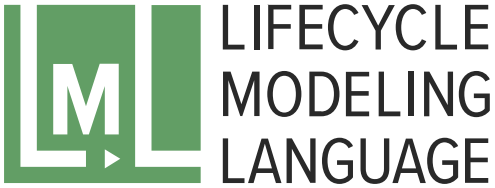


Essential LML

Lifecycle Modeling Language (LML):
a Thinking Tool for Capturing, Connecting and Communicating Complex Systems

Warren K. Vaneman, Ph.D., Jerry J. Sellers, Ph.D., Steven H. Dam, Ph.D., ESEP

Essential LML

Lifecycle Modeling Language (LML): A Thinking Tool for Capturing, Connecting and Communicating Complex Systems

By Warren K. Vaneman, Ph.D., Jerry J. Sellers, Ph.D., Steven H. Dam, Ph.D., ESEP

Published by:

SPEC Innovations

Manassas, VA

www.specinnovations.org

Copyrighted Material

Trademarks

Warning and Disclaimer

GPBU01991-00004 (TBD)

Essential LML. Lifecycle Modeling Language (LML): A Thinking Tool for Capturing, Connecting and Communicating Complex Systems

ISBN: TBD

Publication Date: TBD

Printed in the United States of America

TABLE OF CONTENTS

FORWARD: WHY LML?

Do we really need another modeling language? I know you are asking this as you read the book title but judging from the results of the architecture and system engineering projects that I have seen over the past 40 years, I am afraid that the answer is yes.

My first introduction to a language other than English, Spanish, French or German (and I was never good at any of them), came in the early 1970's when I took my first computer course and was introduced to FORTRAN IV. During that first course I was introduced to flowcharting as a way to describe the software I was supposed to write. Like most of the others in my class, I wrote the code first and then flowcharted it because flowcharting was actually harder than just writing the code.

In the late 1970's, early 1980's I was introduced to a simpler flowcharting language that used just two symbols (a rectangle to show a process step and a rectangle with a point on the end for decisions, such as a GOTO). With this simpler language, I was able to flowchart a large, existing code in order to port it from the Cray to the new VAX's the Lab had just purchased.

In the mid- to late 1980's I became familiar with formal systems engineering techniques. As a physicist who spent most of his time writing code, I had no formal training in the systems engineering discipline, but after my time at LANL, TRW and Rockwell hired me as a Project Engineer, leading teams of systems engineers.

However, even with all this experience, I did not have a formal course in Systems Engineering until I went to Science Applications International Corporation (SAIC). SAIC had a formal course, which I attended when I became the Technical Manager of their West Coast proposal center. Mr. Charlie Zumba was the lead instructor of this course and provided valuable personal insights into the real problems that project team's face in implementing systems engineering for customers who frequently did not appreciate the time and money it took to produce "all that paper."

Part of my role at SAIC was to help proposal writers do a better job of applying the systems engineering discipline to the proposal process and technical approaches. As a result, I was introduced to a new systems engineering tool, RDD-100. This opened a completely new technique for capturing system design information and creating the necessary documentation. It had a new (to me) language using logical constructs and a set of elements to describe systems.

The "sales engineer" who presented the RDD-100 tool, was Mr. Jim Long. Jim later left Ascent Logic to become the President of his son's company that created the successor tool, CORE. Jim explained the tool and its underlying technique so effectively that I insisted he provide the training in the tool for SAIC whenever possible. Since he was the program manager under which that technique was developed in the late 1960's at TRW, he was the best.

After leaving SAIC to form Systems and Proposal Engineering Company (now dba SPEC Innovations), I continued my formal and informal education in systems engineering and architecture development. We applied the systems engineering technique and developed a number of processes for implementing it over the next decade.

In the past decade, I became familiar with Java and the Unified Modeling Language (UML). Again, I found myself writing code first. UML didn't seem to be a great aid in the process. However, I'm sure that reflected in the code I wrote. The object-oriented methodology never felt very comfortable in describing how the code would be used. It certainly didn't work well in trying to describe what the code was going to do or any systems engineering designs to the people unfamiliar with UML.

Subsequently, I have become familiar with the Systems Modeling Language (SysML) profile for UML. It contains many of the same problems as UML. UML and SysML may communicate well to those using it, but it is very difficult for users, developers in other disciplines, test and evaluation personnel, and other participants in the system lifecycle to understand. They are not used to thinking in terms of objects. Instead they need processes and procedures, which are used for training, operations and support. These items are more functionally oriented. Even in the software domain, the latest programming methodology is called "Agile." In Agile Programming, they start with *functional* requirements. Having recently hired a number of programmers fresh out of a very good school, they told me they had limited exposure to UML. Similarly, the systems engineers I've hired have little or no experience with SysML. If this is the case, then who are we developing these object models for?

Years of research and work with a number of my colleagues lead to the development of the Lifecycle Modeling Language (LML) described in this book. The purpose of this language is not necessarily to supplant the other languages out there. After all, electrical engineers (EE) are not going to change their symbology for anyone else. It's been around a long time and all double E's know it. Instead, we propose LML as primarily a systems engineer's language that can easily translate into other notations and languages, such as Business Process Modeling Notation (BPMN), UML, SysML, EE diagrams, etc.

To standardize the LML language, the LML Steering Committee was formed and began meeting in April 2013. It consists of several representatives from academia, government and industry. I acted as the Secretary for the committee and tried to keep my opinions to a minimum.

We called it the *Lifecycle* Modeling Language because it supports the full lifecycle: conceptual, utilization, support and retirement stages, along with the integration of all lifecycle disciplines including, program management, systems and design engineering, V&V, deployment and maintenance into one framework.

LML is an *open standard* and published at www.lifecyclemodeling.org. Comments and contributions to the standard are welcome. Minor extensions to support SysML were added in version 1.1, published in December 2015, thus demonstrating that SysML was subsumed by LML.

NOTE: *LML is not a proprietary language*. No one owns it, and no one is going to force it on the community. We hope you will adopt it by consensus. It is only the "80%" solution and you can add to it to meet your specific needs.

For those of us who see that the primary job of a systems engineer is to communicate the needs for the system to the system developers and help prove that those needs were met, LML provides a common language that anyone can use. I think you will enjoy this book and using LML.

Steven H. Dam, Ph.D., ESEP
Secretary, LML Steering Committee
Manassas, VA
February 2018

WHO IS THIS BOOK FOR?

This book is offered as a general overview of the Lifecycle Modeling Language. The aim is to present the basic philosophy and concepts of the language to highlight its application to capture, connect and communicate the details of complex systems. The reader may be an expert at model-based systems engineering (MBSE) or totally new to the discipline. However, we assume the reader has at least a basic understanding of the systems engineering framework and its application to projects and problem solving. In fact, we hope it is the reader's frustration with their current tools and techniques that has led them to explore LML.

We designed the book with short chapters, so they can be quickly digested by busy people as well as provide a quick reference for experts. The concepts and terminology should be familiar to almost anyone working in these fields. This book is not intended to be a detailed specification of LML. The specification for LML version 1.1 is available for free at www.lifecyclemodeling.org. Engineers who want to perform a full implementation will want to download the specification and go through it in detail. An Owl file is available upon request as well. We anticipate that future books, courses and other material will be developed to aid systems engineering practitioners who want to examine LML in greater detail.

We hope you enjoy learning this new language and find it to be a useful tool in your systems engineering and project management toolbox.

LML Steering Committee

- Dr. Warren Vaneman, Chair
- Dr. Dennis Buede
- Dr. Steven Dam
- Mr. Gerard Fisher
- Mr. Edward Gabb
- Dr. Kristin Giammarco
- Dr. Edward Huang
- Dr. Kathy Laskey
- Dr. Jerry Sellers
- Mr. Stephen Zini

BACKGROUND: MBSE, ONTOLOGIES AND LANGUAGES, OH MY!

Before diving into an overview of the Lifecycle Modeling Language (LML), it is instructive to first step back and examine model-based systems engineering in general, and the role taxonomies, ontologies and languages play in it. More than just a language for building diagrams, Model-Based Systems Engineering (MBSE) leverages the power of relational database software tools to:

- Capture—details about complex systems, including elements that make up various architectural views (i.e. functional, physical, requirements, verification)
- Connect—various elements of the architecture to define key relationship (i.e. a requirement is "satisfied by" a specific system)
- Communicate—the details of complex systems to different project stakeholders in ways that make the most sense to them using diagrams and other outputs tailored to their perspective.

To understand the advantages of MBSE it is useful to consider the various dimensions of a systems engineering project in which we might be interested. As illustrated in Figure 1, we want to have insight across the entire lifecycle of a project from concept through operation and disposal, think of this as the *width* of a model.

We also want to be able to zoom from very high system of systems architecture levels down to the component level or even to the part level to understand the model more in-depth. In a similar way we will want to identify the requirements, functions, risks, etc. at these same levels. We can think of this as the *height* of the model.

Finally, within the width and height we want to be able to understand the complex relationships between systems, functions, requirements, etc. We can think of this as the *depth* of the model.

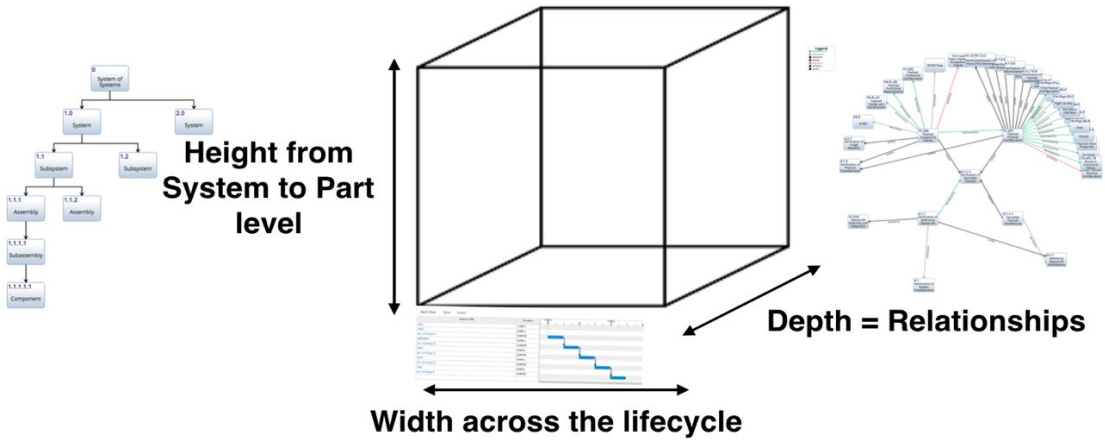


Figure 1. Modeling Dimensions. Throughout a project, systems engineering needs insight across the entire lifecycle of a project from concept through operation and disposal, think of this as the width of a model.

To define MBSE it is important to first understand what is meant by model-based engineering in general. The Department of Defense (DoD) defines a model as a "physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process." (DoD 5000.59 -M 1998) In other words, a model is anything (physical or logical) that is used to describe, discuss, design or otherwise understand the system of interest. By this definition, all of engineering is, "model-based."

Practicing engineers use all kinds of models to do their jobs, including sketches, drawings, CAD models, spreadsheets, even the proverbial back of an envelope as illustrated in Figure 2. While physical models are essential to the systems engineering process, especially during the design phase, often come from more abstract models developed in early requirements development phase of the lifecycle.

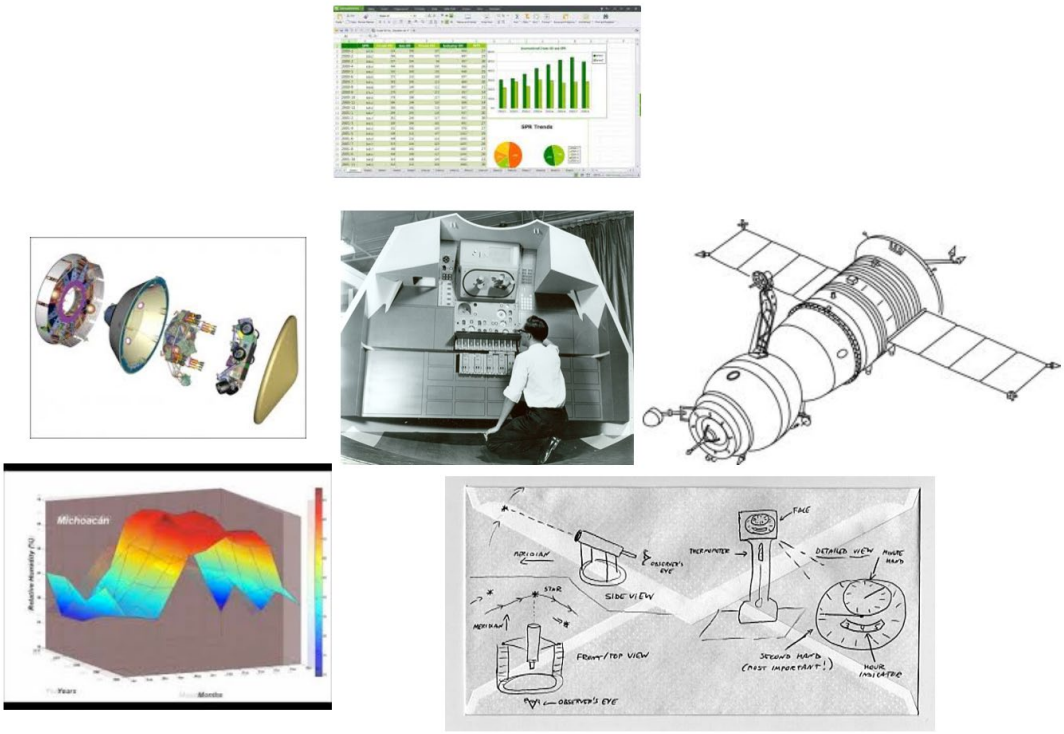


Figure 2. Types of Models. Engineering efforts depend on a wide variety of different types of models. Some are descriptive (e.g., CAD models), others are analytic (e.g., thermal models).

Continuing that metaphor here, we can view MBSE and its supporting tools as providing a new pallet of colors to allow the systems engineering "artist" to "paint" in ways they couldn't before, providing insight into complex problems in new and unique ways. Equally valuable, MBSE tools can perform much of the "heavy lifting" of the science side of the problem, reducing the number of documents that need to be produced for example and making it far easier to put our hands on the latest, best technical data.

The universe of systems engineering is typically a four-dimensional universe of cost, schedule, performance and risk as shown in Figure 3. Every decision made in systems engineering affects one or more of these dimensions in some way. Sometimes in unexpected ways. One definition of systems engineering suggests that it is about "managing the unintended consequences of systems." These unintended consequences occur along one or more of these dimensions.

