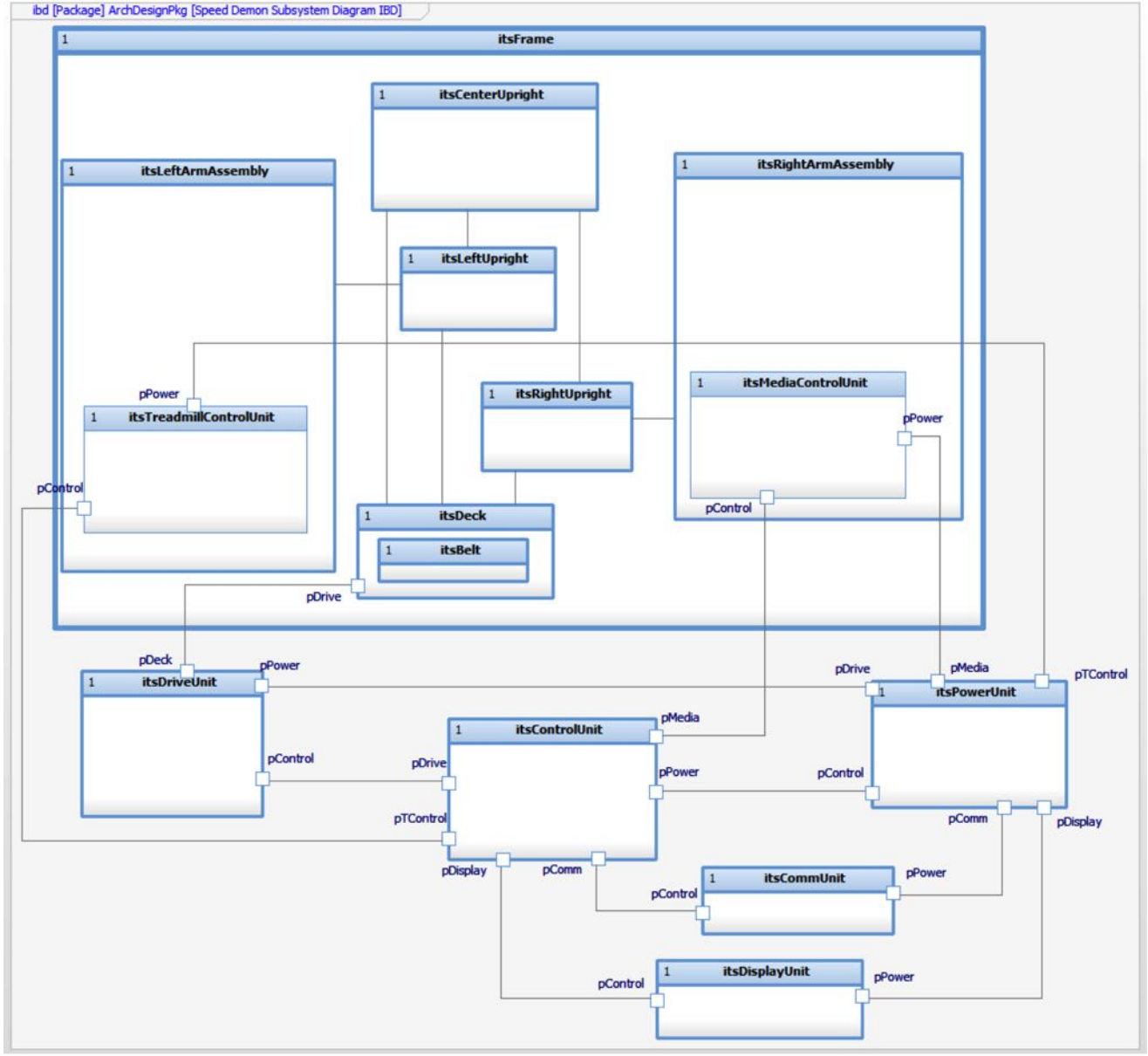
Architecture View for Control Flow (SysML 1.3)



