2018 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
SYSTEMS ENGINEERING (SE) TECHNICAL SESSION
AUGUST 7-9, 2018 – NOVI, MICHIGAN

The NeMO Orbiter: A Demonstration Hypermodel

Michael J. Vinarcik, P.E., FESD
University of Detroit Mercy
Booz Allen Hamilton
Detroit, MI

Michael J. Vinarcik, P.E., FESD
University of Detroit Mercy
Booz Allen Hamilton
Detroit, MI

Peter Hodges
Fiat Chrysler Automobiles
Auburn Hills, MI

Lizardo Amador Marin
Kyle Ebner
Gary Kliczinski
Natalya Matevossyan
Ford Motor Company
Dearborn, MI

Jesus Mata Castaneda
General Motors Corporation
Milford, MI

ABSTRACT

System modeling is continuing to grow in importance as the enabling discipline for digital engineering. Descriptive system models can be used as the "central nervous system" of a system development effort (to federate a constellation of analytical models and other engineering content).

Hypermodeling is a methodology focused on maximizing model elegance through the efficient generation of a descriptive system model (with appropriate supporting content). It emphasizes the most simple, direct approach to rigorously capturing relevant information. Hypermodels use a limited set of model elements, relationships, and properties and seek to maximize the amount of information derived from the model.

The NeMO hypermodel, an example built by students at the University of Detroit Mercy, provides a comprehensive demonstration of this approach and includes behavioral, structural, and analytic information as well as metrics and requirements.

It is hoped that this large example will serve as a focus for discussion and experimentation in the system modeling community. Links to hypermodeling tutorial videos are available for study and comment at the hypermodeling website: http://hypermodeling.systems.

INTRODUCTION

Hypermodeling was born of necessity, there were several systems modeling needs emerging at the beginning of 2018 that this effort was intended to address:

- There was a need to unify a variety of modeling techniques that the author had developed in the past several years and demonstrate their utility and coherence in a larger effort.
- The system modeling community needed a publicly available reference model, drawn from unclassified and non-proprietary sources, that could be used as a testbed for new modeling techniques, analyses, and development.
- A large model was needed for further development of adjacent and ancillary analysis techniques such as design structure matrices, Salado's tension matrices [1], network graph visualizations, and other potentially useful emerging practices.
- Finally, there was a need to challenge the *status quo* in modeling and demonstrate that there was a way to model systems effectively using relatively few relationships and element types while still maintaining a coherent and rigorous model narrative of the system of interest.

HYPERMODELING

The Elegance Equation

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:

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

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

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.

Agile proponents have described software development in two ways:

WET = Write Everything Twice

DRY = Don't Repeat Yourself

A corollary of these principles is directly applicable to system modeling: *Don't Create What You Can Infer or Query*. As long as these inferences and queries are unambiguous, leveraging them has a significant and direct impact on reducing the number of modeling elements.

Controversial Aspects of Hypermodeling

There are several controversial aspects of hypermodeling that challenge assumptions in traditional systems engineering and modeling approaches (See Figure 1). First, requirements are subordinated and are considered "just another model element." Source requirements, capability documents, or other upfront goals provided by stakeholders, management, or regulatory bodies should be respected and are collected at the beginning of the model development process or if they are imposed later. However, the authoring of individual system, subsystem, and component requirements is deferred until very late in the hypermodeling process. This is intended to free up resources because, in the author's experience, significant effort is spent trying to synchronize text-based requirements with system models that are still in flux.

Modeling of the complex systems that characterize the modern age is an exercise that is

both iterative and collective. Behavior, structure, interfaces, flows, properties, and relationships typically evolve rapidly over the course of the modeling effort; their dynamic nature and the constant emergence of new information makes attempts to keep textual requirements in sync with models difficult and time-consuming. The effort this requires results in a drain on resources and saps modeling effort away from more fruitful discovery and analysis.

Many programs attempt to conduct requirements development before the integrated system model is matured and this leads to an endless cycle of requirements updates. To make matters worse, requirements typically are placed under change control and this necessitates involving administrative, engineering, and management personnel in the review and approval of every proposed change. This usually entails pre-work and attendance at change control meetings.

For that reason, requirements authoring is purposely deferred until the entire model (or at least key sections of it) is stable enough to warrant requirement authoring. At that point, it is a straight-forward matter to author requirements based on functions, messages and signals, interface information, and other relevant model content (See *Requirements Churn: The Hidden Drain on Systems Engineering* [2].)

The second controversial aspect of hyper modeling is that very few relationships are used between requirements. It should be noted that the SysML extended requirement types (functional, performance, design constraint, etc.) are strongly preferred because system modeling tools are capable of rigorously validating requirement relationships. For example, functional requirements *must* be <<satisfied>> by activities or operations, and interface requirements *must* be <<satisfied>> by flows, connectors, or ports. For this reason, <<satisfy>> is the only relationship that is permitted between requirements and descriptive model elements in the construction of a hypermodel. It forces a crispness in requirements because the element, behavior, or property that "makes them true" must be present in the model to serve as the other end of the connection. The <<derive>> and <<refine>> relationships are permitted between requirements and between requirements and certain model elements (such as use cases). These allow the maturation of the model to drive additional requirement content (for example, the creation of functional and performance requirement couplets connected by <<refine>> relationships). <<trace>> relationships are used between requirements and any upstream content, artifacts, or standards. Finally, the <<verify>> relationship is used between requirements and test activities, which are essentially activity diagrams that have a special property that returns a *result* of pass or fail. This ensures the clear identification of the step in the verification process that adjudicates the outcome and determines whether the requirement has been appropriately satisfied.

Another controversial aspect of hyper modeling is the absence of swimlanes on activity diagrams. It is the primary author's opinion that swimlanes are a serious misrepresentation and misuse of a modeling tool. They attempt to use spatial positioning on a diagram as a surrogate for properties of interest. That may have been of some utility with drawing tools but is completely inappropriate for modern modeling approaches. Several individuals have challenged the author, claiming that "crossing the swimlane" is a useful way to identify needed interfaces. The author's approach, which rigorously associates each action node with a specific owning class or usage, enables simple, query-based identifications of functions that need interface assignments because they are owned by different model elements. The hypermodeling approach also allows for the assignment of object flows between action nodes to specific connectors, sequence diagram messages, and state transitions. There is no need to rely on cosmetic approaches as a surrogate for rigor when rapid, tool-driven queries are possible.