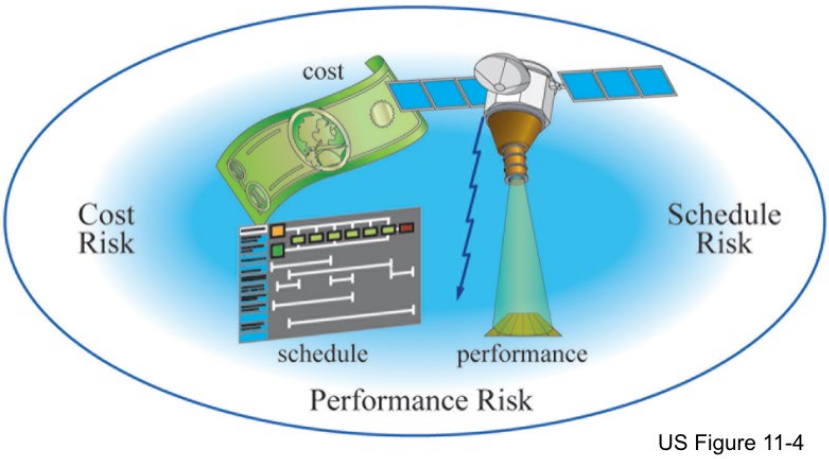


Figure 3. The Systems Engineering Universe. The systems engineering universe has four dimensions—cost, schedule, performance and risk.

Model-Based Systems Engineering has evolved to help make better decisions in the systems engineering universe. According to INCOSE, "Model-Based Systems Engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases." [INCOSE SE Vision 2020 (INCOSE-TP-2004-004-02), Sept 2007]

While the INCOSE definition is a good starting point, it does not give any indication how MBSE is different from traditional systems engineering, nor how to implement it. To develop a more thorough definition of MBSE, it is useful to think of systems engineering being comprised of both an art and a science. The art of systems engineering focuses on technical leadership, on a system's technical design and technical integrity throughout its lifecycle. The science of systems engineering is art's ally, systems management, focused on managing the complexity associated with having many technical disciplines, multiple organizations, and hundreds or thousands of people engaged in a highly technical activity.

In the approach to LML offered in this book we will use the MBSE definition offered by Dr. Warren Vaneman. "*Model-Based Systems Engineering is the formalized application of modeling (static and dynamic) to support system design and analysis during all phases of the system lifecycle, through the collection of structures, modeling languages, model-based processes, and presentation frameworks used to support the discipline of systems engineering in a model-based or model-driven context."* (Vaneman, 2016)

The four tenets of MBSE highlighted by this definition are further expended on below, and are depicted in Figure 4:

- **Structure** – An ontology is a collection of standardized, defined terms and concepts (taxonomies) and the relationships among them. Structures use ontologies, and defined relationships between the systems entities, to establish concordance, thus allowing for the emergence of system behaviors and performance characterizations within a model.

- **Modeling Languages** – Serve as the basis for tools and enable the development of system models. Modeling languages are based on a logical construct (visual representation) and/or an ontology.

- **Model-Based Processes** – Provides the analytical framework to conduct the analysis of the system virtually defined in the model. The model-based processes may be traditional systems engineering processes such as requirements management, risk management, or analytical methods such as discrete event simulation, systems dynamics modeling, and dynamic programming.

- **Presentation Frameworks** - Provides the framework for the logical constructs of the system data in visualization models that are appropriate for the given stakeholders. These visualization models take the form of traditional systems engineering models. These individual models are often grouped into frameworks that provide the standard views and descriptions of the models, and the standard data structure of architecture models.

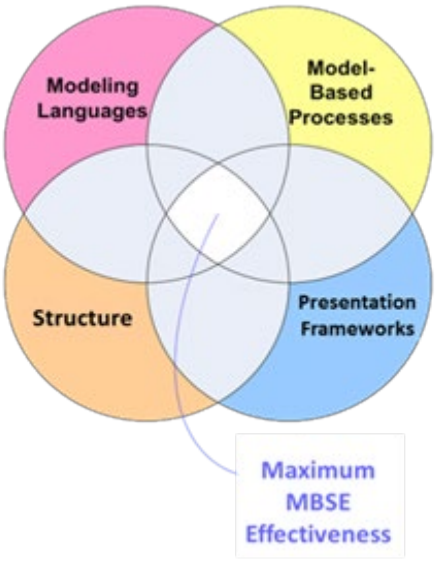


Figure 4. Four Tenets of MBSE.

Maximum effectiveness occurs at the convergence of the four MBSE tenets in Figure 4. This is where most MBSE tool vendors strive to create a proper balance of the tenets to develop their modeling environments.

Model-Based Systems Engineering was envisioned to transform systems engineering from a reliance on document-based work products to a deeper understanding of the engineering environment based on models and model-generated products. This transformation means more than using model-based tools and processes to create hard-copy text-based documents, drawings, and diagrams. In a MBSE environment, a model is created to virtually represent the system. As such, each entity is represented as data, only once, with all necessary attributes and relationships of that entity being portrayed. This data representation then allows for the entity to be explored from the various engineering and programmatic perspectives (viewpoints). A viewpoint describes data drawn from one or more perspectives and organized in a particular way that's useful to management decision-making. The compilation of viewpoints (e.g. capability, operational, system, programmatic viewpoints) represents the entire system, allowing the system to be explored as a whole or from a single perspective."

In traditional systems engineering, the system is represented in diagrams, tables, textual descriptions, spreadsheets, etc. As a result, each entity is often represented several different times, possibly in conflicting ways. For example, a requirement for a single entity may be represented at several levels of the requirements documentation hierarchy. Traditional systems engineering captures the different perspectives and requirements of the system but fails to depict what the system will "look like" after it is developed because it fails to capture the underlying ontology and structure of the system. The essence of

MBSE, where the system is virtually represented in the model, thus represents a significant departure from traditional systems engineering.

As we launch further into our discussion of MBSE, it is also useful to understand something we can coin the "Law of Conservation of Systems Engineering" which states:

> ***For all systems there is a minimum amount of systems engineering effort required to successfully turn a stated need into a delivered capability***

Regardless of the modeling approach you use (paper-developed or software-based), this systems engineering minimum effort can be added to, moved around, consolidated, or divided, but it cannot be reduced below the minimum and still achieve success.

Our goal for MBSE is to capture information so that we can make design decisions throughout the lifecycle. To implement MBSE, we need a language that consists of specialized words and their relationships (ontologies), as well as the associated visualizations of these words and relationships in the form of diagrams.

Modeling languages capture a formalized ontology (internal logic) to capture and connect relationships along with standards to facilitate data sharing between tools as part of a MBSE implementation. In a MBSE environment, ontologies become increasingly important, because it not only provides the entity definitions, it defines the relationships that allow the system structure to be built. Structure not only allows the system to be virtually modeled, it also allows for the discovery of emergent system behaviors.

However, it is critical not to conflate one with the other. Don't confuse languages with MBSE. For example. SySML is to MBSE as C++ is to Software Engineering. C++ is a language. Software engineering is a discipline. Here we will treat MBSE as a discipline that draws upon formalized languages and standards as part of a practitioner's tool kit.

Numerous modeling languages have been defined. A short list, in alphabetical order, includes:

- *Architecture description language (ADL)* is a language used to describe and represent the systems architecture of a system.
- *Business Process Modeling Notation (BPMN, and the XML form BPML)* is an example of a Process Modeling language.
- *C-K theory* consists of a modeling language for design processes.
- *DRAKON* is a general-purpose algorithmic modeling language for specifying software-intensive systems, a schematic representation of an algorithm or a stepwise process, and a family of programming languages.
- *Fundamental Modeling Concepts (FMC)* modeling language for software-intensive systems.
- *IDEF* is a family of modeling languages, which include IDEF0 for functional modeling, IDEF1X for information modeling, IDEF3 for business process modeling, IDEF4 for Object-Oriented Design and IDEF5 for modeling ontologies.
- *Lifecycle Modeling Language (LML)* (the subject of this book) is an open-standard modeling language designed for systems engineering. It supports the full lifecycle: conceptual, utilization, support and retirement stages. Along with the integration of all lifecycle disciplines including, program management, systems and design engineering, verification and validation, deployment and maintenance into one framework.
- *Specification and Description Language (SDL)* is a specification language targeted at the unambiguous specification and description of the behavior of reactive and distributed systems.

- *System Modeling Language (SysML)* is a Domain-Specific Modeling language for systems engineering that is defined as a UML profile (customization).
- *Unified Modeling Language (UML)* is a general-purpose modeling language that is an industry standard for specifying software-intensive systems. UML 2.0, the current version, supports thirteen different diagram techniques, and has widespread tool support.

For most of the languages listed above, diagrams can be generated from the underlying data (or, in some cases, this data can be generated after the diagrams are drawn). The beauty of these diagrams (and the information they capture) is in their ability to convey complex details about the system within the ontology in a compact, understandable way. However, it very important to emphasize that the practice of MBSE is ***more*** than simply generating nifty diagrams. It is the underlying data structure with related artifacts that capture the system description and provide a means for defining complex relationships between the data elements that is the true value of the model. The diagrams provide a convenient way to communicate this complex information and relationship. There are numerous types of diagrams possible from the underlying data (more on diagrams later).

- Hierarchical
- Relational
- Operational
- Logical sequence and flow (pseudo-code)
- Interfaces

We can use this collection of data and their relationships to generate traditional paper (or virtual paper) products, such as CONOPS, System Requirements Documents, Interface Control Documents, etc. One of the greatest advantages of this approach, especially by using software-based tools, is that the model is the ***single source of truth***, the documents are only a convenient snapshot of the truth.

To understand these languages and their associated ontologies we recommend you review specifications for a given language. We will examine the details of LML over the rest of this book.

THE LML TAXONOMY

A *taxonomy* describes the vocabulary for information categories you want to collect. Often in science we call this *classification*. You probably remember your high school biology class when you first learned about how we classify animals into domains, kingdoms, phyla, etc. Figure 5 shows an example from biology.

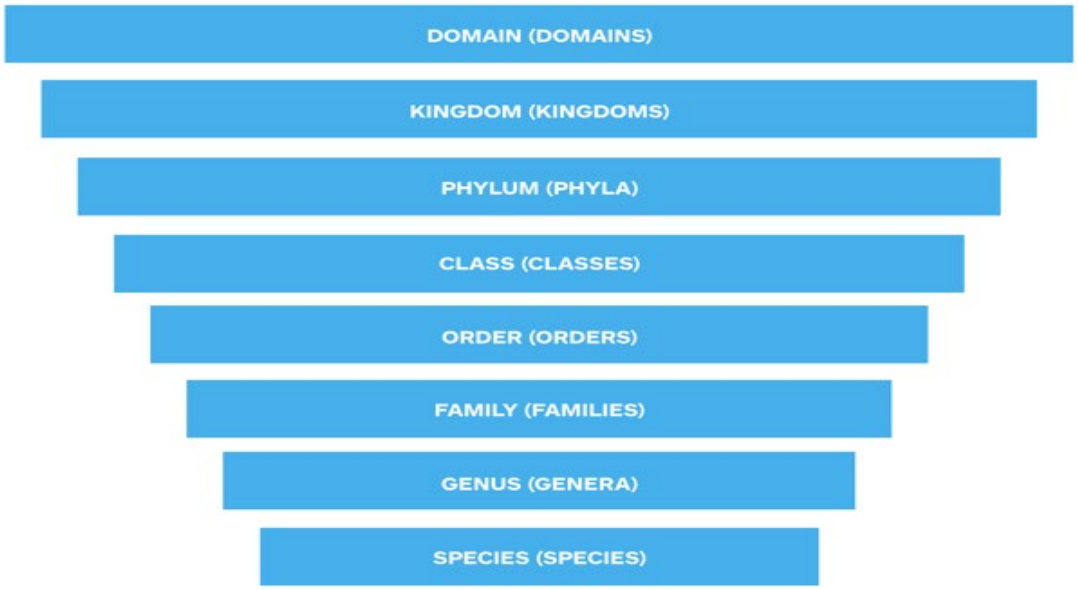


Figure 5. An Example of Taxonomy from Biology

In systems engineering, we are interested in the functional, physical, and programmatic information needed to design, develop, build, test and operate a system. The purpose of this description is to specify the parts of a system in enough detail that they can be built, tested, and delivered to provide capabilities to help people do various jobs, from fighting wars, to going into space, to developing medical devices, to building energy sources for the future. In other words, any system that we need in any field.

Most of the systems we develop today are very complex and, if we aren't careful, it's easy to develop a very complex taxonomy to match. But that makes the job of describing the system clearly and succinctly very difficult. With a complex taxonomy we run into problems as to "what bin does this information go in?" For example, the DoD Architecture Framework versions 1.0 and 1.5 used a data model called the Core Architecture Data Model. A data model includes a taxonomy. A couple of the elements in that taxonomy are "operational activities" and "system functions." Both elements are functional in nature. Many architecture practitioners assumed that operational activities are functions performed by people and system functions are functions performed by hardware and software. In actuality, these elements are essentially the same thing, but at different levels of abstraction. Hence, the confusion.

When we have many "bins" to put the information in, we need to be very careful not to duplicate the information in multiple places. Such duplication causes confusion and may result in poor specifications of the system.

Developing your own mental map of systems engineering nouns, verbs and adjectives requires a lot of careful thought and can be quite time consuming and would involve much re-inventing of the wheel. Fortunately, that is where the Lifecycle Modeling Language (LML) comes in. We've already invented a "wheel" with pre-defined ontology that we propose is simple, complete and applicable to any system.

LML was developed with this concern in mind. We wanted to balance the number of "bins" or "entity classes" with the need to gather a sufficient amount of information about the system and associated project. We quickly recognized that although people use different names for something, they were really just different types of the same thing. For example, "operational activities" and "system functions" are really both different types of a functional element. LML calls the functional element an "Action," to avoid the confusion that was occurring between an operational activity and a system function.

As a result of the LML Steering Group's analysis, we came up with the set of entity classes seen in the Figure 6. Each of these entity classes have distinctive attributes (which is an inherent characteristic or quality of an entity) that distinguish them from each other. If we think of an *entity class* as the noun in the language, the *attribute* serves as the adjective.

