Chapter 3

Modeling and SysML Modeling

3.1 INTRODUCTION

This chapter serves two major purposes: describes models and their role in the engineering of systems and introduces several modeling techniques associated with SysML. The modeling techniques introduced in this chapter are use case diagrams, sequence diagrams, IDEF0 (Integrated Definition for Function Modeling), enhanced Function Flow Block Diagrams (EFFBDs), block diagrams, and parametric diagrams. IDEF0 is a process modeling technique that is not part of SysML but will be utilized throughout this book.

Models, abstractions of reality, are critical in the engineering of systems. These models start as very high level representations that address what needs the system should meet and how, then progressively define how the system will meet these needs. These models contain increasingly more mathematical and physical details of the system as the design portion of the development phase ends. The various engineering disciplines create even more detailed mathematical and physical representations of the configuration items (CIs) before the final prototype of each CI is produced for testing and integration. During the qualification of the system design, these CI prototypes are tested with a test system that itself is comprised of many models of the system's components, other systems and the context with which the system interacts, models of scenarios that depict how the system will be used, and analysis and simulation models for creating and analyzing the test results. In fact, models are so pervasive in the engineering of systems that engineers must always remind themselves not to confuse *reality* with the *models of reality* that are being created, tested, and used.

Every modeling technique is a language used to represent some part of reality so that some question can be answered with greater validity than could be obtained without the model. All languages have a set of symbols or signs, known as *semantics*, that are used like we use letters and numbers to form expressions. Similarly, every language has a *syntax* that defines proper ways of combining the symbols to form thoughts and concepts. Section 3.2 summarizes the descriptive versus normative purposes of models and then categorizes models as physical, quantitative, qualitative, and mental.

SysML is a modeling language that is a modification of the Unified Modeling Language (UML) for software engineering. SysML matches somewhat closely with the Traditional Top-Down Systems Engineering (TTDSE). In Section 3.3 we will introduce SysML, including use case and sequence diagrams for high level metasystem interactions, IDEF0 for process or activity modeling, EFFBDs for dynamic behavior modeling, and block diagrams for the structural modeling and parametric models for modeling equations. Note SysML also addresses requirements but uses textual representations of the requirements. Chapter 6 of this text addresses textual representations of requirements.

Use case diagrams capture the various systems that comprise the metasystem, one of which is the system of interest. The use case diagram also identifies a number of scenarios in which the systems in the use case diagram interact during the relevant life cycle phase of the system of interest. Each of these scenarios is then defined in more detail in a sequence diagram. Section 3.4 defines these diagrams and gives examples.

Process models address how outputs are transformed from inputs via some function, activity, or task. There are numerous process modeling techniques in use today, one of which is IDEF0. Other process modeling techniques (data flow diagrams and N^2 charts) are described in Chapter 12. Process models are graphical representations that provide qualitative descriptions to explain how inputs are transformed into outputs. These process models can be used at both shallow and detailed levels of abstraction. IDEF0, presented in Section 3.5, is a popular modeling technique because it has a rich and standardized semantics and syntax.

Function flow block diagrams (FFBDs) and enhanced FFBDs (EFFBDs) are part of SysML for capturing dynamic behavior in a representation that can be simulated. EFFBDs are discussed in Section 3.6.

Block diagrams are used within SysML to capture the interconnections between pairs of components within the physical architecture so that interfaces between these pairs of components can be defined. Section 3.7 presents this material.

Requirements diagrams are introduced in Section 3.8. Parametric diagrams, used to capture variable relationships in systems of equations for simulating system performance, are discussed in Section 3.9.

Exercise Problem 3.1 introduces a process model of the TTDSE process using IDEF0 model. Selected pages of this IDEF0 model will be used in Chapters 6 through 11 to describe the methods that comprise this engineering process.

3.2 MODELS AND MODELING

A *model* is any incomplete representation of reality, an abstraction. Models can be physical representations of reality. A subscale aircraft is used in a wind tunnel to test the aerodynamics of the real aircraft; this subscale aircraft does not contain the instrument panel used by the pilot or the seats in which passengers sit because they are not relevant (we think) for testing the aerodynamics of the aircraft. Similarly, models can be mathematical. A random number generator can be used to model the propensity of a coin to turn up heads or tails in a flip. Similarly, we can develop either an analytic or a simulation model of an aircraft's aerodynamics or an information system's response to user inputs. The wind tunnel data taken from the physical model of the aircraft can be used to refine the simulation data. The simulation data can be used to guide additional wind tunnel tests. Qualitative models are also quite useful. The set of requirements for a system is an example of a qualitative model that serves as a model of the system's performance and capabilities. Finally, each of us has a number of mental models that we use in everyday life. However, in every case the *essence of a model is the question or set of questions that the model can reliably answer* for us.

