One Model,

Vitech

Zane Scott, Vice President of Professional Services
David Long, President
Vitech Corporation

www.vitechcorp.com

Introduction

In 1995, Jim Long presented a seminal paper at the International Symposium of the International Council on Systems Engineering in which he set out the relational context for a range of behavioral views used to depict the logical architecture of systems under study or design. That document has guided our thinking and context for the ensuing years, giving us a way of thinking about the structure of our presentation of design information.

Over the years, we have come to the realization that the work Long started in that paper was an invitation for us to extend its application to other views. His first steps have inspired us to continue the journey and to spread the value of his approach to an even wider audience. In that spirit we offer this paper in the hope that it brings discipline and rigor to the systems engineering conversation and proves as helpful to the reader as his paper has for audiences across the years.
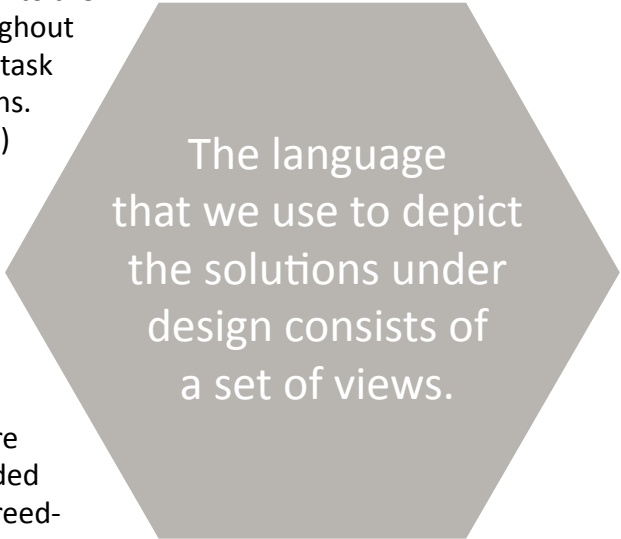
Communication

The *INCOSE Systems Engineering Handbook* lays out five essential benefits of model-based systems engineering:

- **Improved communications** among the development stakeholders (e.g. the customer, program management, systems engineers, hardware and software developers, testers, and specialty engineering disciplines).
- **Increased ability to manage system complexity** by enabling a system model to be viewed from multiple perspectives and to analyze the impact of changes.
- **Improved product quality** by providing an unambiguous and precise model of the system that can be evaluated for correctness and completeness.
- **Enhanced knowledge capture** and reuse of the information by capturing information in more standardized ways and leveraging built-in abstraction mechanisms inherent in model-driven approaches. This in turn can result in reduced cycle time and lower maintenance costs to modify the design.
- **Improved ability to teach and learn SE fundamentals** by providing a clear and unambiguous representation of the concepts (*INCOSE 2015 Systems Engineering Handbook*, p. 189).

The goal of language is to give voice to the meaning that we assign to the thoughts, experiences, and observations we experience throughout our lives. In the context of model-based systems engineering, this task bears the burden of conveying the meaning of our solution designs. We build a data model of the solution (and often alternatives) composed of the elements, relationships, and attributes of the system solution.

The language that we use to depict the solutions under design consists of a set of views. These may be graphical or word-based, but in any case, they represent a subset of information from or about the model arranged for presentation according to a set of rules prescribed for constructing that view. Views are (or should be) constructed by querying the model for the needed information and then assembling the information into an agreed-upon format.

> The language that we use to depict the solutions under design consists of a set of views.

This paper discusses that language and its building blocks. We will examine various systems engineering views in some detail, paying particular attention to the information they convey, the format they use to convey it, and the intended audience they are designed to reach. The intent is not to provide an exhaustive treatise on the detailed notation (a purpose better served by guides, textbooks, and formal specifications), but instead an overview of many views, the information that underpins them, their interrelationships, and their effective use.

Considering the Audience

Any consideration of the choice of expressions must begin with the audience. The purpose of communication is to transfer information and the "meaning" assigned to it in a way that creates a picture in the mind of the audience that matches the picture in the sender's mind. Therefore, the first criterion in choosing a representation or view is that it must speak to the intended audience. In this way, the sender can cast the information or meaning in a form that will be understood by the audience in the same way it is by the sender.

What will "speak" to a given audience is determined by the background and experience that shape the way the audience communicates. If the audience is a group of business administration professionals, they are likely to be accustomed to and comfortable with flow charts as a way of depicting process flows. Other expressions of process flows (sequence diagrams, for example) may show inputs, outputs, and sequences, but do so in a way that need explanation and an orientation to the view. The choice of an unfamiliar view slows or obstructs communication, so the sender must be conscious of the potential for this problem given the audience background and composition.

In linguistic terms, this is a problem of "productive" and "receptive" vocabularies. Our productive vocabulary is composed of the words we can use to produce messages. Our receptive vocabulary includes all the words we can recognize and understand in receiving messages from others.

When we "speak" or otherwise transmit ideas, we need to use a productive vocabulary that matches the receptive vocabulary of our intended audience as closely as possible. When that happens, we produce messages that can be received and understood by the audience. If we use a productive vocabulary that isn't a part of the receptive vocabulary of the audience, our meaning will not be conveyed.

Obviously, different backgrounds and experiences result in the development of different receptive vocabularies. The greater the diversity of audiences that systems engineering senders must communicate with, the wider the senders' productive vocabularies must be in order to match all potential audiences. The systems engineering audience is large and getting larger. As acknowledged by the International Council on Systems Engineering in their 2025 vision, *A World in Motion, Systems Engineering Vision 2025*:

> Stakeholder Expectations Drive System Trends. System performance expectations and many system characteristics will reflect the global societal and technological trends that shape stakeholder values. Examples of system stakeholders are:
>
> - System Users
>   - o The general public
>   - o Public and private corporations
>   - o Trained system operators
> - System Sponsors
>   - o Funding organizations
>   - o Investors
>   - o Industrial leaders and politicians
> - Policy Makers
>   - o Politicians
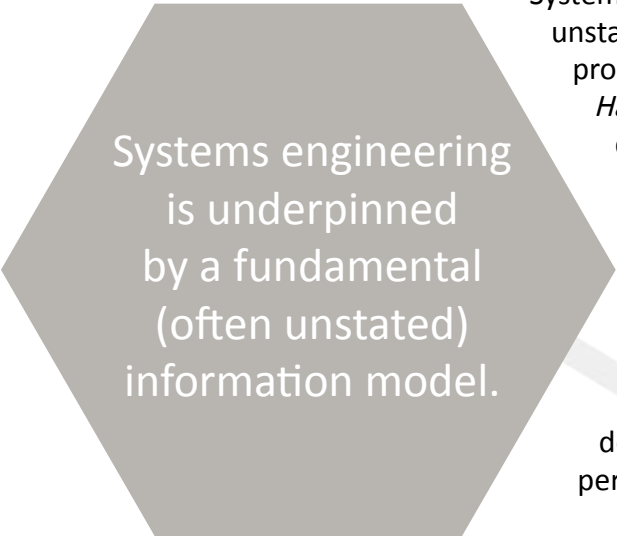>   - o Public/private administrators
>
> Across a wide variety of domains, stakeholders are demanding increased functionality, higher reliability, shorter product life cycles, and lower prices. Stakeholders are also demanding environmentally and socially acceptable solutions that assure safety and personal security while delivering more value to the users. In maximizing value to stakeholders, systems engineers have to cope with greater levels of complexity and interdependence of system elements as well as cost, schedules and quality demands. (INCOSE 2014, *A World in Motion, Systems Engineering Vision 2025*, p. 10.)

Because of systems engineering's history with defense and aerospace disciplines, it is tempting to see it in terms that are confined to those spaces. Whether it is the blizzard of acronyms that are particular to that market sector or the use of terms in ways that are confined to narrowly related disciplines, the use of jargon and specialized communication limits the breadth of communication possible.

It is therefore incumbent on the systems engineering profession to adopt as large a set of expressions as possible in order to expand the effective reach of systems engineering and, in the process, realize the economic opportunity represented by that expansion. By communicating with a large and diverse audience, systems engineering can best serve the global demand for system solutions.

The Model

By definition, the essence of model-based system engineering is found in the models it creates. A systems engineering model is at its root a representation of a physical reality which can be a problem or its solution. It represents the elements, interrelationships, and characteristics that make up the system being modeled. The views used to describe such a model should be drawn directly from the model itself. In the views we will consider, the model resides in a single repository – a single source of truth. That repository consists of elements that are modified by attributes (often referred to as properties) and related to other elements. This structure corresponds to the object-oriented approach.

Systems engineering is underpinned by a fundamental (often unstated) information model. As you execute systems engineering processes as reflected by the *INCOSE Systems Engineering Handbook* or other guides, you are implicitly eliciting, developing, analyzing, reviewing, and ultimately controlling this information. Good model-based systems engineering is far less about the diagrams and notations used to communicate this model than it is about having a clear, defined information model that captures the elements, attributes, and relationships essential to successfully engineering a system. (Perhaps most of all, it is about the relationships, because systems and systems engineering are defined by the interactions between parts that deliver the performance of the whole.)

Systems engineering is underpinned by a fundamental (often unstated) information model.

4

Vitech

Systems engineering has not yet arrived at a single metamodel that defines our concepts, their context, and the interrelationships in a formal way that underpins the necessary knowledge capture, analysis, and communication. The following is one representation of critical systems engineering concepts and their interrelationships spanning requirements, behavior, architecture, and test. Based upon 50 years of practical application and continued evolution, this integrated model presents a high-level view of not only the ultimate specification of a system, but also the journey to that specification – concerns opened and closed, risks identified and managed, and more.
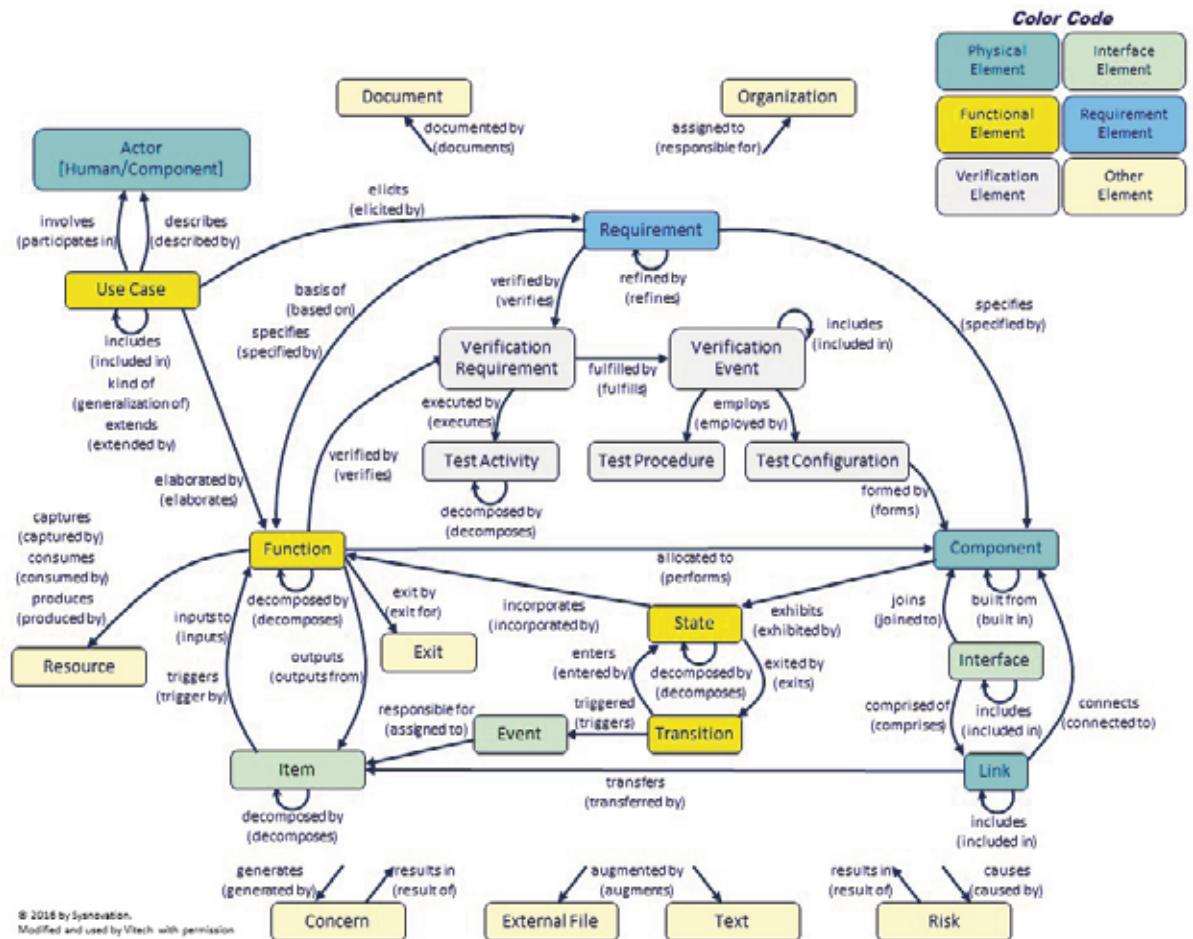
In this single repository setting, views are dynamically generated directly from the system design repository, ensuring that they are consistent with current design details. A change made in any view changes the design information in the repository and, conversely, a change made to the database is automatically reflected in the views. In this way, the communication through the views of the model is "real time" in representing the current state of the design at the moment the view is generated. Furthermore, because any changes to a view are reflected directly into the model itself, all other views drawn from the model will communicate the model consistent with the latest changes. In short, everyone interested in the model can be certain that they are seeing it – regardless of the chosen view – consistently and concurrently with the rest of the model's constituency.

With that background, we will turn our attention to the language of model communication.

Representing Requirements

Hierarchy Diagram

The hierarchy diagram – one of the oldest and most used systems engineering graphical views – represents relationships between several layers or types of elements. There is no pre-defined semantic for a hierarchy, allowing the creator to define the specific set of relationships to deliver the desired representation and insight. Sample uses include representing requirements hierarchies as well as functional composition, physical composition, and traceability across the design.

Level of Detail: Low
Audience: General
Content: Names and relationships
Use: Multi-level decomposition of requirements
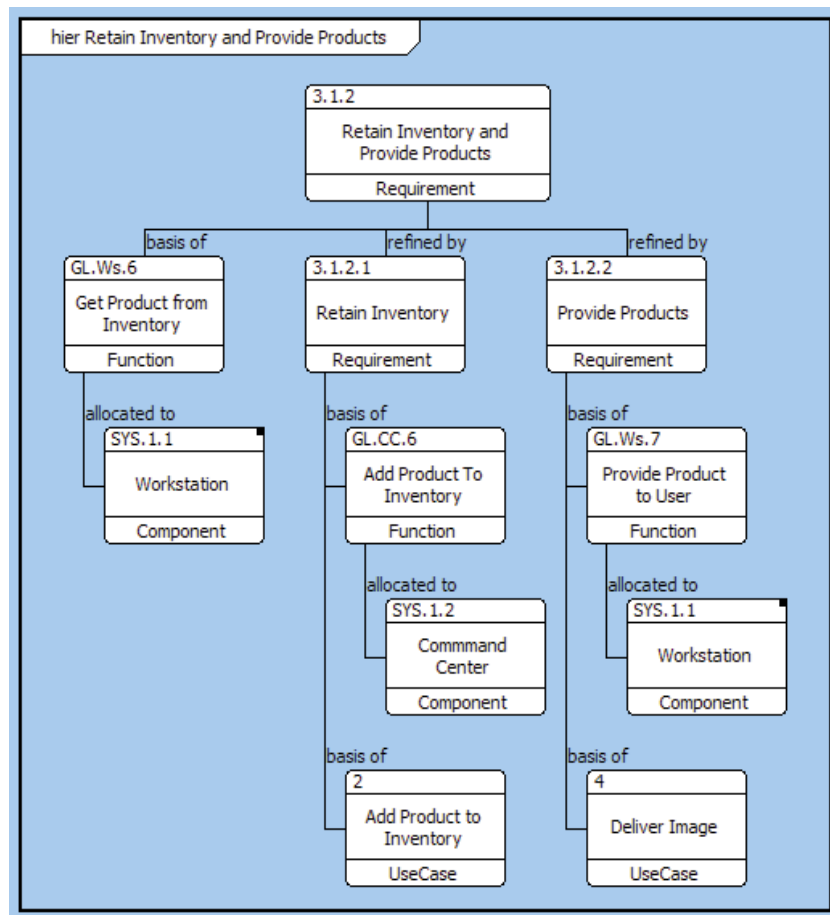
A hierarchy diagram is based on the combination of a root element and a set of relationships to display. The root element defines the starting point for the diagram and is classically shown as a node on the top (in a top-down representation) or left (in a left-to-right representation). Individual elements are shown as nodes with the relationships between the elements shown as connecting lines. Classically, the information content is kept to a minimum with nodes showing element names and perhaps number, type, or class (although any information can be displayed, as desired). The emphasis in a hierarchy diagram is on interrelationships, with connecting lines frequently labeled to clearly communicate the nature of the relationship between the elements unless the diagram only shows composition (a pure hierarchy of requirements, functions, or components).

Individual nodes can be expanded or collapsed to show additional relationships or hide additional detail as desired to enhance the communication value of the diagram. When a node is collapsed, a black square is placed in the upper-left corner of the node as an indicator that there are more relationships which have not been shown. When an element occurs multiple times on the same diagram (as with "Workstation" in the sample hierarchy diagram), a black square is frequently placed in the upper-right as a cue to the reader.

Because of its classic format and the absence of any specialized symbology, the hierarchy diagram is well-suited for all types of audiences. The information content is intentionally kept low to maintain focus on the interrelationships between system elements – composition, traceability, or both. At its core, the hierarchy diagram represents a generic visual query with no defined semantics.