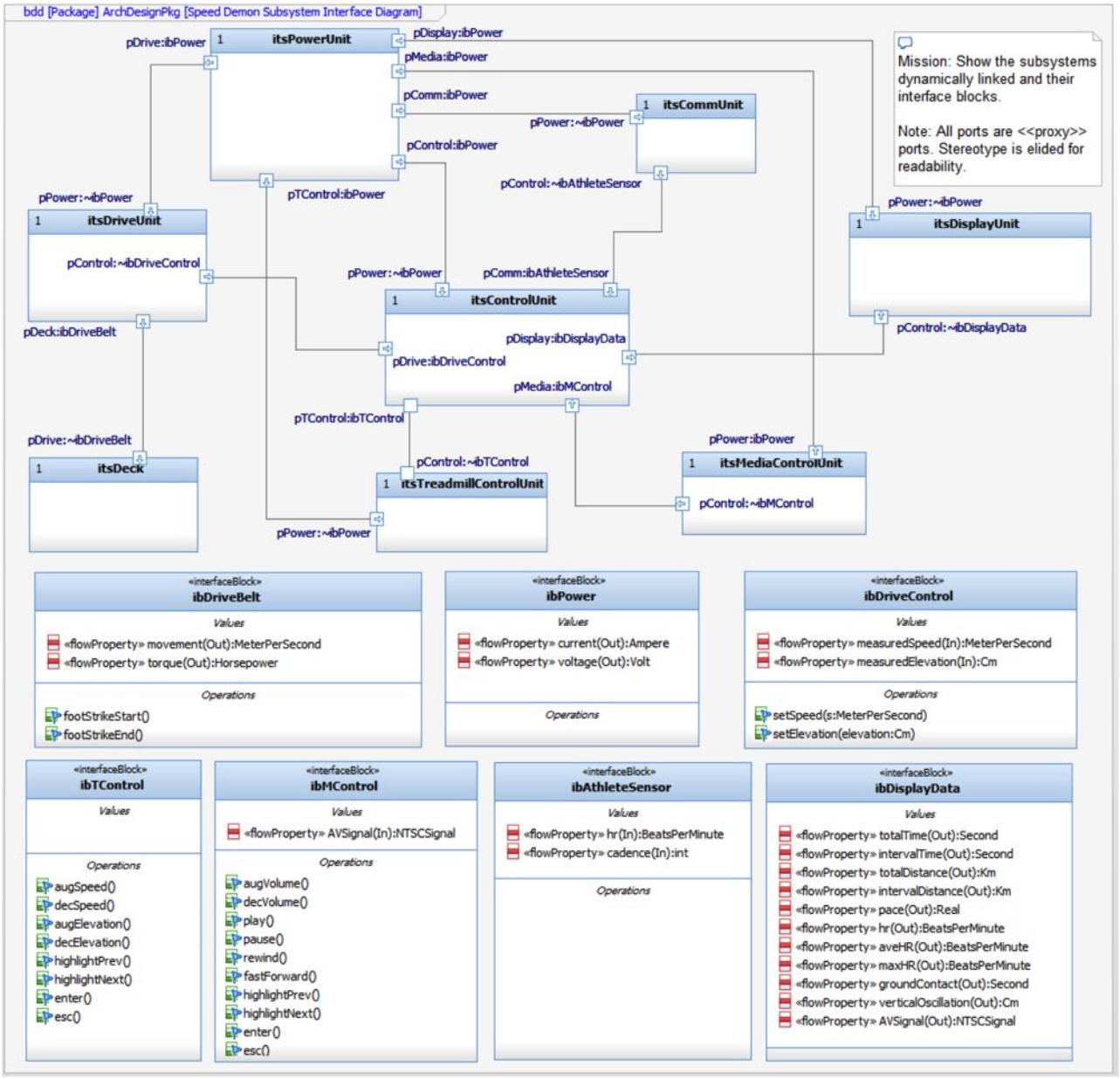
Architectural View – Subsystem Diagram



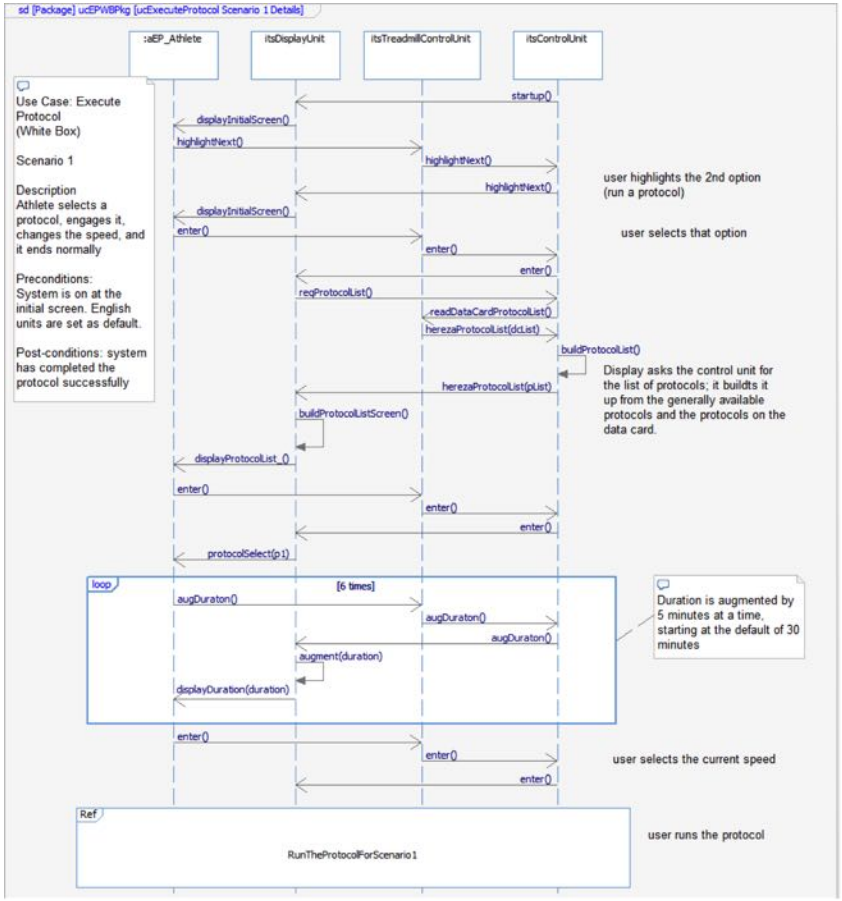
Architectural View – Subsystem Detail Structure