Another commonly used practice prohibited in hypermodeling is the use of the <<allocate>> relationship. It is the primary author's opinion that allocation is inherently sloppy; it is easy to allocate thousands of requirements to model elements with little rhyme or reason. This forces the downstream consumer of the requirements (typically text statements) to construct lossy mental models of the system intent. The use of the <<satisfy>> relationship, in contrast, forces a crispness in the construction of both the model and the related requirements. It forces singularization and careful selection of each element, property, interface, or behavior that <<satisfies>> the requirement and "makes it true." This improves clarity and facilitates the identification of duplicates or conflicts because instead of sifting through thousands of requirements allocated to a system element, the relative handful that are <<satisfied>> by a given element or property are easy to compare in a check for errors and inconsistencies.

Note that more than one element may <<satisfy>> a requirement if that is appropriate.

Other Aspects of Hypermodeling

Hypermodeling relies on the <<realization>> relationship to connect different layers of architectural abstraction. If a purely functional layer is constructed it is <<realized>> by logical architecture elements and these are then <<realized>> by elements of the physical architecture. This relationship construction also allows error checking and the tailoring of functions at each level. For example, by having every physical element own its own copies of relevant functions, inputs and outputs can be adjusted, additional content may be added, and code snippets can be embedded. These relationships improve the fidelity and rigor of the model significantly. For example, a physical model element may <<realize>> a logical element that in turn <<realizes>> multiple functions…each of which is associated with functional hazards, regulatory requirements, or other relevant information. Using structured queries, these relevant pieces of information may be collected and displayed at the physical level without the need for direct connection or duplication. In addition, if any changes are made to these higher level analyses or artifacts the information presented to the team members at the physical level is immediately updated; this approach is inherently DRY.

MagicDraw

One of the primary author's Ten Commandments of Modeling (see Figure 2) is that one should ruthlessly subordinate a modeling effort to what the chosen modeling tool does well. MagicDraw™, developed by No Magic, is arguably the most standards compliant SysML modeling tool available today. It is also arguably one of the most user-friendly, due in large part to its user community's feedback. The primary author is a member of No Magic's Client Advisory Board, a formal group representing some of the largest and most demanding user communities, but the company also considers a multitude of feature requests submitted by individuals worldwide. As a result, each new release or service pack has a host of improvements that savvy modelers can exploit to increase their productivity.

Because of these factors, the author believes that the η_{tool} ε_{tool} for MagicDraw are relatively close to 1, *but only if modelers know how to effectively use it.* Structuring each model to make use of MagicDraw's internal query language (known as Structured Expressions) or other languages (such as Beanshell and JavaScript) is how hypermodeling delivers on its goal of economizing modeling while maintaining rigor.

QED

In traditional geometric proofs, QED was the final line a student wrote to indicate he was finished. *Quod erat demonstrandum* means "that which was to be proven" in Latin.

This acronym has been adapted for hypermodeling:

What is the **Q**uestion we need to answer?
How can we **E**xtract relevant information from the model?
How should we **D**isplay it to stakeholders in a meaningful, easy to consume way?

By appropriately harnessing and answering these three questions, a competent system modeler is able to provide value for his program by allowing the program team's engineering staff to have insight into the system of interest.

As Frederick Brooks wrote:

"Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.[3]"

Tables (or matrices or relationships maps) are often much more useful and clear than diagrams. For this reason, competent modelers should carefully select *how* information should be presented (remembering that the model and its content is independent of its display). Subject matter experts, modelers, decision makers, and stakeholders may have different cognitive styles and preferences. Modelers must be willing to adapt without compromising model integrity. There is a fine line between fruitful challenge of the *status quo* (such as abolishing swimlanes) and fruitless conflict.

Observing QED principles requires that the modeler(s) find a way to represent all relevant information in a well-defined structure so that it can be found and serve as the authoritative source of technical truth; for each piece of useful information:

Should it be owned by an element?
Should it be owned by a relationship?
Should it be owned by a usage?

BUILDING BLOCKS OF A HYPERMODEL

Operations

Operations are the most important behavioral element in hypermodeling:

- They own *parameters* (allowing rigorous definition of inputs/outputs).
- They can be further decomposed with *methods*.
- They must be owned by *blocks* or *activities*.
- They may <<satisfy>> functional requirements.
- Their use streamlines behavioral decomposition and because they have clear ownership many complex queries are simplified.

Opaque Behaviors

Opaque behaviors (a model element type) encapsulate structured expressions, metachains, scripts, queries, and other tool-specific behaviors.

They may be used to drive tables, matrices, derived properties, and metrics. A library of useful opaque behaviors may be created by expert modelers and shared throughout an organization. In many ways, opaque behaviors enable DRY development of the system model by facilitating reuse and enabling maintenance of a smaller set of custom queries.

Opaque behaviors may also be used to facilitate analysis; in the case of this case study, an opaque behavior was used to facilitate scenario-based power consumption analysis.

Item Flows

Item flows are one of the most important elements used in hypermodeling. They may be used to fully integrate behavior, structure, and flows. A given *item flow* connects specific *parts* or *ports* (at the usage level) and may be mapped to *connectors* (internal block diagrams), *object flows* (activity diagrams), *messages* (sequence diagrams), and *transitions* (state machines). Derived properties using *item flows* allow extended information to be displayed on any diagram in this chain of

connection (for example, mapping a functional output to a transition on a state machine).

This represents the most efficient way the author has yet discovered to fully describe the relationships between these diagrams with a minimum number of elements and relationships.

THE BIRTH OF THE NEMO HYPERMODEL

A unique opportunity presented itself in the fall of 2017: a number of students who previously had studied systems architecture and systems engineering as students in the University of Detroit Mercy's Master of Science in Product Development (MPD) program wished to complete incremental requirements and obtain systems engineering certificates from the University. They were required to take additional classes and elected to enroll in the newly-created dedicated systems modeling with SysML class. (SysML, the System Modeling Language, is the industry standard, general purpose modeling language. It is administered by the Object Management Group). Because they had previous experience with SysML modeling integrated into the MPD curriculum, the author felt they would benefit from an extended modeling project and gave them the option of developing a reference model.

Once they had agreed to participate in this effort, a suitable subject for the model needed to be selected. Past modeling projects in the MPD program included notional space probes (with missions and capabilities specified by the primary author), a notional polar exploration submarine, personal survival pods (intended to sustain disaster survivors or explorers in remote areas for a period of time without resupply or other resources), and a Next generation Mars Orbiter (NeMO). The author selected the NeMO for redevelopment (starting from published NASA goals, objectives, subobjectives, and investigations) because it was based on public information and non-automotive (the students all work in the automotive industry and this precluded the possibility of introducing any proprietary information). It also had the advantage of being large enough to exercise the modeling approach and small enough to be manageable by the team.