Requirement Diagram

The requirement diagram is a SysML extension of the classical hierarchy diagram standardizing the representation of key aspects of requirements – notably decomposition into child requirements and traceability to system elements that satisfy or verify the given requirement. To convey greater information, diagram nodes often show the element description. As a result, requirement diagrams quickly become quite large and therefore are frequently limited to display context for just a handful of requirements. Recognizing this, requirement diagrams are frequently complemented with a tabular representation.
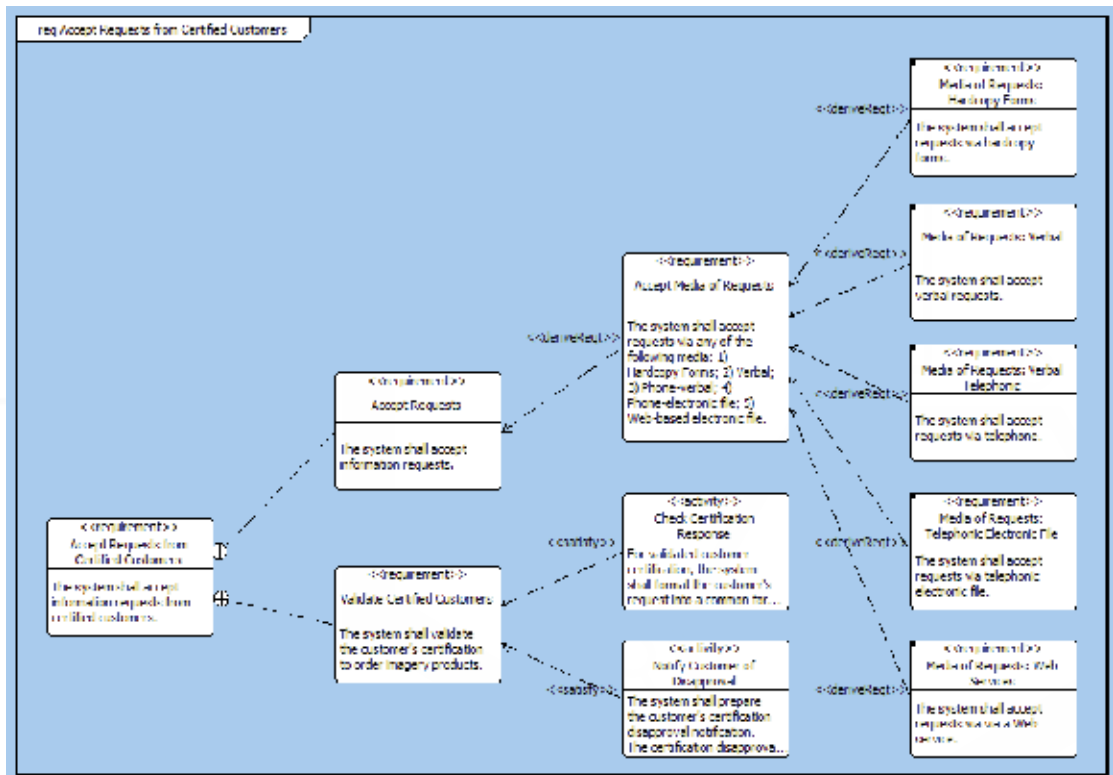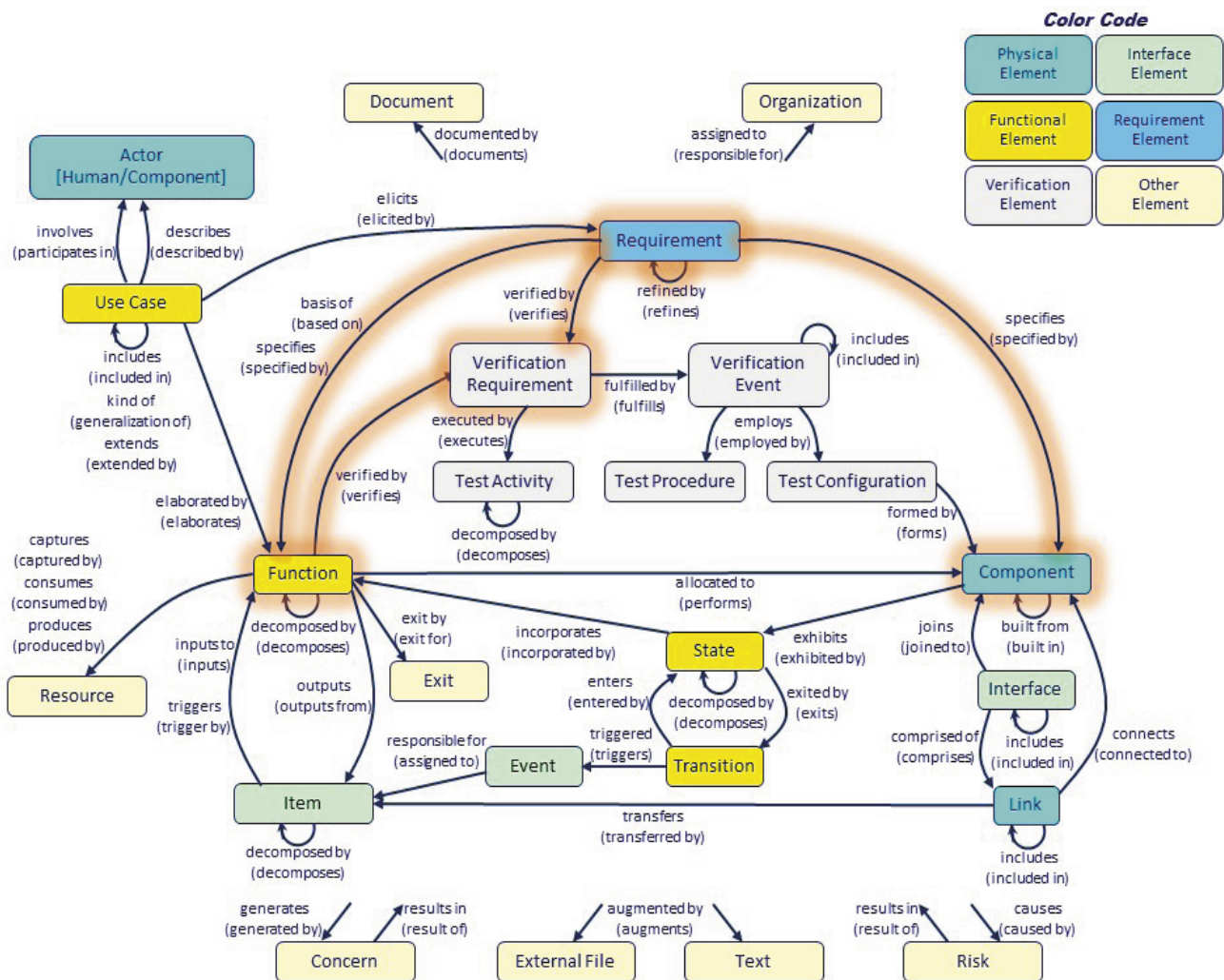
Level of Detail: Medium
Audience: System/ software engineers
Content: Names, relationships, and descriptions
Use: Context for limited set of requirements

Mapping for the Requirement Diagram

A node on a requirement diagram represents a given element and shows its type, name, and description. The lines connecting nodes represent relationships with the requirement diagram standardized to represent the following semantics:

- A circle with a plus sign (or, less frequently, the label <<containment>>) reflecting the partitioning of a complex requirement without the addition of meaning (e.g., the decomposition of a compound requirement into its atomic statements);
- <<deriveRqt>> reflecting decomposition into a more detailed requirement where analysis was used to arrive at the new requirement statement;
- <<satisfy>> reflecting an assertion that the model element (generally an activity or a block) satisfies the requirement; and
- <<verify>> referencing a test case or other aspect of verification.

The enhanced symbology of requirement diagrams serves those who need to understand how the requirements are satisfied and provides an easily traceable view of the relationship of the requirements to the applicable activities. However, that same symbology can be an inhibitor when communicating with those who have not been trained in the notation. Frequently, it proves most effective to use hierarchy diagrams with general audiences and equivalent requirement diagrams with systems and software engineers.

Additional Requirement Views

Requirements are textually based and can be represented in other, more textual, views. These include tables (such as requirements lists, traceability matrices, and verification matrices) and specifications of various configurations tailored to the needs and uses of the modeler. It is important to note that these representations are views as much as any diagram. Properly implemented, they are queries of the underlying system model that extract the desired information and then present it in a structured textual format with the information content and layout tailored to the communication purpose and the corresponding audience.

Level of Detail: High
Audience: General
Content:
Requirements, properties, and relationships
Use: Requirement lists; traceability matrices; verification matrices

Tables

Level of Detail: High
**Audience:** General (including contract officers)
Content: System or subsystem requirements
Use: Textual representation of requirements – generally compliant with a specific document format – used for milestone reviews and transmission across contractual boundaries matrices

3.3.7 Customer-Workstation Interface

Description:

This Customer-Workstation Interface shall handle all interactions between the Workstation subsystem of the Geospatial Library and its Customers.
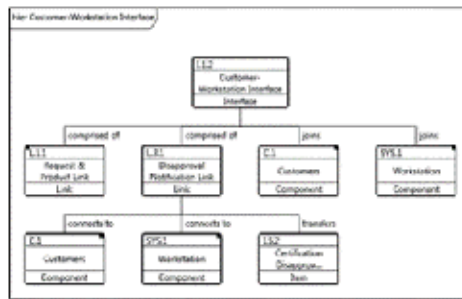
3.3.7.1 Disapproval Notification Link

Description:

The transport mechanism shall be for delivering certification disapproval information from the Geospatial Library's Workstation to the Customers.

Direction: in

Table 10. Disapproval Notification Link Items

Items	Description	Attributes
Certification Disapproval Notification	Customer certification disapproval is returned to the customer.	Type: Mixed

3.3.7.2 Request & Product Link

Direction: inout

Table 11. Request & Product Link Items

Items	Description	Attributes
Collection Products	The collection product is the material provided to the customer in response to an imagery product request.	Type: Mixed Size: 96000
Information Request	The information request is the request for imagery products from a customer. The request may be in a variety of formats and on a variety of media.	Type: Mixed Size: 3600

ical Image Management
ystems Interface Description

nterconnection Table

	Resources Exchanged	Destination
es all the ectors.	collector data [I.3.1] *type* = Mixed, collector tasking [I.2.1] *type* = Digital	Collectors [C.2]
es all the tomers.	collection products [I.4.1] *type* = Mixed, estimated delivery schedule [I.5.1], information request [I.1.1] *type* = Mixed	Customers [C.1]
		Tracking Software [ES.2] *Distribute Service [ES.2] External Services [ES]*

ated Element Definitions

Description
Component

tem, collectors, is responsible for acquiring data to support the information but does not reside in the inventory. There are multiple collectors, of d capabilities.

agement System is intended to serve as a means to demonstrate the use of n engineering support tools. As defined, this demonstration system accepts ery information, determines the best way for the system to respond to the provides the requested information to the requestor. In the process of uested information, the system may generate tasking orders for a set of lectors.

Command Center Subsystem [SYS.1.1]

Specifications

Vitech

Representing the Journey from Requirements to Behavior

Spider Diagram

The spider diagram provides a complete contextual view of a set of elements and their interrelationships. Unlike a hierarchy diagram, each entity is represented once and only once. In addition, the free-form presentation does not artificially imply a hierarchical relationship that may not exist. The result is an extremely powerful representation – neither traditional nor SysML – for analysis and communication as you continue the engineering journey through systems design.

Level of Detail: Low
Audience: General
Content: Names and relationships
Use: Contextual view of objects of interest with no implied meaning

As with a hierarchy diagram, a spider diagram is based on the combination of a root element and a set of relationships. The element defines the starting point for the diagram. The set of relationships identifies which links to traverse when building the diagram. The relationships (potentially multiple) between the elements are then shown as connecting lines. The lines of convergence and divergence help identify critical aspects in the model.

Like the hierarchy diagram, the spider diagram is a generic visual query with no implied semantic meaning. As a simple node and line diagram with no special symbology, it is easily accessible by a broad audience. Good communication in a spider diagram is often heavily influenced by the creator. As a free-form diagram, it should be configured in a way calculated to display the relationships depicted in an understandable and informative manner.

Use Case Diagram

Classically used to help elicit requirements from stakeholders, use case diagrams describe the functionality of a system from the user perspective. The use case diagram is a graphical representation of actors (humans), blocks (components), and use cases. As such, they are also a very effective bridge in the systems engineering journey from requirements to system behavior.

Level of Detail: Low
Audience: General
Content: Use cases and corresponding actors (components)
Use: High-level tool to elicit requirements; bridge from requirements to system threads
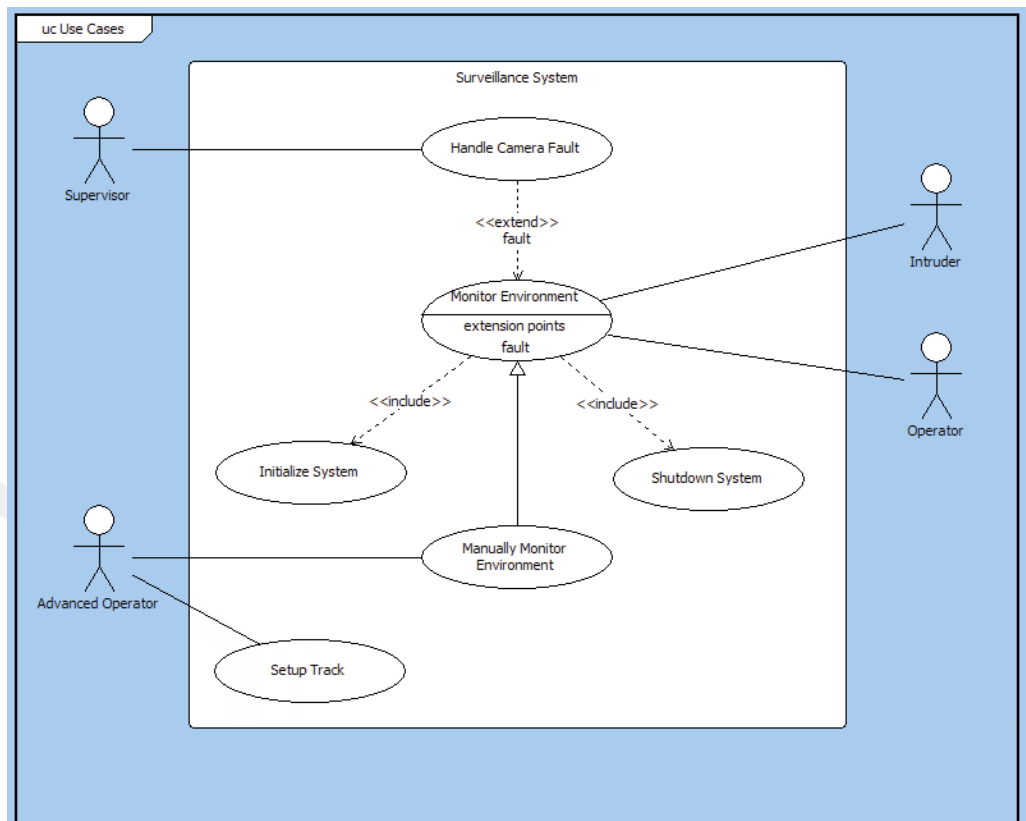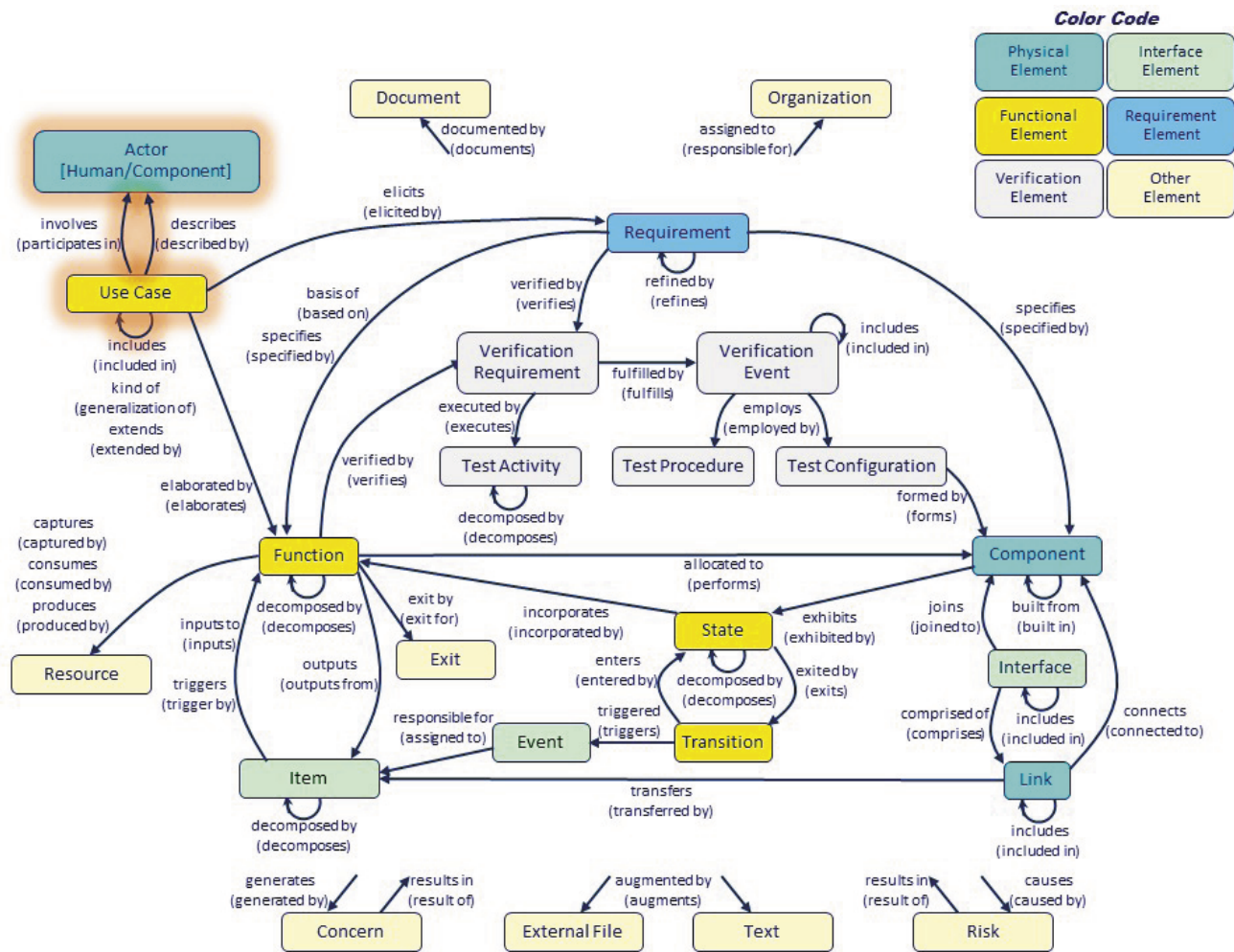
Mapping for the Use Case Diagram

On a given use case diagram, the use cases are drawn as ovals within a containing frame. The label at the top of the frame represents the system or component described by the diagram. The use cases shown on the diagram can be related to other use cases with three relationships:

- **Inclusion** (designated with the label <<include>>) which is a parent-child relationship between use cases with the child use case shown at the end of the arrow.
- **Extension** (designated with the label <<extend>>) which reflects the expansion of the main use case under specific conditions (shown under the <<extend>> label). In the example, the Handle Camera Fault use case extends the Monitor Environment use case under the fault condition.
- **Classification** (designated with the standard UML / SysML unfilled arrowhead decoration) representing a generalization / specialization relationship between use cases. In the example, Manually Monitor Environment is a specialization of the Monitor Environment use case.

Actors and blocks are classically shown around the boundary of the diagram. These are the humans and system components involved in the use case. Human actors are almost always represented by a stick figure. System components (hardware or software) can be shown either as a rectangular block or a stick figure. Because different teams follow different practices, you should be careful about drawing inferences as to whether an actor is a human or a component based upon the graphical representation. Actors and blocks are connected to the use cases with which they are involved by unlabeled lines.

There are two cautionary notes when dealing with use cases. First, the meaning of "use case" has somewhat drifted over time. An original use case as conceived by renowned computer scientist Ivar Jacobson is more analogous to a sequence of activities or a behavioral thread. Today, the use case is more the title or container of that scenario, which is subsequently elaborated by a detailed behavioral thread. Second, though the use case diagram is the most frequent representation of use cases, the diagram in isolation is largely worthless. The greater value comes from capturing at least the pre-conditions and post-conditions associated with each use case. These become the essential context and ensure that various team members are communicating effectively as they leverage use cases to better understand the system and begin the design process.

While the notation and symbology of some SysML diagrams can prove intimidating for general audiences, this is not the case for the use case diagram. Whether it is the relatively lightweight nature of the diagram or the disarming nature of stick figures, the use case diagram is a very effective way of representing use cases and the related actors and blocks largely independent of the composition of the audience. This makes the use case diagram an ideal high-level view to support requirements elicitation sessions to better understand the problem as well as design sessions to bridge from requirements to system threads.

Representing Behavior

Having set the stage for transitioning from requirements to behavior, it is now time to consider how the system behavior (logical architecture) is represented. The building blocks of the behavioral architecture are the activities / functions that are based on the requirements. These are connected into a behavioral flow through the use of control constructs. We will first consider the control constructs and then see how these are represented in combination by a variety of diagrams.

Representing Behavior — Control Constructs

System behaviors are represented through a combination of control constructs that define how logical (behavioral) control flows from one function to another. Regardless of the problem and the domain, at a system level, behavior can be represented by a combination of the following executable constructs.

In the interest of completeness, we are including a discussion of the control constructs that determine the logical flow of the system behavior. For those not familiar with modeling logical architectures, it is helpful to understand the representation of logical control. For those who are already familiar with this concept, this discussion may serve as a convenient reference.

Sequence

The simplest construct is the sequence. When drawn horizontally in a diagram, control is represented as flowing from left to right.
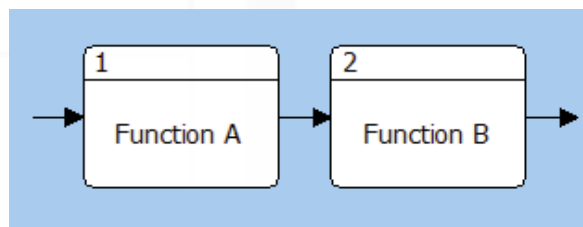
In a sequence construct, control enters the first function in the sequence – in this case, Function A. When the first function finishes its execution, control is passed to the next function in the sequence (Function B). In this simple construct, the completion of Function A enables the execution of Function B. (Function B can never begin before Function A completes.) But, a simple sequence is hardly the most sophisticated logic that can be modeled.