Before describing the types of models, discussing the types of questions that can be answered is important. The questions can be divided into three categories: descriptive (or predictive), normative, and definitive. A *definitive* model addresses the question of how should an entity be defined; this is the major category of questions that will he addressed in this book. The focus is building a definition of how the system is being designed, in terms of its inputs and outputs, functions, and resources. A *descriptive* model attempts to predict answers to questions for which the truth may or may not be obtained in the future. Descriptive models are the most commonly used in science and engineering. Executable models, which will be discussed in Chapter 12, are descriptive models because they are predicting the behavior of the system's design in specific situations given the modeled design definition of the system. *Normative* models address how individuals or organizational entities ought to think about a problem and guide decision making. A normative model for decision making, in particular deciding about the engineering of a system, is developed in Chapter 13.

Every modeling technique requires a language to establish a representation of reality. Models should be used to provide an answer to one or more questions; these answers should provide greater validity or insight than is possible without the model. Any language has semantics, a set of symbols or signs, which form the basis of representations in the language. In addition, every language has a syntax that defines proper ways of combining the symbols to form thoughts and concepts.

Definitive models require a rich language, both in terms of semantics and syntax, since these models are used to establish an interpretation of some aspect of reality and communicate that interpretation to a broad range of people and

possibly computers. This language must be understandable to its audience. Unfortunately, richness and understandability often conflict with each other. That is, making a modeling language richer usually makes it less understandable. A third aspect, formality, is useful for proving that certain characteristics exist or do not exist; formality tends to conflict with both richness and understandability.

Descriptive models are measured by their power or richness for addressing a wide range of problems, understandability to both wide and narrow audiences, and accuracy or precision with which they can be used to define the relevant entity. Descriptive models can sometimes be tested as to their predictive accuracy in various situations. This predictive accuracy must be understood by those using the descriptive model because the ability to predict accurately in the situation in which the model is being used cannot be known exactly. Nonetheless, talking about descriptive models as being right or wrong is fruitless—all models are wrong. Rather, the model's usefulness in terms of predictive accuracy in general and the cost of building and using the model are very relevant.

Normative models, on the other hand, cannot be tested but are judged on their understandability and appeal across all disciplines in which they can be used. A normative model for making decisions cannot be tested because the world can never be examined in the same conditions with and without the use of the normative model. Rather, the normative model is tested by decision makers based upon the model's ability to reflect the intuitions of the decision makers or provide logical arguments that refute this intuition.

One possible taxonomy of models is shown in Table 3.1. This taxonomy begins by breaking models into physical, quantitative, qualitative, and mental models. A *physical model* represents an entity in three-dimensional space and can be divided into full-scale mock-up, subscale mock-up, breadboard, and electronic mock-up. Full-scale mock-ups are usually used to match the interfaces between systems and components as well as to enable the visualization of the physical placement of elements of the system. The design of the Boeing 777 replaced the physical mock-ups with a very detailed three-dimensional electronic mock-up. Subscale models are commonly used to examine a specific issue such as fluid flow around the system. A breadboard is a board on which electronic or mechanical prototypes are built and tested; this phrase was legitimized in dictionaries in the mid-1950s but is not used as much now.

Quantitative models provide answers that are numerical; these models can be either analytic, simulation, or judgmental models. Simulation models can be either deterministic or stochastic, as can analytic and judgmental models. Similarly, these models can be dynamic (time varying) or static snapshots (e.g., steady state). An analytic model is based upon an underlying system of equations that can be solved to produce a set of solutions; these solutions can be developed in closed form. Simulation methods are used to find a numeric solution when analytic methods are not realistic, such as when friction in some form is introduced as an element of the model. When the equations involve the

TABLE 3.1 Taxonomy of Models

Model Categories	Model Subcategories	Typical Systems Engineering Questions
Physical	Full-scale mockup	How much?
	Subscale mock-up	How often?
	Breadboard	How good?
		Do they match?
Quantitative	Analytic	How much?
	Simulation	How often?
	Judgmental	How good?
Qualitative	Symbolic	What needs to be done?
	Textual	How well?
	Graphic	By what?
Mental	Explanation	All of the above!
	Prediction	
	Estimation	

movement through time of a number of variables, we say the simulation is dynamic, involving differential or difference equations. However, a simulation need not involve time; the model may address spatial issues. Simulations that include uncertainty are often called ''Monte Carlo'' simulations; Monte Carlo simulations involve the repetitive solution of the same set of equations based upon different samples of the underlying probability distributions for the uncertainty specified in the equations. Judgmental models provide representations of real-world outcomes based solely on expert opinions. Explicit judgmental models are not used as often as the other types discussed here, but many analysts have found them to be an extremely useful precursor to other quantitative modeling activities.

