Hypermodeling: A Profile for Teaching SysML Modeling

Mr. Michael J. Vinarcik P.E., University of Detroit Mercy

Michael J. Vinarcik is a Chief Solutions Architect at SAIC and an adjunct professor at the University of Detroit Mercy. He has nearly thirty years of automotive and defense engineering experience. He received a BS (Metallurgical Engineering) from the Ohio State University, an MBA from the University of Michigan, and an MS (Product Development) from the University of Detroit Mercy. Michael has presented at National Defense Industrial Association, International Council on Systems Engineering, and American Society for Engineering Education regional and international conferences. He is a regular speaker at the No Magic World Symposium. Michael has contributed chapters to Industrial Applications of X-ray Diffraction, Taguchi's Quality Engineering Handbook, and Case Studies in System of Systems, Enterprise Systems, and Complex Systems Engineering; he also contributed a case study to the Systems Engineering Body of Knowledge (SEBoK). He is a licensed Professional Engineer (Michigan) and holds INCOSE ESEP-Acq, OCSMP: Model Builder – Advanced, Booz Allen Hamilton Systems Engineering Expert Belt, ASQ Certified Quality Engineer, and ASQ Certified Reliability Engineer certifications. He is a Fellow of the Engineering Society of Detroit. He is the President and Founder of Sigma Theta Mu, the systems honor society.

Hypermodeling: A Profile for Teaching SysML Modeling

Abstract

Competent execution of descriptive models in SysML, the system modeling language, facilitates rigor and analysis in support of systems architecture and engineering activities. However, this requires mastery of SysML, the selected modeling tool, and the method used. A semester-long course is not long enough to provide students with adequate time and experience to independently construct a high-quality model.

This paper details the content and use of the hypermodel profile, originally released by the author in 2017. It contains an organizational structure, stereotypes, queries, analysis aids, metrics, and quality checks that can be leveraged by students. Use of the profile allows students to focus on the intellectual content of their assignments while modeling in compliance to a provided style guide. It permits them to experience the benefits of automated quality checks, detailed inferential queries, and other modeling aids without having to have the advanced knowledge to construct them independently. This approach also exposes students to the full benefits of a sophisticated model and encourages them to explore and gain deeper insights into their system of interest.

The specifics of the hypermodel profile will be presented, including its organization, content, and customizations. Guidelines for its use will be presented in conjunction with lessons learned from its use at the University of Detroit Mercy in the Master of Science Product Development, Systems Engineering Certificate, and Advanced Electric Vehicle programs.

The Digital Transformation

Modern systems are increasingly complicated and complex; the number of components (and software elements) continues to grow. Document-Intensive Systems Engineering (DISE) is unable to keep pace with the need to keep stakeholders, program managers, design engineers, and other individuals informed about the consequences of their decisions and the current state of the system under development. This problem is particularly acute in military and aerospace development; a United States Air Force general recently stated, “Our current defense acquisition system applies industrial age processes to solve information age problems [1].”

Emergent behaviors (wanted, unwanted, and unanticipated) are particularly difficult to manage with traditional systems engineering approaches. Although functional decomposition and related deconstructive approaches are useful, they fail to fully manage interactions. As David Cohen, Director of Naval Air Systems Command’s Systems Engineering Department, recently stated: “We have been using Newtonian systems engineering. We need quantum or string theory systems engineering to manage modern system development [2].” Model-Based Systems Engineering (MBSE) is one solution to this need.

Descriptive system modeling, making descriptive information available in an evolving, organic system model (typically using SysML, the OMG Systems Modeling Language) is an important enabler to transform DISE into MBSE.

The Department of Defense released a Digital Engineering Strategy in June 2018; it outlines five goals:

1. Formalize the development, integration, and use of models to inform enterprise and program decision making
2. Provide an enduring, authoritative source of truth
3. Incorporate technological innovation to improve the engineering practice
4. Establish a supporting infrastructure and environment to perform activities, collaborate, and communicate across stakeholders
5. Transform the culture and workforce to adopt and support digital engineering across the life cycle [3]

These goals are foundational and applicable to any organization that seeks to develop complicated and complex systems. The development of hands-on system modeling exercises directly supports Goals 1, 2, 4, and 5 by:

- Showing students how to create descriptive system models that serve as the authoritative source of truth for a system under development [Goal 2]
- Demonstrating how models may be queried to answer relevant questions [Goal 1]
- Illustrating collaborative techniques using modeling tools and collaboration servers (such as No Magic's MagicDraw and TeamWork Cloud) [Goal 4]
- Showing the value of MBSE through student experience [Goal 5].

Hypermodeling

System modeling is inherently difficult; language, tool, and method must be mastered sufficiently to competently model. Because there is no simple visual analog to a system model (unlike the case of CAD, in which a solid model can be viewed and easily compared with the desired design), modelers must rely on a variety of secondary work products (validation suites, metrics, tables, and matrices) to judge the quality of their work.

In addition, "Every modeling effort has several factors that may be used to describe it:

η = Efficiency factor = output/input ($0 < \eta < 1$)

ε = Effectiveness factor = ability to accomplish intended outcome ($0 < \varepsilon < 1$)

φ = Elegance value ($0 < \varphi < 1$)

$$\eta\,\varepsilon = \varphi$$

Language, tool, and method each have their own contributions to this equation:

$$\eta_{\text{language}}\,\varepsilon_{\text{language}}\,\eta_{\text{tool}}\,\varepsilon_{\text{tool}}\,\eta_{\text{method}}\,\varepsilon_{\text{method}} = \varphi$$

Once the tool and language are selected, those terms are effectively constants…so any modeler is only able to directly influence $\eta_{\text{method}}\,\varepsilon_{\text{method}}$.

Therefore, productivity, effectiveness, and elegance depend heavily upon the methods used to construct the descriptive system model. One critical, inescapable fact is that every model element has a cost associated with its elicitation, creation, definition, and maintenance. Therefore, if a system can be described rigorously and completely with n elements, each $n + i$, where $i > 0$, element adds no value and only increases cost [4]."