Parallel

The next construct is the parallel. In contrast to sequences of functions where the next entity cannot be executed until the previous one completes, the parallel construct designates that the parallel branches can be executed concurrently (even though they may interact through triggers). Though termed a parallel construct, the better description is a "don't care" sequence – as the creator of the model is maintaining flexibility in the system design by avoiding specifying required sequencing.
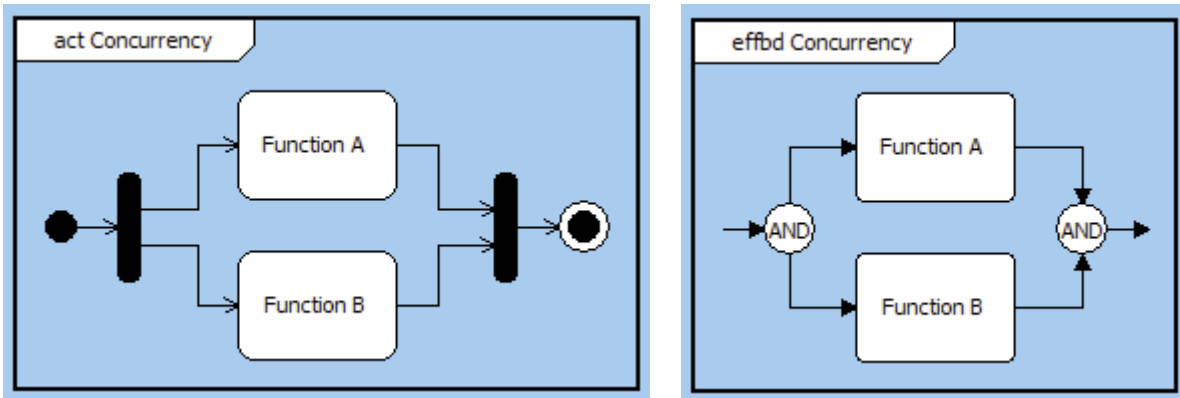

A parallel construct consists of a fork node followed by separate branches that rejoin and terminate at a matching join node. The construct can contain any number of branches, and each branch can contain any number and combination of functions and control constructs.

The construct cannot be exited (from the join node) until all branches have completed their processing. Control is then passed to the next function or construct after the parallel construct.
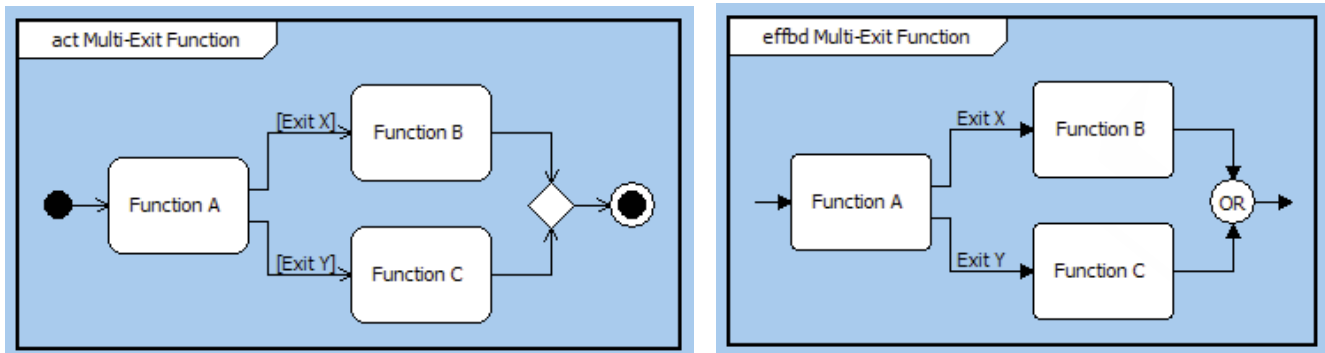
Select

A select construct consists of a decision node followed by multiple branches that rejoin at a node. As in a parallel construct, there may be any number of branches, and each branch may contain any number of functions and control constructs. But in contrast to a parallel construct in which all branches are executed, with a select construct, only one branch is executed. Thus, the select construct is an exclusive OR.
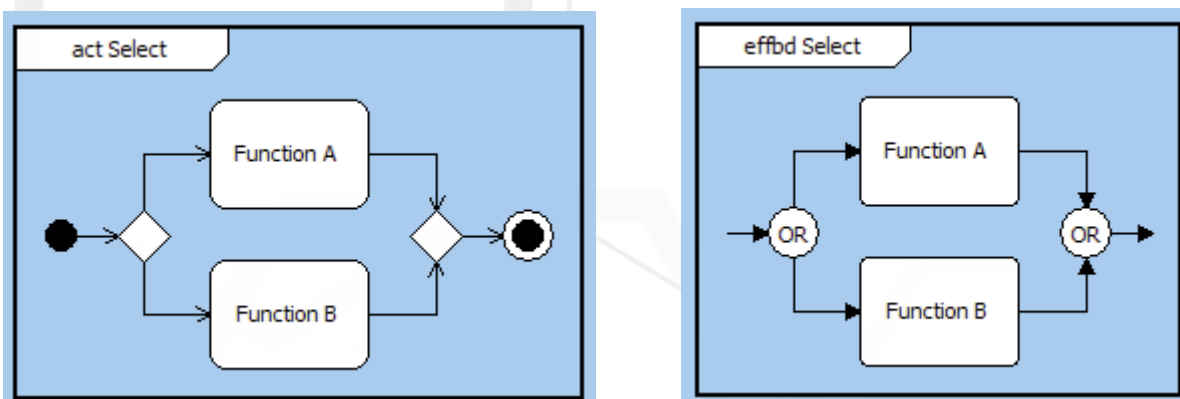
Due to system or contextual considerations, one branch of the OR construct may execute more or less often than the other. For simulation purposes, in addition to descriptive annotations, each branch may be assigned a selection probability based on this likelihood (if known) to determine how often it is executed during the simulation. If there are no branch selection probabilities, each branch is assumed to have equal likelihood of being selected for execution.

Multi-Exit Function

A multi-exit function is a control construct where multiple branches exit from a function and rejoin at a closing decision node. Like a select construct, only a single branch will be selected. (The construct operates as an exclusive OR.) What differs is the manner of selection.

In a multi-exit function, each branch is labeled with the name of its associated exits and can contain any number of functions and control constructs. At the conclusion of execution, the logic within the main function (Function A above) selects the exit branch for execution either via logical statement or via a corresponding exit node within its decomposition.

Exit Node

An exit construct terminates execution of the process and returns control to the parent activity. In the example below representing the decomposition of activity Multi-Exit Function, when the exit node is reached (the target node in an activity diagram or the EXIT node in an enhanced function flow block diagram), the behavior of Multi-Exit Function is completed and whatever function or construct follows Multi-Exit Function in the parent process would be enabled.
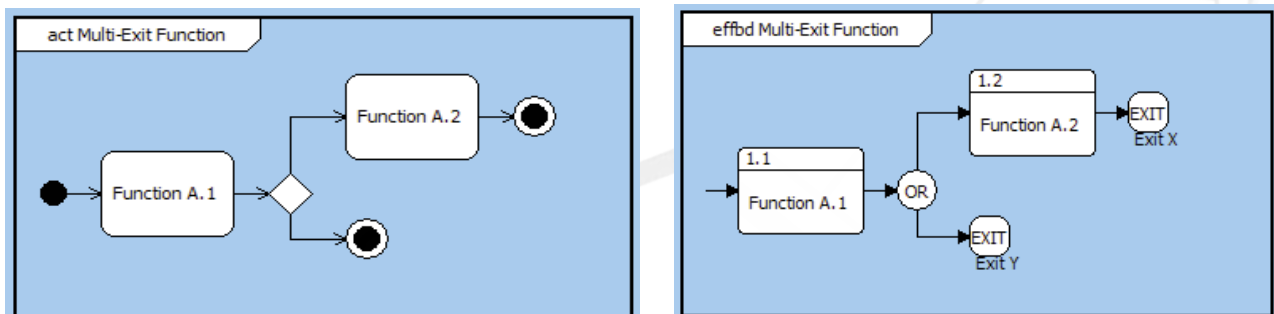
Exit nodes establish the mapping between the completion of the decomposition behavior and the exit branches of the parent function. There should be at least one exit node in the function decomposition for each exit branch for the function. The name for the corresponding exit branch is shown below each exit node icon.

Loop

A loop construct repeats a sequence of functions or constructs until a logical condition is satisfied.
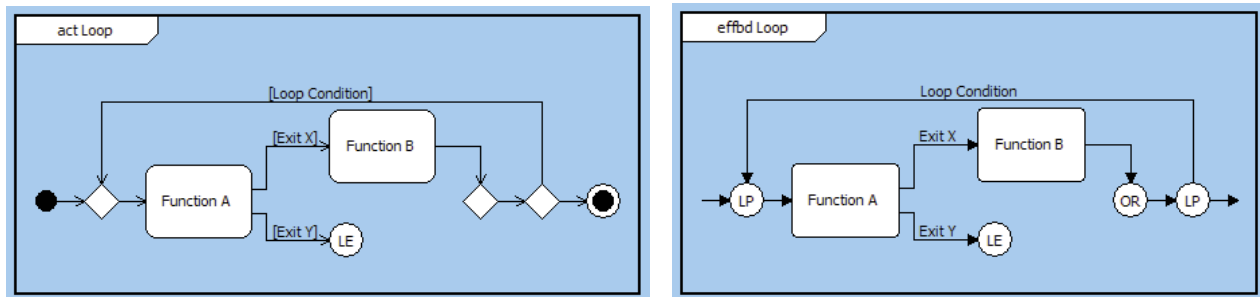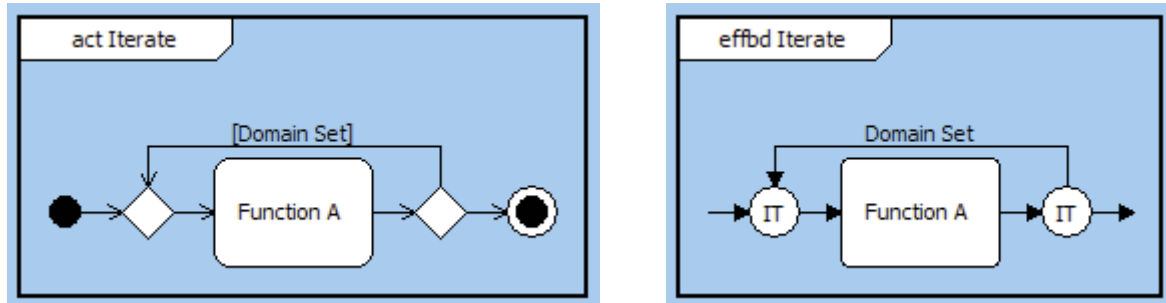

A loop construct consists of a pair of decision nodes that enclose a branch and are connected with a loop-back line. The branch can contain any number of functions and control constructs. These will be repeatedly executed in sequence. The branch will typically contain a loop exit construct to conditionally exit the loop construct. A descriptive Boolean annotation is generally provided for each loop construct and is displayed above the loop-back line.

Loop Exit

The loop exit construct provides the mechanism for exiting a loop. When the loop exit construct is encountered, the innermost loop is immediately terminated, enabling the construct or function following the loop.

Iterate

An iterate construct is similar to a loop construct in that it repeats a sequence of functions or constructs. Unlike a loop, which is controlled by a logical condition, an iterate construct is controlled by a domain set which specifies the number of iterations.
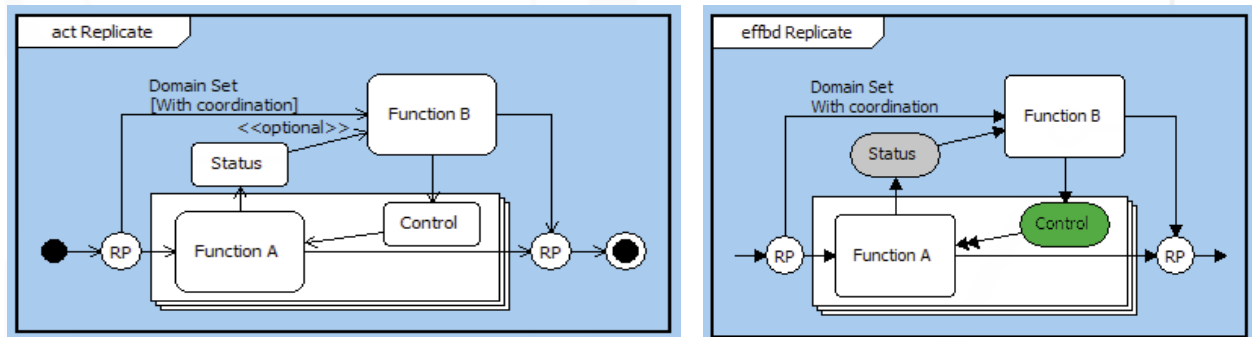

An iterate construct consists of a pair of decision nodes that enclose a branch and are connected with a loop-back line. The name of the specified domain set which determines the number of iterations (either a count, a frequency, or a specified set of objects) is shown above the loop-back line. The branch can contain any number of functions and control constructs. These will be repeatedly executed (in sequence) as specified by the domain set.

Unlike the loop construct in which behavior on the main branch is guaranteed to be executed at least once, the main branch of an iterate may not be executed depending upon the domain set.

Replicate

The replicate construct is a shorthand notation for identical processes that operate in parallel.
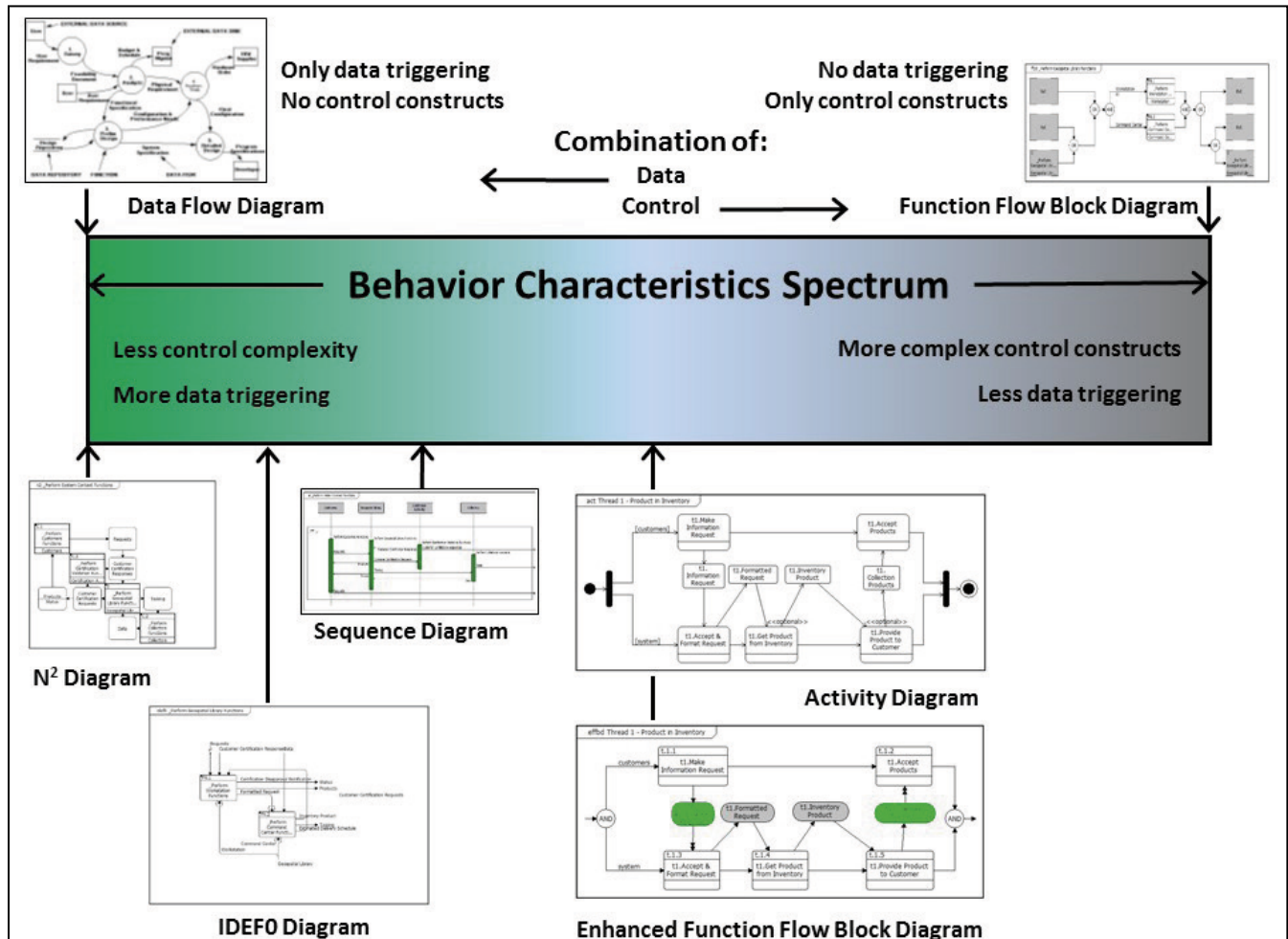

A replicate construct consists of a pair of nodes labeled "RP" that enclose a main branch and are connected with a coordination branch. This coordination branch is labeled with the name of the associated domain set.

The main process logic is shown on the main branch. This logic will be repeated for each occurrence as specified by the domain set. The coordination between these processes is handled via the coordination branch. Coordination includes assigning items to specific processes, inter-process communication, and the instantiation or termination of process branches. An example of a situation handled by the replicate construct would be a supermarket in which multiple checkout lanes support shoppers (represented by the functions on the main branch), and a manager supports the various checkout lanes as required (represented by the functions on the coordination branch).

Representing Behavior — Diagrams

In his original paper, Jim Long noted that although they evolved largely independently to support varied analysis for different domains and audiences, the rich set of behavioral representations is fundamentally linked by a few primary concepts. Composition captures the parent-child aspect. Control reflects the logical structure of behavior (the constructs previously noted). Data flow reflects the transfer of items between processes and the corresponding components. Triggers indicates the special nature of certain item relationships which serve to initiate activities and synchronize processes.

Reflecting upon these concepts, the wealth of representations can be plotted along a single spectrum reflecting the key differentiation in diagram content – data flow and triggering vs. complete representations of control. On the left end of the spectrum are representations that focus exclusively on data flow and triggering (e.g., data flow and N2 diagrams) with no representation of structure. On the right end of the spectrum are representations of control flow (e.g., function flow block diagrams) with no representation of data. Falling in the middle of the structure are diagrams that represent a blend of these aspects at different levels of fidelity with the activity diagram and enhanced function flow block diagram fully reflecting both data and control dimensions of behavior.

Only data triggering
No control constructs

No data triggering
Only control constructs

Combination of:
Data
Control

Data Flow Diagram

Function Flow Block Diagram

Behavior Characteristics Spectrum

Less control complexity

More data triggering

More complex control constructs

Less data triggering

N^2 Diagram

Sequence Diagram

Activity Diagram

IDEF0 Diagram

Enhanced Function Flow Block Diagram

Note that more content does not necessarily equate to a better representation. As with systems themselves, the measure of goodness is "fit for purpose." Choosing the right representation for a task is a function of the kind of information needed (data flow, control flow, or both) and the audience that must successfully interpret the diagram.
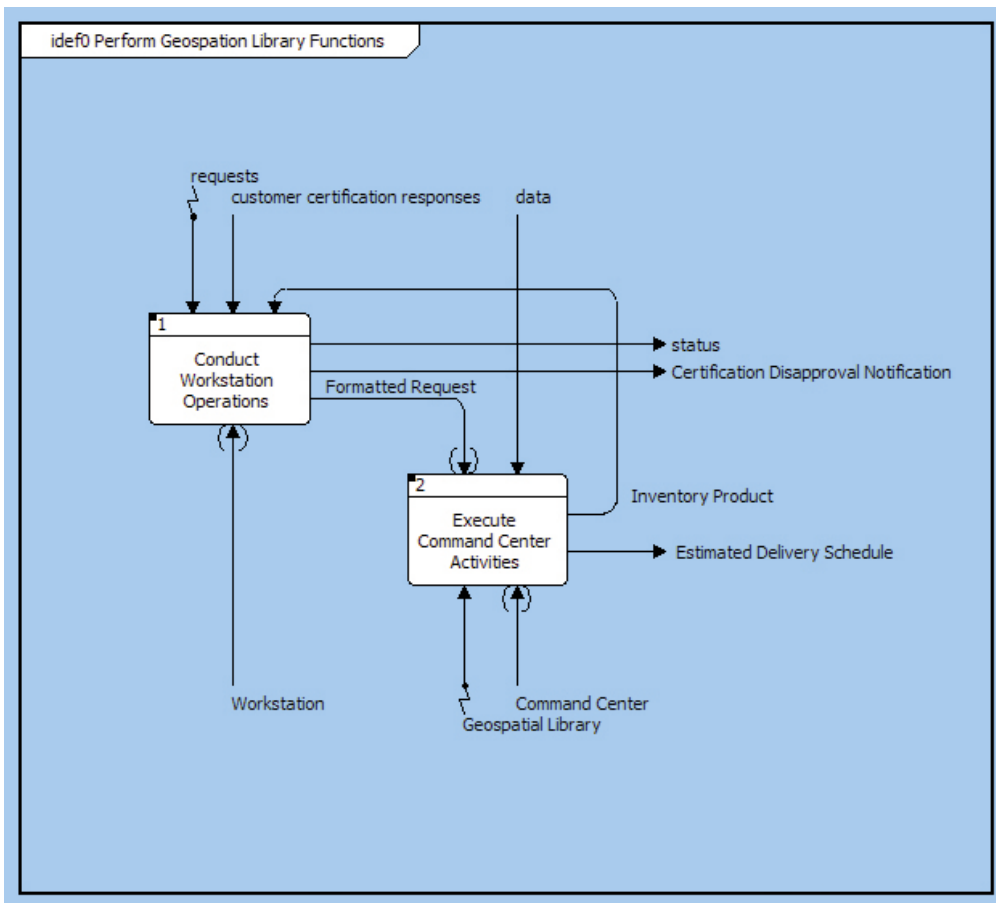
IDEF0 Diagram

Originally specified by National Institute of Standards and Technology (NIST) Standard FIPS-183, the IDEF0 diagram presents an integrated picture of the inputs, control, outputs, and mechanisms (ICOM) for a function's decomposition. The IDEF0 diagram displays a great deal of context information on the interrelationships of decomposition and implies sequencing, but displays no actual control logic / structure of the decomposition.