Modeling Process

The NeMO hypermodel was constructed in one term by six students. One class session each week was augmented with one or more feedback and help sessions to assist the students with advanced modeling techniques and the resolution of quality checks to improve model health.

The modeling effort followed the hypermodeling loop displayed in Figure 3. It should be noted that this cycle is highly iterative; as information or gaps are exposed in one area they may impact other areas. The model's package structure (see Figure 4) allowed the students to work in a relatively linear fashion since each "lower" package tends to be less abstract and more concrete than the one above it.

The instructor imported the NASA goals, objectives, sub-objectives, and investigations and their relationships (see Figure 5) and provided these to the students as a basis for their modeling. Behavior modeling was conducted using the relationships shown in Figure 6. They identified capabilities and <<traced>> them to investigations (these served as the connection point to the upstream goals and objectives, see Figure 7); they created use case diagrams to analyze these capabilities (see Figure 9). Each student also developed instrument architectures that implemented those capabilities and was assigned a subsystem that provided or enabled generic satellite behaviors and capabilities.

Elements identified in the use case diagrams were used as the basis for the system context (see Figure 8). Students were given the option to create a functional architecture (using *activities* to own *operations*) or begin with a logical architecture (since this project already made numerous assumptions about which elements would perform various functions). See Figure 10 for notional architectural elements and Figure 11 for a relation map that decomposes the logical architecture.

Interfaces were defined using Internal Block Diagrams (see Figure 12); these allowed the use of *item flows* to unify behavioral, structural, and flow elements. State machines were created (and owned by specific system elements) to integrate desired system behaviors (see Figure 13 and Figure 14); note that entry/exit points are extremely useful SysML elements that facilitate the use of submachines.

The use of generalizations between use cases (see Figure 15) allows the cloning of associated behaviors and rapid tailoring at each level of abstraction (or to create a variant).

The final graphical example provided illustrates the relationships between requirements and test activities (see Figure 16).

Detailed parametric diagrams were not constructed; however, a scenario-based power consumption analysis was developed. Customized *power usage* relationships were created between part properties and the power scenarios in which they consumed power. A scale factor (defaulting to 1) was defined to allow lower-power usage of a given part. An opaque behavior was defined that rolled up all power usage for a given scenario (power consumption for a part property was provide by the defining block, multiplied by the multiplicity and scale factor, and then summed for the power scenario). Although not as rigorous as a state-machine and parametric diagram-based analysis, this approach demonstrated that a rapidly constructed, less rigorous approach still had significant qualitative value.

Quality Checks

Numerous quality check tables were created to enable the rapid detections of errors in the model. For example, one table displays "trapped parameters." These are parameters owned by operations that cannot legally flow over the available interfaces owned by the block. These may be resolved by adding interfaces or generalization relationships to enable the "trapped" signals to flow over existing interfaces.

Other tables displayed elements without documentation or required <<trace>> relationships. The creation of the custom queries that drive these tables requires some advanced skills but when they are placed in a library as opaque behaviors every model that uses the library may benefit from them.

IMPACT

The NeMO hypermodel and related tutorial videos were released at the No Magic World Symposium in May 2018. As of June 2018, the model is already in use by one Ph.D. student who is analyzing it as part of his thesis and it is serving as the basis for improving interoperability between MagicDraw and another analysis tool.

Several debates (primarily in LinkedIn's MBSE group) have been sparked by the video series and model and the author has provided it to the teams developing proposals for SysML 2.0. Ongoing discussions with various modelers and corporations are underway.

CONCLUSION

The NeMO hypermodel was born from a set of needs and the effort of a dedicated group of students. It illustrates a number of advanced modeling techniques and a unified approach intended to maximize the value of an integrated system model.

The following students built the NeMO hypermodel:

- Lizardo Amador Marin
- Kyle Ebner
- Peter Hodges
- Gary Kliczinski
- Jesus Mata Castaneda
- Natalya Matevossyan

The primary author is deeply grateful for their willingness to tackle this challenge and share their work with the system modeling community as the basis for further development and refinement.

DISCLAIMER

The model and paper were prepared by the authors in their personal capacities as instructor and students. The opinions expressed in this article are the authors' own and do not reflect the views of their respective employers

REFERENCES

[1] *The Tension Matrix and the Concept of Elemental Decomposition: Improving Identification of Conflicting Requirements*, A. Salado and R. Nilchiani, in IEEE Systems Journal, vol. 11, no. 4, pp. 2128-2139, Dec. 2017.

[2] *Requirements Churn: The Hidden Drain on Systems Engineering*, Systems Architecture Guild YouTube channel, published 10/8/2016. https://www.youtube.com/watch?reload=9&v=T84WZ4WLqw8.

[3] *The Mythical Man-Month: Essays on Software Engineering* (1975, 1995) [Originally published in 1975; Brooks, Frederick, page numbers refer to the substantially expanded Anniversary Edition (2nd Edition), 1995, Addison-Wesley, ISBN 0-201-83595-9], Pp. 102–3.