Because of these considerations, it is critical to teach students to model economically from the start. If students' initial modeling instruction is diagram-centric (the "pictures") and ignores the reality that a competently-executed system model is model-centric (focused on elements, attributes, and relationships), they will need to overcome this intellectual deficit to become capable modelers.

Hypermodeling is an attempt to unify pragmatic modeling techniques supported by a shareable modeling profile that extends SysML. It includes stereotypes, customizations, enumerations, metrics suites, and validation suites. It is paired with a blank "stub" model that organizes modeling artifacts into a progression that supports systems architecture engineering processes. A reference model based on a next-generation Mars Orbiter (NeMO) was released for public use (and served as the basis for demonstration of a state-machine based interdiction of cybersecurity threats) [5].

Conceptual Framework

The hypermodeling approach builds on work previously published [6] and expands upon it by increasing the focus on state machines as the unifying behavioral construct and explicitly including analysis before requirements generation. Note that although needs/objectives/mission statements/etc. are considered critical up-front source content, the authoring of system and subsystem requirements is deferred until late in the process. The intent of hypermodeling is to use the descriptive model as analytical support to facilitate an entire spectrum of analysis without resorting to traditional "shall" statements. Those can be authored, if necessary, once appropriate sections of the system model are sufficiently mature.

Although Figure 1: Content Relationships shows a cascade, feedback loops and iterations are implicit to the hypermodeling approach. As new information is captured in the model, either through analysis or exploration of another level of abstraction, model elements, attributes, and relationships are updated to remain synchronized.

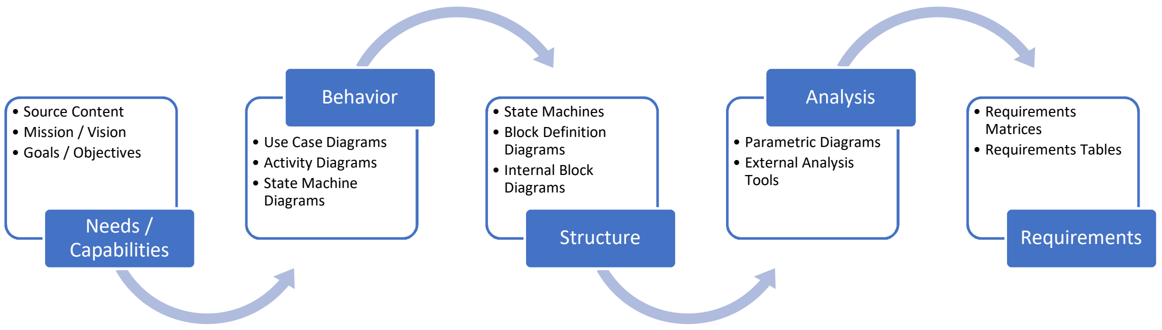

Figure 1: Content Relationships

Basic Structure

The structure of the stub hypermodel consists of the following packages:

- 00 Source Content
- 10 Behavioral Analysis
- 20 Context
- 30 Functional Architecture
- 40 Logical Architecture
- 50 Physical Architecture
- 60 Verification
- 70 Analysis
- 90 Tables and Matrices
- QC Quality Checks

00 Source Content is intended to contain packages for relevant source content, such as Goals and Objectives, Mission Statement, Regulatory Requirements, and other source material. It is listed first in the containment tree as a matter of convenience and to subtly emphasize that it is authoritative information or guidance imposed on the system of interest.

10 Behavioral Analysis contains high-level analysis of goals, objectives, capabilities, and use cases. As a matter of convenience, use cases, use case elements, and use case diagrams are collected in separate subordinate packages. This facilitates reuse of elements and prevents duplication (a common rookie modeling mistake).

20 Context contains contextual elements (typically *external* and *domain* blocks). There is one and only one *system context* block (named "[*system of interest*] Context"); it owns part properties typed by the contextual elements. Each part property must be connected to a use case element (such as an *actor, boundary system, or environmental effect*) by a <<trace>> relationship. These relationships are used as the basis of a quality check in the validation suite to ensure that all context elements represent use case elements and vice versa. Although the use case elements could be used to directly type the part properties of the *system context* block, the author finds it convenient to use <<traced>> blocks instead (since they can own ports and be used to fully describe the interfaces between them).

30 Functional Architecture is a package to contain the functional architecture of the system, if desired. The author often omits this step because, in his experience, purely functional decomposition is often more of an academic exercise than a useful one. If one is aware that functions can be re-assigned to different logical or physical elements as needed, a purely functional decomposition is unnecessary. If a functional architecture is constructed, the author recommends that *activities* be used as basis for collecting *operations* (his preferred atomic behavioral element). *Operations* must be owned by either *blocks* or *activities*; using a *block* implies a specific architectural element whereas using an *activity* makes it clear that is a purely behavioral element. Because operations own *parameters* that are typed by *signals*, they allow the rigorous capture of inputs and outputs for each function.

40 Logical Architecture contains one or more logical architectures (made up of *blocks* with the <<logical>> stereotype applied). Each should be organized in a sub-package. Logical *blocks* own *operations*, *value properties*, and *ports* (the author prefers the exclusive use of *proxy ports*). Note that *value properties* at this level should be related to performance or measures of effectiveness and not physical properties (which should only be present in the physical architecture). If a functional architecture has been constructed, *operations* in the logical architecture use <<realization>> relationships to provide traceability to the related functional architecture *operation*.

50 Physical Architecture contains one or more physical architectures (made up of *blocks* with the <<physical>> stereotype applied). Each should be organized in a sub-package. Physical *blocks* own *operations*, *value properties*, and *ports* (typed by *proxy ports*). *Operations* and *ports* in the physical architecture are connected to their logical architecture analogues with <<realization>> relationships. This allows another quality check to ensure consistency between layers of abstraction. By having each architectural element own its *operations* and *ports*, this approach allows tailoring (eliminating or adding *parameters* to an *operation*, for example). It also permits the creation of variant-specific activity diagrams, state machines, sequence diagrams, and internal block diagrams.