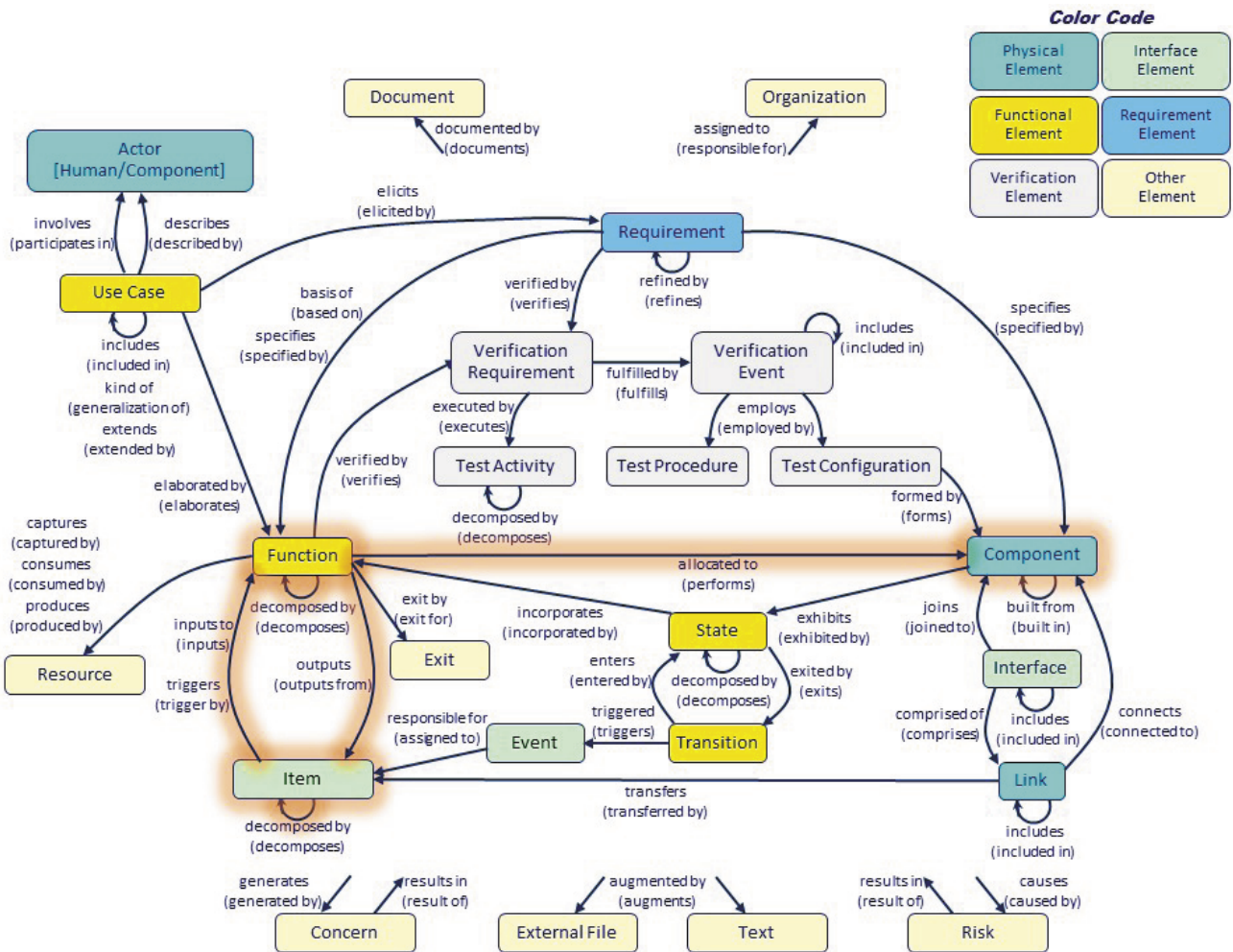**Level of Detail:** High
Audience: Traditional SEs and process engineers
Content: Data flow, triggering, and allocation
Use: Analysis of data flow with diagnostics of inconsistencies across behavioral decomposition requirements to system threads

Color Code

Physical Element	Interface Element
Functional Element	Requirement Element
Verification Element	Other Element

Mapping for the IDEF0 Diagram

On an IDEF0 diagram, the subfunctions are shown as nodes on the main diagonal. For each functional node:

- Inputs enter on the left. These can either come from the edge of the diagram (external inputs) or from another function on the diagram.
- Controls (triggering data) enter on the top. These can either come from the edge of the diagram (external triggers) or from another function on the diagram.
- Outputs exit on the right. Outputs can either connect to another function on the diagram, exit to the edge of the diagram, or both (representing an output that is input to / triggers both internal and external functions).
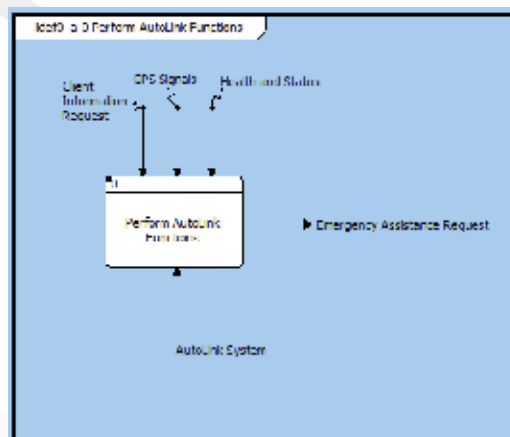- Mechanisms (allocation) enter on the bottom.

The ICOM representation on an IDEF0 diagram has two special aspects not shown on other behavioral representations:

- Branching – Individual inputs, control, outputs, and mechanisms (ICOM) arrows fork and join on the diagram. An arrow forking represents the relationship between a parent element and a child element. Two or more arrows joining represents the relationship between a child element and a parent element. In this way, the IDEF0 diagram elegantly represents multiple levels of hierarchy in items and components, bringing additional clarity to the model.
- Tunneling – Tunneling is a technique within IDEF0 to hide an ICOM in part of the model. The use of parentheses around either the head or tail of an arrow depicts a tunnel in IDEF0. A parenthesis around the head of an arrow that is entering a function box indicates that the ICOM associated with that arrow will not be seen on the decomposition of that function. If the ICOM does reappear, it will have parentheses around its tail.

Though the IDEF0 diagram has largely fallen out of favor in systems engineering, it still finds use with senior systems engineers and maintains a strong following within the process engineering community. The simple box and line representation is widely accessible by diverse audiences as long as the diagram does not become overloaded with too much ICOM and too much forking / joining of ICOM. The IDEF0 diagram does present unique visual diagnostics of inconsistencies across behavioral decomposition. For this reason, it remains a useful representation and is frequently used in the training of new systems engineers.

IDEF0 A-0 Diagram

The IDEF0 A-0 variant (pronounced "A minus zero") provides a contextual ICOM view of a function at any level in your behavioral hierarchy. As such, it is an ideal "functional context diagram" at any level and is often the first behavioral representation drawn alongside the system context diagram.

Though a context diagram that shows only the functional node itself, the A-0 variant follows the same ICOM rules as the IDEF0 diagram: inputs enter on the left, controls enter on the top, outputs exit on the right, and mechanisms (allocation) enter on the bottom.

Because it displays all functional context information in a simple form, the IDEF0 A-0 remains a uniquely valuable representation in the suite of behavior representations. No other diagram conveys the complete functional interface for an activity in a single picture. The lack of special symbology – beyond recognizing the ICOM standard for locating arrows – makes the IDEF0 A-0 ideal for communicating the functional context and functional interfaces with general audiences.
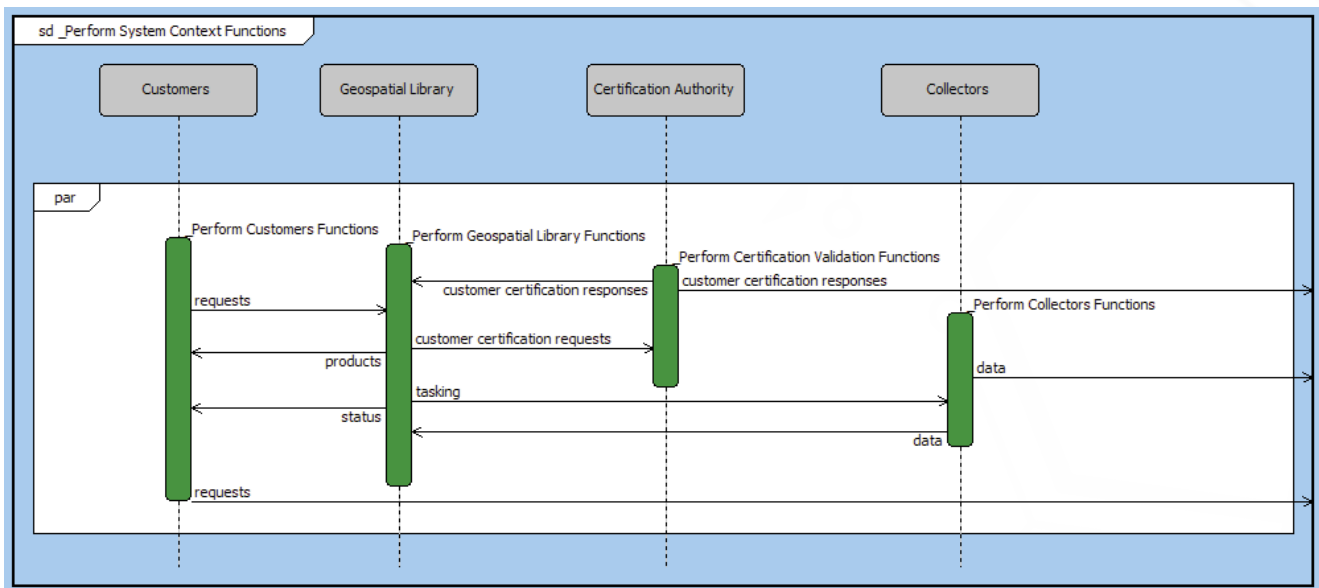
Sequence Diagram

The sequence diagram emphasizes the interaction between collaborating parts of a system. Previously known as a function sequence diagram, the modern sequence diagram is part of the SysML specification. By minimizing the representation of control flow and representing allocation of functions along lifelines, the sequence diagram enables you to focus on triggering data and the resultant flow of control between components.
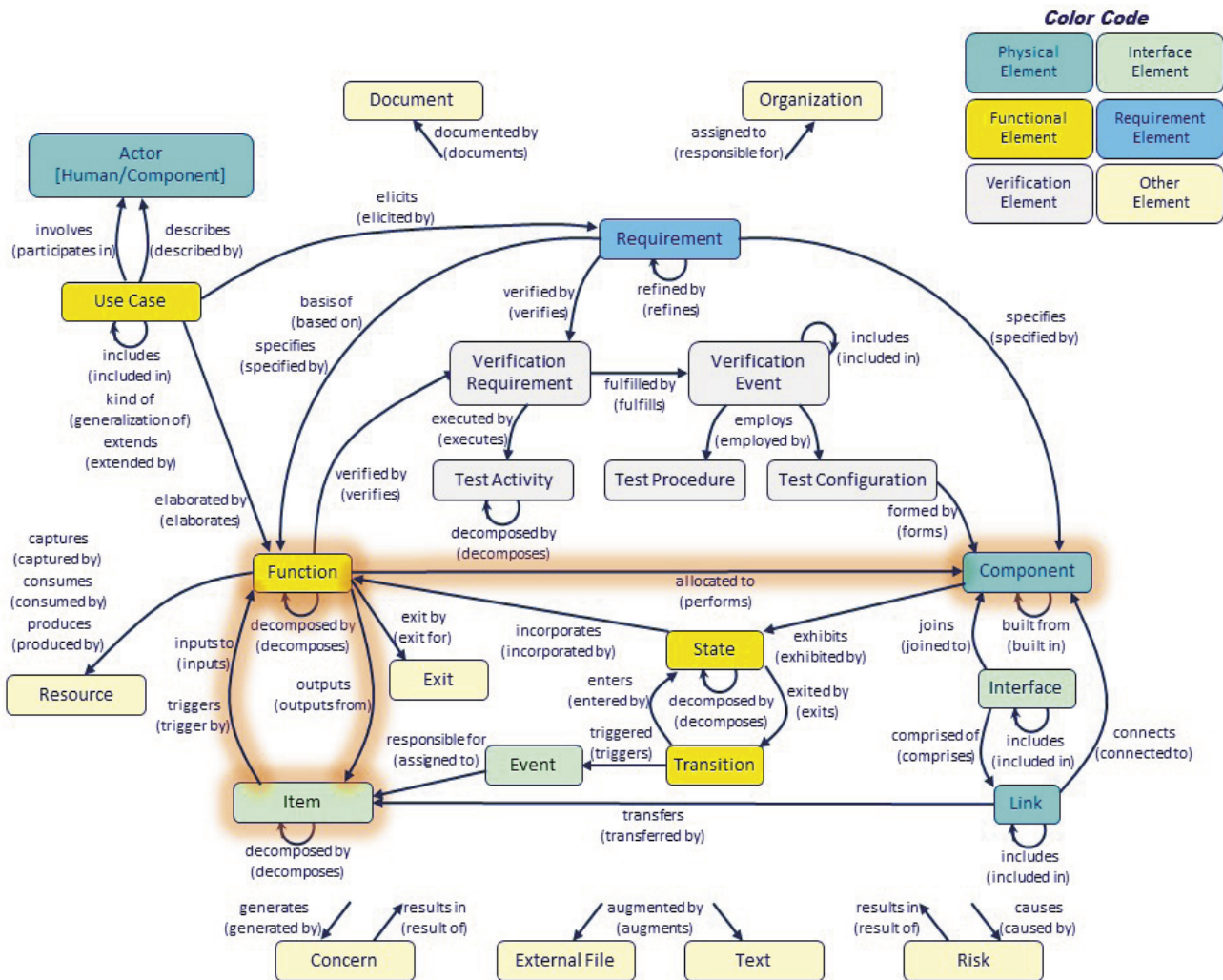
Level of Detail: Medium
Audience: General
Content: Specification of sequence (but not control), allocation, and triggering
Use: Initial capture of threads when focusing purely on triggering aspects; communication with software engineers

Mapping for the Sequence Diagram

The interacting blocks involved in a function's decomposition are displayed at the top of the diagram. Lifelines are shown as a dashed line extending downward from each interacting block. Individual function nodes in the decomposition are placed along the corresponding vertical lifeline (depending upon their allocation) in the sequence in which those functions occur. Often, these function nodes are unlabeled to focus attention on the interaction between the blocks.

Control constructs (termed "interaction operators" in the language of the sequence diagram) are displayed in a lightweight manner and enclose the nested functions and constructs. This representation is much less complete than the control representation on an activity diagram or enhanced function flow block diagram, but it conveys essential nesting.

The arrows on a sequence diagram represent messages sent and received between interactions. These can be synchronizing messages (triggers) or simple data exchanges (inputs). Often, basic inputs (data stores) are not shown on the sequence diagram in order to focus on interactions which synchronize activities across blocks. An arrow exiting the node is an output that is input to or triggers another function. Arrows entering from the left edge of the diagram are external messages that originate outside of this decomposition. Arrows that exit the right edge of the diagram are outputs that are consumed elsewhere in the system model.

Given its long history of use and rather simple semantics, the sequence diagram is an effective representation when used with any audience to convey message passing and interactions between systems or blocks. The sequence diagram is particularly useful in developing logical threads to elaborate use cases. (As logic becomes more complex, complete sequence diagrams often become overloaded.) The sequence diagram is frequently a diagram of choice in communicating behavioral dimensions with software engineers, but it must be used with care. The diagram is an incomplete specification of the logical architecture and should always be used in conjunction with a more complete representation (classically an activity diagram) when used as a specification for implementation.
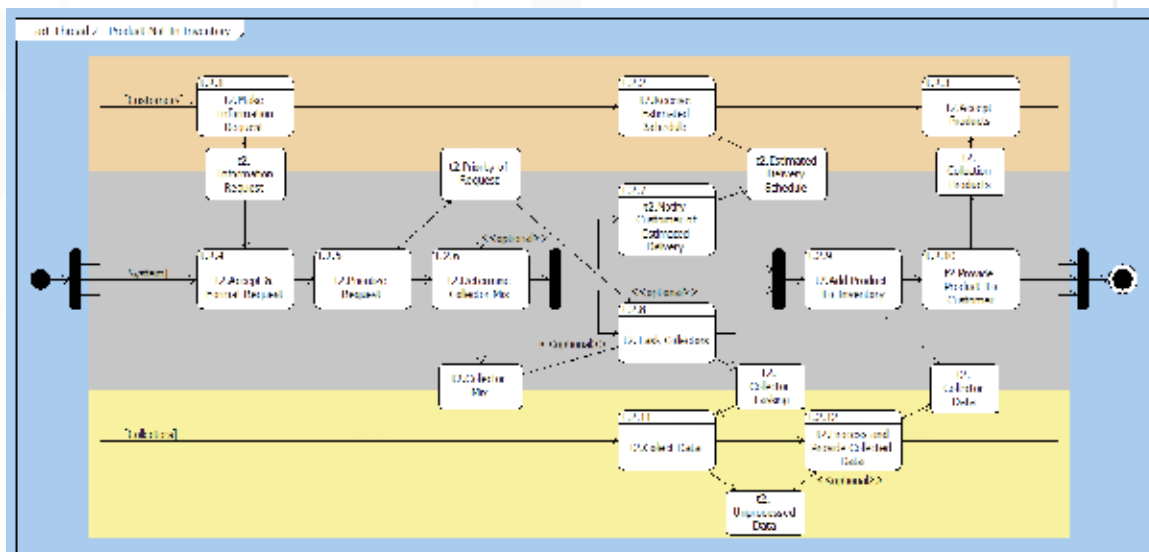
Activity Diagram