APPENDIX A: FIGURES

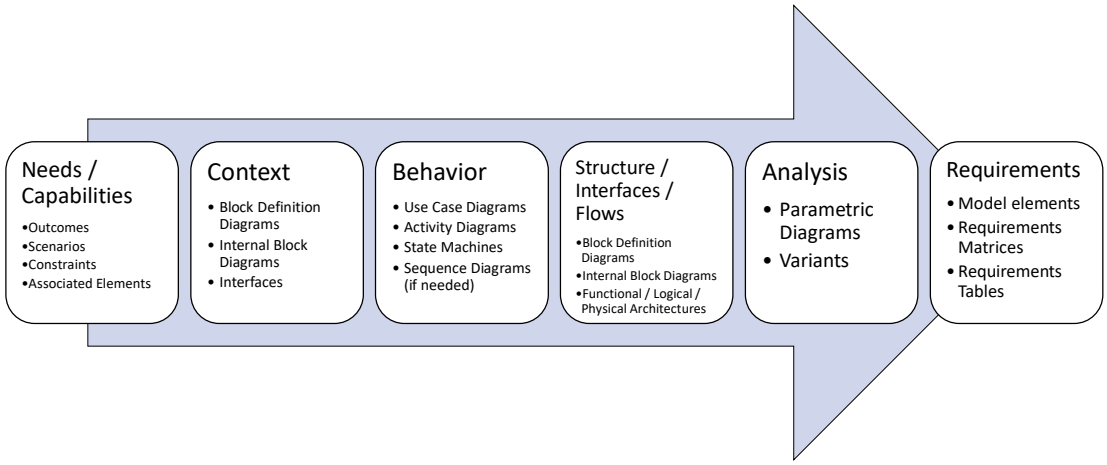

Figure 1: The Hypermodeling Approach

The 10 Commandments of Modeling

I. Thou shalt not make shelfware
II. Thou shalt not add any model element without reason
III. Thou shalt not add any model element that can be derived
IV. Thou shalt document all model elements
V. Thou shalt always apply units to value properties and tags
VI. Thou shalt type all model elements
VII. Thou shalt integrate the model and help it grow organically
VIII. Thou shalt delete unneeded elements to prevent clutter
IX. Thou shalt not be afraid to say “no”
X. Thou shalt always do what is right for the model

The Greatest Commandment:

Thou shalt ruthlessly subordinate thy approach to what the modeling tool does easily and well. There is no point in trying to model and automate document-based processes with their inherent inefficiencies.