60 Verification is a location to store *test cases* that <<verify>> requirements. A test architecture can also be stored here (by creating *blocks* to own the *operations* called on *test case* activity diagrams). Alternatively, the operations can be owned by the *test case* activities themselves.

70 Analysis is a catch-all package for collecting any ancillary analysis, parametric diagrams, or model elements used to interface with modeling and simulation tools.

90 Tables and Matrices is provided as a convenient storage location for tables and matrices that make up the bulk of the derived work products. Because blocks can also own tables and matrices, it is usually beneficial to allow context-specific products to reside with the block that described them (for example, a table listing *ports* owned by a given subsystem should be owned by that *subsystem* block).

QC Quality Checks is a repository for tables and matrices used for quality checks (e.g., *parameters* unable to flow over interfaces because the *block* owning them lacks a suitable *port*). These may overlap with the validation suite rules, but the author finds tabular representations are often useful additions to the model.

Ontology

Figure 2 and Figure 3 illustrate the allowed relationships between source content elements. <<trace>> relationships are the primary relationship used; <<include>> relationships connect *goals, objectives, sub-objectives,* and *investigations*. Proper use of these relationships enables inferential querying, metrics suites, and validation rules to operate correctly.

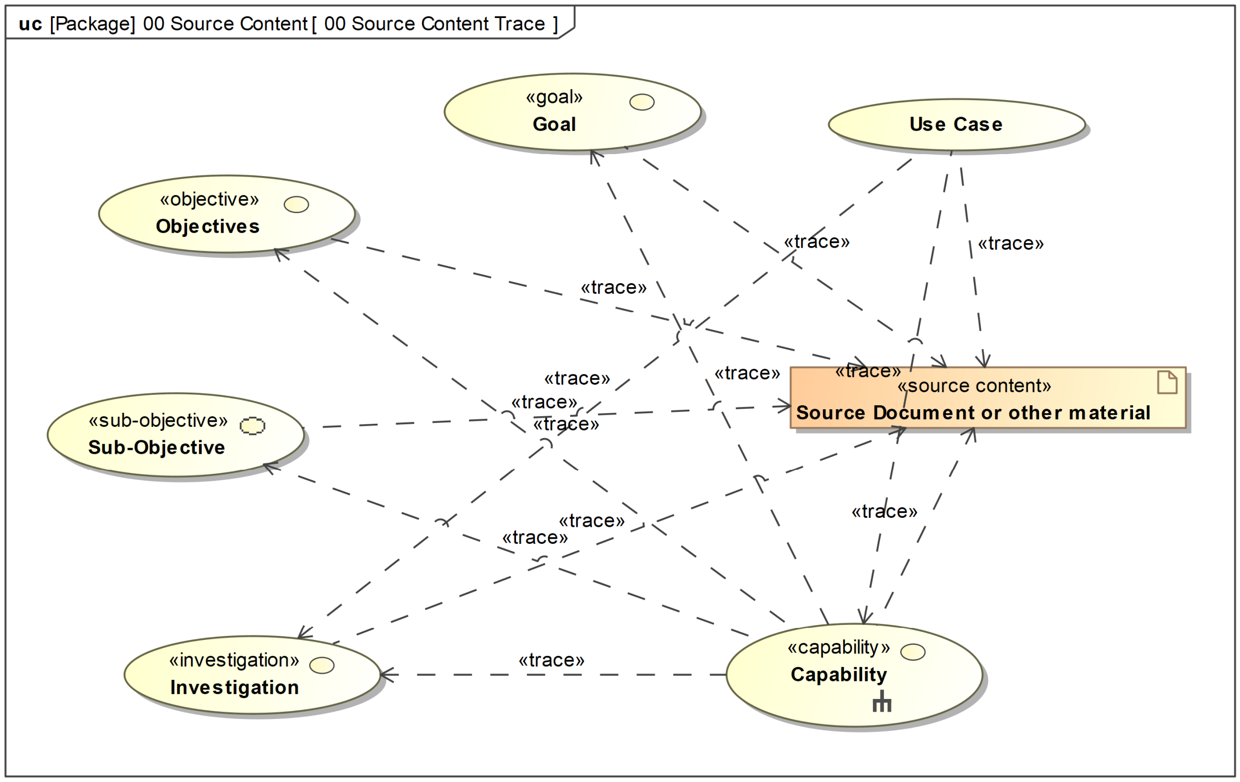

Figure 2: Source Content <<trace>> Relationships

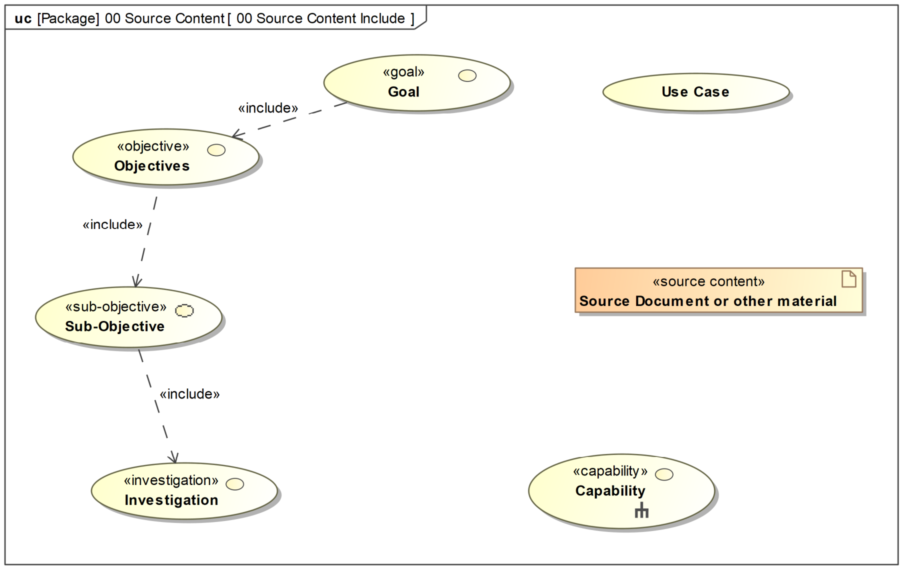

Figure 3: Source Content <<include>> Relationships

Figure 4 and Figure 5 illustrate the allowed relationships between behavioral elements. Note that if any *goal, objective, sub-objective, investigation, capability,* or *use case* elements are further decomposed with activity diagrams, every node on the activity diagram must be a *call operation*.