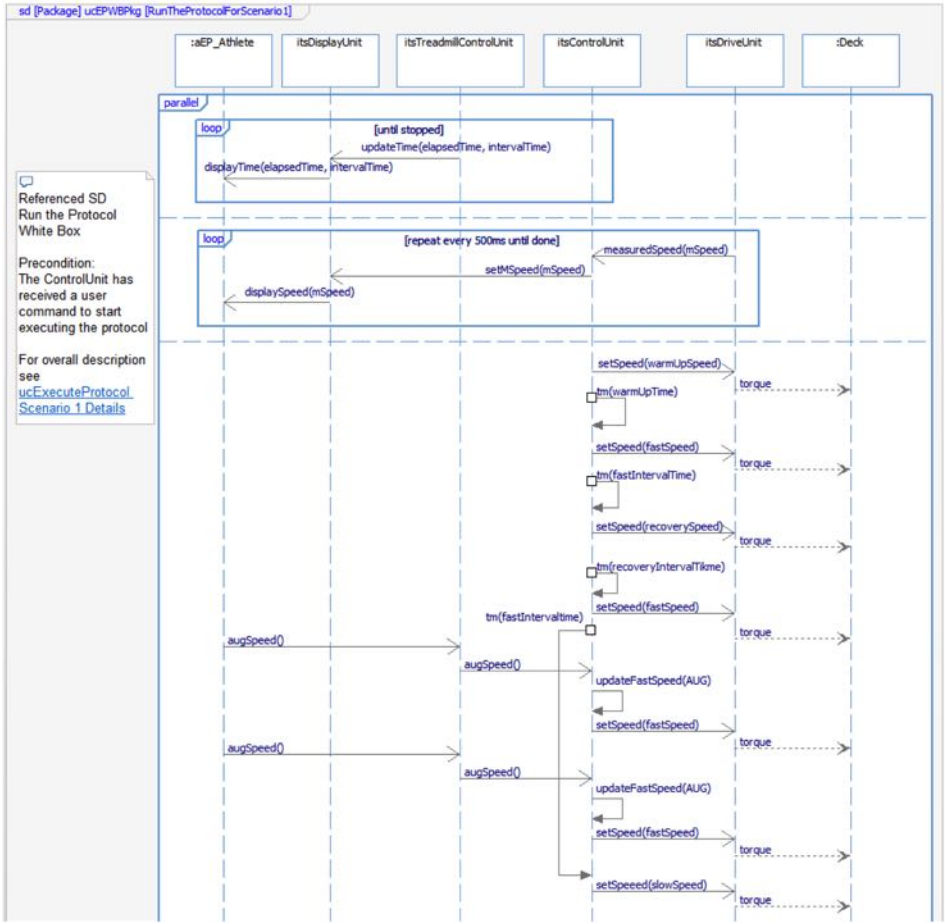


Architecture with (logical) subsystem interfaces

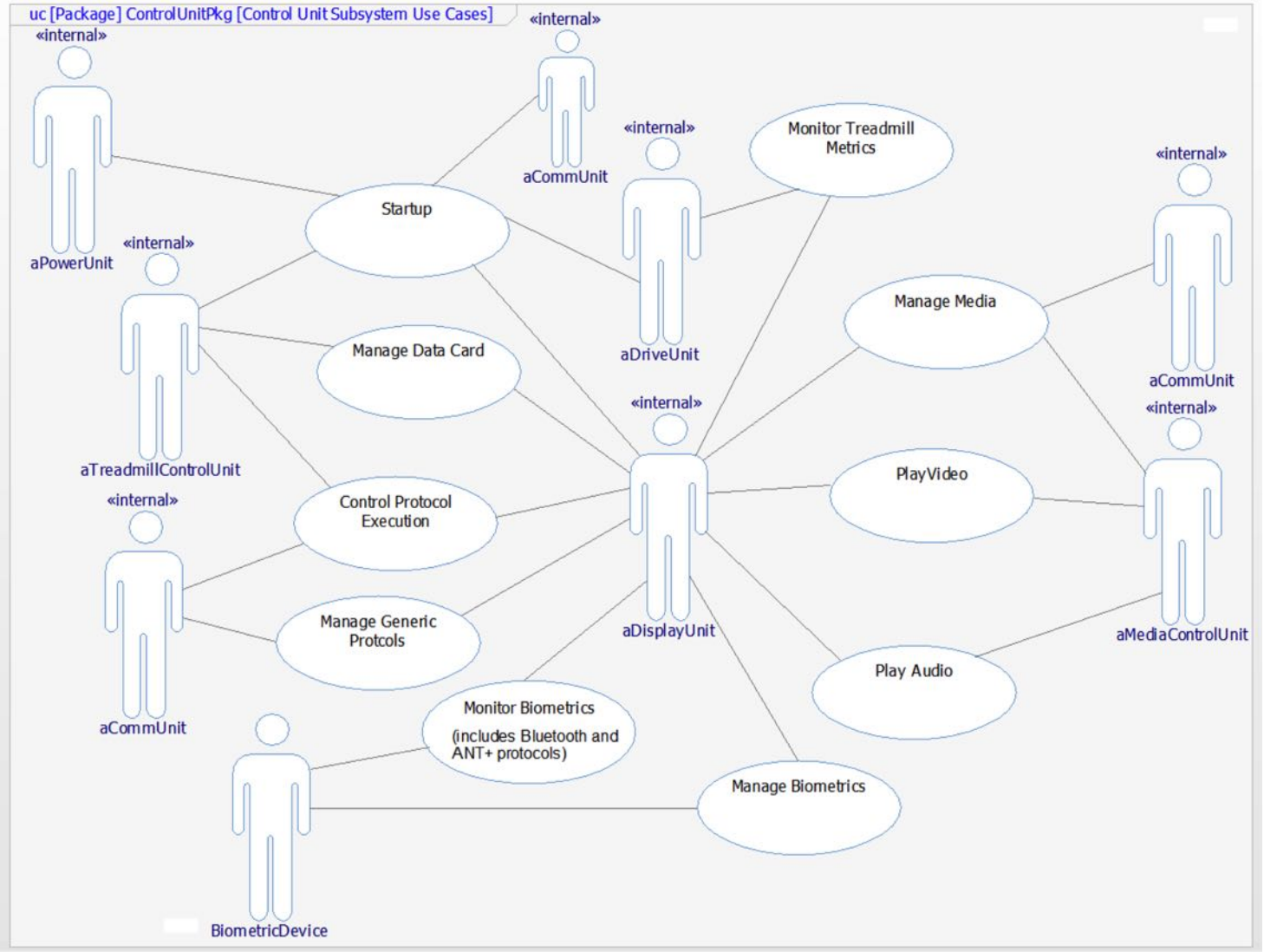


Architectural Verification via Simulation/Execution