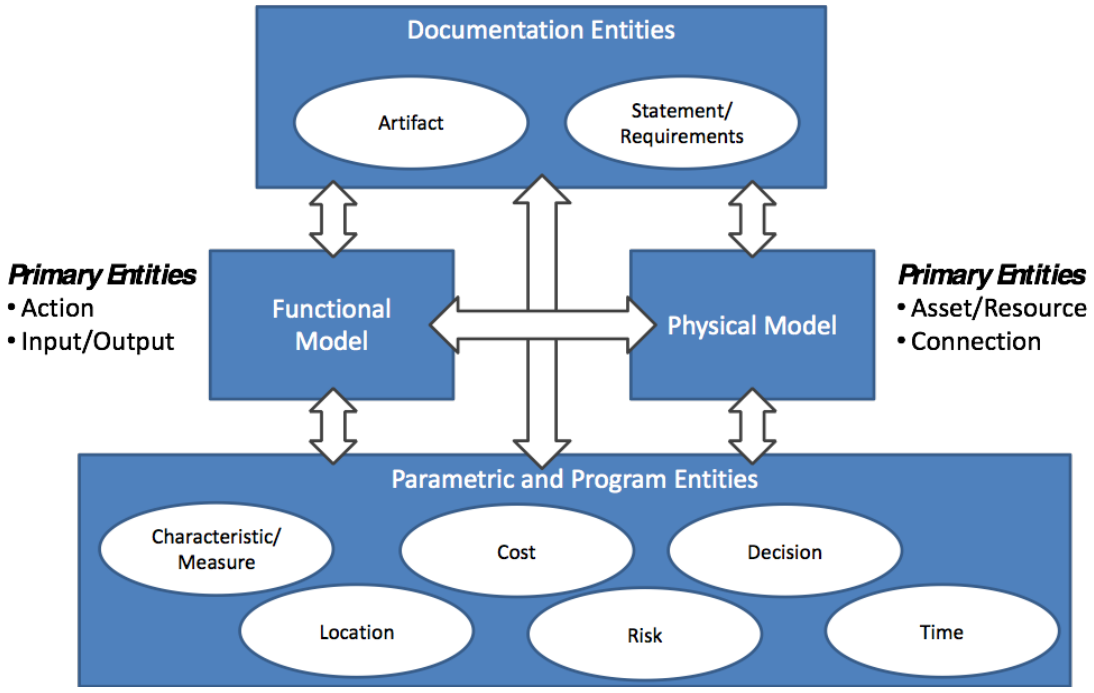


Figure 6. The basic building blocks of LML. LML is composed of entities. The entities are grouped as shown here.

Let's go through each one of the entity groupings to help you better understand them. To enhance that understanding and make the following discussion less abstract and more tangible, we will introduce a specific system of interest to give us a threaded example to refer to. Our system of interest will be a simple solar-powered fan shown in Figure 7. As you can see, this simple system consists of a tiny solar array, a motor, and a propeller.

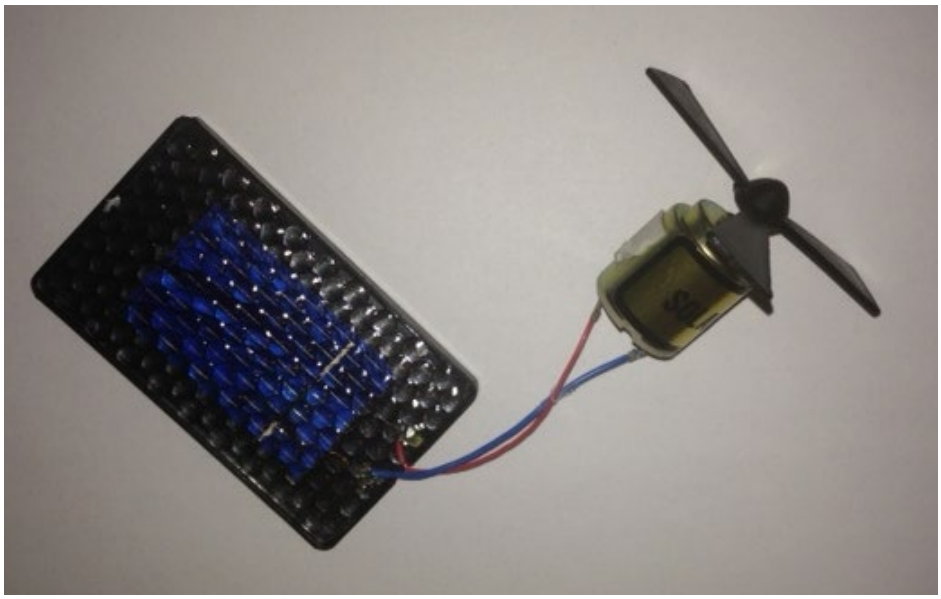


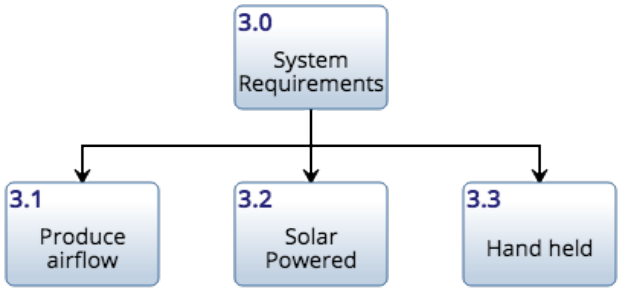
Figure 7: Solar Fan System of Interest

Let's start with the Documentation Entities. Documents are a critical tool for communicating every aspect of a project. This book is a document. It contains the information in a simple to read, linear form, enabling a person to understand the subject in a logical flow. Documents can include pictures (graphics) as well as text. To be useful then, a modeling language should address documentation entities.

In LML, the documentation entity classes include *Artifact*, *Statement*, and *Requirement* (with Requirement being a sub-class of Statement). The Artifact entities specify a document or other source of information that is referenced by or generated by another entity. Examples of an Artifact include: Document, E-mail, Procedure, or Specification.

Assuming our solar fan system came with a User's Manual, it would be an important document to capture as an Artifact. Statement entities specify text from an Artifact. Examples of Statements include: the project Need, project Goals and Objectives, and Assumptions. A Requirement entity identifies a capability, characteristic, or quality factor of a system that must exist for the system to have value and utility to the user. Examples of Requirements include: Functional Requirements, Performance Requirements, and Safety Requirements.

We can define some simple requirements for our solar fan system shown in Figure 8 as both a diagram and textual description. ***NOTE:*** These are not offered as examples of "good" requirements, that discussion is outside the scope of this book. We are simply offering these as examples of requirement entities that LML would capture.



Number	Requirement Description
R3.0	System Requirements
R3.1	The system shall produce cooling airflow.
R3.2	The system shall be powered by solar energy alone.
R3.3	The system shall be hand held.

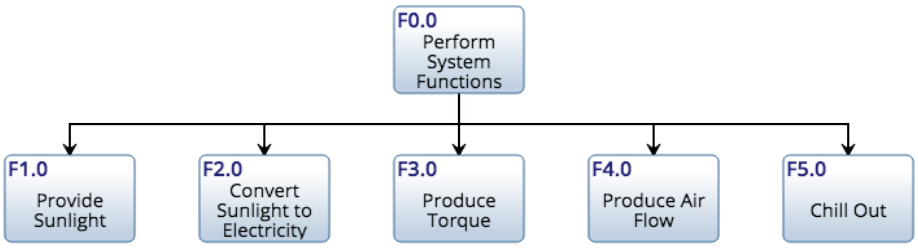
Figure 8. Requirements for the Solar Fan

The next grouping of entities in LML to consider is the functional model. The functional model provides a way to describe how a system behaves. As with children, some behavior is good, and some is bad. Our job as systems engineers, just like parents, is to emphasize the good behavior and root out the bad behavior. We also want to make sure the behavior is predictable. The last thing a user or operator of a system wants is "emergent behavior," also known as "undocumented features" that lead to degraded performance and serve to frustrate operators and vex maintainers.

The *functional model entity classes* consist of Action and Input/Output. An Action specifies the mechanism by which inputs are transformed into outputs. Examples of Actions include (but are not limited to): Activity, Capability, Event, Function, Process, or Task. An Input/Output entity specifies the information, data, or object input to, trigger, or output from an Action. Examples of Input/Outputs include (but are not limited to): Item, Trigger, Information, Data, and Energy.

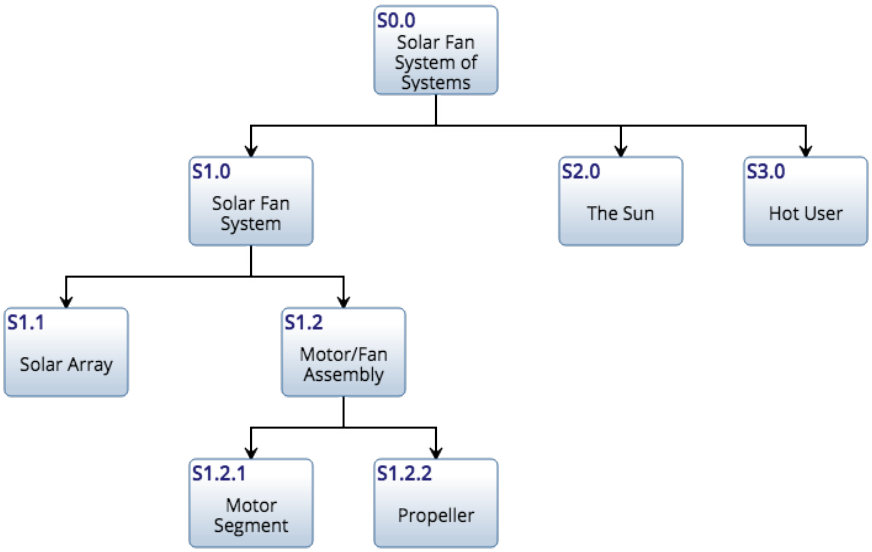
To help that make more sense, let's return to our solar fan system of systems. Figure 9 lists all the functions for the system.

A physical model provides a means for describing the pieces of the system and how they are connected to each other; what we often call interfaces or relationships. The *physical model entity classes* in LML consist of *Asset*, *Resource*, and *Connection*, where a Resource is a sub-class of the Asset. An Asset specifies an object, person, or organization that performs Actions, such as a system, subsystem, component, or element. Examples of Assets include (but are not limited to): Component, Entity, Service, Sub-system, and System. The list of physical assets for the solar fan system of systems is shown in Figure 10. A Resource entity specifies a consumable or producible Asset, such as fuel, bullets, energy, and people. For the solar fan example, solar energy and electrical power could be considered resources.



Number	Functions
F0.0	Perform System Functions
F1.0	Provide Sunlight
F2.0	Convert Sunlight to Electricity
F3.0	Produce Torque
F4.0	Produce Airflow
F5.0	Chill out

Figure 9. Solar Fan System of System Functions



Number	Assets
S0.0	S0.0 Solar Fan System of Systems
S1.0	S1.0 Solar Fan System
S1.1	S1.1 Solar Array
S1.2	S1.2 Motor/Fan Assembly
S1.2.1	S1.2.1 Motor Segment
S1.2.2	S1.2.2 Propeller
S2.0	S2.0 The Sun
S3.0	S3.0 User

Figure 10. Solar Fan System of System Assets.

The *Connection* is an abstract entity class that specifies the means for relating Asset instances to each other. It has two sub-classes: *Conduit* and *Logical*. A Conduit entity specifies the means for physically transporting Input/Output entities between Asset entities. It has limitations (attributes) of capability and latency. Examples of a Conduit include (but are not limited to) Data Bus, Interface, and Pipe. A Logical entity represents the abstraction of the relationship between two entities (e.g., Asset entities with the type "Entity"). Examples of a Logical connection include: "has," "is a," and "relates to." ***NOTE:*** This Logical entity class supports data modeling. Figure 11 shows the "Array to Motor Connector" conduit between the Motor/Fan Assembly and the Solar Array. As implemented, this conduit is the power and ground wires between the two assets.

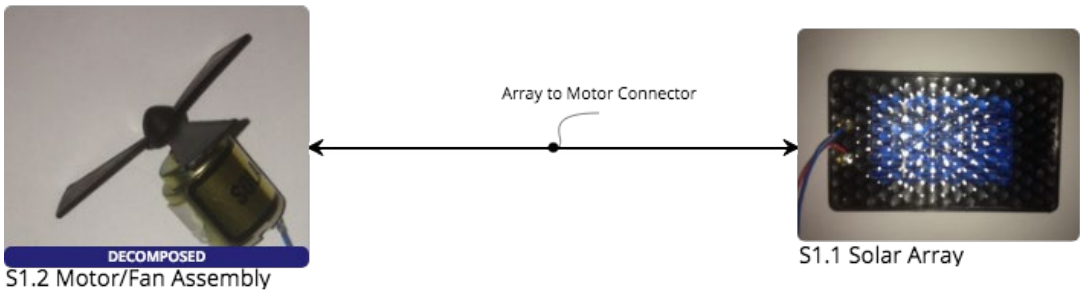


Figure 11. Motor/Fan Assembly to Solar Array Conduit

The last grouping of classes consists of parametric (properties of the system) and program (information needed by the program management to control the development and operations of the system) entities. The *parametric and program entity classes* consist of *Characteristic* (with its sub-class of Measure), *Cost*, *Decision*, *Location* (with its sub-classes of Physical, Orbital, and Virtual), *Risk*, and *Time*. Let's start with the parametric entities of Characteristics and Measures.

A *Characteristic* specifies the properties of an entity, such as Attribute, Category, Color, Power, Role, Size, and Weight, and hence they have a *value* attribute with each Characteristic. A Measure entity specifies properties of measurements and measuring methodologies, including metrics. The Measure entities include the Key Performance Parameters (KPP), Measures of Effectiveness (MOE), Measures of Performance (MOP), and other metrics.

The program entities consist of the remaining classes. A *Cost* entity specifies the outlay or expenditure (as of effort or sacrifice) made to achieve an objective associated with another entity. Examples of types of Cost entities, include (but are not limited to): Earned Value, Work Breakdown Structure, Actual Cost, and Planned Cost. A *Decision* entity specifies a challenge and its resolution. Types of Decisions include (but are not limited to): Major Decision, Challenge, Issue, and Problem. ***NOTE:*** Capturing decisions should be a key part of a "decision database," which is used to determine the course of a project or program.

Another entity class that is very important to both the systems engineer and program manager is *Risk*. A Risk entity specifies the combined probability and consequence in achieving objectives. Examples of Risk types include (but are not limited to): Cost Risk, Schedule Risk, and Technical Risk. Figure 12 depicts the risk of injury from using the solar fan as moderate.