This eliminates the need for swimlanes, since every *operation* has an unambiguous owner; it also allows rigorous elicitation of inputs and outputs for each function.

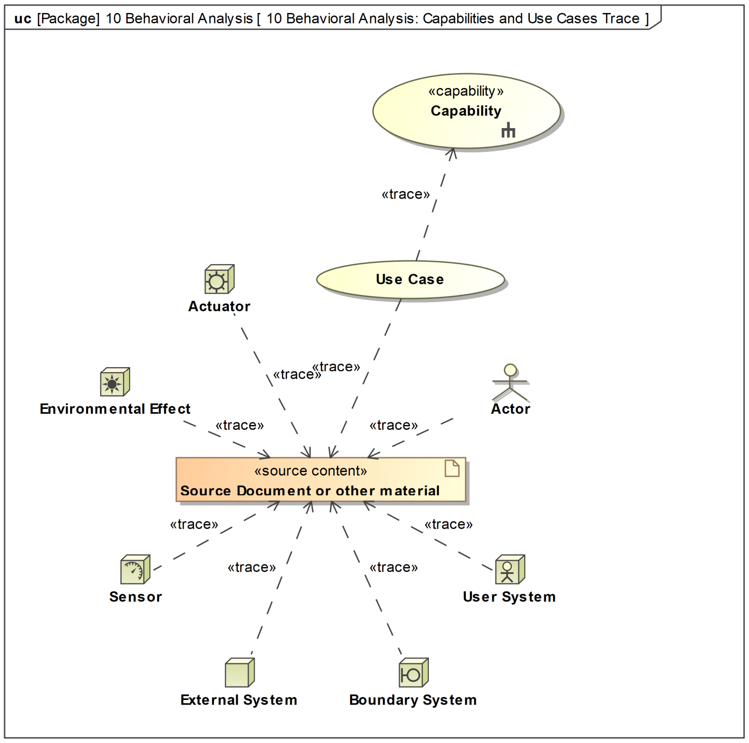

Figure 4: Behavioral Analysis <<trace>> Relationships

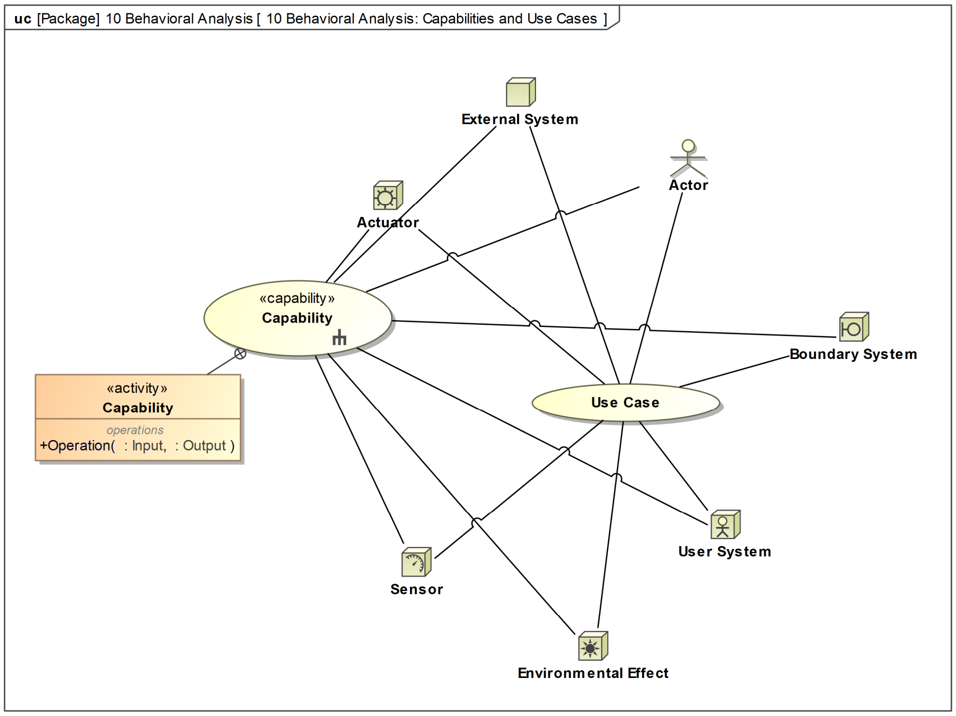

Figure 5: Behavioral Analysis Relationships

Figure 6 shows the <<trace>> relationships between context elements and behavioral analysis elements.