The activity diagram (and the Enhanced Functional Flow Block Diagram, or EFFBD, its cousin in traditional representations) are the most complete representations of behavior. The activity diagram unambiguously represents the flow of control through sequencing of activities and control constructs as well as the data interactions overlaid to present a more complete picture.
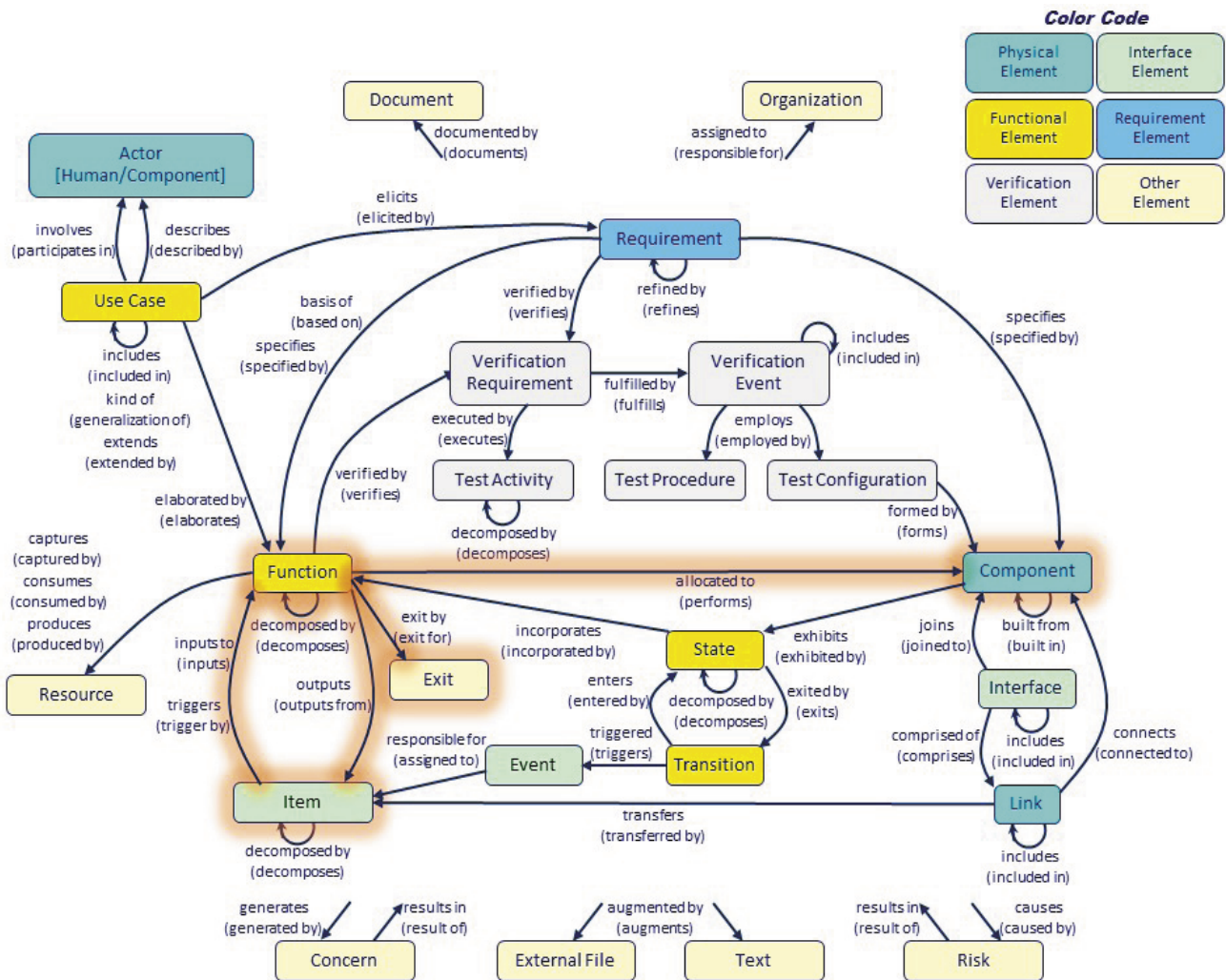
Level of Detail: Highest
Audience: System and software engineers
Content: Composition, triggering, and allocation
Use: Full specification of system behavior; best at lower levels of decomposition (design view)

Mapping for the Activity Diagram

Control classically flows from left to right (when drawn horizontally) or top down (when drawn vertically). Rounded rectangles on branches represent activities or functions. Where an activity has a decomposition specifying greater detail, a "pitchfork" in the corner of the node indicates the decomposition is present.

As noted in the "Representing Behavior – Control Constructs" section, diamonds (decision nodes) and bars (fork and join nodes) represent control constructs upon which behavior is built. As each activity is completed, control flows along the branch lines to the next activity or control construct. Every construct has a precise definition that prescribes how control will be passed within the construct and when the construct itself will end. This structure results in a complete specification of control flow which itself is fully executable.

The rectangles on an activity diagram represent the items or the data interaction aspect of behavior. Where most behavioral representations focus on either control or data, the activity diagram (and the EFFBD) represent both aspects to provide the full specification of behavior. The activity diagram distinguishes between the two primary roles that items play:

1. Triggers control the execution of a function by their presence or absence. Triggers can be simple signals or actual objects. Items that trigger a function are drawn with a standard arrow to that function with no additional decoration.

2. Data stores are input to or output from a function with no control implications. Items that are input to a function are drawn with a standard arrow to that function with a label decoration indicating <<optional>> at the point of connection with the function.

To visually represent allocation, activity diagrams frequently display swim lanes. These bands are labeled with the name of the block or component which performs the activities drawn within that band. There are additional techniques for representing allocation – such as annotations on branches or footers on the activity nodes – but swim lanes are the most common approach.

The similarities between activity diagrams and EFFBDs are not coincidental. Not only do they address the same need for a more comprehensive representation of behavior, but the EFFBD notation was also used for both guidance and verification by the SysML team during the development of the activity diagram. The net result is a pair of closely coupled representations from which you can select to best meet your analytical and communication needs. Because of their representational similarity to UML diagrams, activity diagrams generally appeal to the software community while EFFBDs are often more easily understandable by process engineers, customers, domain specialists, and end users. Additional detail present on the activity diagram – such as the specification of ports – also makes the activity diagram an ideal representation at lower levels of decomposition when dealing with detailed design.

Enhanced Functional Flow Block Diagram (EFFBD)

A variant of the traditional function flow block diagram (FFBD), the EFFBD, like its SysML cousin the activity diagram, is a complete representation of behavior. EFFBDs unambiguously represent the flow of control through sequencing of functions and constructs as well as the data interactions overlaid to present a more complete picture. EFFBDs also display resources – the third critical aspect of executable behavior.
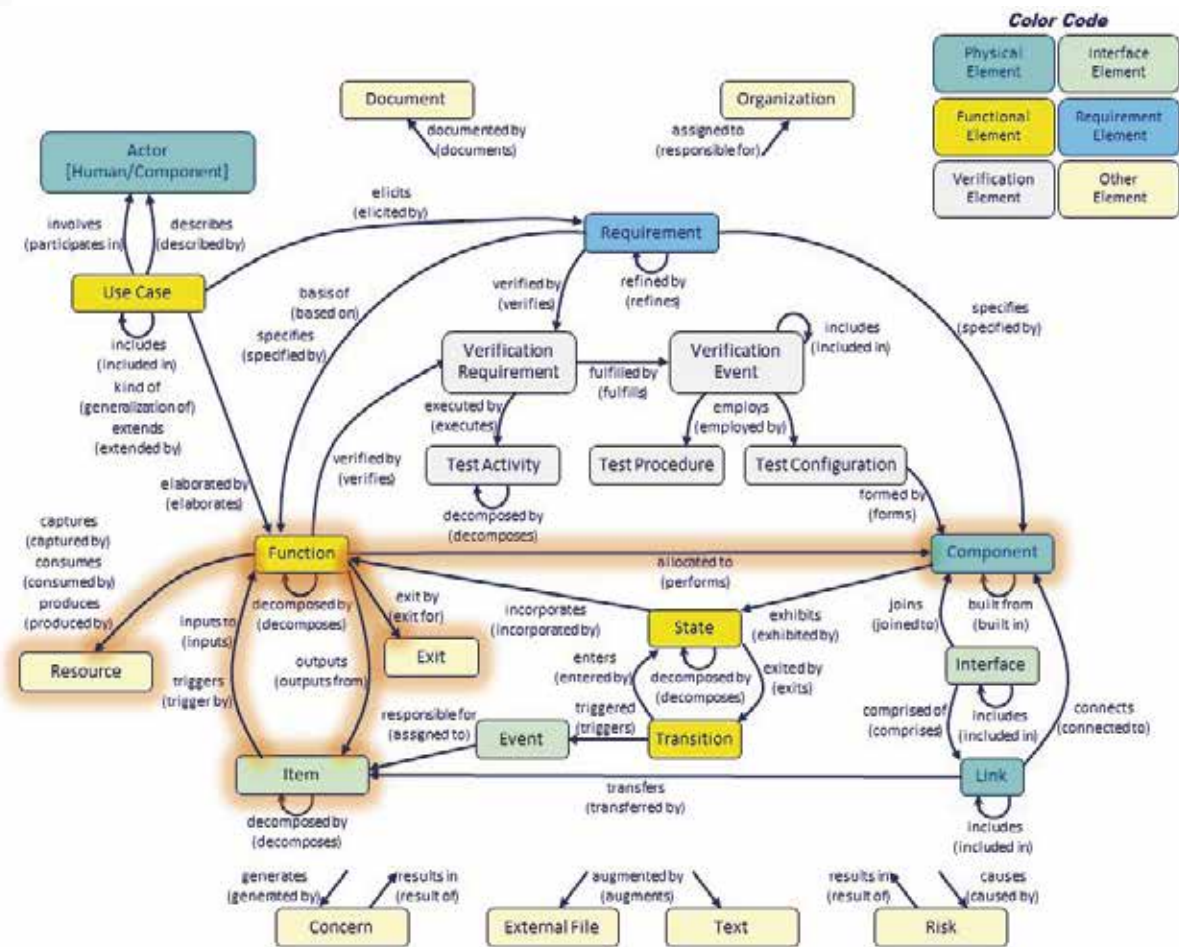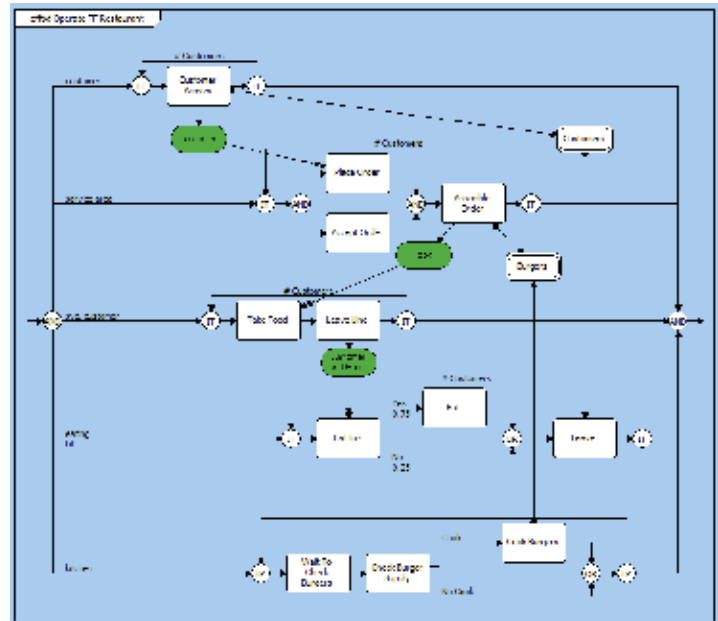
Level of Detail: High
Audience: Diverse audiences beyond system and software engineers
Content: Composition, triggering, resourcing, and allocation
Use: Full specification of system behavior; best at higher levels of decomposition (level 0, level 1, …) when dealing with broader audiences

Mapping for the Enhanced Function Flow Block Diagram (EFFBD)

Rectangular nodes drawn on branches represent functions. Circular nodes and branching structures represent control constructs – the building blocks of behavior. As a function completes execution, flow of control proceeds along branch lines to the next function or control construct. Each construct has a precise definition that prescribes how control will be passed within the construct and when the construct itself will end.

The rounded rectangles on an EFFBD represent the items or the data interaction aspect of behavior. The EFFBD distinguishes between the two primary roles that items play:

1. Triggers control the execution of a function by their presence or absence. Items that trigger a function are drawn with a double arrowhead to that function.

2. Data stores are input to or output from a function with no control implications. Items that are input to a function are drawn with a standard arrow.

Resources are also optionally displayed on EFFBDs. Resources are drawn with a double border to help distinguish them. Resources can be related to functions in three different ways:

1. Consumes – Resources that are consumed during a function's execution (electrical power, for example) are indicated with a half circle decoration on the resource and an arrowhead indicating the flow of resources into the corresponding function.

2. Produces – Resources that are produced during a function's execution (again, electrical power or perhaps fresh water) are indicated with a half-circle decoration on the function and an arrowhead indicating the flow into the resource.

3. Captures – Resources that are utilized during a function's execution and then released (a human operator responsible for overseeing a task, for example) are indicated with arrowheads at both the function and the resource.

A function begins execution when it has received all of its triggers, and its necessary resources have been acquired. If the flow of control has reached a function, but either the triggers or resources are not available, the function is said to be enabled but waiting. Obviously, this has notable impacts in the sequencing and synchronization of behavior as well as in the overall performance (how quickly the process completes) and whether or not it can complete at all due to live-locks and deadlocks.

Allocation is sometimes shown on EFFBDs via swim lanes. More frequently, branches are annotated or functional nodes are tagged to represent allocation.

A special aspect of some FFBD and EFFBD representations are reference nodes. Reference nodes reflect the context immediately surrounding this behavior. A function shown with a broken frame on the left edge represents the last function to complete before this decomposition begins (the source of control flow). A function shown with a broken frame on the right edge represents the next function to enable when this decomposition completes. When there is no previous or next function, the boxes are simply labeled "Ref." When a function appears multiple times in a system model or when the previous / next construct is complex, reference nodes can begin to branch, showing all of the paths into and out of a given function's decomposition. In this way, the reference nodes provide valuable context information.

With the heavy (but not complete) overlap between activity diagrams and EFFBDs, it is incumbent upon the presenter to choose the diagram carefully. In practice, this largely comes down to the composition of the audience. Whether it is the more classic feel of a flow chart or the absence of software-style decorations, the EFFBD is typically better understood and better accepted at higher levels of decomposition (level 0, level 1, etc.) when dealing with more diverse audiences. When working with those trained in SysML or UML, the activity diagram is the far better choice. Given the similarities in content and style, there is little value in engaging in a religious debate regarding the merits of one diagram over the other. Instead, suffice it to say that any communication that begins with "let me explain to you how to read this diagram" is poor communication indeed, as the audience is now focused on the form of the communication rather than the critical content.

N^2 Diagram

Largely overlooked these days, the N^2 (pronounced "N-squared") diagram represents the logical data flow for a system or system segment. The N^2 diagram has no representation of control constructs or sequencing. It displays only the data dimension of the behavior model and helps focus attention on this subset of the model. In particular, this is helpful in partitioning and allocating the system behavior to manage internal and external interfaces.

Level of Detail: Low
Audience: General
Content: Data flow with possible inclusion of allocation
Use: Understanding of data flow and implied interfaces; clustering analysis
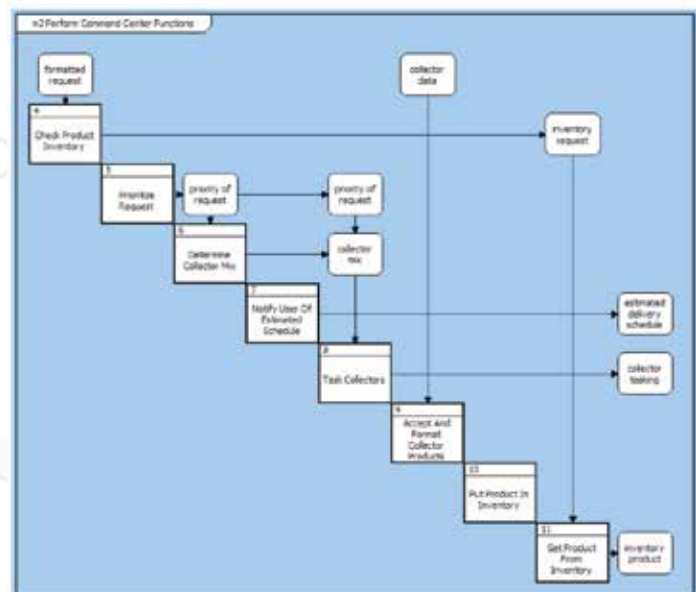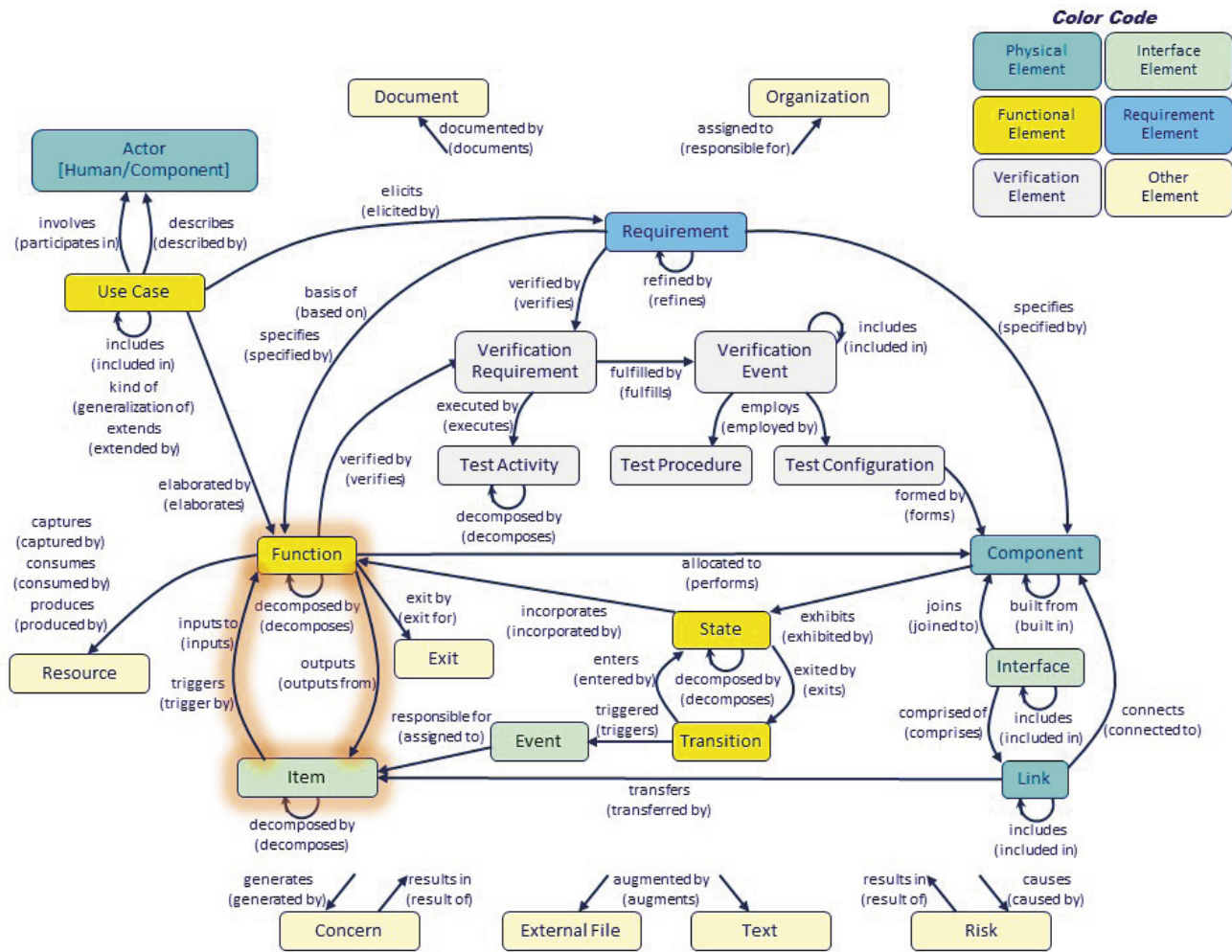
Mapping for the N^2 Diagram

On a functional N^2 diagram, the subfunctions are shown on the main diagonal forming an N x N matrix of cells. Items that are output from a function are shown in the function's row. Items that are input to or trigger a function are shown in the function's column. (There is no notational difference to visually differentiate an input from a trigger.) If multiple items are output from and input to / trigger the same pair of functions, multiple items will be shown in the same item cell. If no items are exchanged between a pair of functions, the item cell will be empty.

The N^2 diagram can be extended to display external inputs and outputs which represent external interfaces for this function. Items appearing in the top row are inputs / triggers for the function that are output by a function not displayed on this diagram. Similarly, items in the right-hand column are outputs that are input to / trigger a function not displayed on this diagram. This extension of classic N^2 diagrams provides valuable context, but can be included or not, as desired.