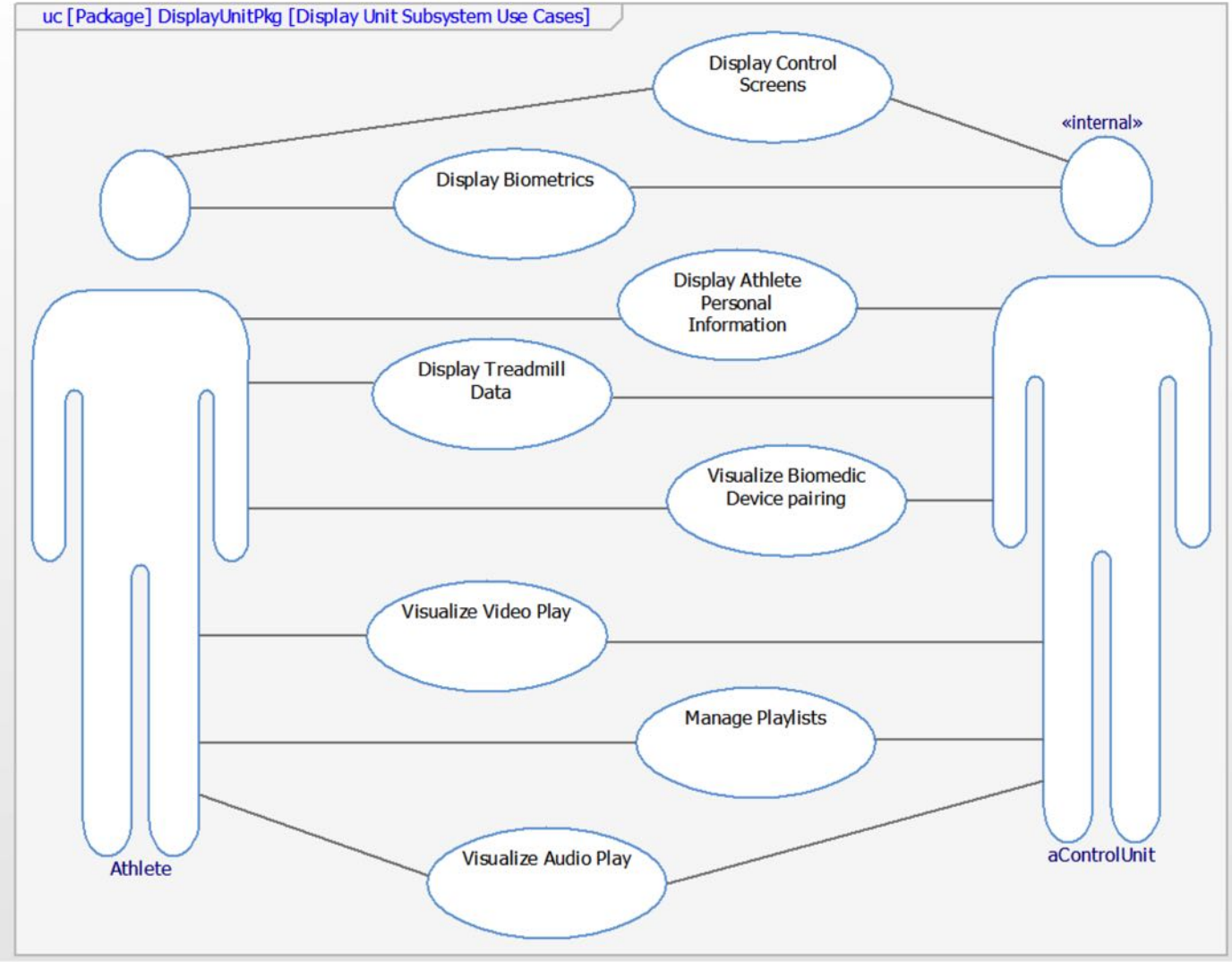




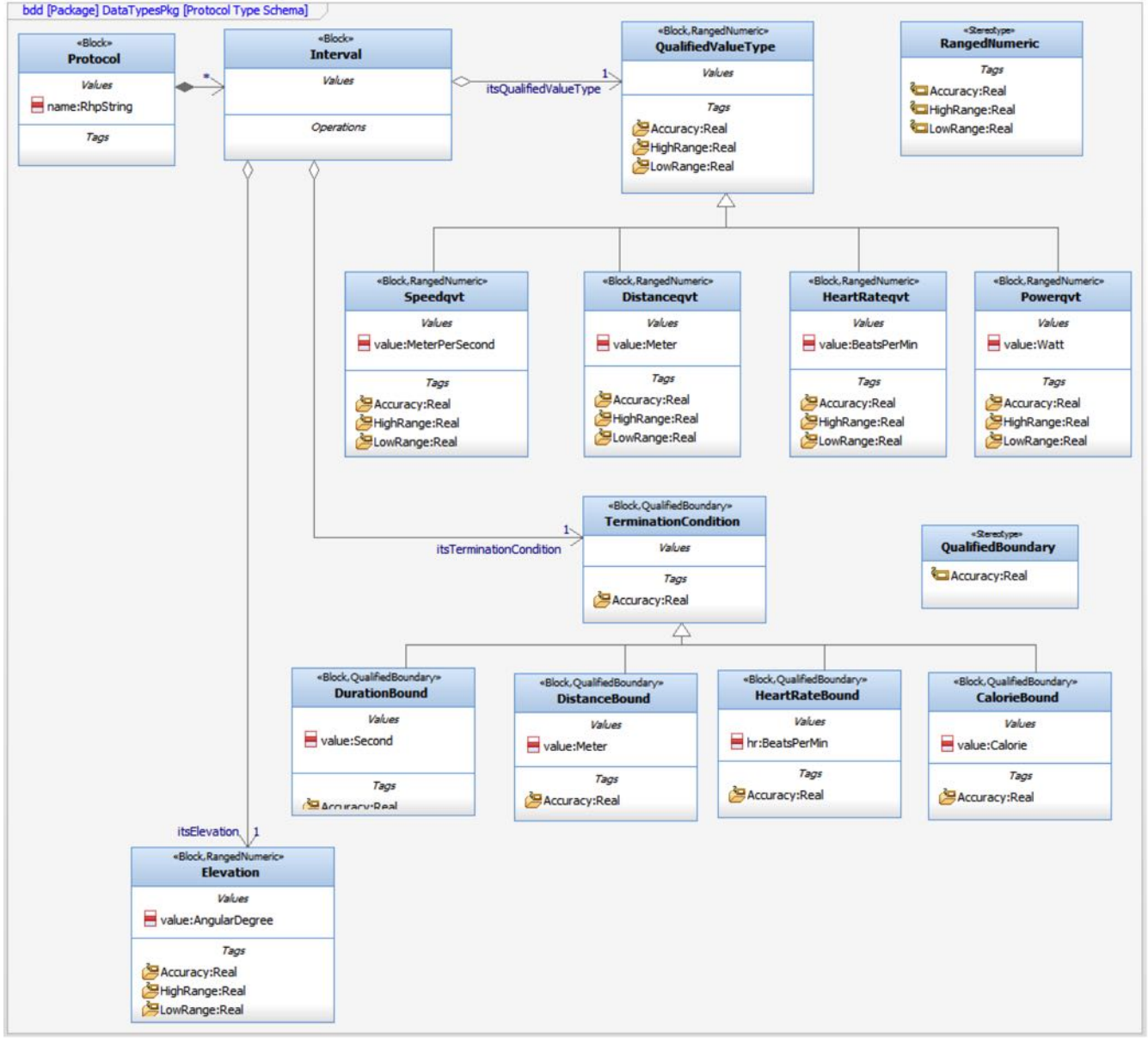
Subsystem Specification (Control Unit)



Subsystem Specification (Display Unit)