Figure 2: The Ten Commandments of Modeling

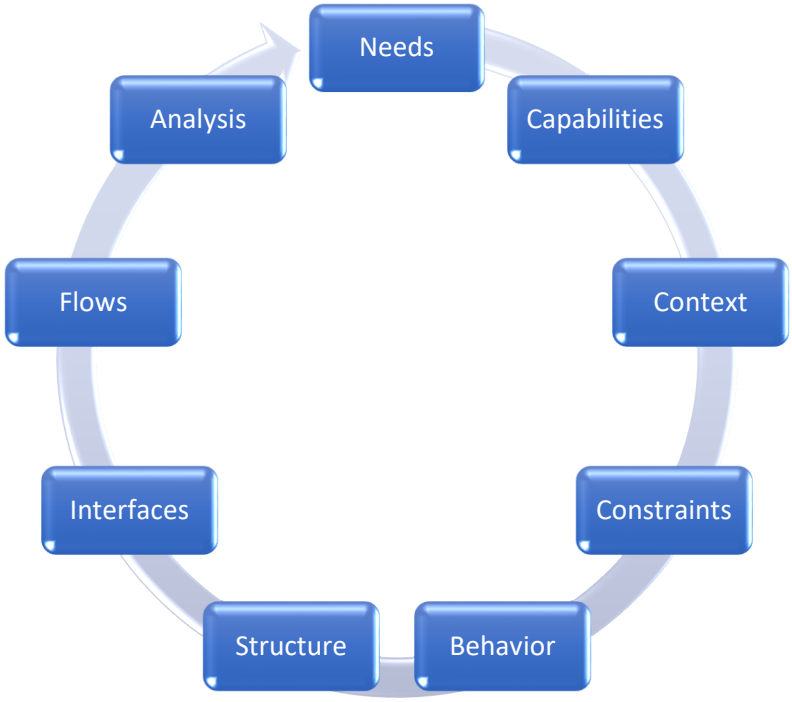

Figure 3: The Hypermodeling Loop

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

Figure 4: Hypermodel Package Structure

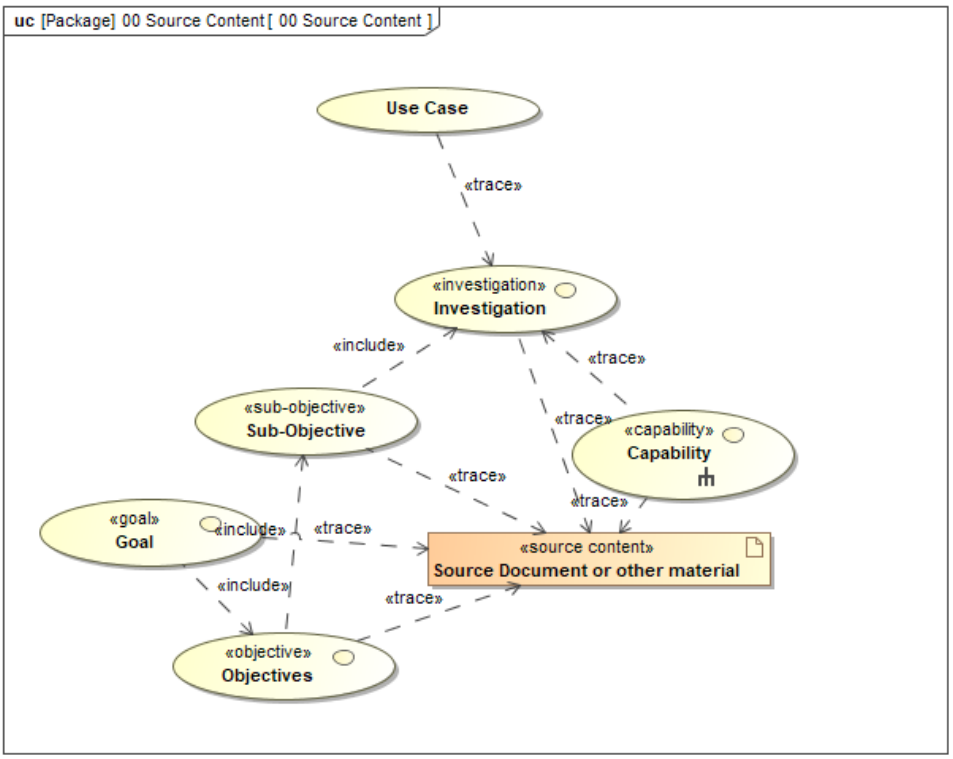

Figure 5: Source Content Relationships

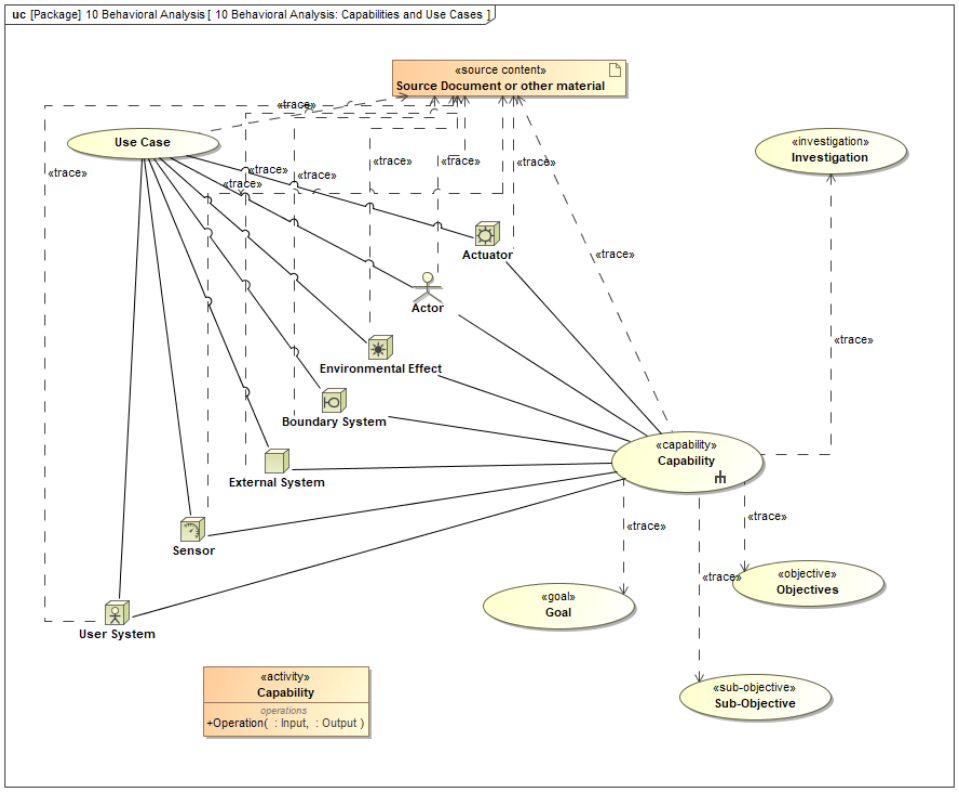

Figure 6: Behavioral Analysis Relationships

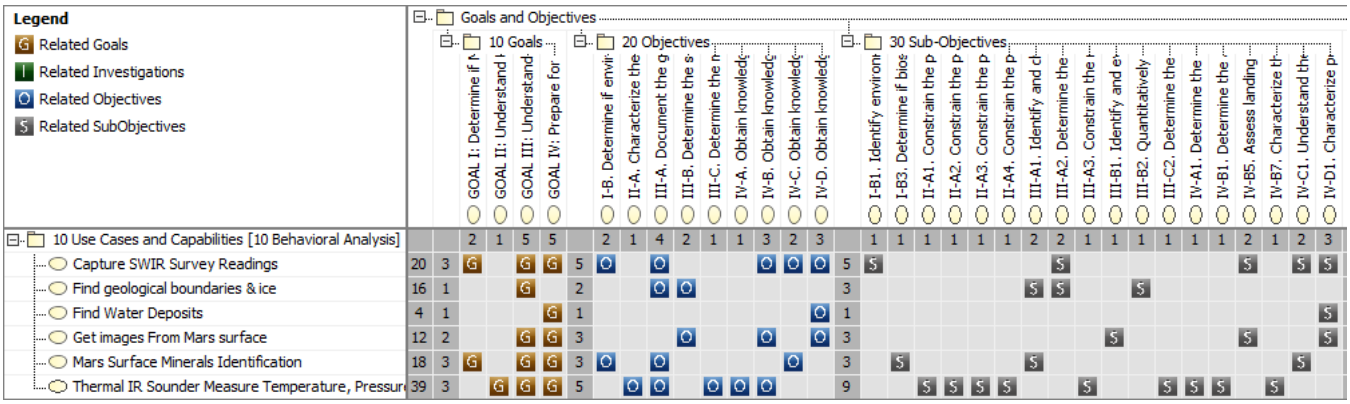

Figure 7: Goal, Investigation, and Capability Matrix

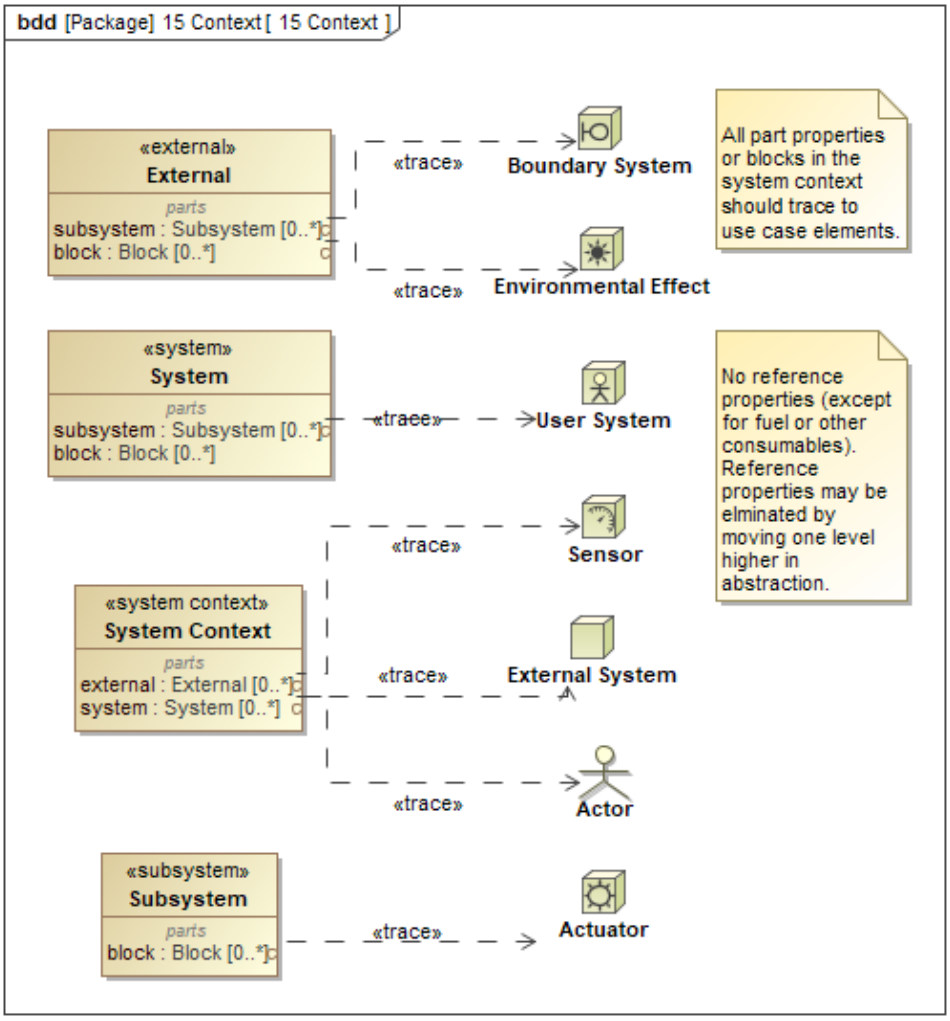