The ordering of the function nodes on the diagonal is arbitrary. This allows the creator to reorder the functions as desired, which is particularly useful for clustering analysis. Among other uses, clustering functional nodes which exchange a lot of data together helps highlight partitioning strategies to simplify interfaces between subsystems. Other uses align with the Design Structure Matrix (DSM) concept.

Beyond clustering analysis, the N^2 diagram is infrequently used today. The lifeline representation of a sequence diagram better communicates interactions. Likewise, most audiences prefer the block and line format of a simplified IDEF0 diagram to the block-line-block or matrix format of an N^2 diagram.

States, Modes, and Transitions
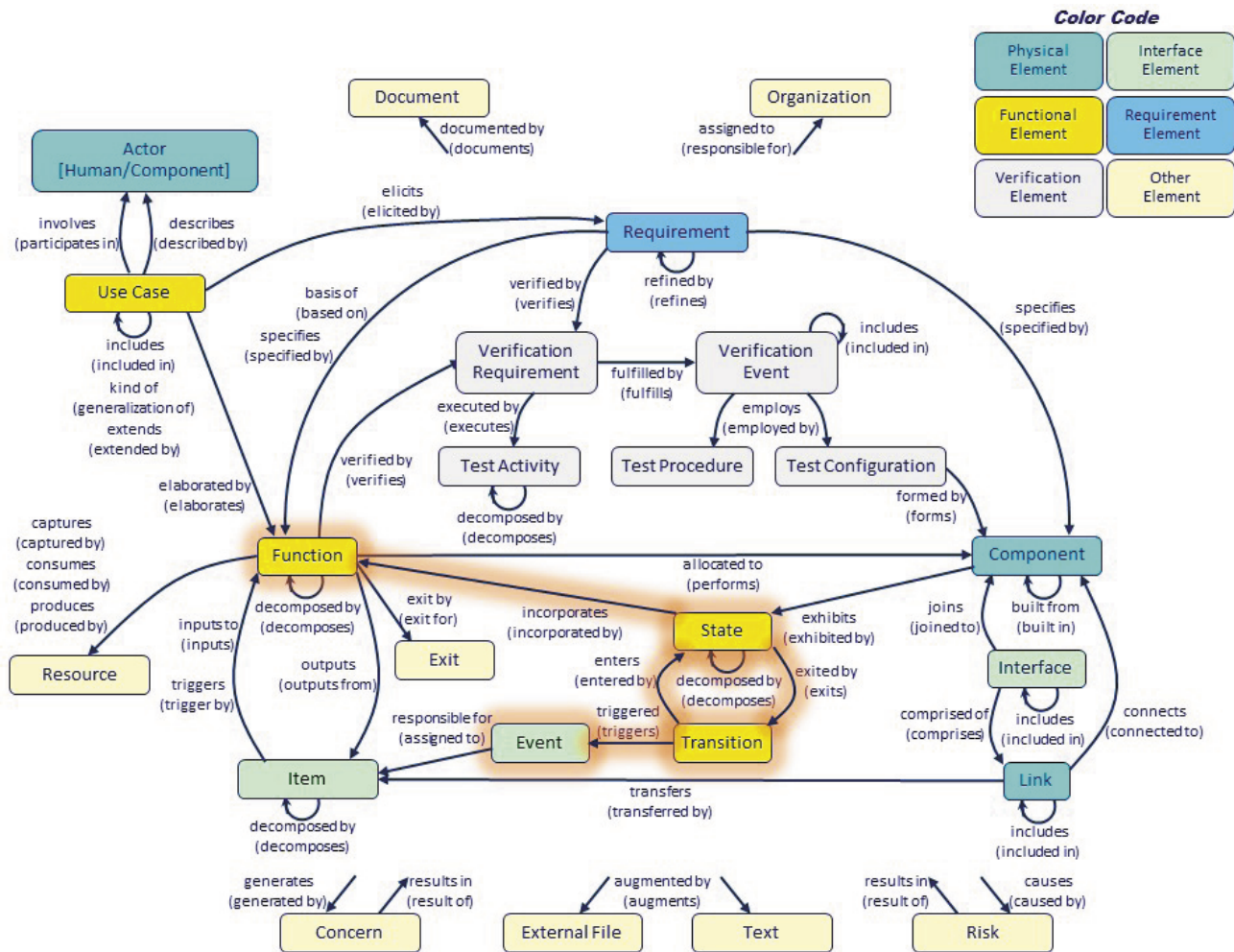
State Transition Diagram

State transition diagrams describe the logical transition of a system through various states of operation. This is a classic systems notation which has been included in the SysML specification. Presented in a free-form layout, the state transition diagram represents states, the transitions that connect them, and the events that trigger transitions.

When discussing behavior, the question of states and the state transition diagram always arises. States are an orthogonal approach to looking at the behavior of a system. Put simply and somewhat loosely, a concept that would be drawn as a block on an activity diagram or EFFBD becomes a line on a state transition diagram. Likewise, a line on an activity diagram would become a block on a state transition diagram.

Some systems are well suited to a state transition representation, and many individuals naturally think this way. Other systems are well suited to a behavioral representation, and many naturally think in this pattern. Ultimately, it is up to the team and the individual whether to use state, behavior, or both in their analysis and modeling. If both are used, then states, their transitions, and the related events are higher-level concepts that are realized by behavior.

Level of Detail: Medium
Audience: System and software engineers
Content: System states and the corresponding transitions
Use: Insight into the system by taking an orthogonal look at behavior
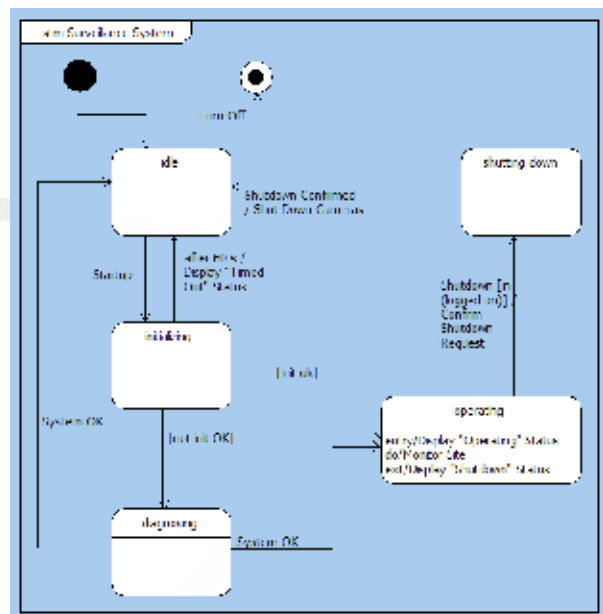
Mapping for the State Transition Diagram

For the state transition diagram, child states are drawn as rounded rectangles. The lines between states represent valid transition paths. Transitions are directional, exiting from one state and entering another. While states may have multiple transitions, transitions are limited to a single entry and a single exit. While transitions can be named and have properties, the focus is generally on the event that triggers the transition and the corresponding conditions:

- Calling events are written in the form "EventName (condition)", and the parentheses are written even if the condition is empty;
- Signaling events are written in the form "EventName (condition)", but the parentheses are not written if the condition is empty;
- Events based upon a Boolean condition are written in the form "when (condition)";
- Events based upon time are written in the form "at (condition)" if they occur at an absolute time or "after (condition)" if they occur after a certain amount of time has passed;
- Any guard condition shown in square brackets.

If a service function supports the transition, the name of the function is shown after the trigging event and call information.

In addition, the nodes representing states optionally display entry (what functions occur when transitioning into the state), exit (what functions occur when transitioning out of the state), and do functions (the behavior that elaborates this state).

The ability to effectively read a state transition diagram corresponds more to an individual's mental model than their role or background. That said, systems and software engineers are classically trained to understand state transition diagrams. For that reason, the view is an effective representation when taking a higher-level, orthogonal look at the behavior of the system.

Representing the System Implementation

Much as Jim Long noted that the various systems engineering diagrams of behavior could be plotted along a spectrum representing the degree of data and structural content, diagrams representing the physical architecture can be plotted in two dimensions. The first dimension (the X axis) parallels Jim's concepts of a behavioral spectrum in the physical architecture domain. The spectrum reflects the two key physical characteristics of composition (the parts tree of a system) and connectivity (how those parts are interconnected externally and internally). The second dimension is the level of detail moving from representations best suited for system architecting and diverse audiences at level 0 and level 1 of the system architecture to representations better suited for design and technical audiences at level N of the architecture.
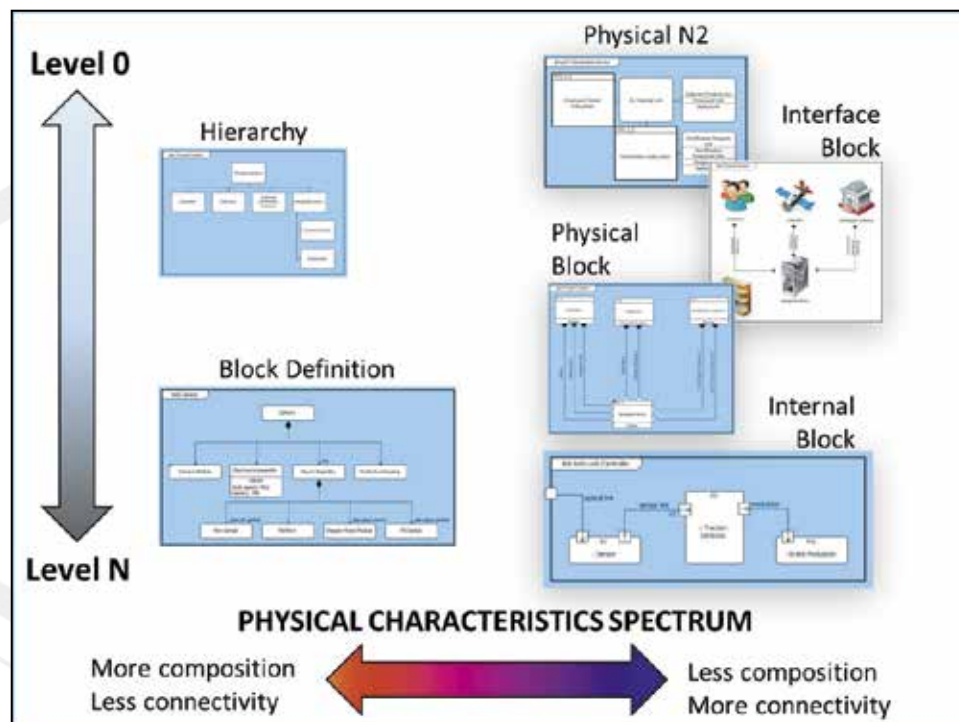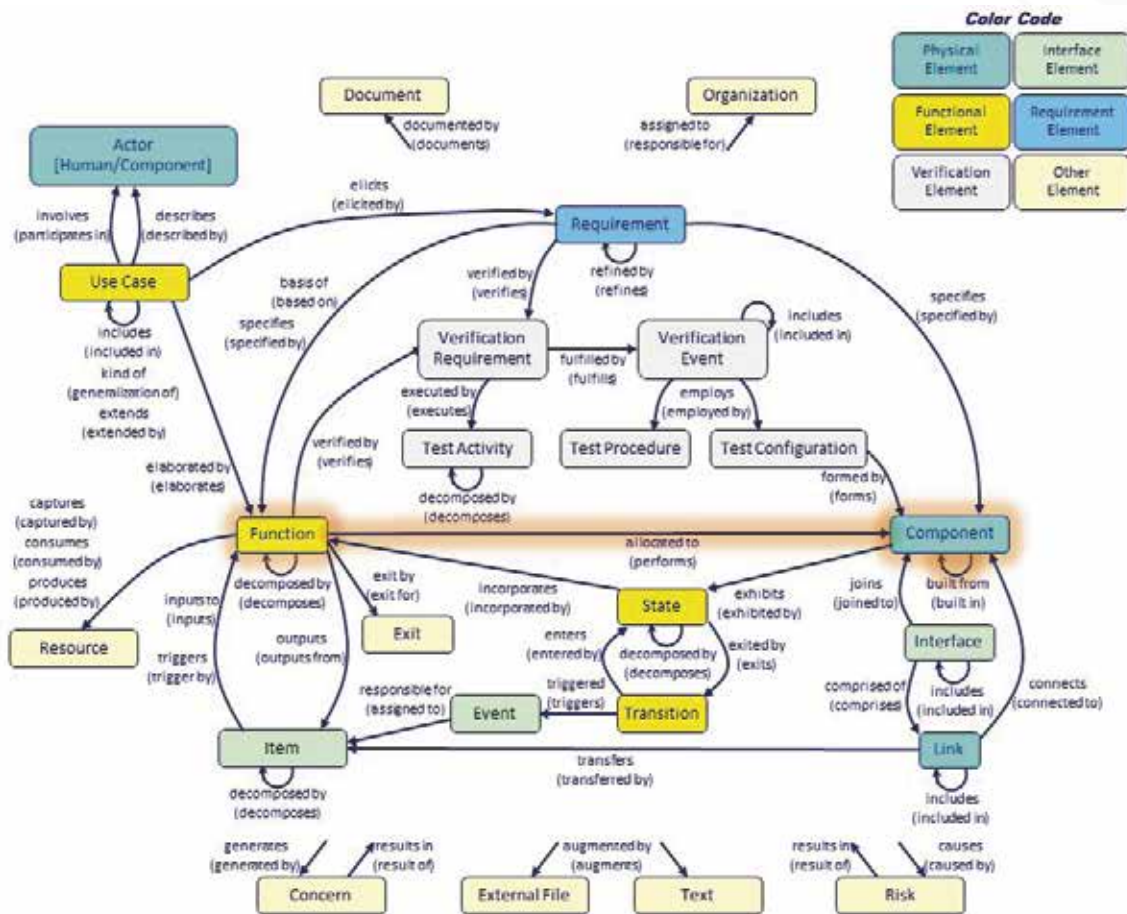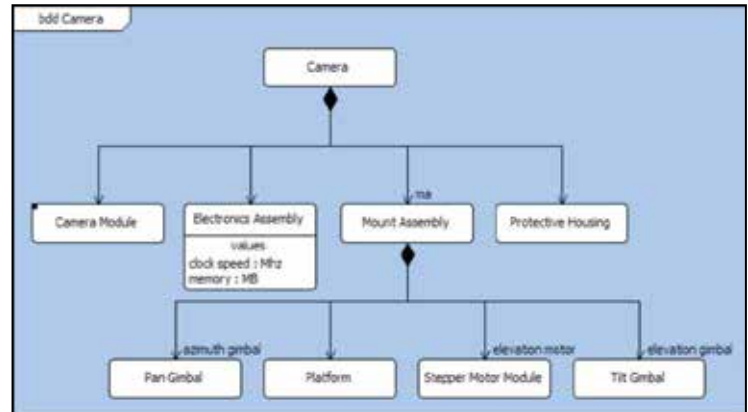
Block Definition Diagram

Block definition diagrams (BDDs) are used to define blocks representing implementation units (hardware, software, and people) in terms of their structure, their classification, and their behavior. They extend classic physical hierarchy diagrams with defined semantics.

Level of Detail: High
Audience: System/software engineers and subject matter experts (SMEs)
Content: Physical composition often including block roles and characteristics; inheritance model
Use: Detailed, multi-level design representation of system composition, inheritance, and corresponding physical characteristics; software class diagram

Mapping for the Block Definition Diagram (BDD)

Nodes on a BDD represent elements (blocks). Nodes always include the element name and frequently include additional information to emphasize design specifics:

- Operations – behavioral aspects allocated to the block. Operations describe synchronous interactions where the requester waits for the request to be handled. Operations reflect a subset of the allocated functions.
- Receptions – behavioral aspects allocated to the block. Receptions describe asynchronous behaviors where the requestor can continue without waiting for a reply.
- Values – represent quantifiable characteristics of a block such as physical and performance characteristics – weight, reliability, etc.
- Parts – are the hierarchical composition of the block (the children). This is classically shown through connecting lines to lower-level blocks, but can be collapsed into the body of the node and shown textually.

The lines on a BDD can reflect either a part-child relationship (in the direction of the arrow) or a generalization / specialization relationship (per UML/SysML standards). When representing decomposition in a part-child relationship, a filled diamond at the connection point with a parent reflects the concept of composition (if the parent is destroyed, the part is destroyed as well). An open diamond reflects the concept of reference (if the parent is destroyed, the child still exists). At the point of connection to the child node, an optional label can be displayed, indicating the role the child plays in the part. Likewise, multiplicity can be shown to indicate the part-child cardinality (the number of elements).

Block definition diagrams can be considered more technical variants of a physical hierarchy diagram. The diagram certainly has more breadth and depth than a classic hierarchy, and this mental model leads to the following rule of thumb when considering its use. The greater technical content of the BDD, including classification, block roles, and multiplicity, make the BDD an ideal replacement for the physical hierarchy when dealing with systems engineers, software engineers, and subject matter experts who crave the detailed, multi-level representation of system composition. For a more general audience, the classic hierarchy diagram conveys the critical composition aspects in a satisfactory manner for their needs and interests.

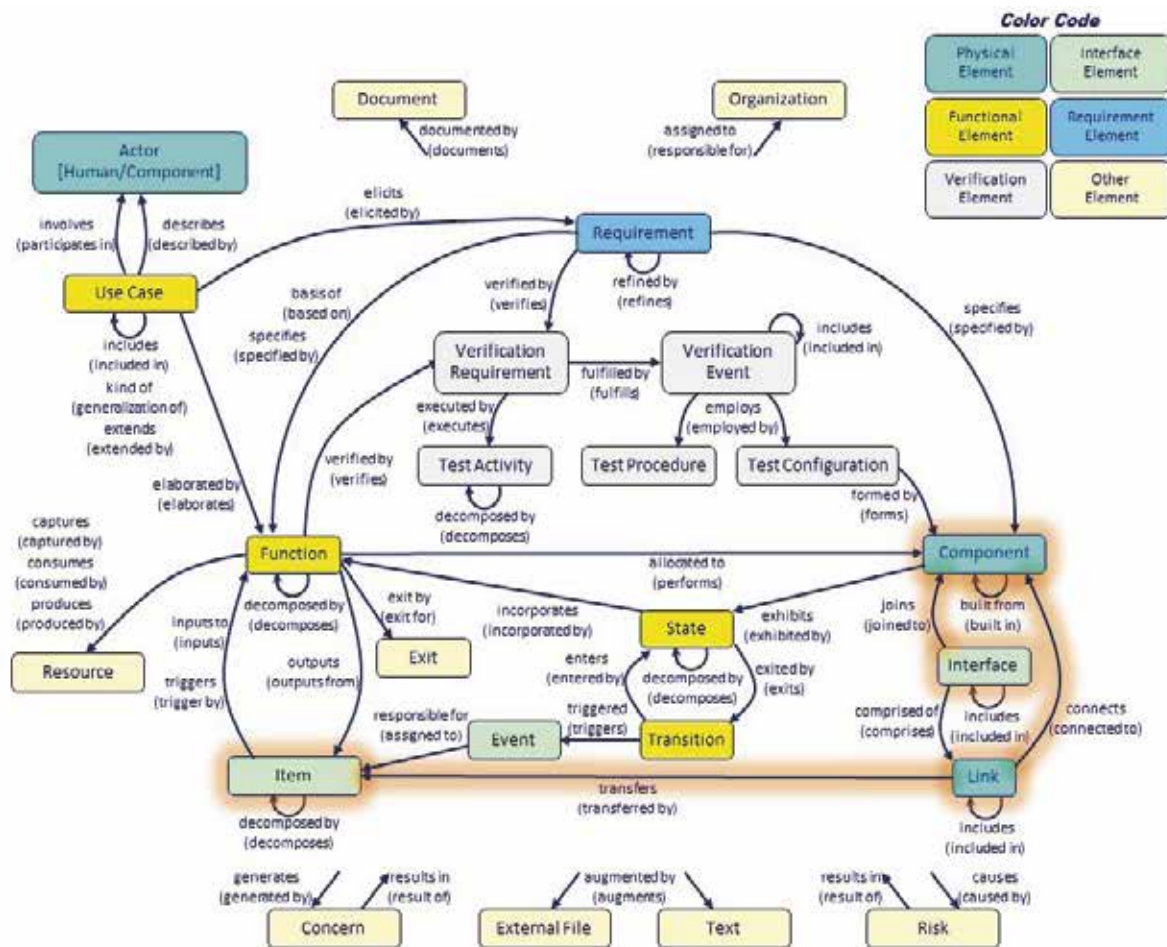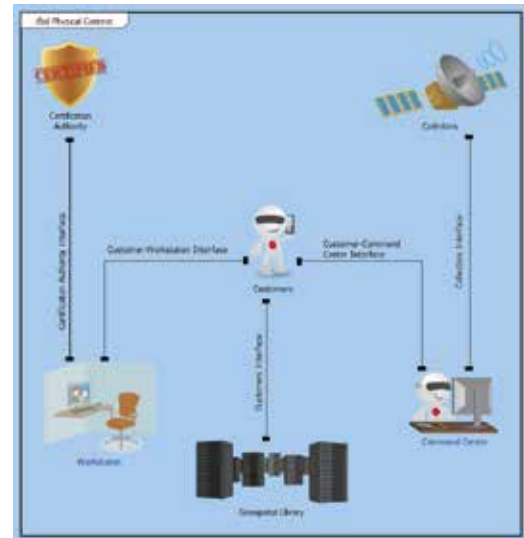## Interface and Physical Block Diagrams

Interface and physical block diagrams are traditional systems engineering box-and-line wiring diagrams representing the logical interfaces and physical connections between components within a system or system segment. The interface block diagram is often the first architectural block diagram that you will develop, focusing first on the fact that logically, A must interface with B before crossing into the details of how that connection is made. At higher levels, these block diagrams often include conceptual communication graphics to enhance communication, leading to the name "architectural cartoons" or "architoons." At lower levels in the system hierarchy, graphics give way to boxes and lines, resulting in a classic "system schematic."