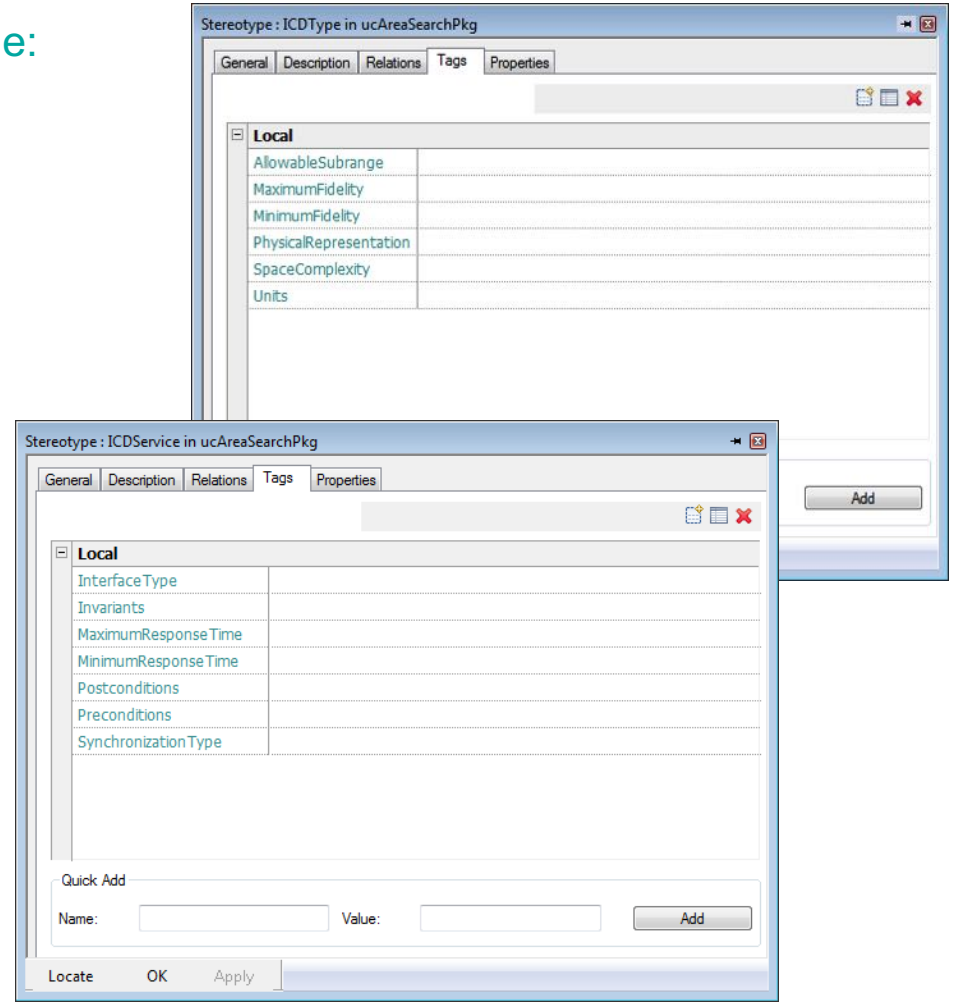


Update / Elaborate Data/Flow Schema

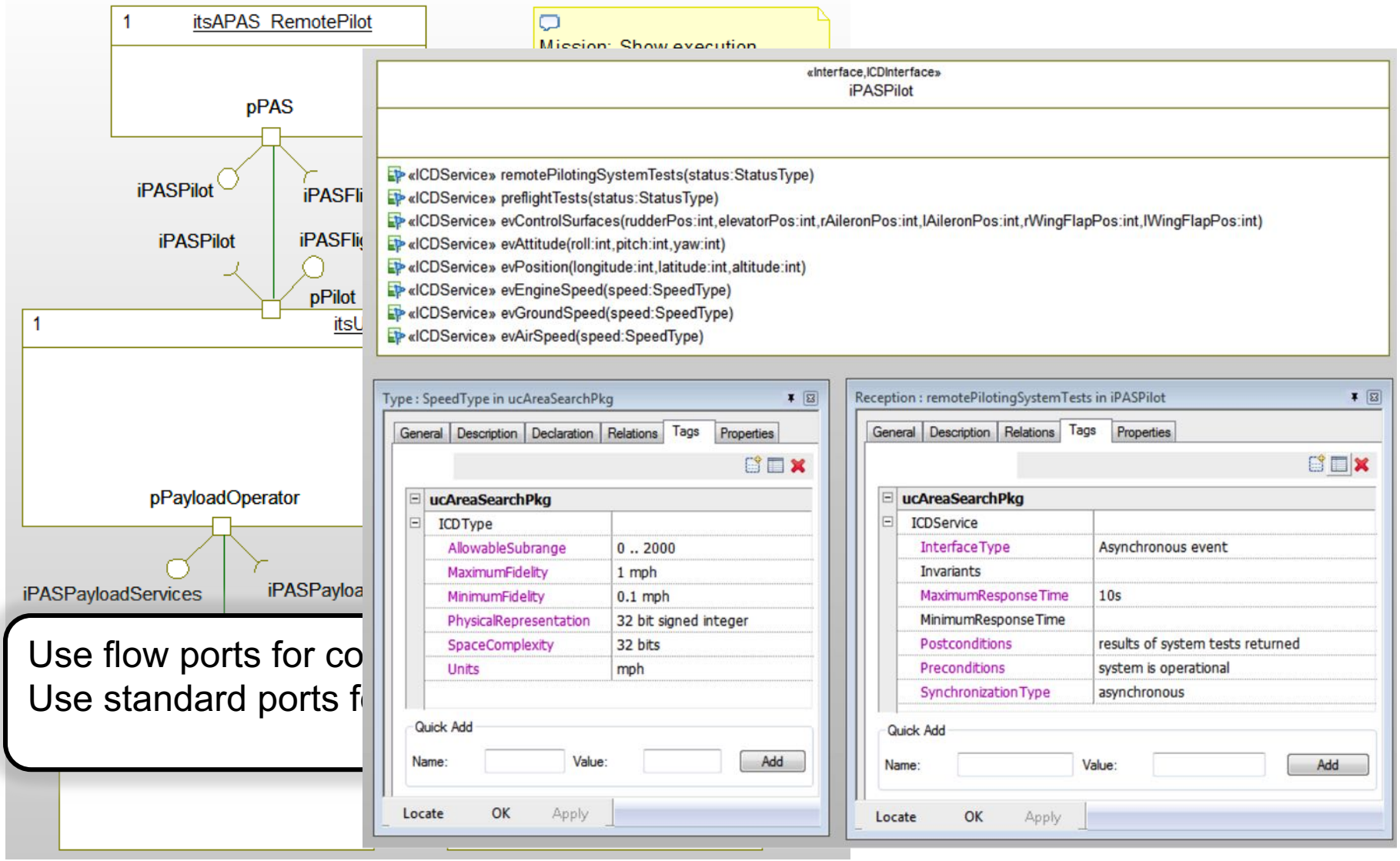


Capturing ICDs in the Model

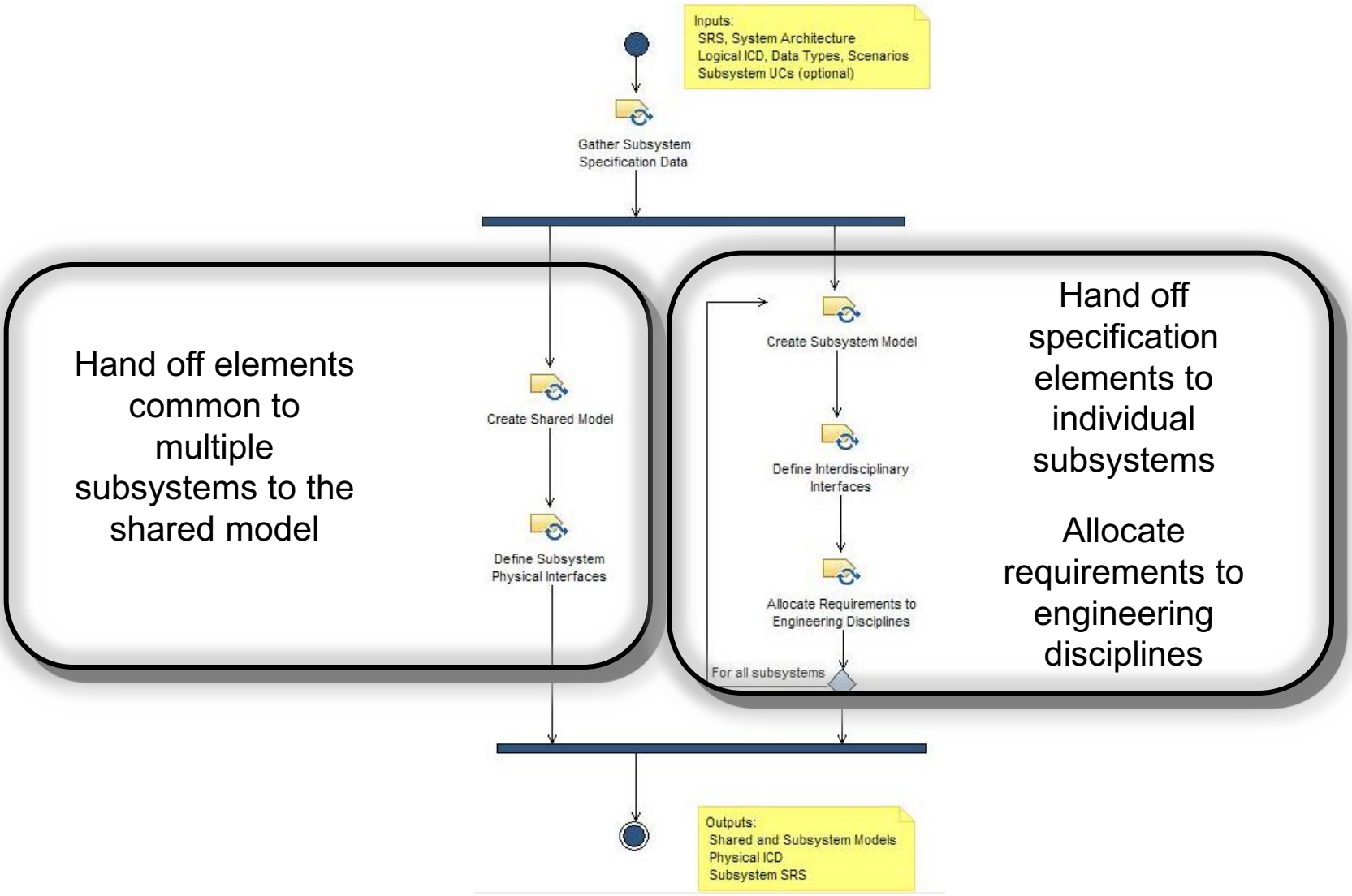
- ICDs are not just a list of services but include:
  - For each Service
    - Functional Description
    - Preconditions
    - Postconditions
    - Invariants
    - Performance
    - Error handling
    - Synchronization type
    - For each parameter
      - Description
      - Type
      - Units
      - Valid subrange
      - Default value
- This metadata can be easily added as tags defined in stereotypes