Qualitative models provide symbolic, textual, or graphic answers. Symbolic models are based on logic or set theory, samples of which are provided in Chapter 4. Textual models are based in verbal descriptions; many models of the social sciences use textual models in which a model is described in one or more paragraphs. Many requirements documents in systems engineering are examples of textual models of the system's ultimate performance. Graphical models use either elements of mathematical graph theory or simply artistic graphics to represent a hierarchical structure, the flow of items or data through a system's functions, or the dynamic interaction of the system's components. This use of artistic graphics as a modeling approach is often given the pejorative name of ''view graph'' engineering. Most engineers view graphical models as one step above textual models. If graphical models can be based on mathematical graph theory, then these qualitative models can be powerful additions to the systems engineers' toolkit.

Finally, we need to address the *mental models* that we all carry around inside of us as abstractions of thought. The concept of a mental model arose in at least

three separate communities relatively independently. Craik [1943] introduced mental models to cognitive psychology as our foundation for reason and prediction. Little was done with Craik's concept of a mental model until the early 1980s when Johnson-Laird [1983] and Gentner and Stevens [1983] published two books on the subject. The research in cognitive psychology has moved from the question of whether people do have mental models to the question of how best to capture and utilize these mental models for educational and other pursuits. The second field in which mental models became popular and useful was that of manual control, comprised of both psychologists and engineers. Early authors in the manual control field [Veldhuyzen and Stassen, 1977; Jagacinski and Miller, 1978; Rasmussan, 1979] addressed the use of mental models by system operators for controlling and predicting system performance. The third field to adopt mental models [Alexander, 1964; Pennington, 1985] is our field of engineering and architectural designers. Alexander [1964] discussed mental pictures as representations of the problem definition and alternate solutions. Do you have a mental model of the street network in your neighborhood, of your residence?

Engineers need to develop a mental model of the system on which they are working to be successful. Modelers who are developing qualitative, quantitative, or physical models clearly have to develop a mental model of the model they are developing. The advantage of these non-mental models is that there is a much clearer communication mechanism; mental models fall down in benefit in terms of enabling communication among people. People engaged in the same conversation may have a very different mental model but due to the imprecise nature of natural language they often feel that they can agree with each other at the end of a conversation even though their models of reality are quite different.

SIDEBAR 3.1

It is tempting to think that a quantitative model is more objective than a mental model and, by extension, that a more complex quantitative model is more objective than a less complex quantitative model. Certainly more complex models are more explicit than less complex models. Also the data inputs to these complex models are more specific and objective appearing. However, we must always remember that any quantitative model is developed via a mental process of one or more people and is the product of their mental models. Therefore, it is a mistake to ascribe objectivity to models. Complex mathematical models often have subjective assumptions throughout their equations and data.

This book emphasizes the *qualitative aspects of systems engineering*. As a result, this chapter introduces the qualitative modeling approaches in SysML

and IDEF0. The next two chapters introduce the mathematics of set theory and graph theory, which should provide some mathematical underpinnings and limitations of these modeling approaches. Chapter 13 introduces decision analysis as the quantitative method for framing the design decisions discussed throughout this book.

The *purpose* in developing a model is to answer a question or set of questions better than one can without the model. Often models are used to check each other; non-mental models should always be used to check mental models. This checking process is a two-way street; each model can be assumed to have certain strengths (answers known to be valid within some degree of accuracy or precision). These strengths can be used to help verify the abilities of the other model. Ultimately, a model is developed to provide answers in an area for which we feel we cannot get reliable answers any other way. However, we are commonly looking for more than just an answer; we want to understand "why" the answer is what it is, that is, obtain insight into how the real world works. Qualitative models are typically created to achieve agreement among individuals (shared visions) and to communicate that agreement to other people. Quantitative and physical models are better mechanisms to provide insight.