Level of Detail: Medium
Audience: Diverse audiences beyond system and software engineers
Content: Composition with logical or physical connectivity
Use: Specification of logical or physical connections; boundary definition; insight into external connections

Mapping for Interface and Physical Block Diagrams

Components are drawn as nodes or graphics and classically labeled with the block name and number. Connections – either logical interfaces or physical links – are represented as lines between nodes. An unconnected interface or link is often drawn as an unterminated line reflecting an open connection. Connections are classically labeled with the element name and optionally display the items being carried (delimited by braces) for additional detail.

The block-and-line representation with no special symbology positions the block diagram for use with a broad audience. This is particularly true when drawn as an architoon to convey the context and top level physical architecture. The interface and physical block diagrams emphasize connectivity rather than composition or design detail. For those aspects, BDDs and internal block diagrams are much better choices. Additionally, those trained in the SysML notation prefer the richness and symbology of the internal block diagram over the classic block diagram, even when abstracted to the same level of detail.
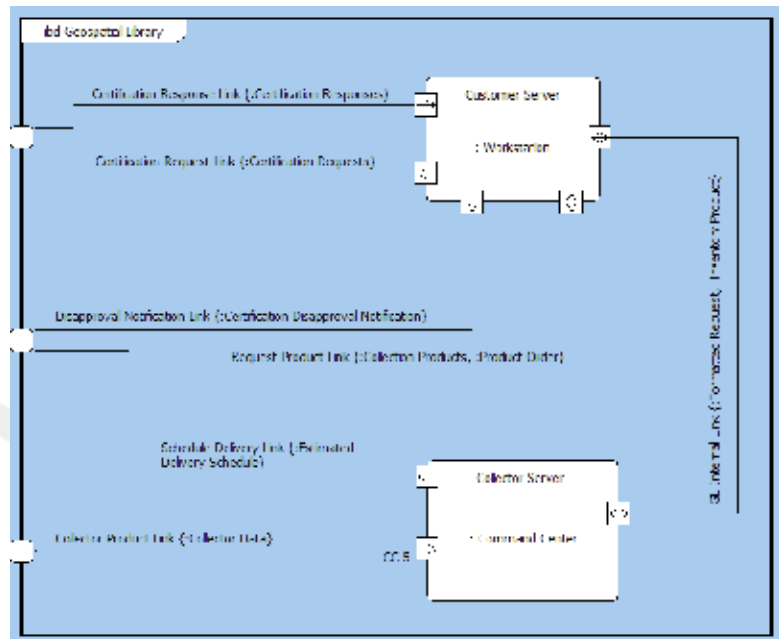
Internal Block Diagram

The internal block diagram (IBD) is a SysML extension of the classical physical block diagram. Though the IBD can be drawn using graphics to create an architoon, the IBD is classically drawn with blocks representing the interconnected parts in a system or subsystem. The IBD goes beyond the classic block diagrams to show additional design detail on nodes, links, and ports where links connect to blocks.

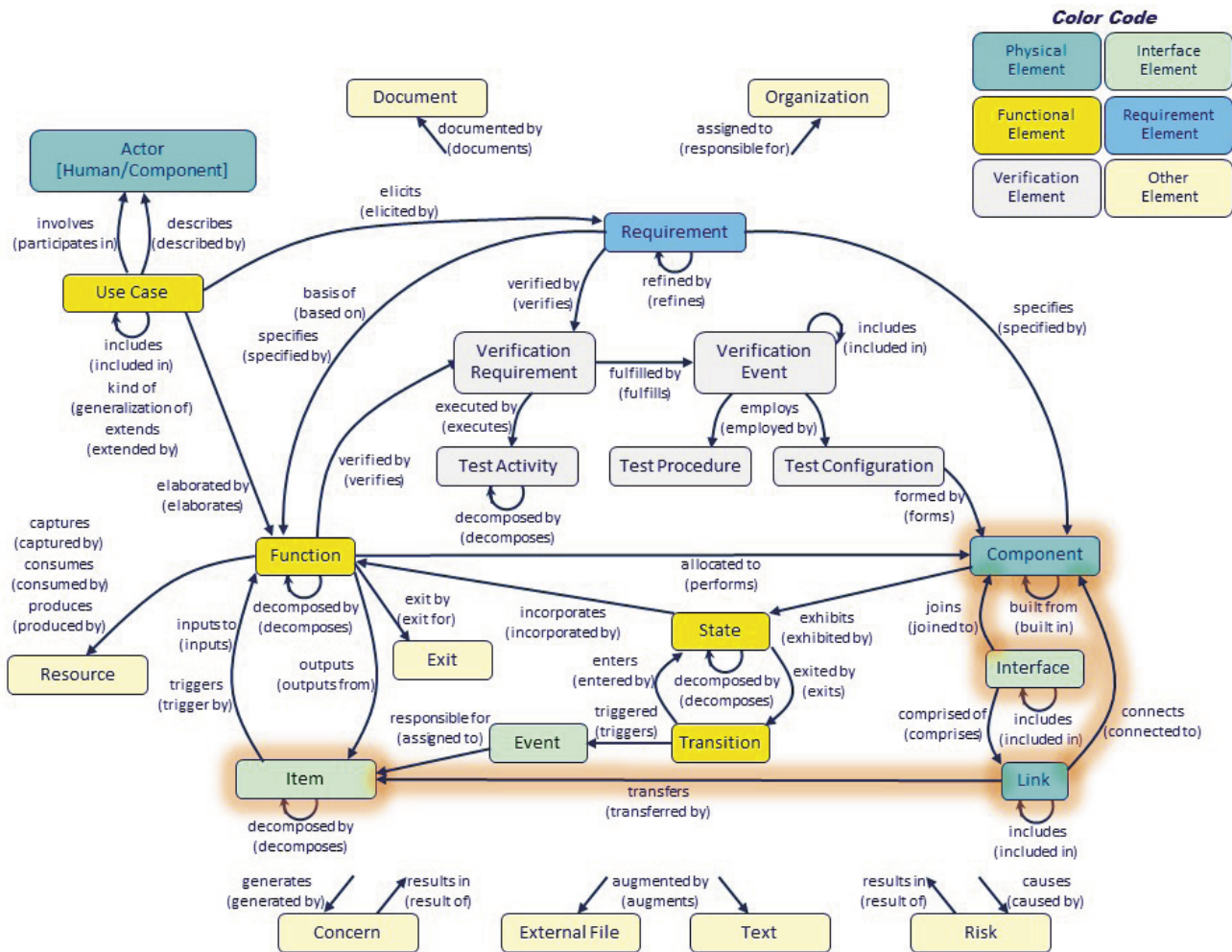**Level of Detail:** High
Audience: System/software engineers and SMEs
Content: Specification of logical or physical connectivity often with ports, directionality, and corresponding data flows
Use: Specification of logical or physical connections

Mapping for the Internal Block Diagram (IBD)

In this variant of a component wiring diagram, the parts are shown as nodes on the diagram. In addition to the part name, the part role is indicated at the top of the node. Parts that are connected but beyond the bounds of the diagram are shown as boxes on the diagram frame.

Lines connecting to a node can reflect either the logical connections (interfaces) between parts or, more classically, the physical connections (links) between parts. As with traditional block diagrams, the connections are labeled with the name of the element and optionally with the items carried by the connection (delimited by braces).

Ports reflect additional design details reflecting how connections connect to the parts. Ports are drawn as squares on the boundary of the part, can be nested within other ports, and can be labeled with their own name reflecting identity. Ports have arrows reflecting directionality of flow (in, out, or inout). Ports optionally display ball and socket style decorations reflecting provided interfaces (drawn as balls connected to the port) and required interfaces (drawn as sockets).

Internal block diagrams have a much higher level of detail than a classic physical or interface block diagram. This detail and the corresponding notations make IBDs ideal for detailed design specification of logical or physical connectivity when communicated to system engineers, software engineers, and subject matter experts. This same level of detail can become problematic at higher levels of abstraction and with broader audiences. In those cases, it is often best to leverage the block diagram with its similar content and structure at an architectural level.
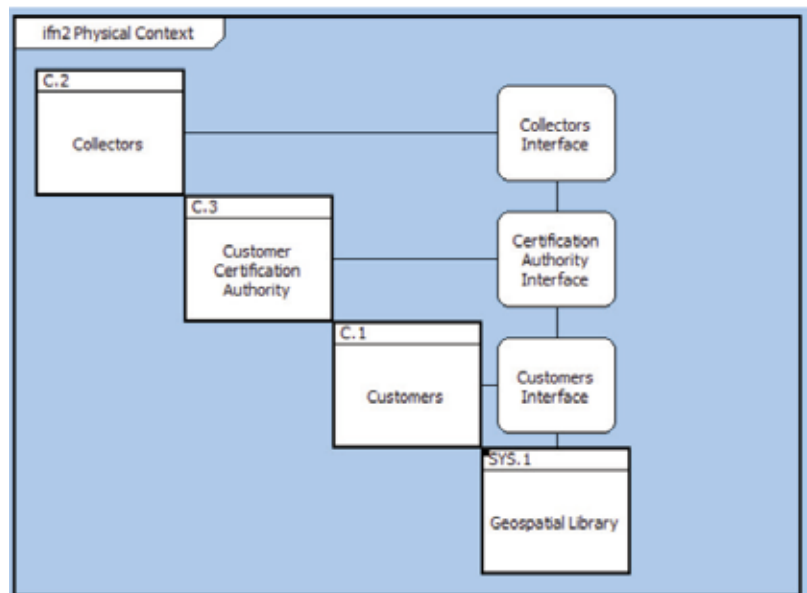
Interface and Physical N^2 Diagrams

Though infrequently used, interface and physical N^2 diagrams leverage the same concepts of the functional N^2 diagram to represent interfaces and physical connections within a system or system segment. These variants of the N^2 diagram present a simplified representation of connectivity between parts. What these diagrams lack in technical detail (and style) of various block diagram representations, they deliver in simplicity and clarity.
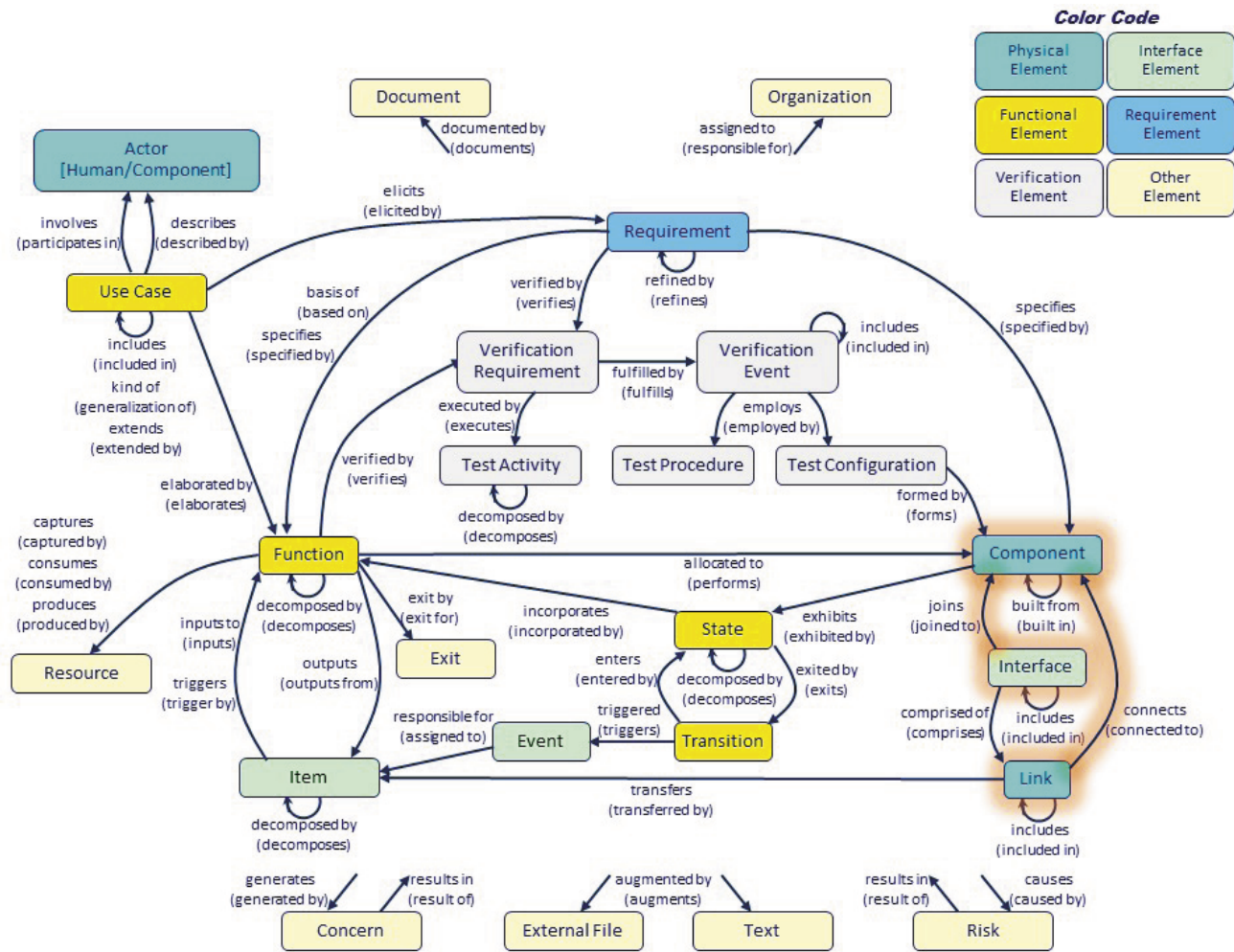
Level of Detail: Low
Audience: General
Content: Single-level composition with corresponding logical (interface) or physical (link) connections
Use: Identification of connections; clustering analysis

Mapping for Interface and Physical N^2 Diagrams

On these N^2 diagrams, the child components are shown on the main diagonal forming an N x N matrix of cells. Connections – either logical interfaces or physical links – that connect a pair of components are shown on the off-diagonal. Since the diagram focuses on physical connection as opposed to directionality, there are no arrows shown on the diagram. Instead, the diagram simply represents who is connected to whom.

The lack of directionality means that half of the off-diagonal locations are redundant. If A is connected to B, we know that B is connected to A. Rather than showing this information twice, only the upper half of the diagram is used. The lower off-diagonal cells will be empty by definition.

As with the functional N^2 diagram, you can manually change the order of the components on the diagonal. This can be useful for clustering analysis. It is the raw simplicity and the clustering analysis that is the primary value of the interface and physical N^2 diagrams.

Connecting Architecture to Analysis

When describing the role systems engineers play, frequently the analogy is drawn to a conductor and an orchestra. While the coordination aspect is appropriate, the greater analogy falls apart. The better analogy is that of connective tissue binding together the various engineers, subject matter experts, managers, users, and stakeholders whose collective knowledge and insights contribute to successfully engineering the right system.

Successfully connecting across the project involves communicating ideas about what is needed, experiences from the past, insights into potential designs, and concerns regarding potential risks and problems. It also requires connecting the many analytical considerations that bring rigor to systems engineering. There are a host of detailed analytical engineering models that govern these considerations – forces, resistance, power, fluid dynamics, reliability, maintainability, and much more. Though the many engineering disciplines and other fields involved may have developed independently, these analytics are not independent. They are often closely coupled and must be properly connected in order to successfully explore possible solutions in the systems engineering trade space.

Much in the way that the systems engineer serves as connective tissue across the project team, the solution architecture is the connective tissue connecting key analytical models that will ultimately govern system performance and viability. Most frequently, these detailed analytical models are interrelated via the physical architecture (components and their interconnections), though the behavioral dimension should not be overlooked. Done properly, the system architecture becomes the "one model to coordinate them all," and several graphical representations help capture and communicate these critical interrelationships.