Capturing ICDs in the Model



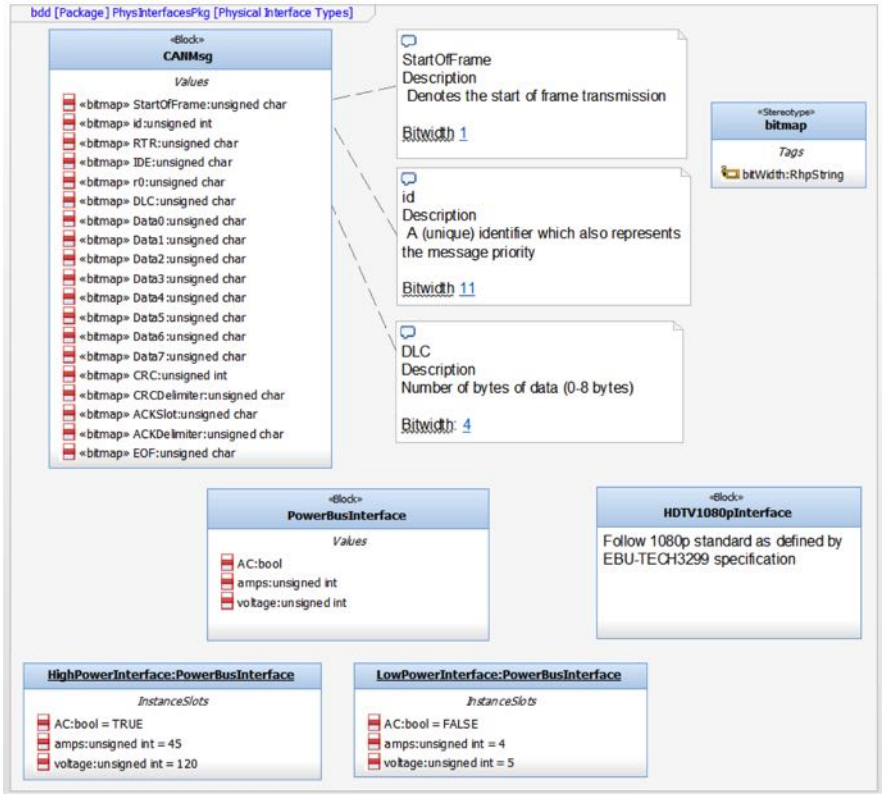
Hand off Workflow



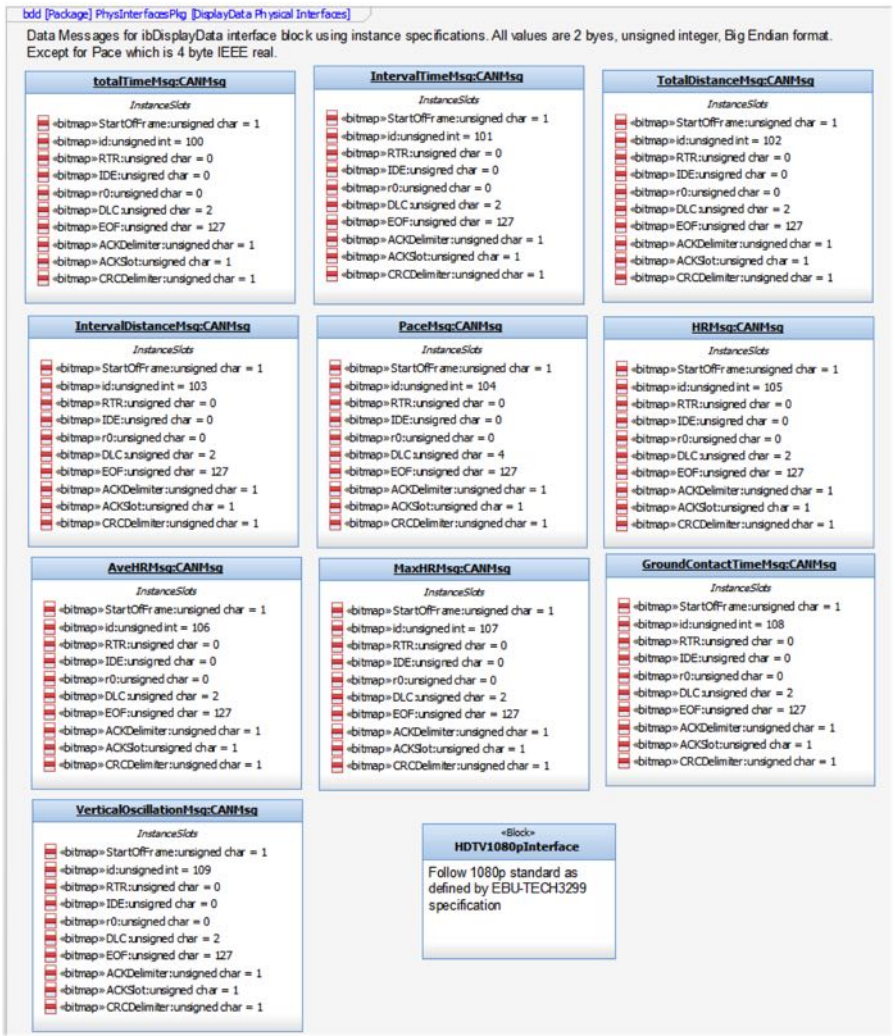
Convert Logical to Physical Interfaces

Logical Interfaces

SpeedDemon Physical Interfaces (using CAN™ Bus)