The more specific a question or set of questions is that a model has to answer; the easier it is to develop a model that can be useful. Models that are expected to answer a wide range of questions or generic questions well are the most difficult to develop and the least likely to provide insight into the logic for the answer. The easiest questions to answer are those for which we are looking for a relative comparison of alternate options: Which aircraft design weighs the most? How much more does one design weigh than another? The hardest questions involve providing an absolute answer: How much does this aircraft weigh?

The most *effective process* for developing and using a model is to begin by defining the questions the model should be able to answer. (This is analogous to defining the requirements for a system.) Then the model should be developed, tested, and refined. The model should be validated, shown to be answering the right questions. Finally, there should be some verification process to show that the model is providing the right answers for known test cases. Now we are ready to use the model for unknown test cases.

Often, there may be existing models that we believe are appropriate for use. In this case we should again begin by defining the questions to be answered. Then we can decide which model to use, perhaps with some enhancements. There should again be a period of verification for the chosen model in relevant cases before usage begins.

The incorrect approach to modeling is to begin by building or revising a favorite model before we know what questions need to be answered. People enthralled with the modeling process rather than the question answering process employ this approach far too often. Modeling enthusiasts are more interested in the intrinsic properties of the model than with the model's ability to answer important questions. Note the more complex the model, the harder it is to obtain that insight we are seeking as to why the answer is what it is. This is

why many experienced model builders opt for the most parsimonious (simplest) model that will provide a reasonably accurate answer.

Before using a model, it is important to establish the validity of the model. Model validity is difficult to establish and must first be defined. Recall from Chapter 1 that a system's validity addresses whether we have built the right system. By extension model validity concerns whether we have built the right model. Validity of a model has several dimensions: conceptual, operational, and data. *Conceptual* validity addresses the model representation, that is, the theories employed, the assumptions made. Conceptual validity addresses whether the model's structure is appropriate to answer the questions being asked. For a qualitative model conceptual validity is the most important. For a quantitative model the *operational* validity is key; that is, does the model's output behavior represent that of the real world for the questions being asked. Finally, *data* validity addresses whether the appropriate inputs were employed in building, testing, and using the model. Data validity for a qualitative model addresses whether the right individuals were involved in creating the model and whether they obtained access to the best set of information about the real world during the creation process. For quantitative models, the selection of a modeling technique may ride on what type of information will be available for running the model. When input data is scarce, judgmental models are often selected. In summary, establishing a model's validity has to be tied to the model's ability to answer the questions that the model was designed to address.

Models have many potential uses in systems engineering: *creation* of a shared vision, *specification* of the shared vision, *communication* of the shared vision, *testing* the shared vision, *estimation* or *prediction* of some quantitative measure associated with the system and *selection* of one design option over other design options. The shared vision could be the inputs and outputs of the system, the system's requirements, the system's architecture, or the test plan for validating the system's design. As can be seen, all but the last two uses involve a qualitative activity. *This is the basis for emphasizing the use of qualitative models as adjuncts to our mental models in this book*. Quantitative models remain important, but qualitative models are not given their due value in engineering.

3.3 SysML MODELING

In Table 1.5 there were four topic areas defined for SysML modeling [Friedenthal et al., 2008]: structure, behavior, interaction, and requirements. This is the decomposition provided by the Object Management Group, Inc. (OMG), which produced the specification for SysML.

Another way of viewing these categories that is more consistent with the organization of this book would be:

1. meta-system modeling with use case and associated sequence diagrams as well as requirements relations with requirements diagram

2. behavior modeling of the system's activities or processes (including both static and dynamic modeling) using activity and state machine diagrams
3. structural modeling of the system's components including block definition and internal block diagrams
4. parametric modeling of performance characteristics of the system
5. the process and structure of that the systems engineering team is taking using package diagrams

Section 3.4 will introduce use case diagrams and sequence diagrams. This material is presented in terms of the very important modeling of the system's interaction with other systems (or the meta-system) that is often not done. Chapter 6 will revisit this material and provide more context about how to use these diagrams.

Systems and software engineers have devised many ways to model processes or activities, at all levels of granularity — meta-system, system, through components. The activity or process modeling category within SysML includes state machines, a new modeling technique that combines some properties of Petri nets (see Chapter 12) and control flow diagrams as well as EFFBDs. Not directly mentioned are the standard static, time-lapsed representations of dynamic processes such as IDEF0, data flow diagrams, and N^2 charts. This text will continue to stress the value of static modeling techniques such as IDEF0 as a stepping stone for getting to the more complex behavior models, as well as for capturing the inputs and outputs that are passed from function to function in the behavioral model. The process modeling approach primarily employed in this book is IDEF0, which is described in Section 3.5. The SysML framers did include state–machine models and activity diagrams (otherwise called extended function flow block diagrams or EFFBDs). State-machine models are discussed in Chapter 12. EFFBDs are described in detail here in Section 3.6. Table 3.2 provides a categorization of process models into static and dynamic, as well as into SysML versus non-SysML approaches. Chatper 12 covers most of the techniques not addressed here in Chapter 3. Chapter 7 will return to this material and provide a process for building these types of models.