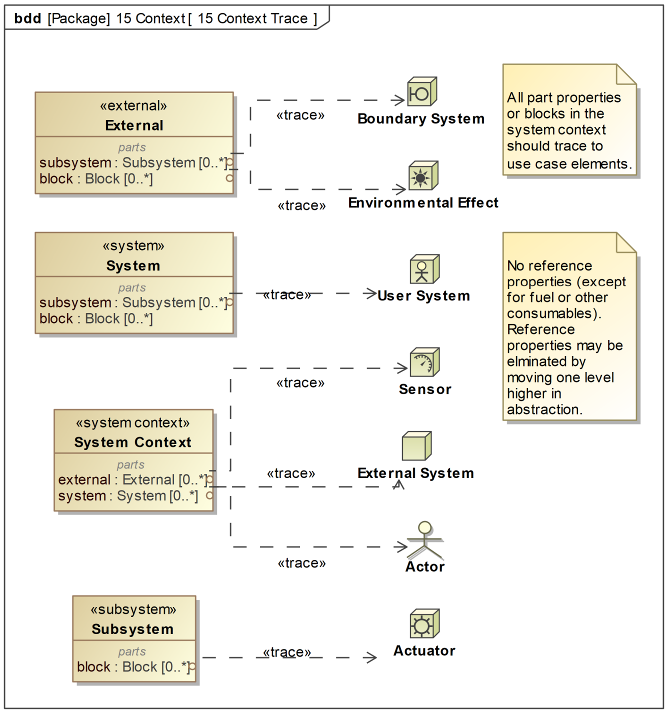
bdd [Package] 15 Context[15 Context Trace]
«external»
External
parts
subsystem : Subsystem [0..*]
block : Block [0..*]
«trace»
Boundary System
«trace»
Environmental Effect
All part properties or blocks in the system context should trace to use case elements.
«system»
System
parts
subsystem : Subsystem [0..*]
block : Block [0..*]
«trace»
User System
No reference properties (except for fuel or other consumables). Reference properties may be elminated by moving one level higher in abstraction.
«trace»
Sensor
«system context»
System Context
parts
external : External [0..*]
system : System [0..*]
«trace»
External System
«trace»
Actor
«subsystem»
Subsystem
parts
block : Block [0..*]
«trace»
Actuator

Figure 6: Context <<trace>> Relationships

Figure 7 illustrates the decomposition of an *activity* in the Functional Architecture and how it appears in the model containment tree.

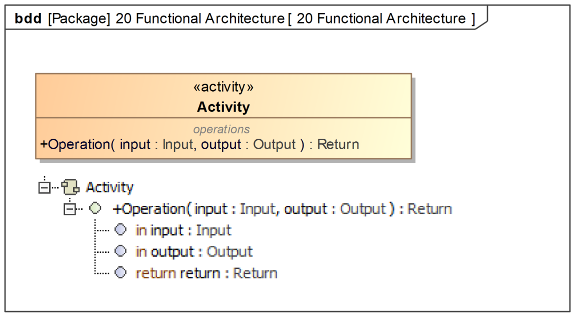
bdd [Package] 20 Functional Architecture [20 Functional Architecture]
«activity»
Activity
operations
+Operation(input : Input, output : Output) : Return
Activity
+Operation(input : Input, output : Output) : Return
in input : Input
in output : Output
return return : Return

Figure 7: Functional Architecture

Figure 8 illustrates the structure of the Logical and Physical Architectures; note that the use of *reference properties* is discouraged. The author has found that any *reference property* can be illustrated by a *part property* at a higher level in the architecture (or context) structure. This eliminated ambiguity and reduces the number of element types in use (slightly easing the cognitive burden on the student modeler).

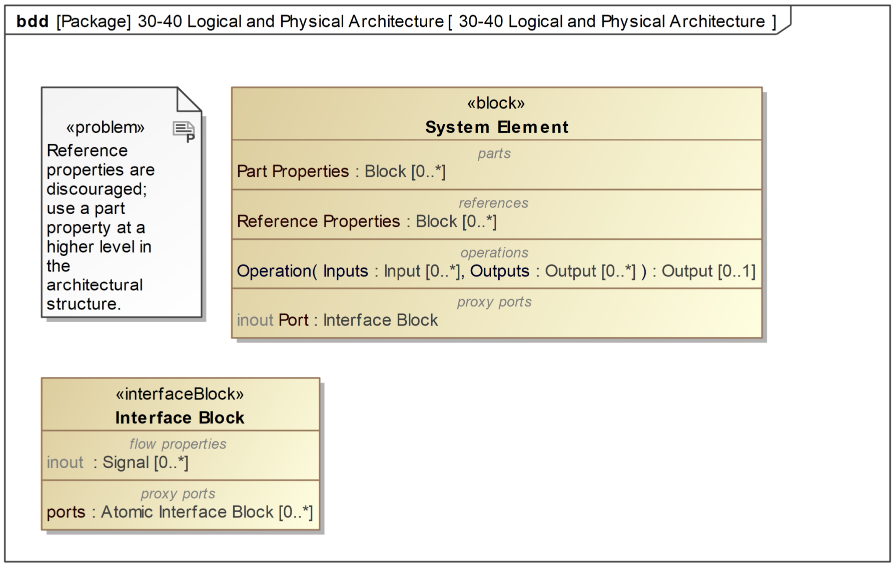
bdd [Package] 30-40 Logical and Physical Architecture [30-40 Logical and Physical Architecture]
«problem»
Reference properties are discouraged; use a part property at a higher level in the architectural structure.
«block»
System Element
parts
Part Properties : Block [0..*]
references
Reference Properties : Block [0..*]
operations
Operation(Inputs : Input [0..*], Outputs : Output [0..*]) : Output [0..1]
proxy ports
inout Port : Interface Block
«interfaceBlock»
Interface Block
flow properties
inout : Signal [0..*]
proxy ports
ports : Atomic Interface Block [0..*]

Figure 8: Logical and Physical Architecture

The allowed relationships for requirements are shown in Figure 9. Note that the <<satisfy>> relationship is used instead of <<allocate>>. This forces the creation of specific model elements that make the relationship true and facilitates automatic requirements verification. It also aids in the detection of redundancy (consider the relative ease of detecting two conflicting <<satisfied>> by the same mass *value property* versus the difficulty in finding them in five hundred requirements <<allocated>> to the same *block*). Also, the ends of <<satisfy>> requirements are intentionally limited by SysML (e.g., functional requirements must be <<satisfied>> by an *activity* or an *operation*…they cannot be <<satisfied>> by a block). This promotes a crispness in requirements formulation and fosters singularity and precision.