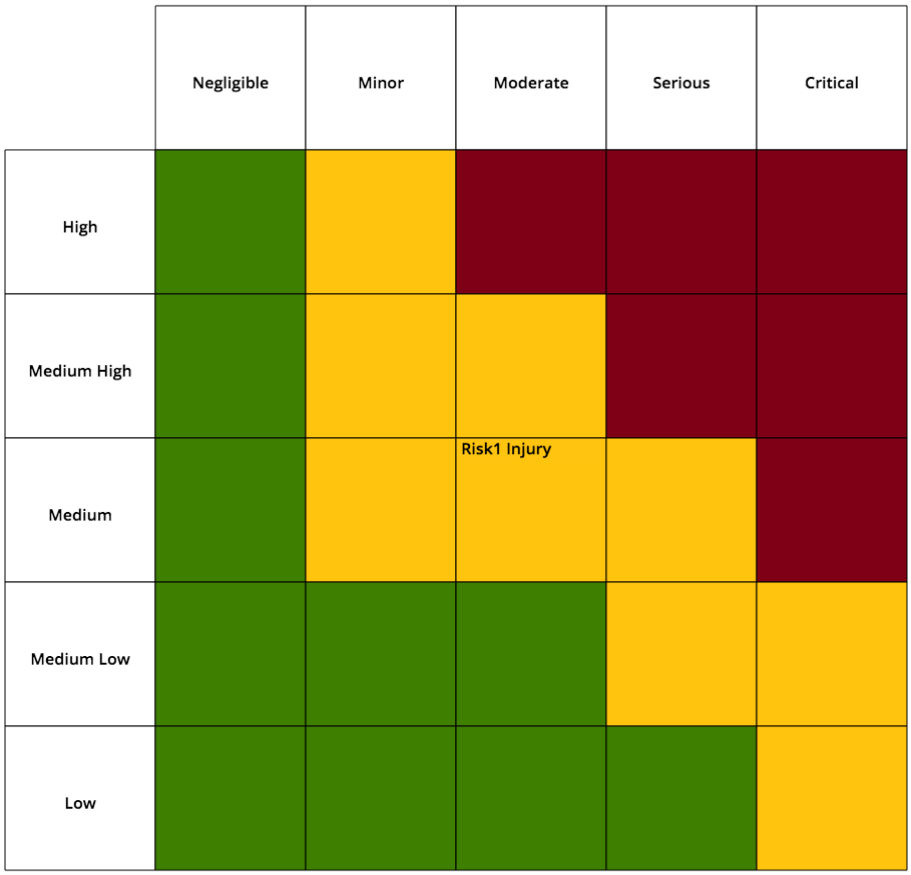


Figure 12. "Stop light chart" risk matrix showing a moderate injury risk from using the solar fan.

Time is another key factor for program managers. A Time entity specifies a point or period when something occurs or during which an action, asset, process, or condition exists or continues. Examples of Time entities are Milestone and Phase.

> ***NOTE:*** *With Characteristics/Measures we capture* ***performance*** *information. With Time we capture* ***schedule****-related information. So, taken along with Cost, we can use these classes of information to* ***optimize cost, schedule, and performance*** *of the system or the program. Many people define the role of a systems engineer and program manager as someone who optimizes cost, schedule, and performance within acceptable risk for the system or program, respectively.*

The last set of entity classes associated with parametrics and programmatics is the *Location* classes. A Location is an abstract entity class that specifies where an entity resides. This is sometimes known as the system context. Locations can include Orbital, Physical, or Virtual. An Orbital Location entity specifies a location along an orbit around a celestial body. A Physical Location entity specifies a location on, above, or below the surface, i.e., map coordinates and elevation. Finally, a Virtual Location entity specifies a location within a digital network. Since we are usually referring to a location on the Internet, this usually becomes the URL for that webpage. ***NOTE:*** When combined with Time entities, the Location provides a way to describe the movement of an object through space and time.

As noted in the forward to this book, LML is intended to be the 80% solution. We know various projects may need more "bins" to capture specific pieces of information. For example, when SPEC Innovations wanted to add a "Test Center" to the Innoslate® tool, they discovered that they needed a "Test Case" class, as it had unique attributes and relationships. They added it as a sub-class of the Action class, which allowed them to inherit the Action's attributes and relationships, as well as diagrammatic representations, including the Action Diagram. The benefit of using the Action as a parent class was that they could model the test processes as well using the test cases, thus reducing the complexity of the modeling and ensuring a better set of tests that met cost and schedule constraints.

The LML taxonomy of twelve primary classes, with eight sub-classes, provides a robust, but simple set of bins to store our information. This is a good starting point, but next we need to define the relationships between these "bins" of information. The relationships between the classes is where complexity is captured, as we will see in the next chapter.

LML ONTOLOGY

The first chapter described the basic relationship between taxonomy, ontology and language. While LML's taxonomy is relatively simple, the relationships between the classes provide the necessary complexity for the language. Almost every class is related to every other class and have multiple relationships between each other. Figure 13 shows the primary set of relationships that form the basic traceability of LML.

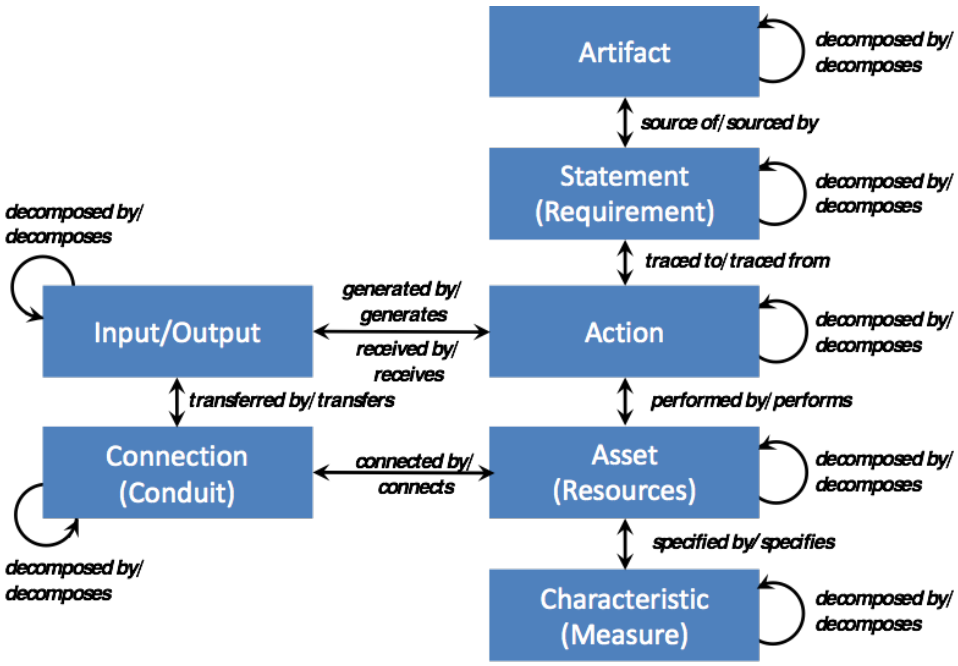


Figure 13. LML ontological relationships.

One of the first things you may notice in Figure 13 is that all entity classes can be ***decomposed***, allowing you to identify high levels of abstraction and break them down progressively into smaller and smaller pieces easy enough to understand and communicate. You also see that every relationship has a reciprocal relationship (e.g., "**decomposed by**" has an inverse relationship "***decomposes***"). To simplify the language, LML uses the same verb for each direction. This feature reduces confusion and makes the relationships easier to learn.

The flow shown in Figure 14[1] follows the classic systems engineering approach of Requirements Analysis, Functional Analysis and Allocation, and Synthesis of the physical implementation or solution. The top down approach begins with the analysis of a requirements document (Artifact) that is broken down into Statements (contextual information from the document) and Requirements (which often include a "shall" statement). So, the document becomes the ***source of*** the Statement/Requirements (see Figure 13).

[1] This figure was originally developed as part of the MIL-STD-499B draft, which was used as part of EIA-632. The overlay of top-down, middle-out, and bottom-up comes from Dr. Steven Dam's book on the DoD Architecture Framework.

Obviously, this isn't the only way that systems engineering evolves. The diagram below shows we can also work in the other direction (Middle-Out), starting with functional analysis and abstracting to create a set of requirements. We often find that enterprise architectures use this approach, as they often don't have an explicit set of requirements defined.

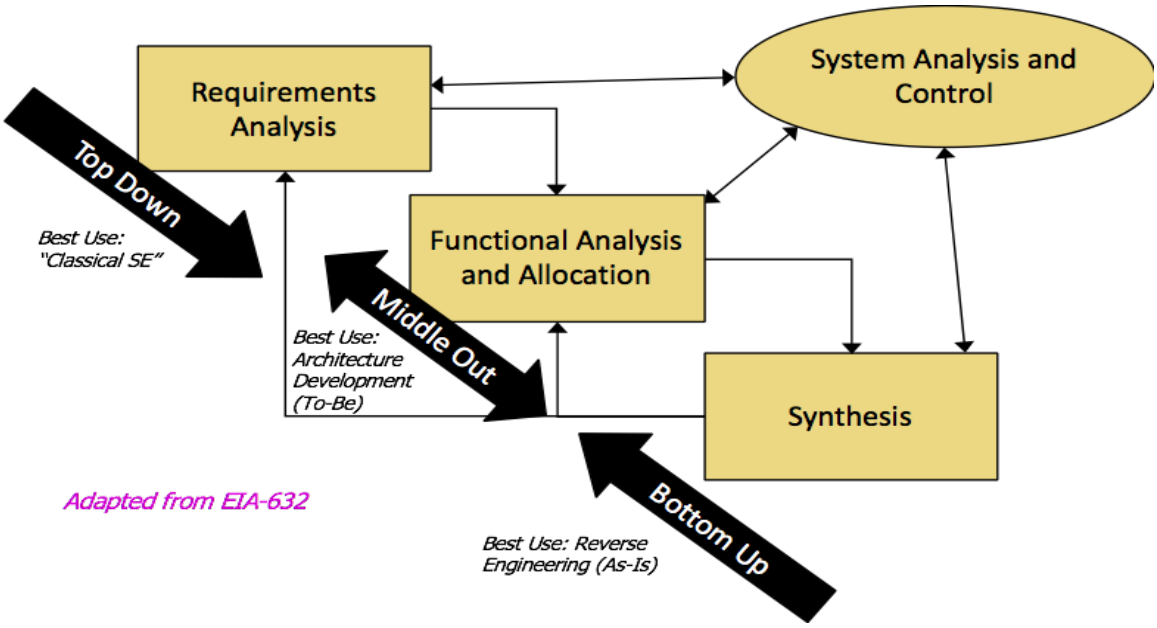


Figure 14. Relationships between systems engineering activities.

Continuing from Figure 13, the functional analysis step uses the Action and Input/Output entity classes then relates them back to the requirements using the ***traced from/traced to*** relationship. That same relationship can be used to trace directly to an Asset as well, when it's a non-functional requirement, although it's not shown on the diagram.

Action are then allocated to Assets using the ***performed by/performs*** relationship. In addition, the Input/Outputs class entities use the ***transferred by/transfers*** relationship to show their allocation to a subclass of Connection: Conduit.

The primary set of relationships is completed by the inclusion of the Characteristic class and its subclass: Measure. A Characteristic or Measure class entity ***specifies*** an Asset class entity as shown in the diagram, but it can also be used to specify any of the other class of entities. You can think of this class as the place where you can capture metrics and relate them back to the appropriate entities.

As with the Characteristic class entities discussed in the previous paragraph, although relationships shown in Figure 13 are the primary relationships, you will find that almost every class needs to be related to every other class, often with more than one relationship possible. Figure 15 shows a portion of the very large table from section 3.2 of the LML specification. In it you can see another common relationship within a class: ***related to/relates***. The relationship is intended to be a "peer-to-peer" relationship, instead of a hierarchical relationship, like the ***decomposed by/decomposes***. You might use this when a requirement in one part of a document (Artifact) is related to another requirement within the same document.

	Action	Artifact	Asset (Resource)	Characteristic (Measure)	Connection (Conduit, Logical)	Cost	Decision	Input/Output
Action	decomposed by* related to*	references	(consumes) performed by (produces) (seizes)	specified by	-	incurs	enables results in	generates receives
Artifact	referenced by	decomposed by* related to*	referenced by	referenced by specified by	defines protocol for referenced by	incurs referenced by	enables referenced by results in	referenced by
Asset (Resource)	(consumed by) performs (produced by) (seized by)	references	decomposed by* orbited by* related to*	specified by	connected by	incurs	enables made responds to results in	-
Characteristic (Measure)	specifies	references specifies	specifies	decomposed by* related to* specified by*	specifies	incurs specifies	enables results in specifies	specifies
Connection (Conduit, Logical)	-	defined protocol by references	connects to	specified by	decomposed by* joined by* related to*	incurs	enables results in	transfers

Figure 15. Partial view of LML ontological relationships.

Another relationship within the same class is the special one added for Assets: ***orbited by/orbits***. Since LML was designed with the aerospace community in mind, we wanted to be sure that we could model the relationship between celestial bodies, like the Sun, Earth, and Moon. So, a Sun entity would be *orbited by* an Earth entity, which in turn would be ***orbited by*** a Moon entity.

Between different classes you can see multiple relationships as well. For example, an Action ***consumes***, ***produces***, or ***seizes*** (i.e. uses the resource for the duration of the Action and then releases the resource) a Resource. An Action also ***enables*** or ***results in*** a Decision. To avoid having too many relationships, we reused as many as possible, as you can see with the ***specifies/specified by*** relationship.

These relationships can be visualized as shown in Figure 16.

The entity in the middle of this diagram (3.0, System Requirements) is the hub of the diagram and the entities related to it are shown with the relationships that have been established. It is ***decomposed by***: 3.1, Produce Airflow; 3.2, Solar Powered; and 3.3, Hand Held. It is also ***satisfied by*** the S1.0, Solar Fan System, which is an Asset class entity. You can also see in this diagram how the Solar Fan System is ***decomposed by***: S1.1 Solar Array; and S1.2 Motor/Fan Assembly.

As you can see much of the complexity of the system can easily be captured using these many relationships. But they are not sufficient to capture the full complexity of the systems we want to model. So, we have included attributes on the entity classes and the relationships. The next chapter discusses attributes.