Figure 8: System Context Relationships

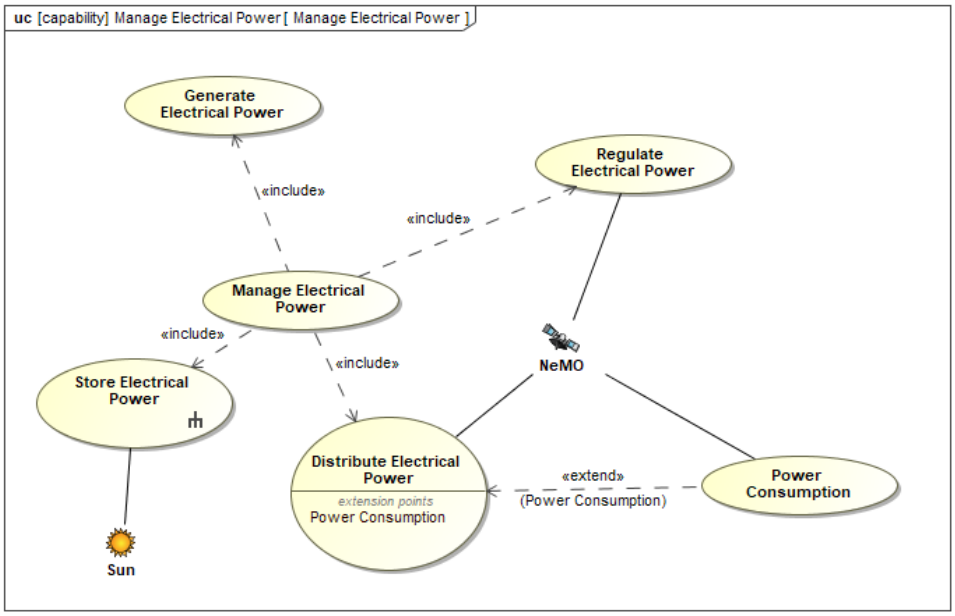

Figure 9: Capability Use Case Diagram

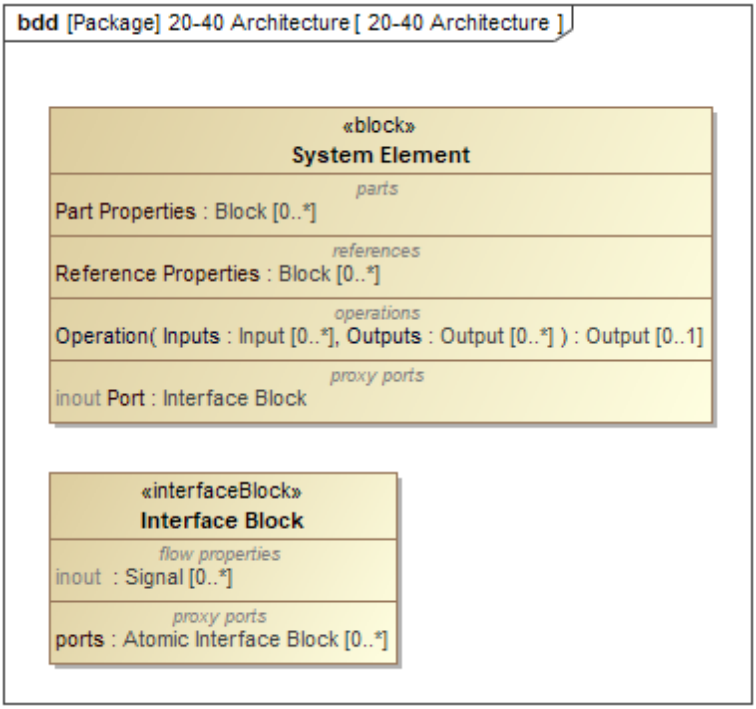

Figure 10: Notional Architectural Elements

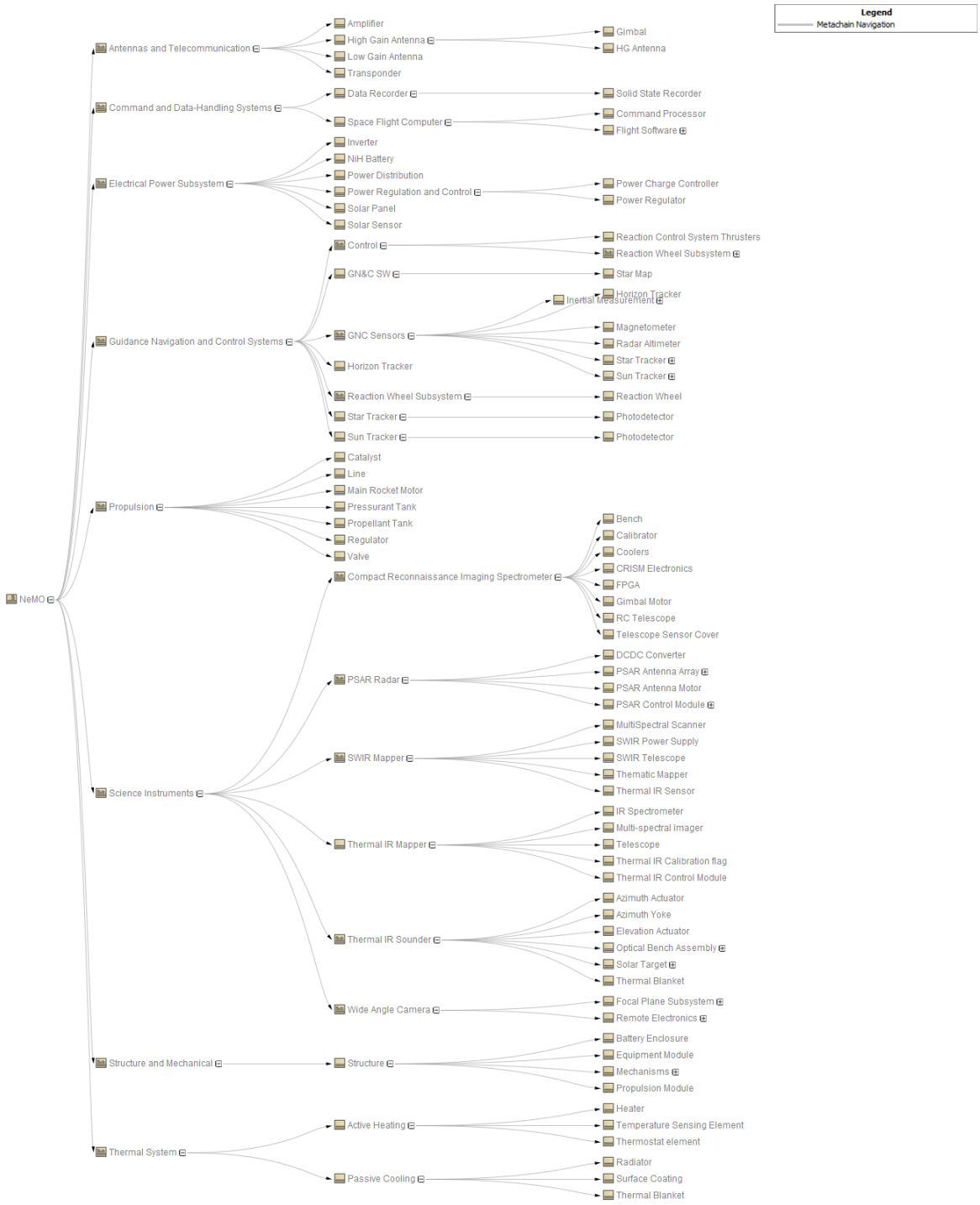

Figure 11: Relation Map of NeMO Logical Architectural

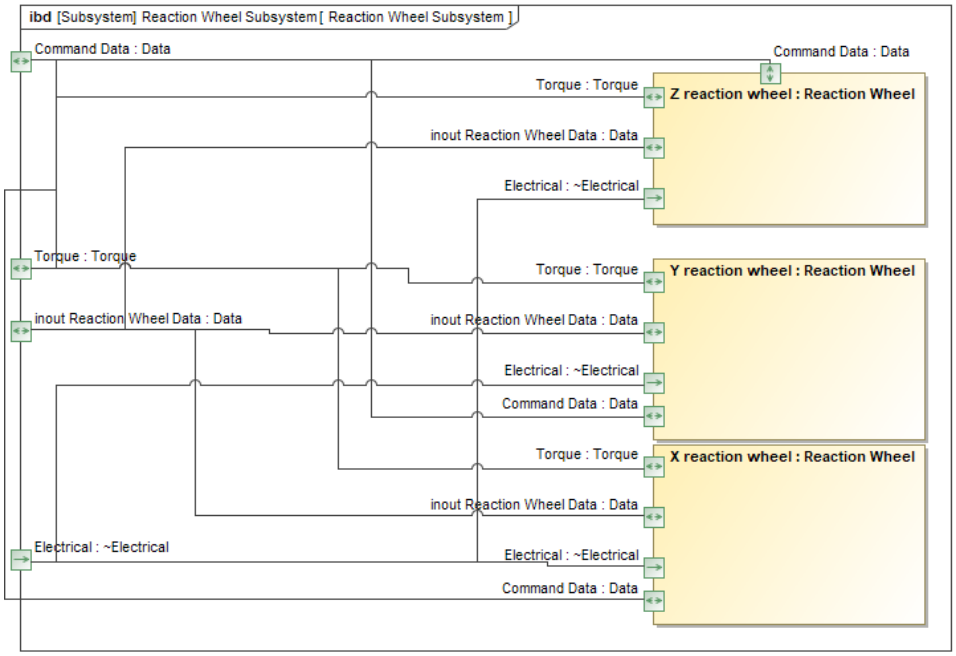

Figure 12: Example Internal Block Diagram

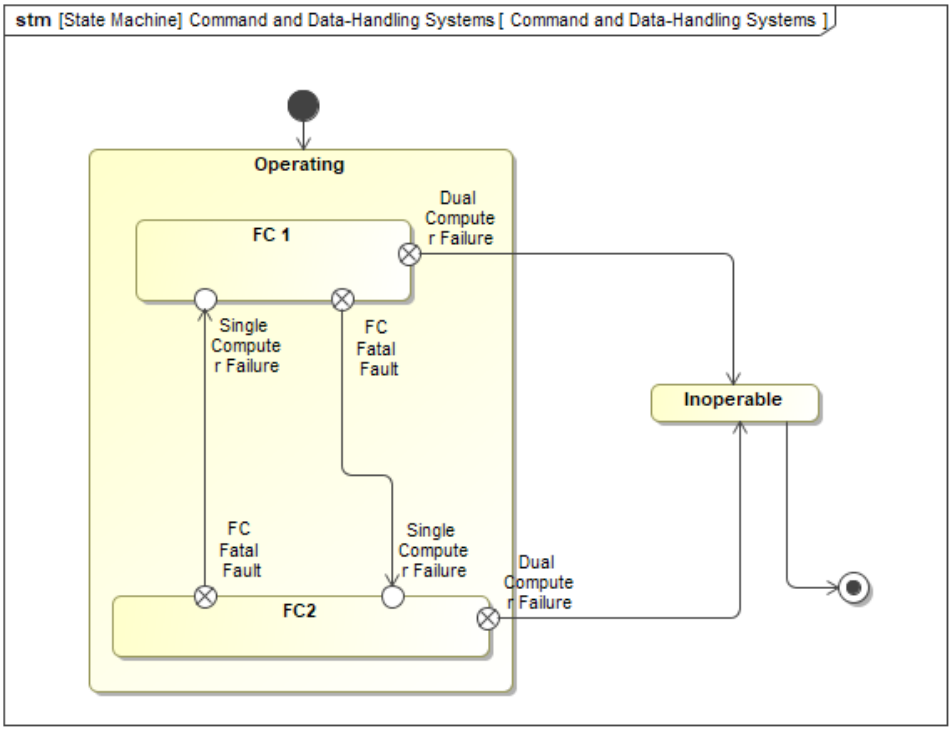

Figure 13: State Machine Showing Entry/Exit Points

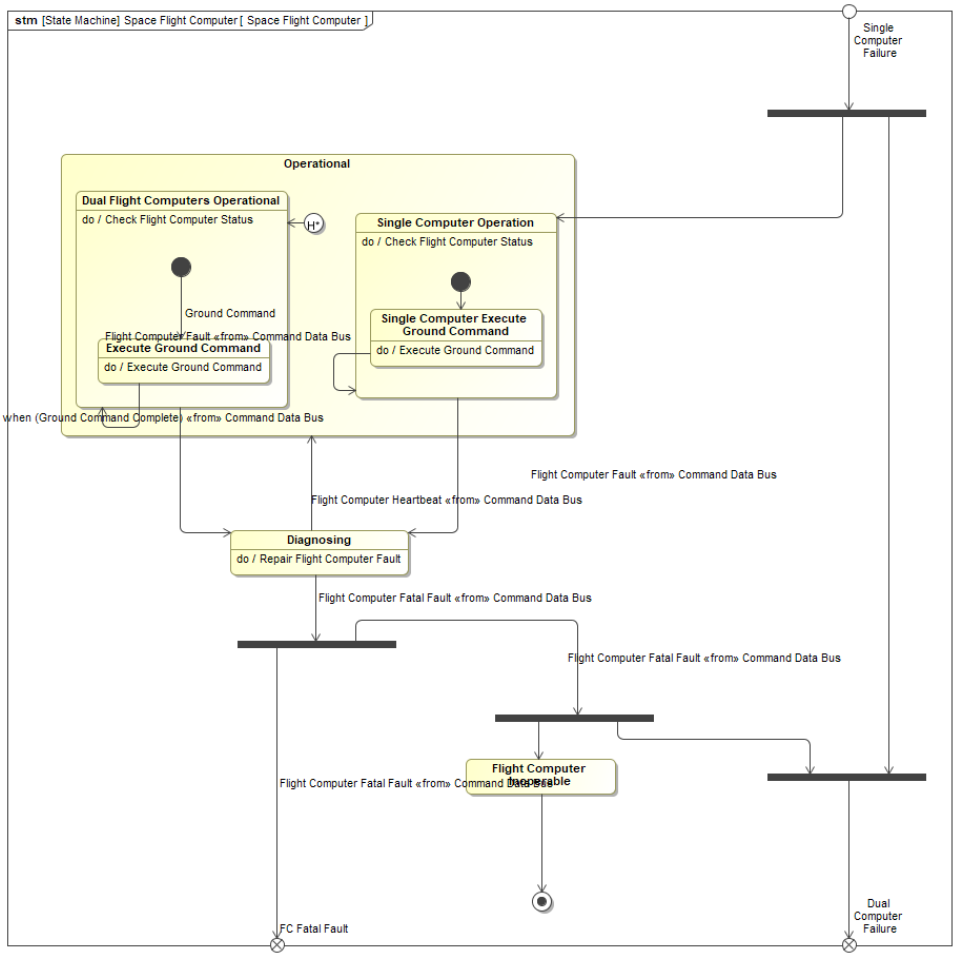