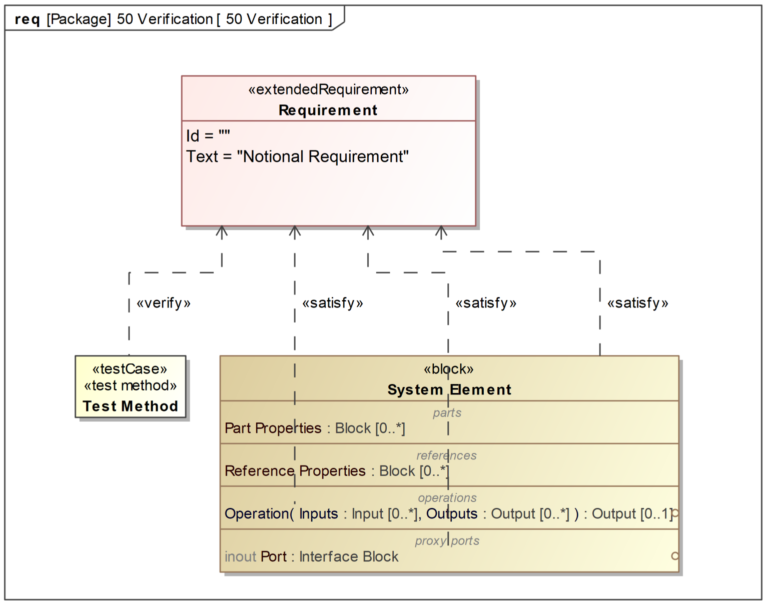
req [Package] 50 Verification [50 Verification]
«extendedRequirement»
Requirement
Id = ""
Text = "Notional Requirement"
«verify»
«satisfy»
«satisfy»
«satisfy»
«testCase»
«test method»
Test Method
«block»
System Element
parts
Part Properties : Block [0..*]
references
Reference Properties : Block [0..*]
operations
Operation(Inputs : Input [0..*], Outputs : Output [0..*]) : Output [0..1]
proxy ports
inout Port : Interface Block

Figure 9: Verification

The author found that despite providing quality check tables (that contained elements that violated a style guide rule), some students had difficulties in using them to improve model quality. The introduction of an on-demand validation suite (See Table 1) significantly improved project quality (no projects have been submitted with violations since its introduction). It allows the codification of the quality checks and when the validation engine processes the rules, violating elements are identified within the model. It is relatively simple for modelers to then correct the issues (receiving satisfying feedback as the element's red highlighting vanishes once the error is corrected).

Name	**Constrained Element**	**Error Message**	**Severity**
CAPDOC	capability [UseCase]	All capabilities must be documented.	error
CAPTRACE	capability [UseCase]	All capabilities must be traced to some authoritative source element.	error
CONNECTPIN	InputPin OutputPin	Pins must be connected	error
EDGEGUARD	ActivityEdge	All activity edges that exit a decision node must have guards defined.	error
GOALDOC	goal [UseCase]	All goals must be documented.	error
GOALTRACE	goal [UseCase]	All goals must be traced to source content	error
IBFLOW	InterfaceBlock [Class]	Interface blocks must own a flow property.	error
INVDOC	investigation [UseCase]	All investigations must be documented.	error
INVTRACE	investigation [UseCase]	All investigations must be <<included>> by an objective or sub-objective.	error
NOALLOC	Allocate [Abstraction]	Allocation relationships are forbidden.	error
OBJDOC	objective [UseCase]	All objectives must be documented.	error
OBJTRACE	objective [UseCase]	All objectives must be <<included>> by a goal.	error
SUBDOC	sub-objective [UseCase]	All sub-objectives must be documented.	error
SUBTRACE	sub-objective [UseCase]	All objectives must be <<included>> by an objective.	error
TRIGGERS	Transition	All transitions (except those from history or initial nodes) must have triggers.	error
TYPEFLOW	FlowProperty [Property]	Flow properties must be typed.	error
TYPEPAR	Parameter	All parameters must be typed.	error
TYPEPART	PartProperty [Property]	Part Properties must be typed.	error
TYPEPORT	ProxyPort [Port]	Proxy Ports must be typed by interface blocks.	error

Name	Constrained Element	Error Message	Severity
TYPEVAL	ValueProperty [Property]	Part Properties must be typed.	error
UCDOC	UseCase	Use Cases must be documented.	error
UCTRACE	Actor	All Use Case Elements must be traced from the context.	error

Table 1: Validation Suite

By providing a comprehensive set of stereotypes (see Table 2), the hypermodel profile enables students to focus on modeling the system of interest rather than creating customizations. Some individual exercises require them to make personalized customizations; the use of the standard profile simplifies the collaborative term projects models.

Name	Metaclass	Definition
cable	Class	This stereotype is used to create cables (1..* connectors and 0..* ports to describe connections).
capability	UseCase	Stereotype applied to use cases to indicate they are capabilities.
CIS control	Class	Stereotype applied to cybersecurity controls from the Center for Internet Security.
conceptual	NamedElement	Stereotype applied to elements of the conceptual architecture.
context	NamedElement	Stereotype applied to context elements.
dissociation	Association	Stereotype used to dissociate elements.
exception	Extend	Stereotype applied to extend relationships associated with "rainy day" scenarios.
goal	UseCase	Stereotype applied to goals of the MEPAG report.
hazard	UseCase	This stereotype is applied to hazard that represent undesired safety-related outcomes.
immutable	NamedElement	This stereotype is applied to model elements that cannot be modified. The documentation should be used to indicate the rationale.
inheritance	NamedElement	Stereotype applied to elements used to inherit relationships.
investigation	UseCase	Stereotype applied to investigations of the MEPAG report.
line contact	Connector	Stereotype for line contact physical interfaces.
logical	NamedElement	Stereotype applied to elements of the logical architecture.
mission	NamedElement	Stereotype applied to elements used to define a NeMO mission.
MPD	NamedElement	Stereotype used to assign students to given model elements.
objective	UseCase	Stereotype applied to objectives of the MEPAG report.
physical	NamedElement	Stereotype applied to elements of the physical architecture.
pinout	Class	Stereotype applied to the pinout-level interface blocks.
power scenario	UseCase	Stereotype applied to use cases that represent power scenarios

Name	**Metaclass**	**Definition**
power usage	Usage	This stereotype is applied to usages that describe power scenarios.
reference	NamedElement	Stereotype applied to reference elements.
social	NamedElement	Stereotype applied to social elements of a system.
software	Class	This stereotype is applied to a block that represents a software element in the system.
source content	Artifact	Stereotype applied to reference elements (such as source documents, mission statements, needs statements).
sub-objective	UseCase	Stereotype applied to objectives of the MEPAG report.
test method	Activity	Stereotype applied to activities to define test methods.
threaded fastener	Connector	Stereotype applied to connectors that describe physical threaded fasteners.
variant	NamedElement	Stereotype applied to elements used to describe variant architectures.