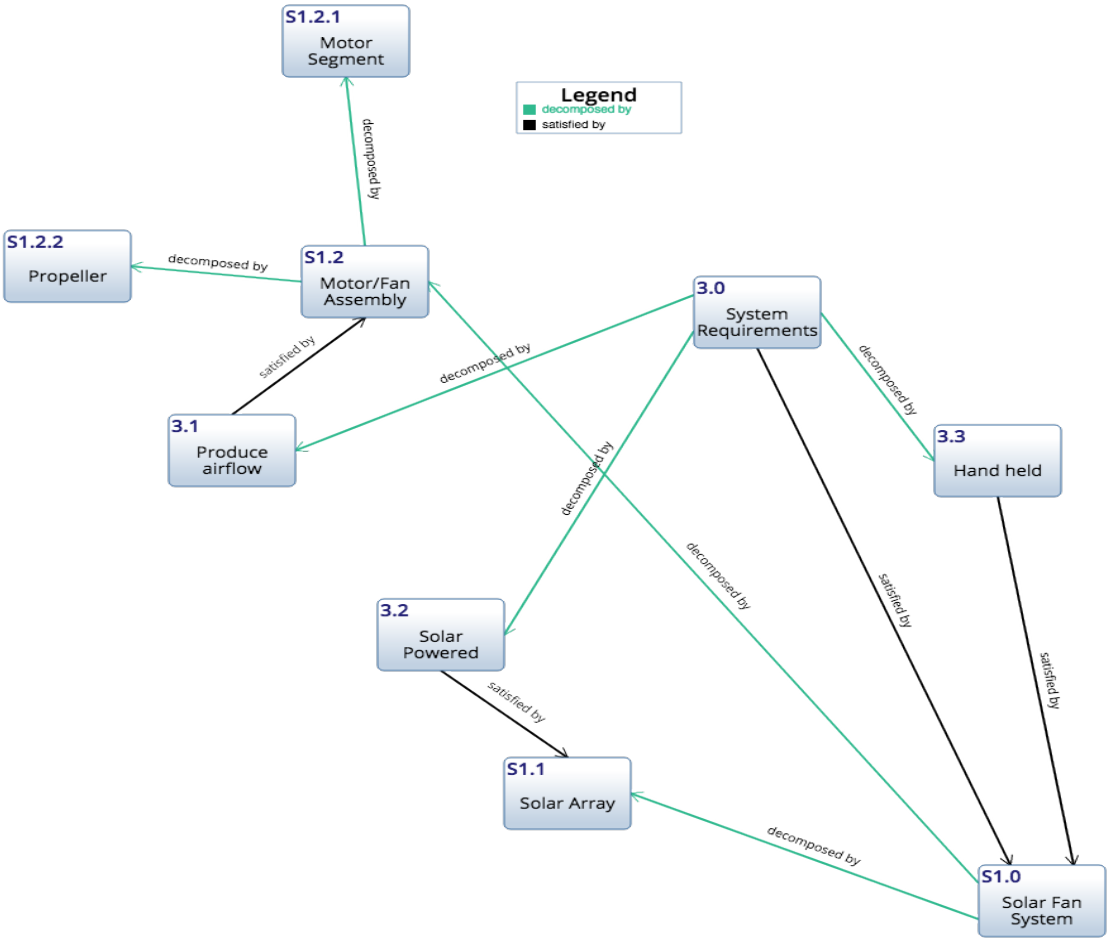


Figure 16. Solar Fan Relationships.

ATTRIBUTES: THE ADJECTIVES AND ADVERBS OF LML

In describing a language, we distinguish between nouns (truck) and verbs (runs). But a language is not complete unless you can modify the nouns and verbs. In LML, the entity classes are the nouns and the relationships the verbs. So, having attributes on each is the equivalent of having adjectives (*red* truck) and adverbs (red truck runs *slowly*). Just like adjectives and adverbs, attributes modify or clarify the noun or verb. Since our goal is to correctly and completely, describe the system and all its capabilities and behaviors, we use these modifiers to support that goal. Let's start with the adjectives.

ATTRIBUTES ON CLASSES

Every entity class in LML has at least three attributes available: *name*, *number*, and *description*. These three attributes provide a way to easily identify the individual entities within a class. The *name* can be anything, but common best practices use nouns to begin the name if it represents a physical entity, such as an Asset or Resource. If the name represents a functional entity, such as Action, we usually use a verb. For example, if we have a satellite as our system, we use the Asset class to capture this piece of information and give it a name, such as "Hubble Space Telescope." If we want to describe the Action entity that launches the satellite into orbit, we might call that Action: "Launch Satellite into Orbit." This is not a hard and fast rule, but it helps people understand the difference between physical and functional things.

The *number* can be anything as well. Although adding a number is not required, it is considered a best practice. Often, hierarchical numbering schemes are used to represent the parent-child relationship, such as in the example in the previous chapter of VR.1 and VR.1.1. We could have used just 1 and 1.1 for this, but with many hierarchies in a database it gets confusing, so another best practice is to use some identifier that lets you know what the name or type of entity is. In this example, the VR stands for verification requirement. LML does not dictate this approach, but as you get used to using an ontological approach to capture and organize your information, you will find these heuristic*s*, help you tremendously.

Finally of the three universal attributes, we have the *description* attribute. Unfortunately, this one, which is arguably the most important one, is the one most often ignored by modelers. The problem comes in that we tend to use a word or few words for the *name* of the entity. Modelers do that because we often use a graphical user interface (GUI) to create a diagram with these entities. Having the name be a complete description of the entity would usually be too long to fit in the large number of boxes we put on a diagram. For example, we might use the word "Tank" to represent the entity of interest and we may know that kind of tank we mean (water, gas, a tracked vehicle with a cannon, etc.), but that usually isn't a sufficient amount of information. The description then gives the modeler a place to add relevant information about the entity. This can be anything from a short definition to a detailed specification.

Another common, but not universal, attribute in LML is the *type* attribute. The *type* attribute is essentially a synonym for the class itself. For example, the types of Action provided in the LML specification include: Activity, Capability, Event, Function, Mission, Operational Activity, Program, Service Orchestration, Simulation Workflow, Subprocess, System Function, Task, Training, Use Case, Work Process, and Workflow. The purpose of the *type* attribute is to avoid creating lots of classes, which essentially have the same attributes and relationships. Other ontologies will often create each of these different types of Actions as separate classes. But this can lead to confusion. What's the difference between an Operational Activity and

a System Function? Both are functional in nature. Both can have the same attributes and relationships. Some people say that one represents the functions performed by humans and the other functions performed by hardware or software. But as any experience systems engineers will tell you, that is simply a matter of *allocation*. In one model you might want a human doing that function, but in another model, you might want the software performing the function. The whole idea of having a physical and functional separation is to enable that kind of reallocation. It's often how we improve a system.

In addition to the universal and common attributes discussed above, each class has its own special set of attributes. If we continue with the Action class, we see in Figure 17 the attributes from the LML Specification.

Attribute	Data Type	Description
duration	Number	*duration* represents the period of time this **Action** occurs
percent complete	Percent	*percent complete* represents the percentage this **Action** is complete
start	DateTime	*start* represents the time when this **Action** begins

Figure 17. Action Class Attributes

The *duration* attribute captures how long the Action entity will take to complete. Timing and sequencing of Actions is a very important concept in functional analysis. Timing of the Actions in LML is handled by a combination of this *duration* attribute and the ***receives*** relationship. The sequencing is usually handled in specific diagrams, such as the well named "Sequence Diagram." In LML, we provide this sequencing using the Action Diagram, which is presented in the next chapter.

The other two attributes for the Action are used primarily for program planning and management. The *start* date and time represents when the action begins on the calendar. Using this information, along with the duration, calendar dates and times can be determined for the end date and time of a task. The *percent complete* attribute provides a way to track progress of completing a task. This attribute is similar to one used in Microsoft Project for project planning.

ATTRIBUTES ON RELATIONSHIPS

The other kind of modifier is the attribute on relationships. Staying with the Action entity class, we find this attribute affecting several relationships: ***consumes/consumed by***, ***produces/produced by***, ***receives/received by***, ***relates/related to***, and ***seizes/seized by***. Three of these relationships are between the Action class and Resource class.

An Action entity ***consumes***, ***produces***, or ***seizes*** a Resource entity. The reason for the attribute on the relationship is to determine the amount of the Resource that the Action uses or produces. The Resource class has attributes that include the *initial amount*, *maximum amount*, and *minimum amount* of the resource. When a dynamic model of a process is created using this information, we see the amount of the resource changes over time as the Actions are executed. A diagram of this using the ***consumes*** relationship is shown in Figure 18.

Figure 18. Attribute on the Consumes Relationship

Another commonly used and important attribute on a relationship is the trigger on the ***received by*** relationship for the Input/Output class. If this "adverb" is set to true, then it means that the specific Input/Output entity must be received by the Action entity before the Action entity can execute. In other words, this attribute on the relationship controls the execution of the functional model or Action Diagram. We discuss this more in the next chapter.

THE ACTION DIAGRAM

One of the mandatory diagrams[2] of LML is the Action Diagram. The purpose of the Action Diagram is to capture the functional behavior of the system. Action diagrams are by no means unique to LML, humans have been drawing simple flow charts for centuries to explain processes and relationships. There are many forms of these kinds of diagrams: Activity Diagrams, Enhanced Function Flow Block Diagrams, and Business Process Modeling Notation (BPMN), and Behavior Diagrams. These diagrams are known by their logical constructs, symbols that represent elements of logic, such as an "AND," "OR," and "LOOP." Figure 19 below shows an example of typical functional behavior diagram.

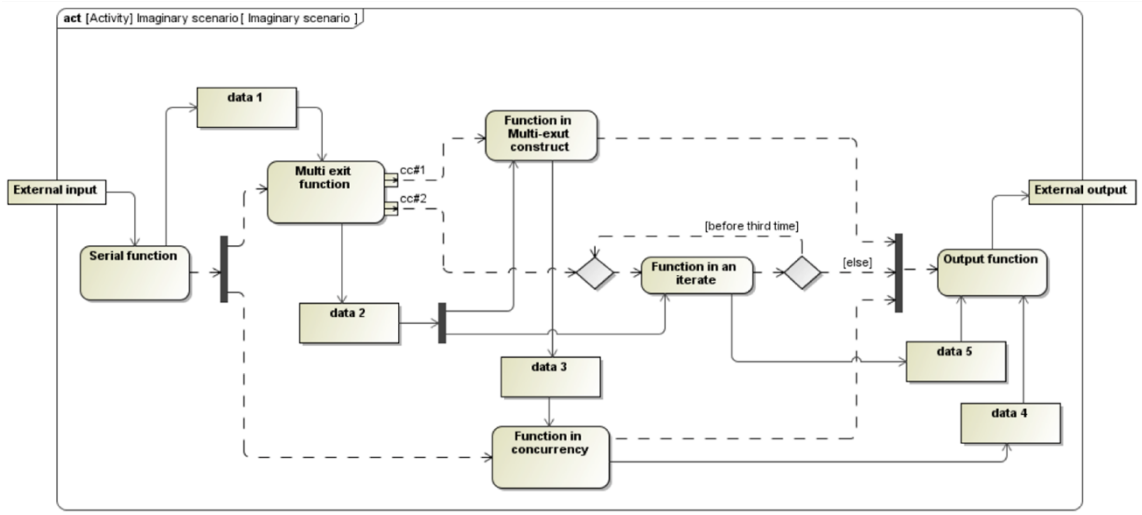


Figure 19. Typical Functional Diagram from SYSML

For those familiar with these types of diagrams, you can guess what each symbol means. The first black bar with two arrows coming out of it is implying that you have two parallel paths (concurrency). These arrows come back together or synchronize in the far-right black bar with now three arrows coming into it, since the multi-exit function split the path again into two branches. The diagram also includes data flow (the rectangular boxes, not the rounded rectangular boxes), which is also shown being produced by the functions and split/input into other functions.

While these types of diagrams can be understood with a bit of study and practice, they are not immediately obvious to the average person, especially when the behavior shown is very complex. Often, when trying to explain this kind of diagram to a senior manager, they either: 1) pretend they understand it; or 2) ask you "what is it you are trying to say with this thing." The question is often phrased more dramatically! Clearly with this approach, you need to be an expert in the language to create these drawings, but also need to be an expert to understand what they mean. By the way, we are not picking on any specific language, the other diagramming languages have the same problem.

[2] The word Mandatory here refers to diagrams that most of the committee felt were minimal to having a complete modeling representation for the logical (Action Diagram), physical (Asset Diagram), and traceability (Spider Diagram).

When trying to come up with a simpler way to depict a functional behavior diagram, we came up with the idea from electrical engineering diagrams of using a special case of the function (Action) instead of the specialized notations for logical constructs. We then decided to make them the same symbol with the name of the logical construct embedded in the box so as not to confuse someone looking at it. That way someone with a basic course in logic or high school math could understand the diagram. Figure 20 shows the symbols used for the logic in LML.

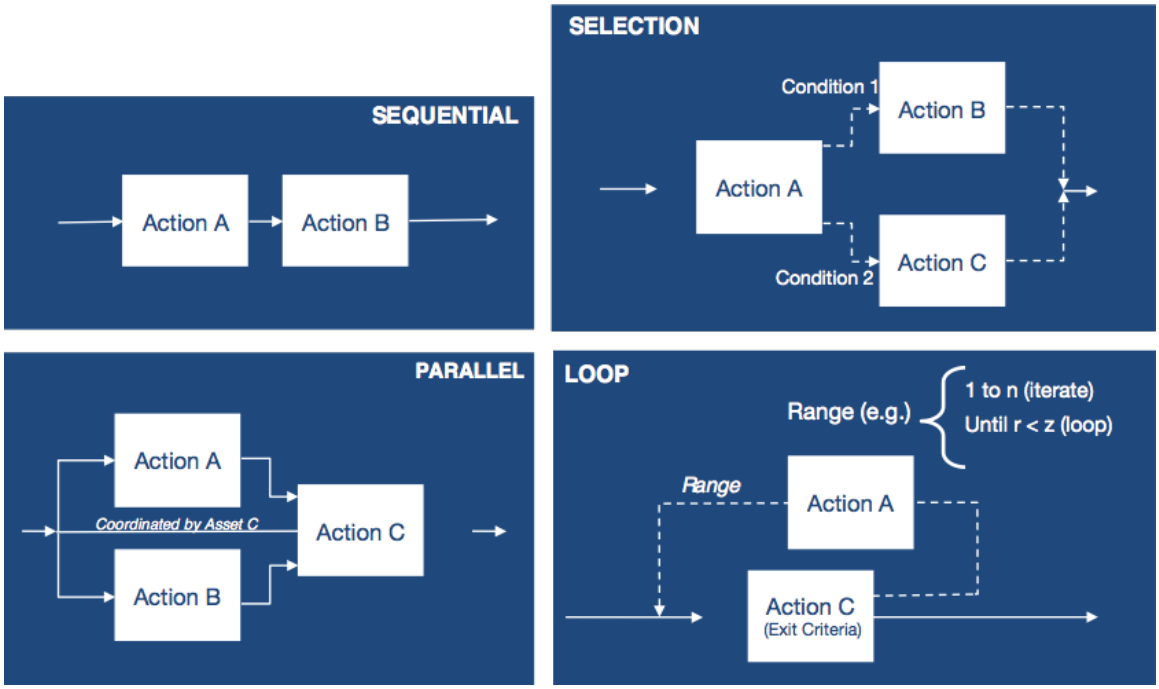


Figure 20. Action Diagram Sequencing "Constructs"

The idea was to use the same basic symbol (a rectangle) for all Actions, but have these special cases of Actions (SYNC, OR, and LOOP) have points on them, represented the overlay of a diamond, which most people recognize as a decision point. The benefit of this is two-fold: 1) now you can easily see what the logical decision point is; and 2) this decision point can now be allocated to who or what makes the decision. This latter benefit allows you now to embed your command and control, information assurance, or other decision-making process into lower levels of the design, thus providing a more realistic and useful model.

For those new to this approach, the sequential functional sequencing (top left box in diagram above) requires that the first action fully complete before the second action starts. Then when the second action completes any following logic can execute.

For the selection functional sequencing (top right box in diagram above), the OR decision point captures the determination of which path to take. If "Condition 1" occurs Action B executes; if "Condition 2) occurs Action C executes. This type of OR is often called an exclusive OR or XOR in other languages. We decided to keep it simple and since the parallel can handle the non-exclusive OR, we didn't want to confuse people.