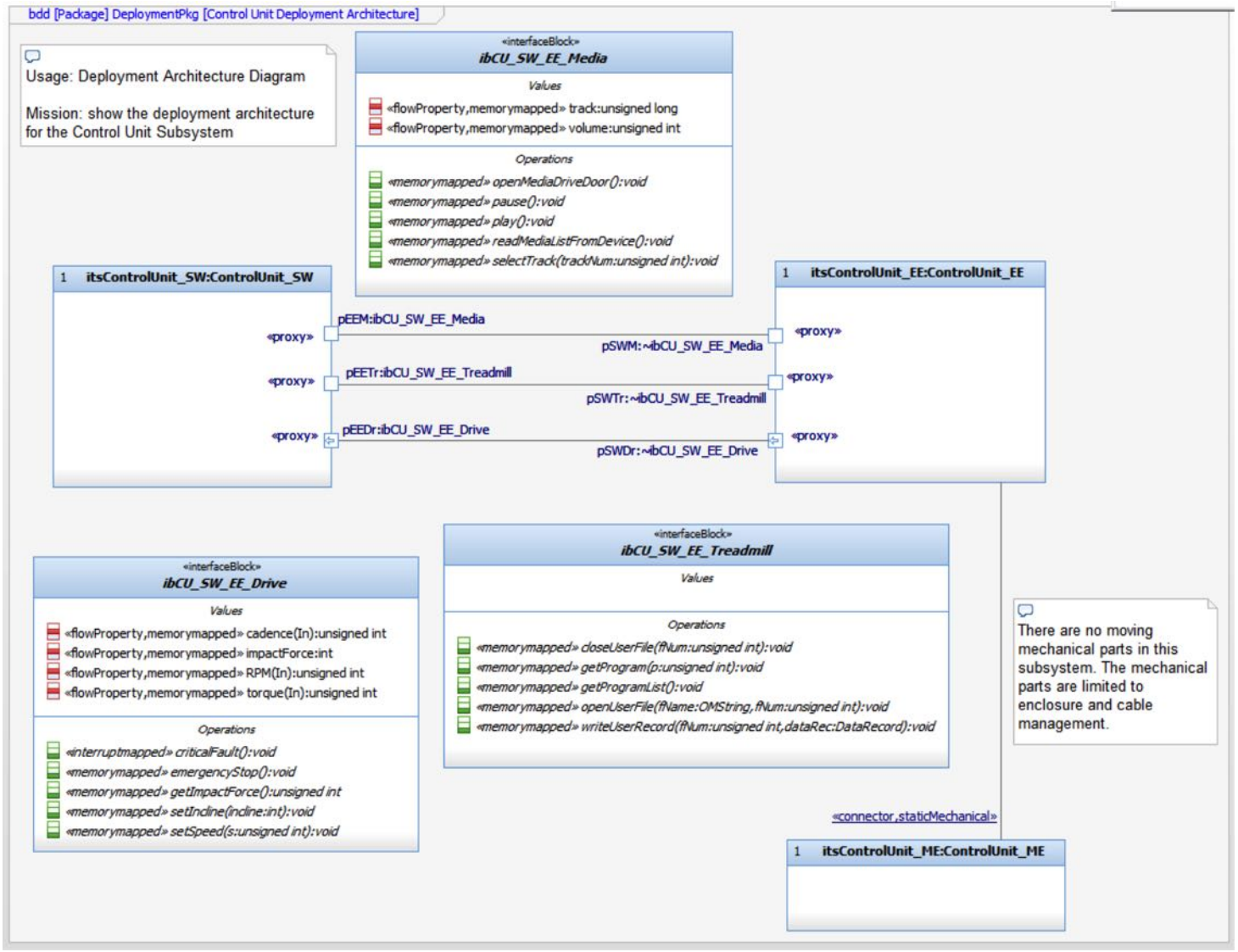


Interface structure specification

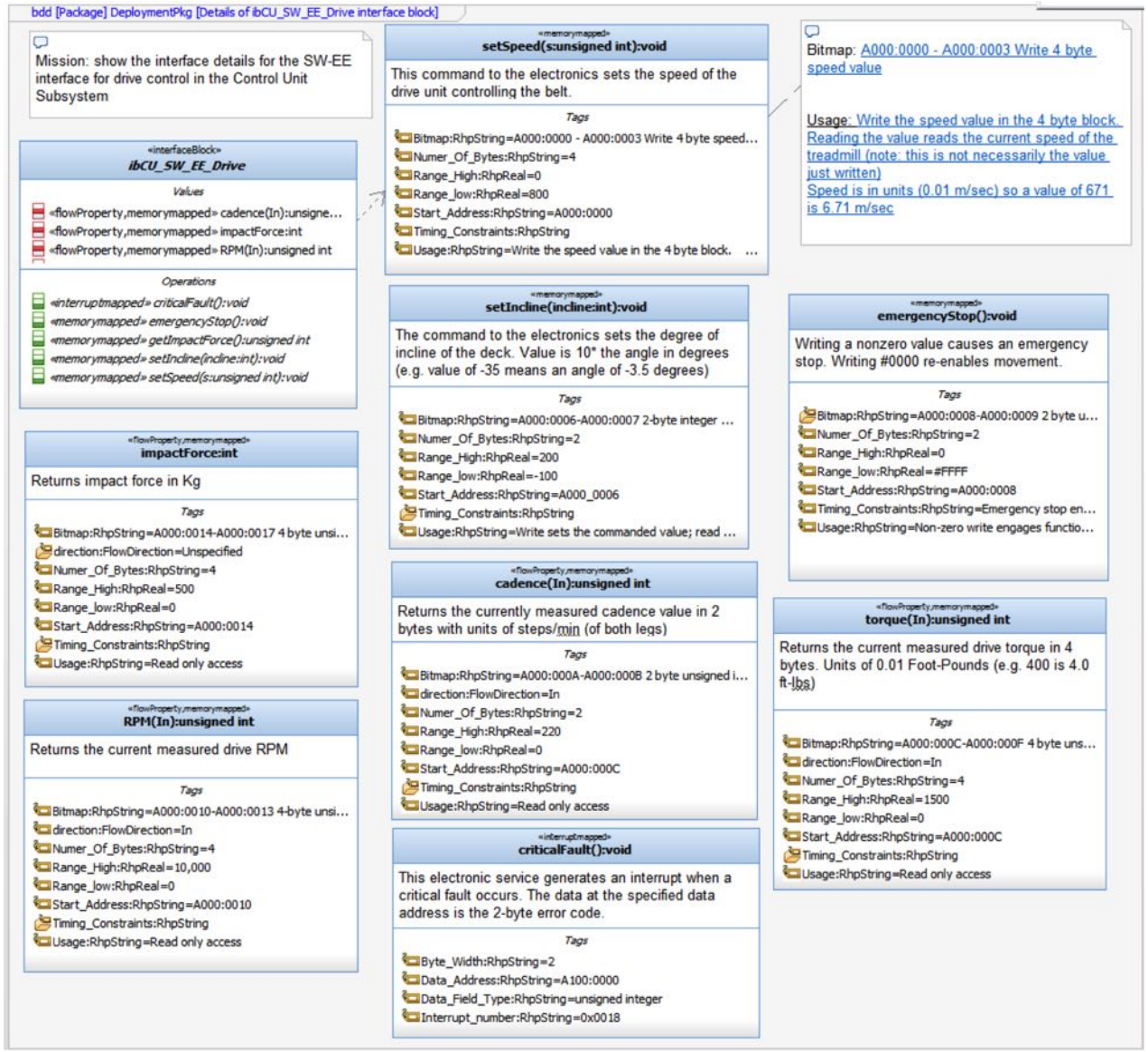


Instance specifications for messages

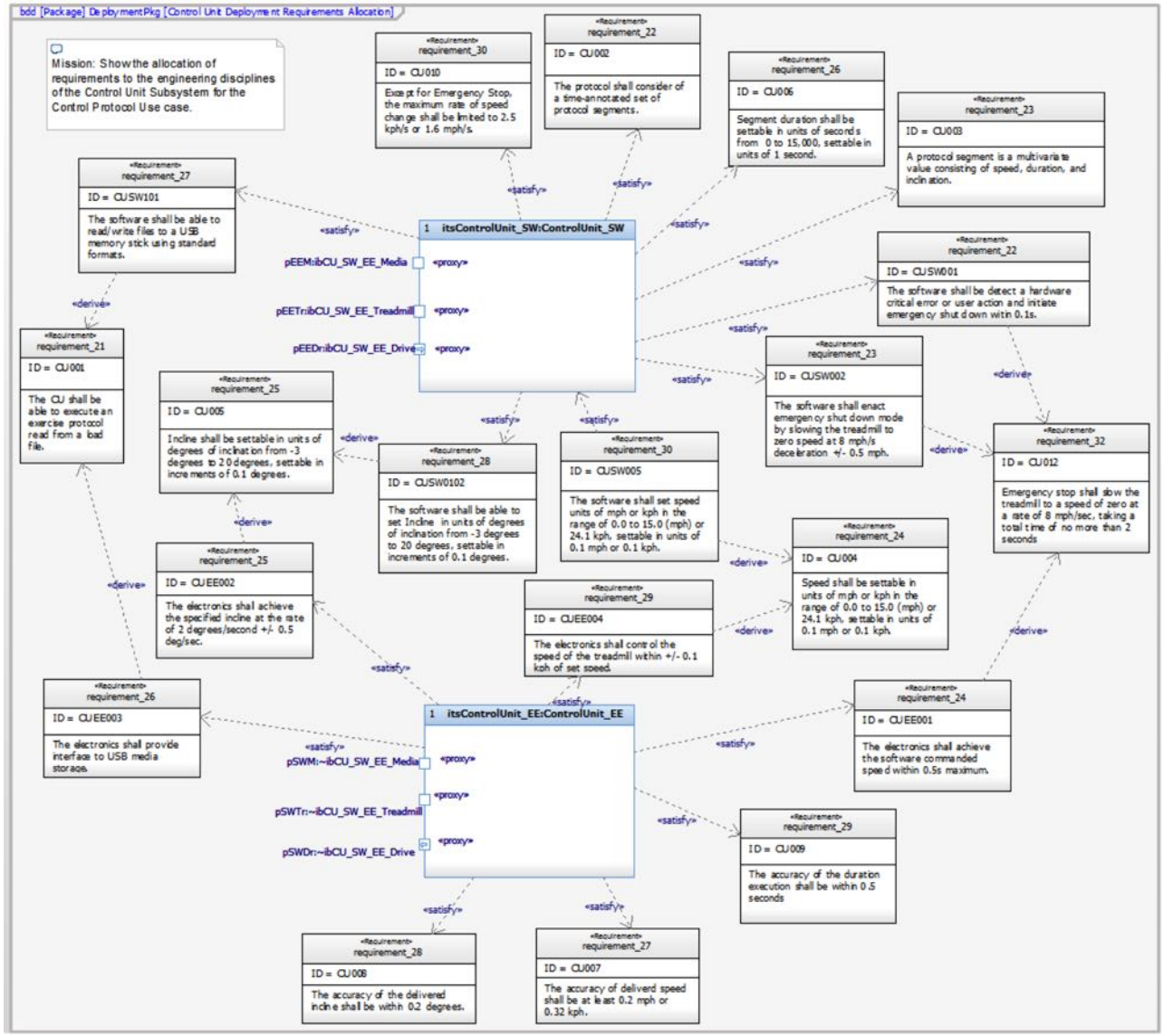
Subsystem Deployment Architecture (SW-EE-ME)



Instance Specs for SW-EE Interfaces in Control Unit

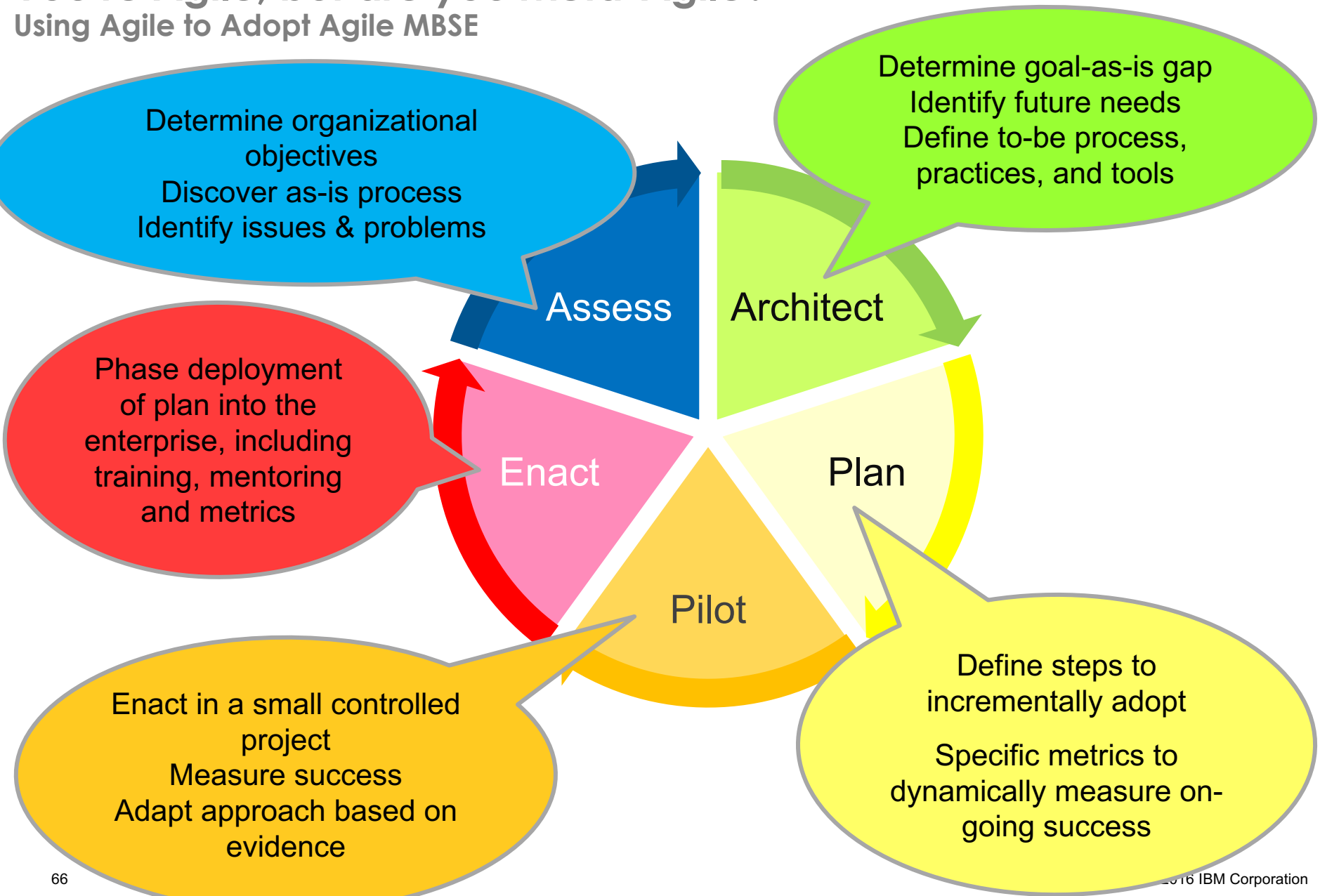


Allocation on Subsystem requirements to engineering disciplines



You're Agile, but are you Meta-Agile?

Using Agile to Adopt Agile MBSE



Summary

MBSE provides precision and verifiability to the SE Process

Agile methods add quality, responsiveness and adaptability to the process

Adopt Agile in an incremental, measured fashion for best results

Continuous verification allows you to avoid costly defects

IBM's Harmony Process defines an agile MBSE process with industry best practices

IBM's Rational tooling supports MBSE and agile methods

References