Table 2: Hypermodel Stereotypes

Metrics suites allow periodic creation of model metrics (e.g., weekly to show model maturation) (see Table 3). Their use will be expanded as the hypermodel profile matures. For example, the count of untraced elements will be eliminated (since that is now an error identified by the validation suite and the expectation is that it will always be zero).

Name	**Owner**
Untraced Use Case Elements	Behavioral Analysis Metrics Suite
Use Case Elements	Behavioral Analysis Metrics Suite
Use Cases	Behavioral Analysis Metrics Suite
Use Cases Without Behavior	Behavioral Analysis Metrics Suite
Logical Blocks	Logical Architecture Metrics Suite
Logical Connectors	Logical Architecture Metrics Suite
Logical Operations	Logical Architecture Metrics Suite
Logical Part Properties	Logical Architecture Metrics Suite
Logical Ports	Logical Architecture Metrics Suite
Unrealized Logical Connectors	Logical Architecture Metrics Suite
Unrealized Logical Operations	Logical Architecture Metrics Suite
Untyped Logical Part Properties	Logical Architecture Metrics Suite
Untyped Logical Ports	Logical Architecture Metrics Suite

Table 3 Hypermodel Metrics Suites

Notable Customizations

The profile provides a stereotype for software (which is modeled identically to hardware, with *blocks, operations, ports, and part properties*). The *exception* stereotype is applied to <<extend>> relationships to identify *use cases* as “rainy day” scenarios.

Power scenarios and <<power usage>> relationships are used to illustrate the use of a model in analysis. For example, a considerable effort is required to analyze power consumption with a fully-executable model; as elements change state, power consumption increases or decreases. A simple, direct approach is to identify power scenarios and associate with them the *part properties* that consume power under those conditions. The customization for *power scenario* contains a script that adds up the power consumption of all *part properties* that are connected to it with <<power usage>> relationships. The <<physical>> blocks typing each *part property* contain the power demand information and the script respects multiplicities to calculate the total power demand. Each <<power usage>> has a usage factor property (default value = 1) to allow individual usage-based tailoring (for example, a low-power mode). The required relationships can be rapidly made in a dependency matrix; the results of the analysis are displayed in tabular form (such as the percentage of demand for a given scenario for each *part property*). This example also shows the value of single authoritative sources of truth (in this case, the power demand of a block); if the power demand is changed, all impacted scenarios update immediately.

Guidelines for Use

The Hypermodel Profile and stub model are provided at http://hypermodeling.systems. The author suggests the following:

Establish a TeamWork Cloud (or comparable server for non-MagicDraw tools) and use it to support modeling classes. Setup and administration of this environment is beyond the scope of this paper. Using the collaboration server minimizes the need for students to manage local files, enables the instructor to actively correct issues during help sessions, and provides valuable practice at collaboration.

Place the stub model and profile on the server and share them with students. As of version 19.0, users can "clone" models; by having students clone their own copy of the stub model, administrative burden on the instructor is reduced.

Instruct the students to update the model from the server upon each open (to ensure they have any changes committed by collaborators) and to run the validation suite as the last step before committing. This instills in them good modeling practices and fosters pride in maintaining a "zero error" model.

Instructors may choose structure-first (*block/part property/port/operation*) or behavior-first (*use case/activity/operation*) organization to their curricula. Students may have an easier time understanding architectural decomposition because they can see example systems; behavioral analysis (particularly when coupled with the need to create *operations* to *call* on activity diagrams) can be a greater cognitive leap. Because the hypermodel approach is capability/behavior-first, if a structure-first approach is used any hypermodel projects should be deferred until appropriate behavior instruction has occurred. Note that the validation suite is independent of instructional sequence (it checks elements that have been created) and that the metrics suites are separated into behavioral and structural suites.

The most fruitful instructional method is to create five-to-ten-minute instructional videos showing tool use and method application. Because system modeling is tool dependent, the use of PowerPoint slides and static images is of limited utility. Live demonstrations are equally problematic, not just because of the possibility of error but also because recorded lectures are often run-on. It is hard for students to locate specific content in a lengthy lecture recording; a playlist of short task-based videos is far easier to review as needed. The author has experimented with several recording and editing programs; Camtasia, by TechSmith, has proven to be the simplest to use. Its productivity aids include easy pan-and-zoom, transitions, annotations, keystroke display, and archival of source files.

The author has posted more than 100 videos to YouTube (more than 60 dedicated to hypermodeling) at http://videos.systemsarchitectureguild.org. Instructors are invited to leverage this body of work in their instructional efforts. Many of these show non-obvious techniques borne of the author's experience as a professional, full-time modeler. For example, the "ferret table" to identify all usages of various elements is particularly useful when refactoring a model.

Whatever approach is taken, it is imperative that the instructor maintain a positive demeanor. No matter the modeling challenge, it is still easier than trying to find information in a stack of documents, disconnected databases, Excel sheets, and analytical models. Showing students that solving a modeling problem elegantly is no different than other engineering challenges and they must be confident that they can find a way to express engineering information in the modeling language. It is also critical to infuse a sense of curiosity into students: what queries can they run to answer important questions related to thought excursions? What tables, matrices, diagrams, and other derived work products can help them explore the system and support analysis? The Q.E.D. mnemonic is particularly useful to help them frame modeling investigations:

- What is the *Question* we need to answer?
- How can we *Extract* it from the model?
- How should we *Display* it to stakeholders in a meaningful, easy to consume way [4]

Future Work

In order to quantitatively assess the improvements related to the use of the hypermodeling profile, a term-long system modeling project created before hypermodeling will be reused with a new group of students. They will be given the same assignment and both system models will be assessed with the same metrics and validation suites. This will enable a quantitative assessment of the model quality improvements related to the profile and the improved instructional videos.