The next decision point is a LOOP (bottom right in diagram above). The LOOP action represents the decision to enter the loop and the decision on when to exit the loop. ***NOTE:*** The range can be an iteration from 1 to

any number, as well as any conditional logic, thus looping until a value meets a specific criterion, such as the one shown above where the value of r is less than the value of z.

Perhaps the most complicated decision point is the SYNC (bottom left in diagram above). The SYNC point represents the decision on whether to wait for the completion of Action A and Action B. The basic parallel construct (without a SYNC point) would require both Actions to complete before the sequencing would proceed. With the SYNC point, a decision can be made to proceed based on any other criteria. For example, if Action A is performed by one group of people and Action B another, then the decision maker performing Action C may decide after a period of time to not wait on one or both to complete before moving on. In this case a "timeout" has occurred, which might terminate the execution, like the end of the game in football. In computers, the act of parallel processing requires a synchronization of information before proceeding. The Action C can represent functionality as well.

In addition to functional sequencing, the Action Diagram can also include Input/Output class entities, which enable triggering of the sequence, as discussed in the previous chapter. The symbol we use for an I/O is the classic parallelogram, as in Figure 21, using our solar fan example. Called an Action Diagram by LML, the inclusion of the I/O makes it what is traditionally called an enhanced functional flow block diagram (enhanced because of the I/O).

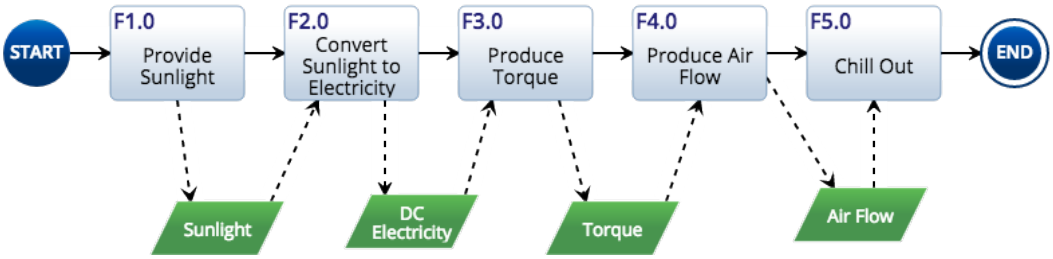


Figure 21. Solar Fan System of Systems Action Diagram showing inputs and outputs.

In this implementation, the gray indicates an optional dataflow, while the green indicates a triggering entity. The dashed lines are used for the I/O, so they are not confused with the solid lines that indicate sequencing of the Actions.

In addition to I/Os, the tool developer may want to include Resource modeling as well. It's not part of the LML specification for the Action Diagram, but it would be a useful visualization of relationships as well. An example is shown below in Figure 22, where the Resources are shown as purple hexagrams.

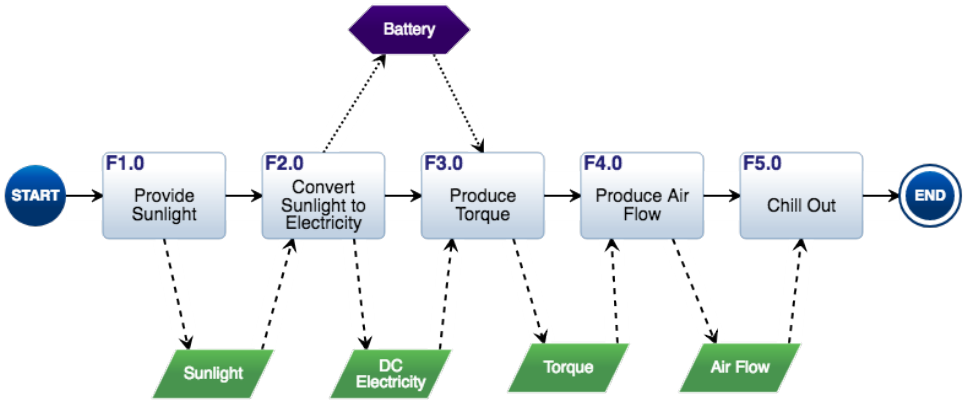


Figure 22. The Action Diagram may contain other information, such as Resources.

In this example, a "Battery" resource is used to capture excess electricity generated and stored in the battery. The "Battery" resource is then consumed when the electricity is insufficient to produce the need torque. The details of this use of the battery would be provided in the decompositions of F2.0 and F3.0. More on this topic will be provided in chapter on "LML Executability."

NOTE: The SysML Activity Diagram shown in Figure 19 does not have an equivalent for the resource modeling but could perhaps implement an "object token" addition to this diagram. Since this resource modeling is not an explicit part of the Action Diagram as provided in the LML specification, it should be viewed as an extension to the standard. We encourage such extensions, as we said LML provides the 80% solution. We just want to make sure when the standard is extended that it fits with the ontology or extends the ontology logically and explicitly. We recommend that the ontology be open and extensible.

Just having a diagram doesn't mean that we have correctly captured the logic. Software may correctly compile (no inherent logic errors), but still fail to execute properly. We can validate the Action Diagram by executing it using discrete event and/or Monte Carlo simulation. The combination of the logical sequencing, Input/Output triggers, and resource modeling provides a rich functional model for simulation. It must also be constrained by the physical limitations of the Assets and Conduits. The implementation above also provides a means for allocating Assets to the Actions in the Action Diagram, thus providing a complete integration of the physical and functional models (as long as we also allocate the Input/Output entities to the Conduits). Again, LML supports such extensions and encourages them.

Speaking of Assets and Conduits, that brings us to the second mandatory diagram: the Asset Diagram.

THE ASSET DIAGRAM

The primary purpose of the Asset Diagram is to provide a simple diagram to communicate the various assets (systems, subsystems, components, people, etc.) and how they are connected. The connection used in this diagram is captured by the Conduit class entities. Figure 23 shows a very basic example from the specification.

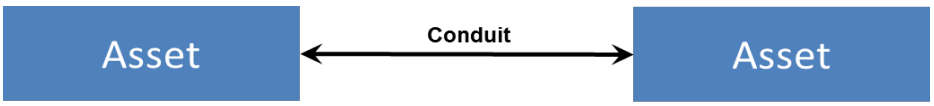


Figure 23. The Asset Diagram can be just simple boxes and lines representing the physical assets and conduits.

The Assets are represented by the rectangular boxes and the line(s) between them represent the Conduit(s), which are the interfaces between the Assets. The arrowheads are optional. Usually you would use an arrowhead only when you want to indicate a specific direction. Figure 24 shows an example of one implementation of the Asset Diagram, using the arrowheads to indicate the flow. The flow is captured in the ontology by using the origin attribute on the ***connects to*** relationship between the Conduit (for example, Sunlight) and the Assets (Solar Fan and Sun).

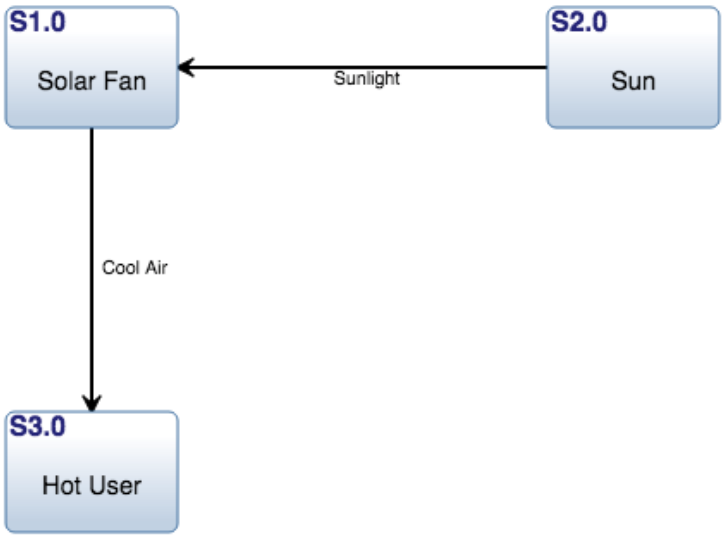


Figure 24. One implementation of the simple Asset Diagram.

Another way to use the Asset Diagram is to substitute pictures for the Assets and have different types of lines to represent the Conduits, as in Figure 25. A diagram like this serves the purpose of a DoDAF OV-1: High Level Concept Diagram. The OV-1 is usually developed in a drawing tool but using LML provides the same capability with an ontological basis. Thus, the individual entities can then be allocated and traced to wherever they are needed. In this case, we have used our solar fan example again. The sun is shown as a picture and a lightning bolt is used to represent the Conduit between the Sun and the Solar Fan System. These images make the diagram more interesting and easier to communicate to a broader audience.

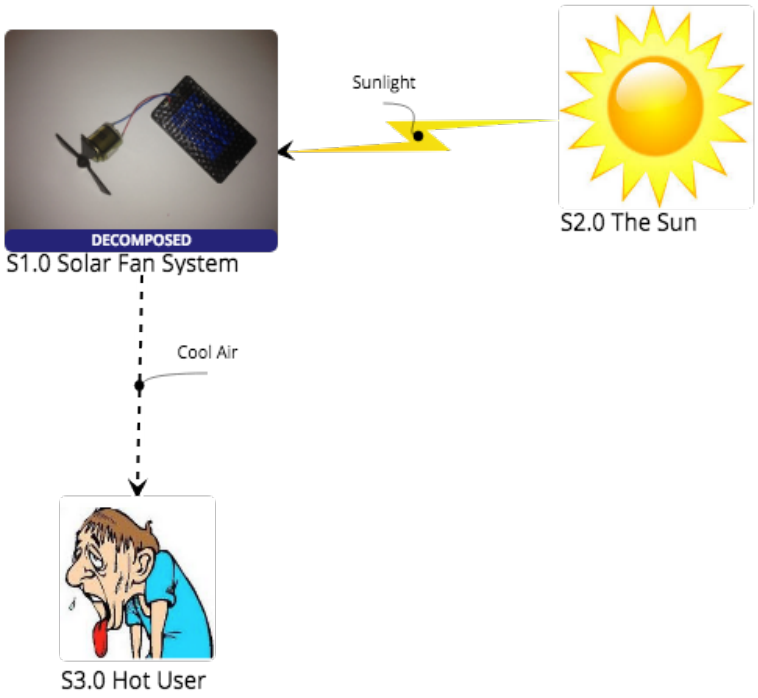


Figure 25. Turing the Asset Diagram into a pretty picture.

All these relationships are critical to capturing information about the system and allow us to communicate it through the decomposition and analysis. The last mandatory diagram, the Spider Diagram, shows these relationships explicitly. The next chapter discusses this diagram.

THE SPIDER DIAGRAM

The last mandatory LML diagram is the Spider Diagram. The purpose of the spider diagram is to provide a way to communicate the detailed traceability of entities and their relationships to other entities. We saw an example of the spider diagram in the second chapter on relationships. Another example of the Spider Diagram is shown in Figure 26.

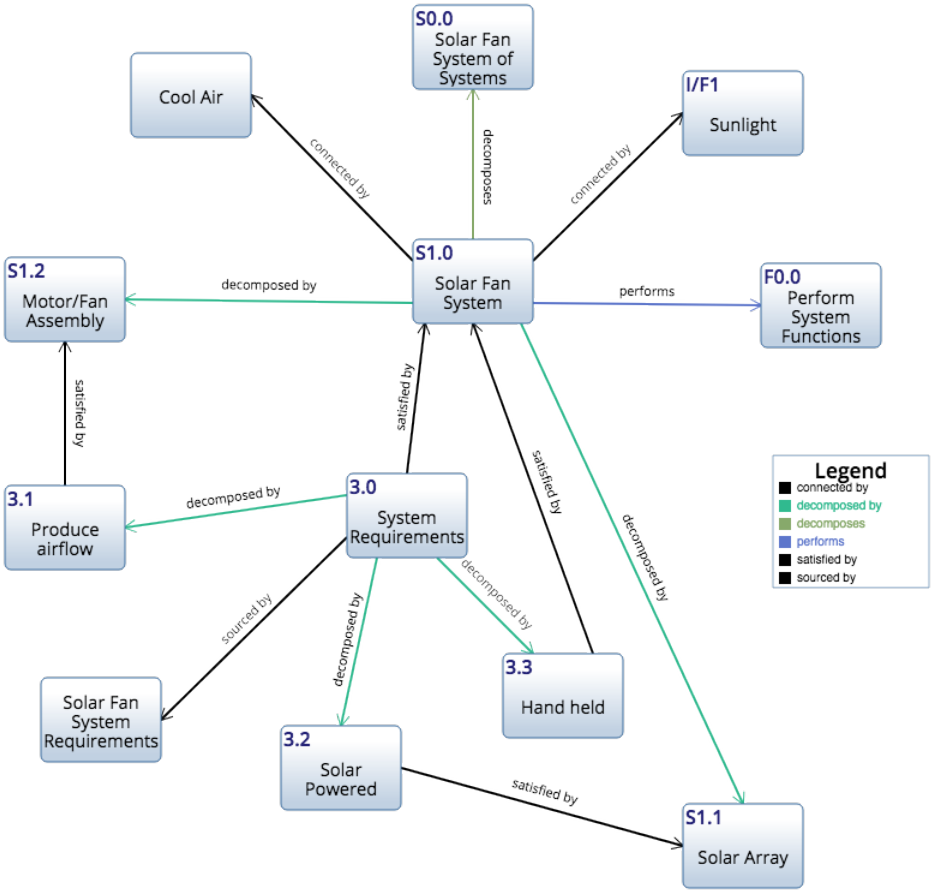


Figure 26. Spider Diagram shows the relationships between entities.

In this example, we can see how the System Requirements (3.0), which is the primary requirement from the Solar Fan System Requirements Artifact (*sourced by* relationship is the clue here), is *satisfied by* the Solar Fan System (S1.0). In turn the Solar Fan System *performs* the F.0.0 Perform System Functions Action entity. ***NOTE:*** LML allows multiple decompositions and allocations, where this same action could be *performed by* other entities. This feature enables creating functional models and tracing them to different ways to implement the action.

Another thing the Spider Diagram provides is a way to go back from the physical entities to their Actions and Requirements using the reverse relationship (reminder: all LML relationships are bi-directional). Starting with the Solar Fan System, we can create a Spider Diagram like the one below.