45

Constraint Block Definition Diagram

The constraint block definition diagram (constraint BDD) is a variant of the physical architecture BDD reflecting the composition of constraints rather than the composition of components. The diagram uses a hierarchical layout to represent the key equations and design parameters that govern system performance.
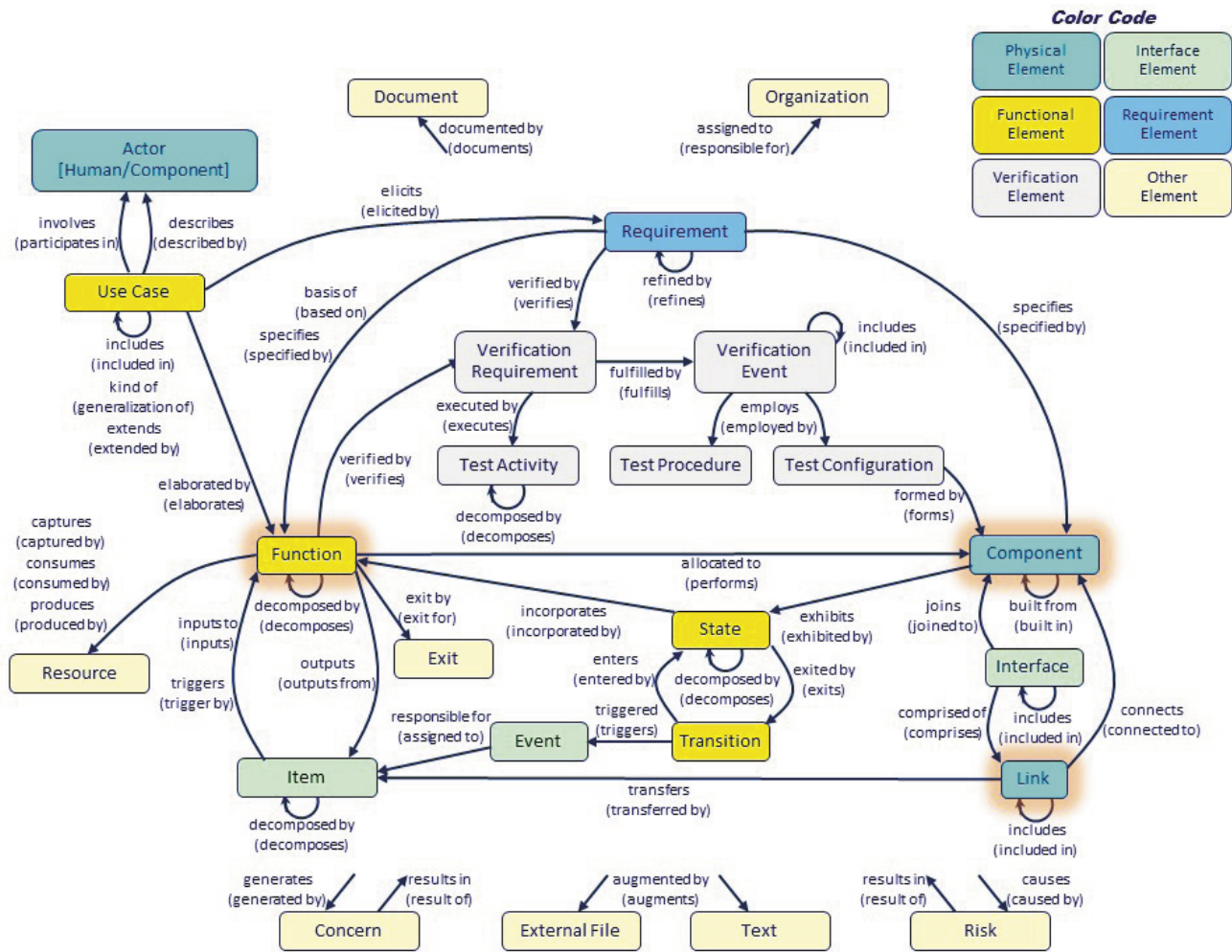
Level of Detail: High
Audience: System engineers and subject matter experts
Content: Physical and logical architecture aspects with associated equations and parameters
Use: Expressing analytics of the system design
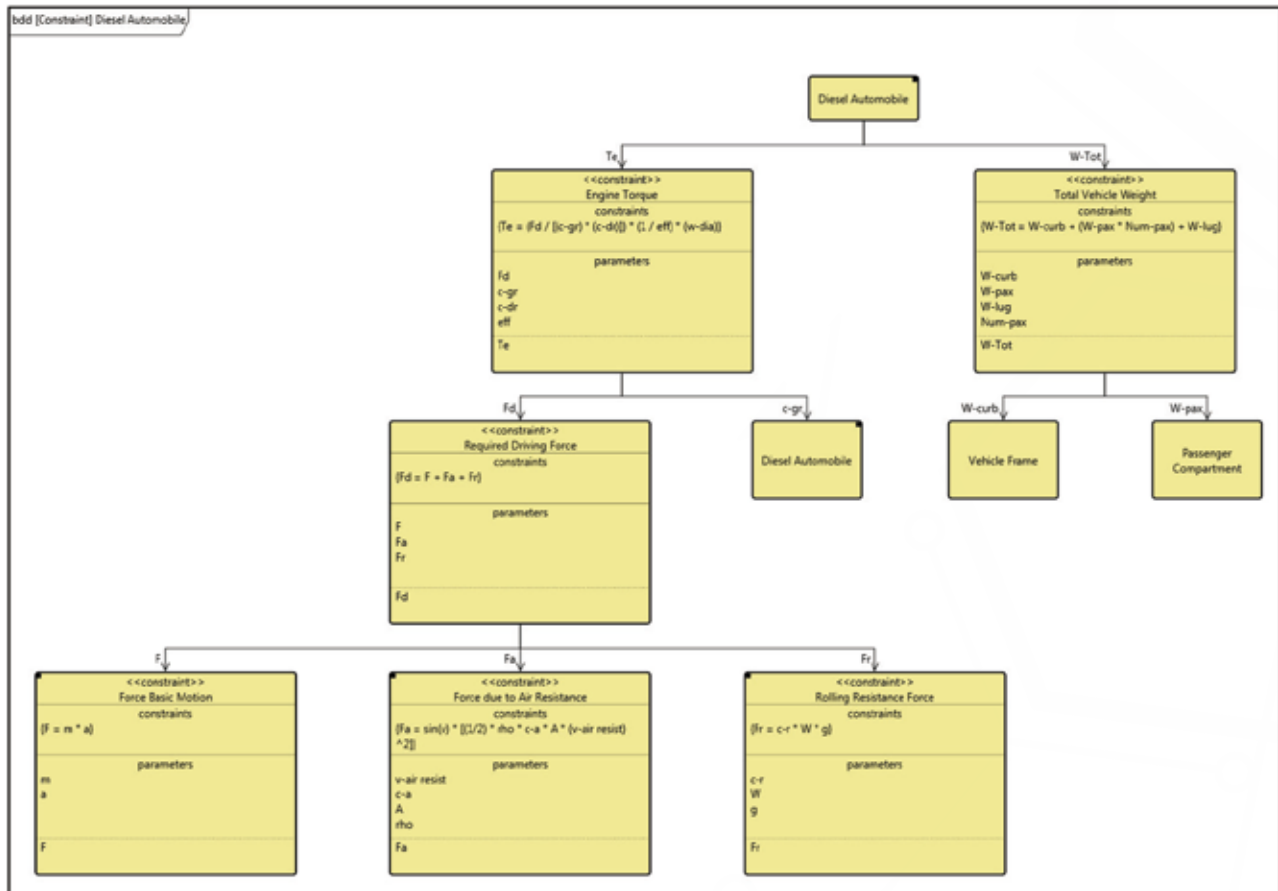
Mapping for the Constraint Block Definition Diagram

Nodes on a BDD represent constraints (the equations governing the system and the corresponding parameters) as well as their connection points (design parameters on elements in the system architecture). Most frequently, the design parameters are associated with the physical architecture (components and links) or behavioral architecture (functions), but they can be drawn from anywhere in the descriptive system model. Nodes display the element name, the constraints (the equations that govern the analytics), and the parameters (the design values of interest).

Constraint BDDs are a rather clean representation of constraints and the manner in which they connect to the physical and logical dimensions of the architecture aspects. Their technical depth makes them well suited for engineers and other subject matter experts. While they are a useful representation, the hierarchical tree structure does not strongly convey the nature and complexity of the interactions. The parametric diagram often does a much better job of visually representing the lines of convergence and divergence to identify critical parameters in your model.
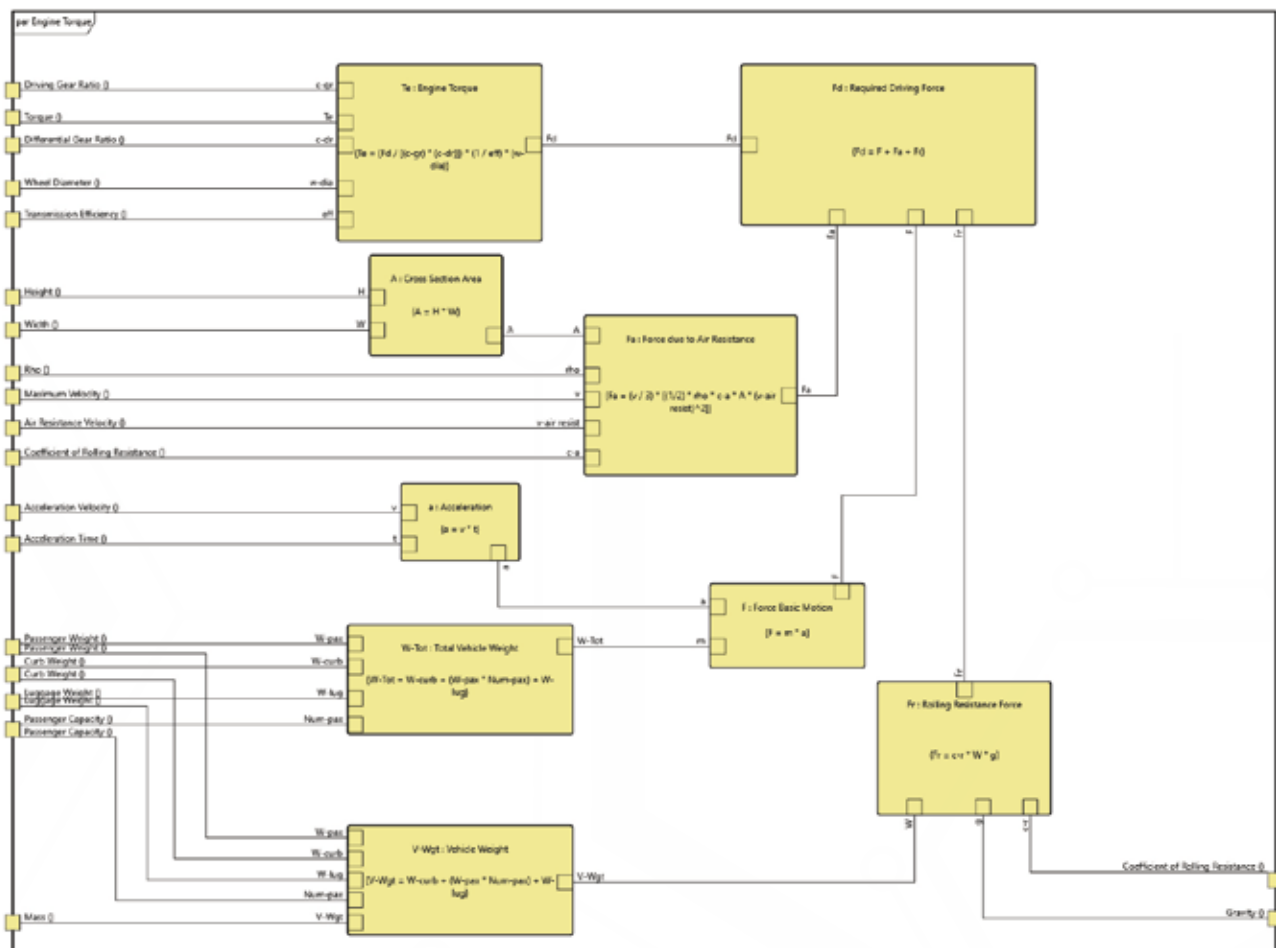
Parametric Diagram

Parametric diagrams represent the constraints and analytics associated with the logical and physical architecture in a box-and-line, wiring diagram format rather than the hierarchical representation of a constraint BDD. This enhances the visualization of the analytical relationships between key systems parameters and the equations that govern systems.
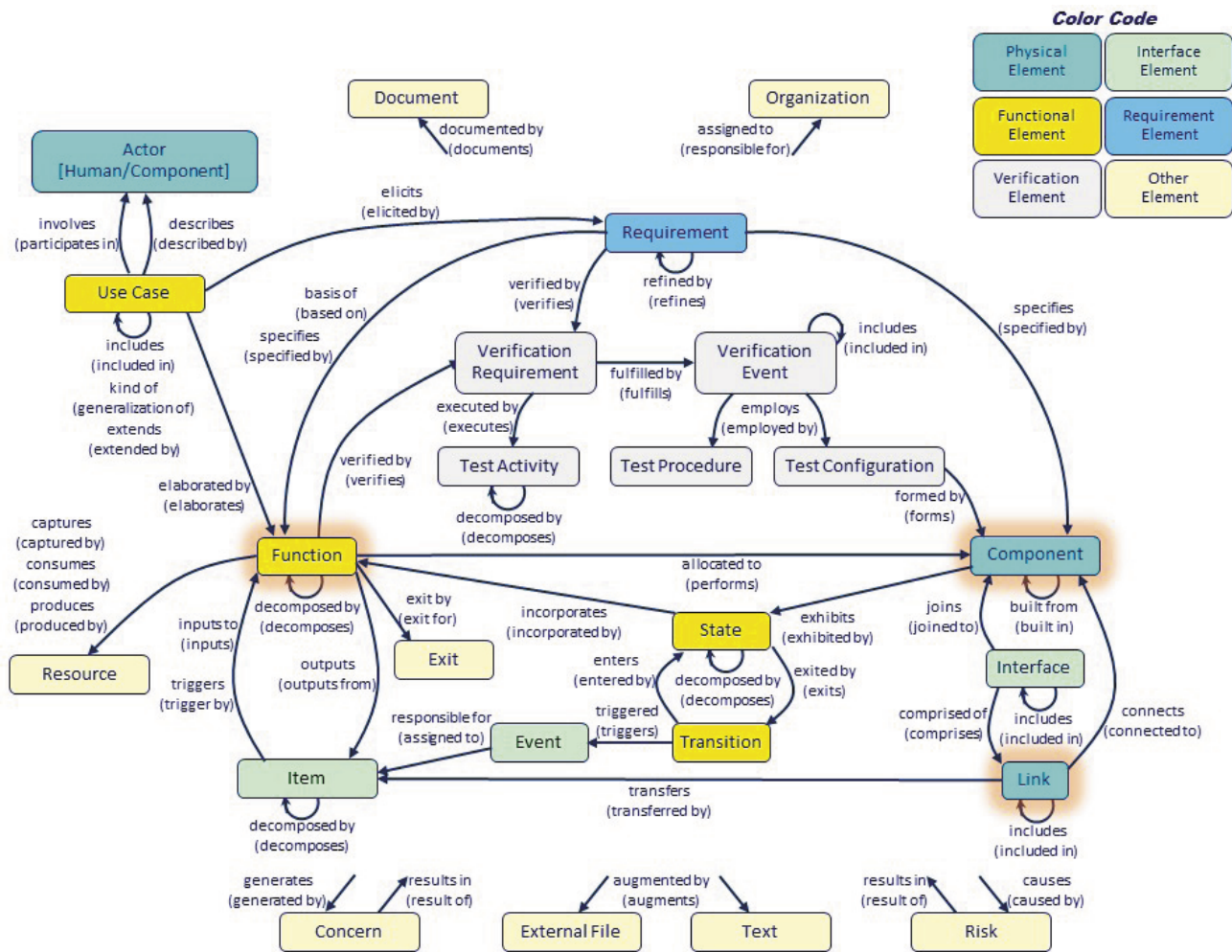
Level of Detail: High
Audience: System engineers and subject matter experts
Content: Physical and logical architecture aspects with associated equations and parameters
Use: Mathematical specification and visualization of relationships between key system parameters

Mapping for the Parametric Diagram

Constraints are shown as nodes with their various equations and variables. Lines connecting the nodes represent the mappings between systems parameters. These lines are labeled with the local variable / parameter name at each end. Parametric values from architectural entities can be represented as ports on the diagram frame or as simple nodes within the diagram.

The interconnected, spider-like nature of the parametric diagram helps communicate the linked nature of the analytics that govern a system. At a detail level, the diagrams communicate the mathematical relationship between key systems parameters to engineers and other stakeholders. At a more abstract level, the same diagram is effective in communicating the interdependencies to a non-technical audience. However, parametric diagrams come with a cautionary note that is relevant regardless of the audience. Simply because one can represent equations and interrelationships graphically does not mean that one should. Parametric diagrams are easily – and often – overdone. Rather than representing absolutely everything to an atomic level of detail, parametric diagrams are often best limited to the key systems equations.
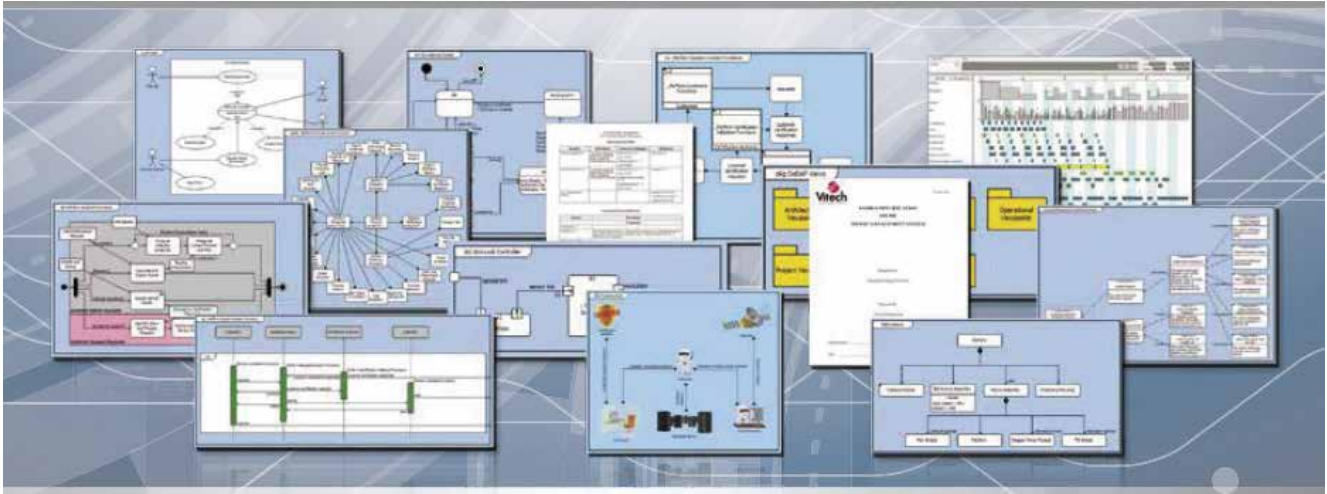
Many Viewpoints, Countless Views, One Integrated Solution

Conclusion

The foundational purpose of all these views is communication. They each represent a specific, defined subset of the information that makes up a system model. When they are drawn from a single model with guaranteed currency and consistency, they become powerful tools in representing and analyzing the breadth of concerns faced when engineering a system. But their fundamental power lies in their ability to communicate richly and effectively across a diverse community of project team members and stakeholders. The systems engineer who draws from this broad collection will have at her fingertips the ability to match the communication needs of her audience with exactly the right vehicle for conveying understanding of the system design.

Any limitation of the set comes at the price of communication with all those who might find the excluded representations helpful. Whether this is done in the name of "standardizing" on some subset of representations or through a failure to understand and use the views correctly, communication is impoverished by it.

In the same manner, any failure to draw these views directly from a model risks both currency and consistency. In this case, not only is communication impaired, but the design integrity of the system itself is also put at risk. If one is forced to maintain drawings by hand, the only choice is to limit the number of representations used, trading off the cost of maintaining drawings against the benefit of enhanced communication.

But linking a rich palette of views with a tool powerful enough to maintain, track, and produce them offers the ability to understand, design, and communicate tailored solutions to solve the problems of a global environment in need of systems engineering.

Additional Resources

Those interested in more information on systems engineering representations and the concept of integrated model-based systems engineering may appreciate the following resources:

- *FIPS-183, Draft Federal Information Processing Standards Publication 183*, NIST, 1993.
- Sanford Friedenthal, Alan Moore, and Rick Steiner, *A Practical Guide to SysML: The Systems Modeling Language, 3rd edition* (OMG Press, 2014).
- Joe Holt and Simon Perry, *SysML for Systems Engineering, 2nd edition: A Model-Based Approach*, (IET, 2013).
- Robert Lano, *The N2 Chart* (TRW Software Series, 1977).
- David Long and Zane Scott, *A Primer for Model-Based Systems Engineering, 2nd edition*, (2012).
- Jim Long, "Relationships between Common Graphical Representations in Systems Engineering," *Proceedings of the Fifth Annual International Symposium of INCOSE*, July 1995 (subsequently updated July 2002 and available from the Vitech website at www.vitechcorp.com).
- Tim Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design* (OMG Press, 2008).

About the Authors

Zane Scott is Vice President for Professional Services at Vitech Corporation. He is responsible for consulting and training operations. A frequent blogger and presenter, Zane has taught the fundamentals of model-based systems engineering in a variety of settings, including the delivery of the "SE 101" tutorial at the INCOSE International Symposium in 2014, 2015, and 2016. With David Long, Zane co-authored Vitech's *Primer for Model-Based Systems Engineering*.

Zane is active in INCOSE as co-chair of the Corporate Advisory Board, a member of the Board of Directors, and a member of the first cohort of the INCOSE Technical Leadership Institute. Zane has a diverse background which includes a B.A. in Economics from Virginia Tech and a J.D. from the University of Tennessee School of Law. He has worked as a litigator, and is trained as a hostage/crisis negotiator and mediator. Prior to coming to work for Vitech, Zane was a senior process consultant assisting government and private clients with process modeling and improvement projects.

For over 20 years, David Long has focused on helping organizations increase their systems engineering proficiency while simultaneously working to advance the state of the art across the community. David is the founder and president of Vitech Corporation, where he developed CORE™, a leading systems engineering software environment. He co-authored *A Primer for Model-Based Systems Engineering*, and is a frequent presenter at industry events around the world. A committed member of the systems community and Expert Systems Engineering Professional (ESEP), David is a past president of the International Council on Systems Engineering (INCOSE), a professional organization focused on sharing, promoting, and advancing the best of systems engineering. In 2006, he received the prestigious INCOSE Founders Award in recognition of his many contributions to the organization.

David holds a bachelor's degree in Engineering Science and Mechanics, as well as a master's degree in Systems Engineering from Virginia Tech.

About Vitech

For over two decades, Vitech has helped organizations raise their systems engineering proficiency through a tailored combination of training, services, and software. By engaging with Vitech, organizations around the globe increase their productivity, enhance agility, and reduce project risk.

Unlike siloed approaches and products that mask critical context and system interactions, Vitech's approach and its GENESYS™ and CORE™ software embrace the holistic aspects of systems engineering. These solutions enable teams to clearly capture and address systems concerns from problem identification through requirements, architecture, and testing in an integrated model. These solutions manage critical interrelationships to guarantee consistency and design integrity. The result is a team empowered to engineer with confidence, free to focus on creativity, innovation, and analysis to effectively deliver against stakeholder needs.

www.vitechcorp.com

Vitech

Insight through integration