The primary focus of hypermodeling has been to improve the quality of students' descriptive models; a near-term objective is to introduce the use of simulation and analysis to allow automated requirements verification and trade studies. This will require more emphasis on *constraints* and parametric diagrams and will be coordinated with instructor of the Systems Optimization course.

Conclusion

Modeling is hard work and requires mastery of language, tool, method, and often, a new way of seeing problems and systems. Sharing a customized profile with students reduces the barrier to entry for novice modelers, allows more time to focus on intermediate and advanced concepts, permits easier collaborative modeling, and demonstrates the value of reuse. The hypermodel profile was created to address this need and will undergo continued development as the author's modeling approach and skills mature.

Bibliography

[1] D. Cohen, "SE Transformation - "Shaping our Future…"," in *NASA Jet Propulsion Laboratory MBSE Symposium*, Torrance, 2019.

[2] Office of the Deputy Assistant Secretary of Defense for Systems Engineering, "Department of Defense Digital Engineering Strategy," Department of Defense, Washington, 2018.

[3] L. R. D. McMurray, AFLCMC/CC, *Keynote address,* Dayton: 2017 Wright Dialogue With Industry Conference, 2017.

[4] M. J. Vinarcik, "The NeMO Orbiter: A Demonstration Hypermodel," in *Ground Vehicle Systems Engineering and Technology Symposium*, Novi, 2018.

[5] M. J. Vinarcik, "Interdiction: The Application of SysML State Machines to Cybersecurity," in *National Defense Industrial Association Systems Engineering Conference,* Tampa, 2018.

[6] M. J. Vinarcik, "A Pragmatic Approach to Teaching Model Based Systems Engineering: The PRZ-1," in *American Society for Engineering Education Annual Conference,* Columbus, 2017.

Appendix: Hypermodel Style Guide

Use Cases:

- Behavioral sketchpad to show behaviors/capabilities.
- <<capability>> stereotype applied to capabilities
- <<extend>> *use cases* are triggered by *extension points*
- <<include>> *use cases* are always executed by the *use case* to which they are connected
- May be more fully described by activity diagrams
- <<dissociation>> relationships used to exclude inherited relationships
- <<exception>> relationships used to capture "rainy day" *use cases*
- Specialized by other use cases realized by variants (provides a basis for variant-specific activity diagrams)

Activity Diagrams:

- Flowcharts of behavior; describe *activities* that are made up of *actions*
- *Call behavior* actions execute other activities (activity diagrams)
- *Call operation* actions execute "leaf node" functions owned by functional (*activities*), logical (*blocks*), or physical (*blocks*) elements (the smallest behaviors we will model)
- *Send* and *accept* event actions model messages flowing into/out of activities and may be assigned to *ports*
- Complicated logical behaviors may be modeled (decision nodes, forking, etc.)

Functional Architecture (optional):

- Composed of *activities* that own *operations*
- They are only used as containers for *operations*
- They should be organized so that most operations within a given *activity* are realized by a logical block (for example, a collection of testing/status/heartbeat functions that always are performed by a subsystem)
- These may be omitted if it is more appropriate to begin modeling at the logical level

Operations:

- Model elements that MUST be owned by a *block* or *activity*
- May own *in, out*, or *result* parameters
- *Parameters* are typed by *signals*
- *Parameters* may have multiplicities

Signals:

- Are used to type *parameters, information flows, item flows, flow properties,* and *send* or *accept* events
- Can own *attributes* that include other signals

Logical Blocks

- Own *part properties* typed by *blocks*
- Own *operations* that realized *operations* owned by functional blocks
- Are connected to other *logical blocks* by *connectors* (*ports* may also be used, if appropriate)
- May own *value properties* typed by *value types (*which are typed by *units*)

Physical Blocks

- Own *part properties* typed by *blocks*
- Own *operations* that realize *operations* owned by logical blocks
- Own *proxy ports* typed by *interface blocks*
- Are connected to other *physical blocks* by *connectors*
- May own *value properties* typed by *value types (*which are typed by *units*)

Interface Blocks:

- Own *flow properties* typed by *signals*
- May own *ports* typed by other interface blocks
- May own *signals* and *interface blocks* (if appropriate)

State Machines

- All *transitions* are defined by *signals*, *change events, time events,* or *operations*
- All *states* have *entry/do/exit behaviors* defined
- Most *do behaviors* will call activities owned by *use cases*; alternately, the *activities* may be moved from the *use cases* to the *states*.

End state:

- All *use cases* are decomposed by activity diagrams
- All activity diagram nodes are either *call behavior* nodes that trigger other *activities* or are *call operation* nodes triggering leaf-node *operations* on *activities,* or logical/physical blocks
- Functional requirements are either <<satisfied>> by *operations* or by *activities*
- All leaf-node functions are *operations* on with *in, out* and *result parameters* typed by *signals*.
- *Ports* have been added to the logical *blocks* (if appropriate) and are typed by *interface blocks*
- Internal block diagrams have been created to show how logical *blocks* connect; all connectors have *item flows* showing what *signals* flow along them.
- *Item flows* are used because of their ability to connect deeply nested *ports* and relate *object flows, conveyed information*, and *messages*
- All *object flows, messages*, and *signal event* transitions are mapped to *item flows*.

- <<physical>> blocks *realize* logical *blocks* and are used to *redefine part properties* of each physical architectural variant.
- All quality checks pass (no untyped elements, documentation fields complete, no unconnected pins, etc.)

Requirements:

- All functional requirements are satisfied by *operations* or *activities*
- All interface requirements are satisfied by *ports*
- All physical and performance requirements are satisfied by *value properties*
- All design constraints are satisfied by *blocks*
- All requirements are *verified* by *test cases*