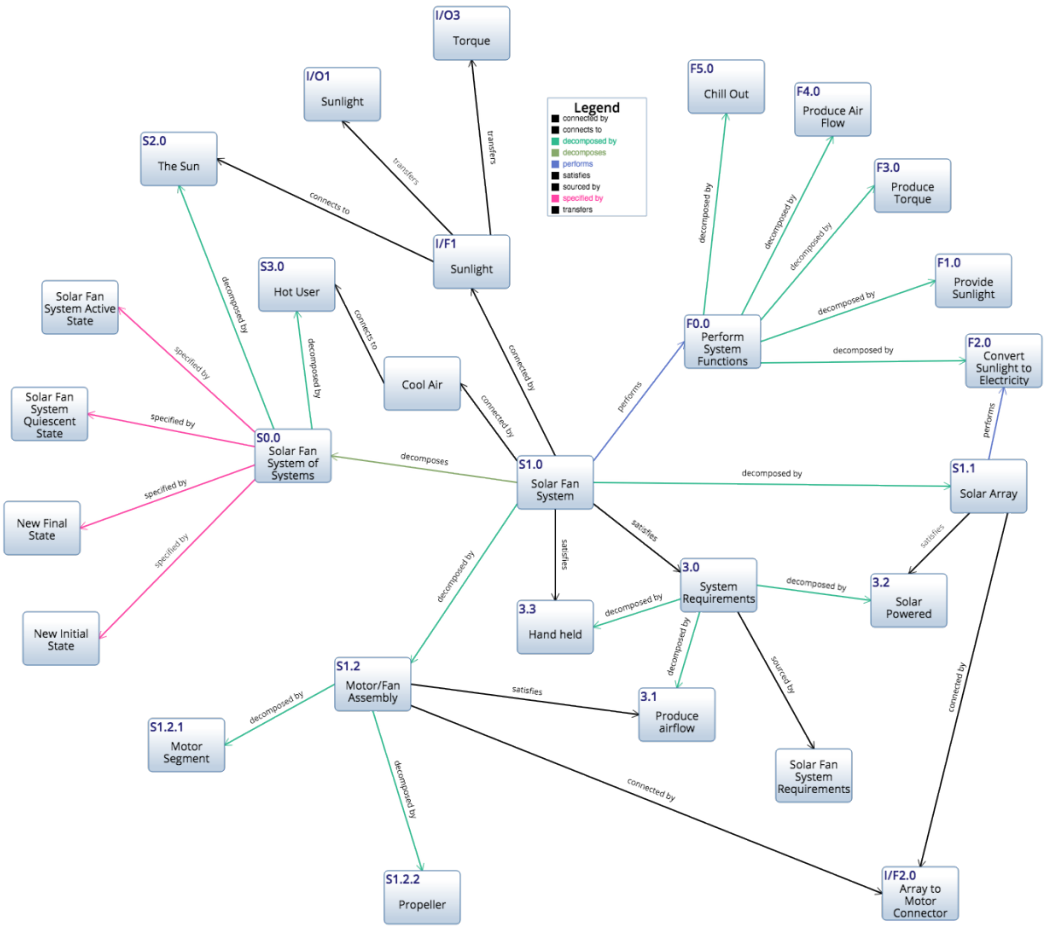


Figure 27. Spider Diagram showing reverse traceability.

Here the Solar Fan System asset entity is in the middle and we can see how it is a component of the Solar Fan System of Systems through the *decomposes* relationship. Now the *performs* relationship brings us back to the F0.0 Perform System Functions action, which in turn is *decomposed by* the action F1.0, F2.0, F3.0, F4.0 and F5.0. This capability to walk back up the relationships enables the bottom-up analytical approach, which we often want to use when understanding existing or legacy systems.

Using our solar fan example, we can focus on the Solar Fan Systems of Systems and see other relationships as well. Figure 28 shows other relationships used in this example. The Characteristics (New Initial State and Solar Fan System Quiescent State) are related to the Action “Add sunlight” through two different relationships (pushes and fetched by). These relationships may appear odd, because they are not part of the original LML specification. These relationships were added to LML version 1.1 to support additional UML/SysML features needed for the State Transition Diagram provided by those languages. You can find more about these additional relationships in Appendix A of the specification.

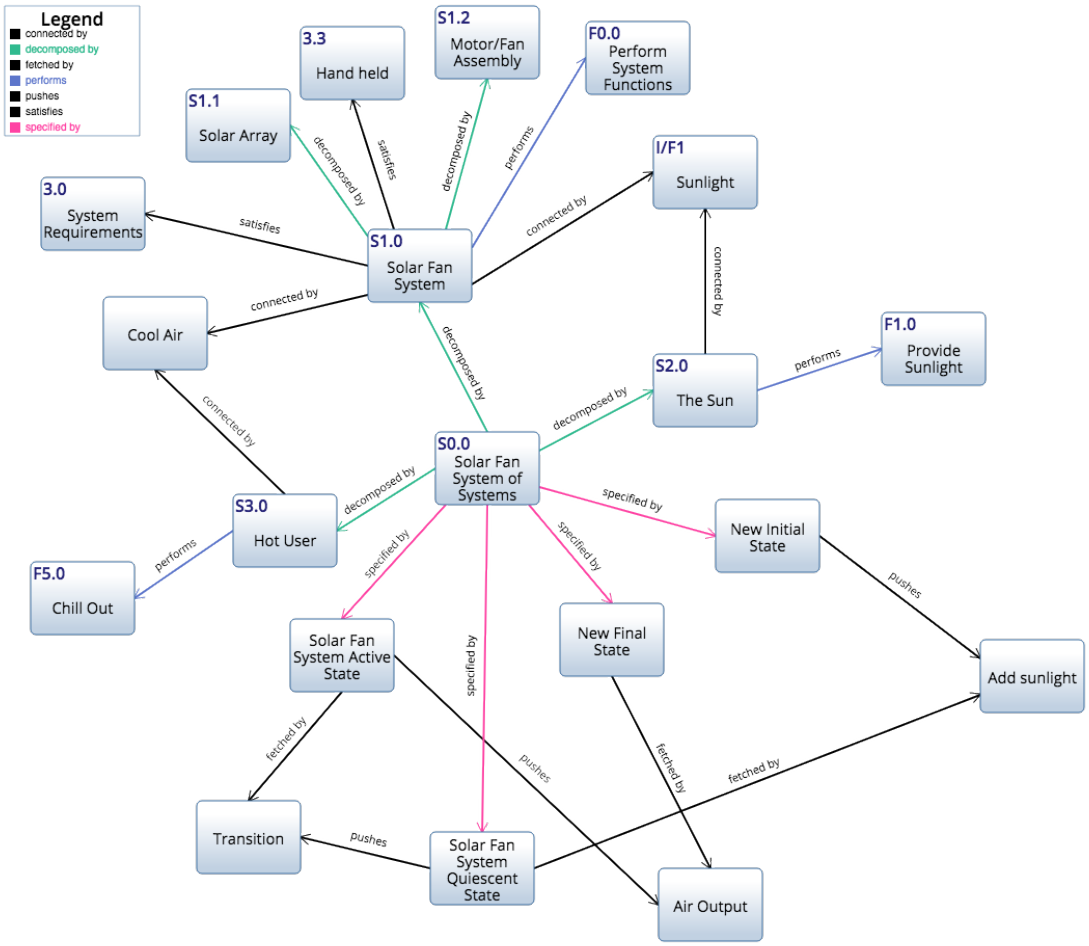


Figure 28. Solar Fan Example showing more relationships.

Clearly these three mandatory diagrams are not enough to represent every aspect of the system or program. Thus, we need other diagrams as discussed in the next chapter.

OTHER DIAGRAMS

LML allows and encourages other representations of the ontology. With twenty classes and subclasses, each with at least three attributes, some with as many as thirteen, more than twenty relationships, their inverses and attributes, you would need a very large number of diagrams to show all this information. We also need to add the decision points from the Action Diagram. The 52 models in the DoDAF, many of which are really overlapping, or the nine SysML diagram types, or the 15 IDEF diagram types, are clearly insufficient to completely communicate all this information. And any specific diagram can only express a few pieces of information at once. For example, the Action Diagram implementation discussed above shows the decision points, Input/Outputs, Actions, Assets, and Resources. That's a lot of information in one place, but it's clearly not everything. If we tried to add more information, we run the risk of making the diagram too complex to fully understand and thus it loses its effectiveness as an analytical tool.

The LML 1.1 specification provides some specific examples of diagrams and how the ontology is used to create them. We will not repeat that information here, because we anticipate that vendors will implement the specification in different ways, so you may not get the same pictures from the same data set. We expect this to occur again and again. That's allowed and encouraged as long as the vendor publishes the way they implemented the ontology. That way other vendors can map their ontologies to each other. In this way, we get to something everyone has been pursuing for more than twenty years: portability of data between tools.

We want to encourage creativity in how you extend and visualize the ontology.

IMPLEMENTING LML

Star Trek's Scotty famously said, "you've got to pick the right tool for the right job," (as did a million other engineers before him, Scottish and otherwise). In this section we will examine MBSE tools. For this discussion it is important to differentiate between:

- Software-based tools used as systems engineering tools—generic software packages (i.e. MS Office or Apple Works) applied to develop products and evidence that communicate systems engineering decisions.

- Model-based Systems Engineering tools—purpose-build software-based tools designed specifically to support the development and communication of products and evidence that support system engineering decision.

Let's start with the first one on this list. It is often said in MBSE circles, that the most commonly used "MBSE tool" in use is MS Office (Word, Excel, PowerPoint). But clearly Office was designed as a generic tools suite to meet the needs of a very diverse group of users. Modeling systems engineering was probably not even on the list of use cases for the product when it was designed. However, because of the ubiquitous availability of the Office suite—and necessity being the mother of invention—systems engineers seized on it as vast improvement over typing documents on a typewriter and drawing diagrams by hand on paper. But as we described earlier (and as summarized in Table 20-2), Office suite tools (with the possible exception of Excel) still only allow us to develop what used to be done on paper using software and thus take us only to Stage 2.

For the remainder of this book, these purpose-built tools will be the focus of our discussion. Certainly, office suite tools will continue to play an important role in the day to day business of doing systems engineering and other project tasks. But to truly reach the potential of MBSE, to have a rich artist's pallet and the ability to do the heavy lifting of systems management, we must evolve to using tools built specifically for that purpose. So, let's examine how these tools work and what they can empower systems engineers to do.

At their core, MBSE tools are relational databases. Individual items in a database are called fields or attributes, a set of fields compose a record. For example, an employee record in a database would have several fields such as last name, first name, date of birth, etc. (i.e. Smith, Mary, 12/3/1982). A set of records comprises a table. This is about the extent of capability you can obtain from a spreadsheet, so in that sense a spreadsheet is also a database. What makes a relational database is the ability to relate fields or records in one table to those in another. So, we could relate an employee to a separate table of departments (i.e. Smith, Mary —> Software Engineering Department). Relationships can be one to many (one department has many employees) or many to one (many employees in one department).

The logical organization of relational databases in general, and MBSE tools specifically is called a schema. Merriam-Webster defines a schema to be:

- a diagrammatic presentation; broadly: a structured framework or plan;

- (from psychology) a mental codification of experience that includes a particular organized way of perceiving cognitively and responding to a complex situation or set of stimuli.

For this discussion, both definitions apply. The schema that we define provides a structured framework for organizing the data. It should be emphasized this is the true art of MBSE—deciding what data to develop and how best to organize it. The MBSE tool is there to make the process easier. How we decide to construct that framework depends on how we cognitively perceive the complexity of the system we are trying to model. The framework we pick will consist of different tables of data (they may be called entities, elements or other names within a specific tool) composed of fields that capture information about the records in the table. The schema the defines specific relationships between tables.

Currently, SysML is being equated by many in the systems engineering community as the foundation of model-based systems engineering (MBSE). We disagree. Without a complete ontology that captures much more than the SysML diagrams need, we do not come close to providing the capabilities required by a language for systems engineers and program managers. We encourage users and developers of other tools that can extend their schemas, such as CORE and CRADLE, to implement LML. We think you will find it much easier to communicate with your customers and reduce the confusion that is common. The only thing these other vendors would need to add is the Action Diagram. They all have the equivalent of the Asset and Spider Diagram. They only need to modify their current eFFBD or Behavior Diagrams to create an Action Diagram that would be serviceable. We hope they do so.

Other tools may find using LML more difficult, but LML has already been extended to provide a common ontology for SysML, so those tools could benefit tremendously by applying the lessons learned encapsulated in the LML. The current plan for SysML is to develop a methodology over the next three years. By adopting LML, they will find they are a long way down that road today. Also, LML will always provide more than the SysML ontology, because it's designed to support the entire systems engineering lifecycle and program management. The future of systems engineering is an integrated, easy to understand language for all stakeholders and tools that implement that language. The language is LML.

EXECUTING LML

LML, and particularly the Action Diagram, was designed specifically to enable dynamic simulation. The basic logical constructs provided by the LML Action Diagram, include LOOPs, ORs, and ANDs. The Input/Output entities include the ability to trigger an Action, thus supporting functional sequencing. The Resource entities provide a means to capture parameters, such as size, weight, power, etc., and these can impact the simulation of the Action Diagram. If these were not enough, physical constraints such as latency and capacity are attributes of the Conduits. Conduits transfer Input/Output entities, which can have a size. If that size is large, a corresponding delay can form due to the time it takes for the Input/Output to be transferred between the Actions associated with the Assets. This tight coupling of the physical and functional model enables all types of simulation from discrete event, to Monte Carlo, to Agent-Based, to System Dynamics.

Each of the classes have attributes that contain the value of a particular parameter. For example, Actions can have durations. The duration can be a fixed value or a distribution. The distribution can provide a range of values (to enable Monte Carlo simulations) or be a time function (to enable continuous simulations, such as System Dynamics). We can also add Location information, since that is another available LML class. Locations could then be used as part of an Agent-Based simulation, where the topology is an important part of the simulation. A complete set of executable semantics is under development. Hopefully the example provided below will aid in that development and in other implementations.

To date the most complete implementation of LML is in the Innoslate® tool as discussed in the previous chapter. It has a discrete event and Monte Carlo simulation capability. So, let's see how we can execute the Action Diagram using it.

We take our solar fan example and now create a more realistic scenario. In this simulation we want to follow the process of energy coming from the Sun (with day/night cycle), filtered by clouds and the atmosphere, and then impacting the solar panel, which if there is excess can charge the battery. When the light is gone, we can then draw power from the battery until it runs out. A model that includes these features is shown in Figure 29.

The top part of this model represents the Sun and the provision to provide sunlight during the day. The decomposition of the F.2 Action (Provide Sunlight) is shown in Figure 30. Action F.2.1 is an OR logic construct and has a script associated with it to determine when the Earth is facing the Sun. That script is shown below.

```
1.  function onEnd() {
2.  var amount = Sim.getAmountByGlobalId('I_FFS0RFZ3SWGRK_822EBJDS4CYEM'); //TIME
3.  if (Math.floor(amount / 720) % 2 === 1) {
4.      return 'Yes';
5.  } else {
6.      return 'No'; }
7.  }
```

This small piece of JavaScript may look a bit complicated at first, but it is actually very simple and easy to learn. The tool will generate most scripts automatically for you, but sometimes you will want to edit these scripts. To better understand this script, we will go line by line through it.

The first line defines the function onEnd(), which is a standard function that executes at the end of the duration of the action. In this case, we have it at 1 second, so it's very quick to execute.