Figure 14: Space Flight Computer State Machine

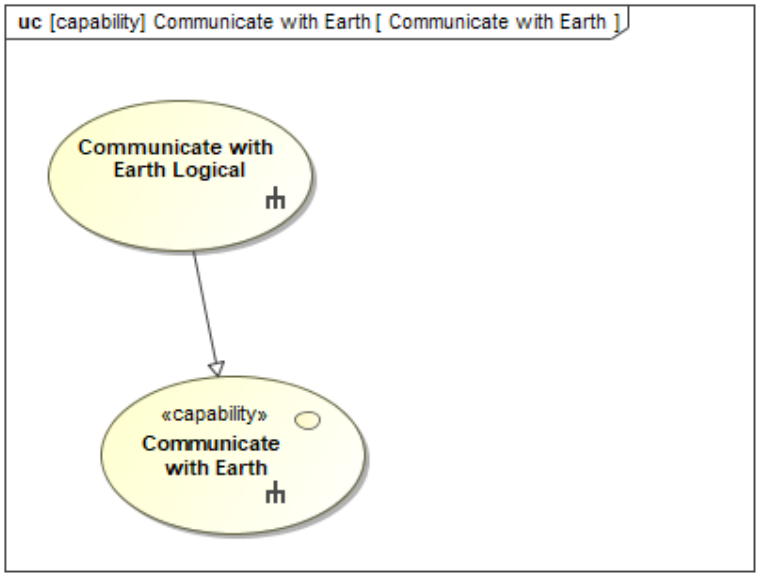

Figure 15: Use Case Diagram showing Generalization to Facilitate Behavior Cloning

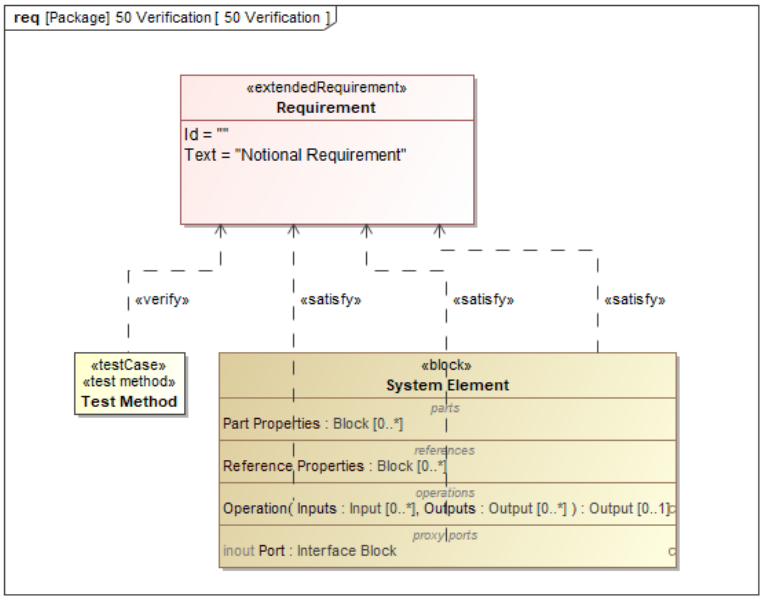

Figure 16: Validation Relationships

APPENDIX B: HYPERMODELING 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.
- 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.)

Capabilities:

- Use cases stereotyped as *capabilities* own *activities* that own *operations*
- They are only used as containers for *operations*
- They should be organized so that the majority of 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 may be 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 realize *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

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*, *flows*, or *connectors*
- All physical and performance requirements are *satisfied* by *value properties*
- All design constraints are *satisfied* by *blocks*
- All requirements are *verified* by *test cases*