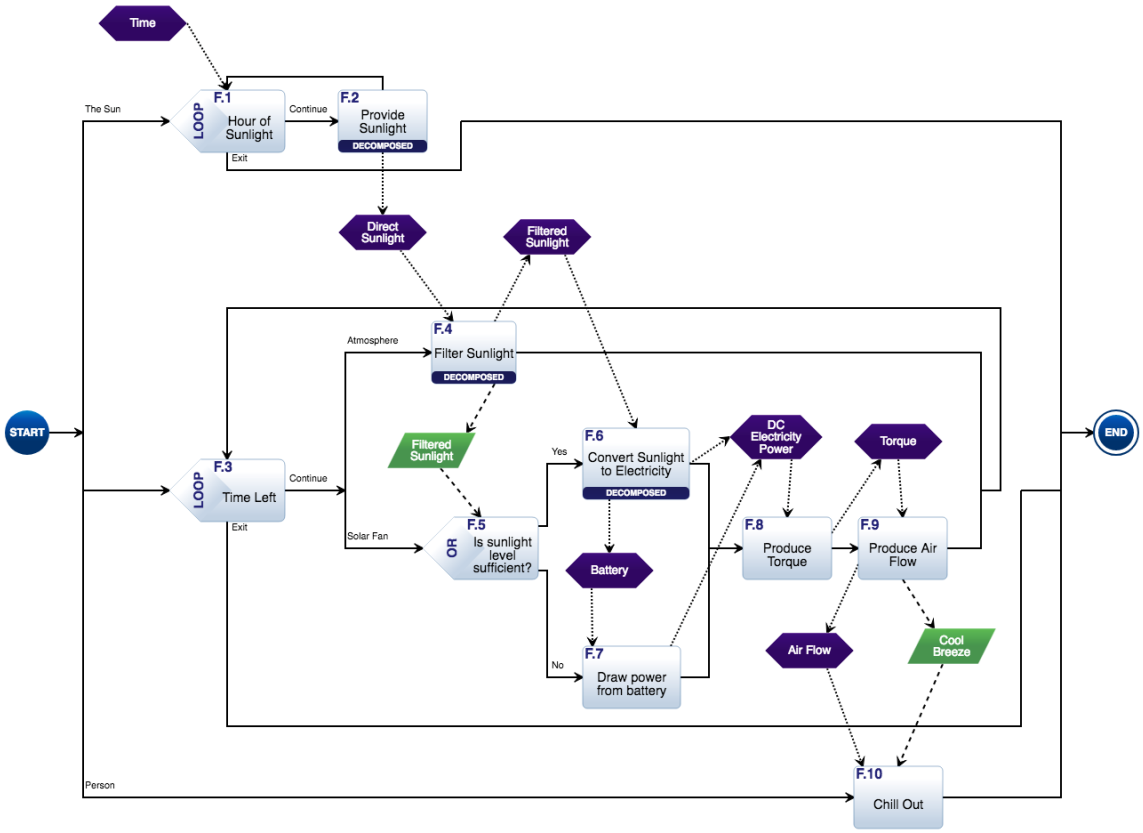


Figure 29. Solar Fan Example with more complete logic and use of resources.

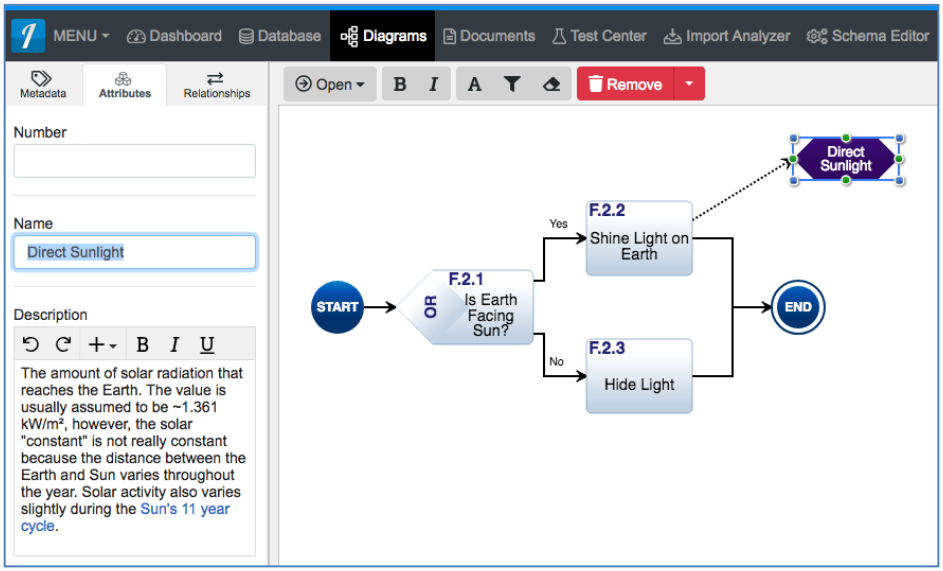


Figure 30. Decomposition of the Sun functionality.

The second line defines a variable “amount,” which represents the current value of the Time resource. The long-complicated number is the global identifier, which is easily found in the tool and allows this script to work if it is moved to another Innoslate project. The third line determines whether the Earth is facing the Sun or not. In this we are dividing the Time Amount by 720 minutes, since we have set most of the Action durations to 1 minute, thus 720 minutes represents a 12-hour period. Obviously, we could create a more complicated algorithm for the period or even pass a feed from a web application or database. The rest just set the return parameter to Yes or No, which means it will execute F.2.2 (Shine Light on Earth) or F.2.3 (Hide Light) respectively.

Since Time is a resource and we can use it to “count down” from an initial value, which was 2880 minutes representing two days. The decrementing of Time comes by using the ***consumes*** relationship, shown in Figure 31. The Amount attribute on the ***consumes*** relationship for this Time Resource accomplishes this. The LOOP Action (Hour of Sunlight) has a duration of 1 hour and the Loop Resource script shown in Figure 32. We could just as easily have done this in the script above, but we wanted to use as much of the language as possible first.

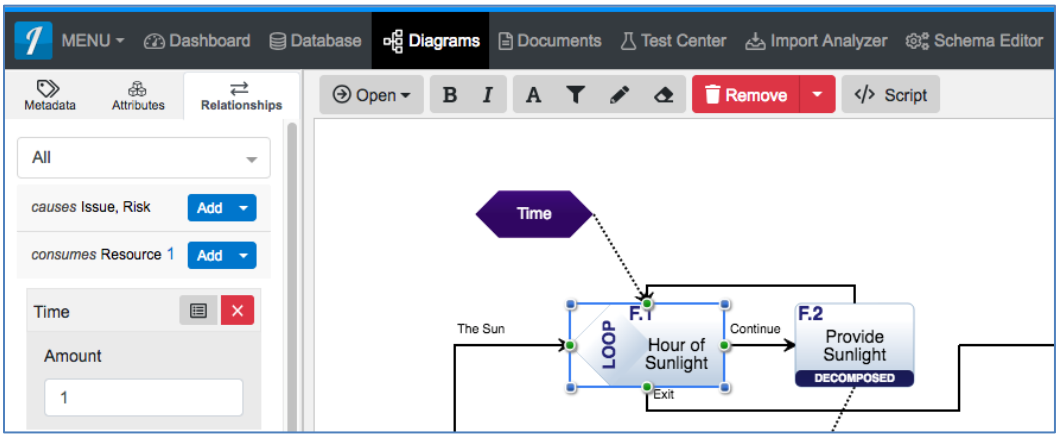


Figure 31. Decomposition of the Sun functionality.

Figure 32. Loop Resource script shows how the loop continues looping as long as Time is greater than 1.

Since this example is meant just to show that you can create an executable model using LML and not a tutorial on a specific tool, we can cut to the final product. When executing this model using Innoslate's discrete event simulation, we obtained the output shown in Figure 33.

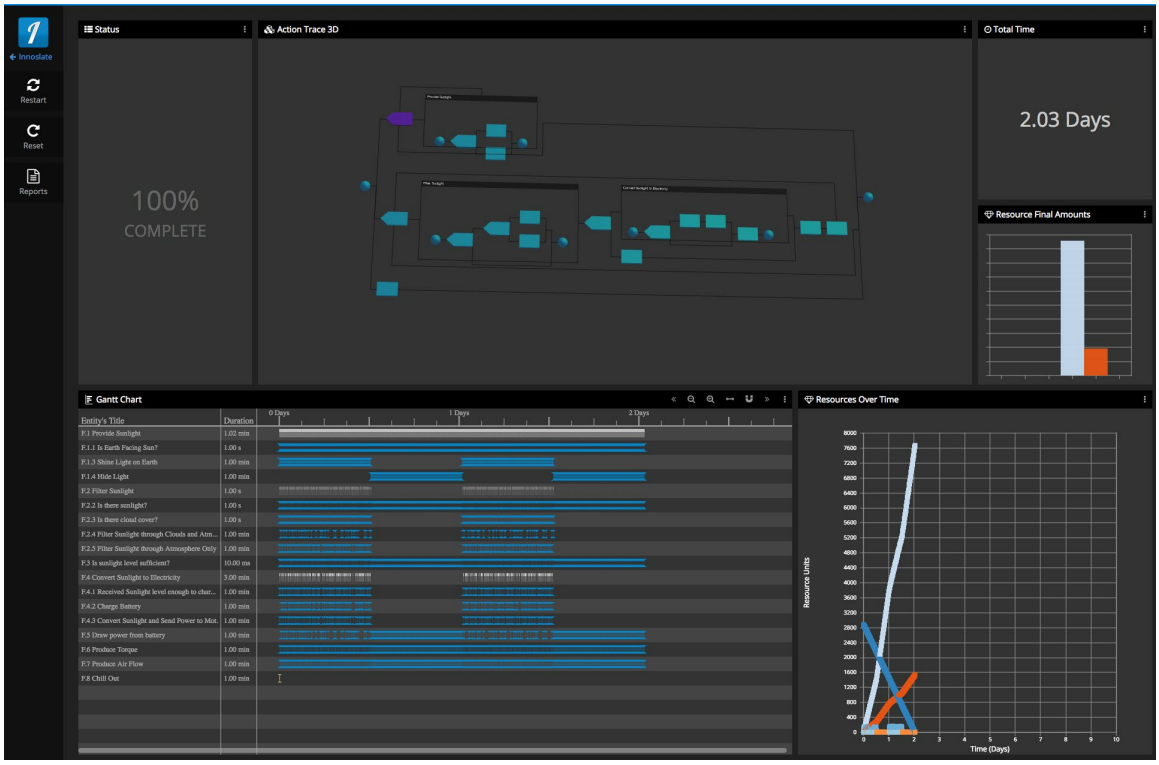


Figure 33. Discrete Event Simulation of the Solar Fan Example.

We can enlarge the Gantt Chart portion of this output using the PNG report download, as seen in Figure 34.

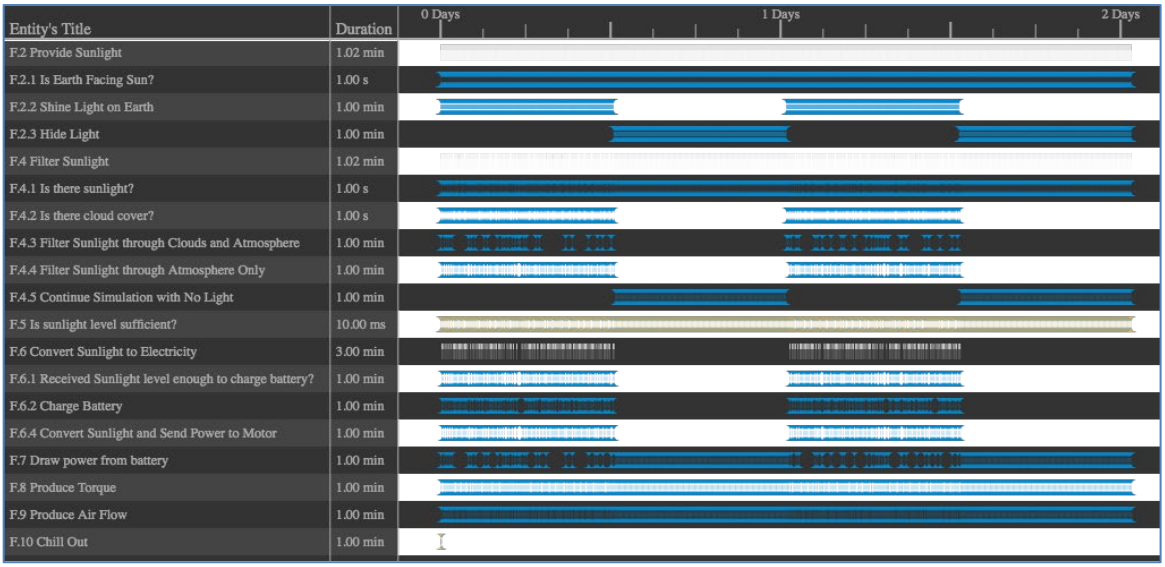


Figure 34. Expanded Gantt Chart Showing Timeline for the Solar Fan Example.

You can see the places where the Earth is not facing the sun and the battery power is drawn. The Resource graphs also available in Figure 33 and other output provide results of the power, current and other parameters as a function of time.

NOTE: That just as we added the power generation resources, we also could add cost, since cost in an explicit class in LML. This kind of cost modeling often occurs as part of using LML for process analysis and program planning.

Now, as stated above, we would likely analyze a problem like this using a time continuous simulation, as those from System Dynamics. LML supports this type of modeling as well, but it has not currently been instantiated in a tool to demonstrate this capability. Further research and development may indicate the need to extend the language, as was done for DoDAF and SysML. Such extensions are welcome.

SUMMARY AND ACKNOWLEDGEMENTS

"If I have seen further than others, it is by standing upon the shoulders of giants."
Isaac Newton

As you may have noticed throughout this book, LML contains very few new ideas and concepts. Instead it builds on things that have stood the test of time. Taxonomies and ontologies have been around since the earliest concepts of science were being developed. This ontology comes from work performed by Jim Long, Mac Alford, and others at TRW in the 1960s. It evolved through the development of several tools: DCDS, RDD-100, and CORE. However, that ontology was never put into a standard, but instead became the basis of these tools and nowhere else. We hope providing LML as a standard provides a well-deserved legacy to these previous works.

Similarly, with the diagrams the only real innovation was in the Action Diagram's use of a special type of action instead of construct notations. As stated in the previous chapter, any type of diagram is acceptable to LML if it conforms to the ontology and helps you to communicate with stakeholders.

The working group was formed to create a standard ontology that was simple and easy to use, understand, and communicate. We use the word communicate again to reinforce our assertion that the primary job of program managers and systems engineers is to communicate with everyone. We don't want a language that is unique to our discipline; as that defeats the goal of teamwork on a project – getting everyone working together toward the same goal – solving the problem.

So, to all who came before us thanks. And to all who come after us, be careful of reinventing the wheel.