TABLE 3.2 Representation of process modeling techniques

	Static View	Dynamic View
SysML	–	State machines Activity diagrams EFFBDs
Non-SysML	Data flow diagrams Control flow diagrams N^2 diagrams IDEF0 diagrams	FFBDs Behavior diagrams Petri Nets Statecharts ROOMcharts

Structural modeling diagrams (block definition and internal block) are described and illustrated in Section 3.7. These diagrams have a long history in systems engineering and have finally been formally defined and standardized by SysML. Chapter 8 will provide more detail on how to build these kinds of models.

Systems engineers have historically built many types of performance models of their system design so as to estimate final performance capabilities of the design prior to fabrication of the initial prototypes. Chapter 9 introduces the types of performance modeling commonly used in the engineering of systems.

3.4 META-SYSTEM MODELING

In order to describe a modeling language we have to describe the way in which the language is used to communicate to its readers/listeners. To describe a language we need to identify the semantics (signs and symbols) and the syntax (composition of signs and symbols) of that language. The following definitions for semantics and syntax are taken from The American Heritage Dictionary [Berube, 1991].

Semantics: study of relationships between signs and symbols and what they represent.

Syntax: way in which words are put together to form phrases and sentences.

This and each of the following sections will describe the semantics of the language tool. The syntax will then be described formally or via examples.

It is critical that there be a team of engineers and domain experts that is performing the systems engineering process. This team can create a huge problem for itself by diving right into the design of the system without first learning about the other systems with which the focus of the design activity is to interact. This is probably the most common and most major problem encountered in the engineering of systems.

Chapter 6 introduces the operational concept, which includes scenarios or use cases that are supposed to describe how the system interest will interact with humans and other systems throughout its life cycle. It is in the operational concept that the use case diagram and many sequence diagrams would be used to describe these scenarios. Developing these sequence diagrams is a major part of getting ready to develop the system's requirements. This section provides the fundamental semantics and syntax for using use case diagrams, sequence diagrams and requirements diagrams within SysML.

The purpose of the use case diagram is to provide a higher level of how all of the individual use cases or usage scenarios combine within the operational concept to describe how the stakeholders think the system will be operated.

This use case diagram originated in software engineering and is now commonly employed within the engineering of systems.

The semantics of a use case diagram contains:

1. labeled stick figures for each class of humans or external systems
2. labeled ovals to define each use case
3. solid lines connecting stick figures and ovals
4. labeled dashed lines connecting ovals

The syntax is that there is a diagram for each relevant phase of the system's life cycle, (e.g., operations, training). For a given phase of the life cycle, the appropriate classes of humans (e.g., operators, maintainers) and other external systems are each given a stick figure. Then all of the possible interaction sequences among the system and these classes of humans and external systems are categorized and labeled as ovals. These interaction sequences are later defined one at a time in a sequence diagram. Actually there is a significant amount of iteration between the first draft of the use case diagram, the defining of individual sequence diagrams, the improvement of the use case diagram, and so on.

Figure 3.1 provides an example of a use case diagram for an elevator. The stick figures are (1) passenger class of humans, (2) maintenance workers, (3) building personnel, (4) a centralized service center including humans and other technology assets, and (5) the building. The high level operational scenario is of course to use and maintain the elevator. There are four extensions of this basic scenario: responding to a fire, keeping the doors open, rescuing people from a stopped elevator, and ensuring that the load on the elevator is within a safe range. These extensions provide more detail about the basic scenario in specific situations that may or may not occur. There are two other ovals on this use case diagram for updates to the basic scenario that must always be present: providing electric power and maintaining a comfortable environment. Finally, there is one use case (fix the elevator) that does not involve passengers. In fact, this would be a basic scenario with extensions and inclusions if we were designing a real elevator.

For each labeled oval in the use case diagram there should be a sequence diagram that defines the interactions among it and the other systems (including people, facilities, etc.) that the use diagram depicts as relevant. The semantics of a sequence diagram are:

- a labeled vertical line
- a labeled horizontal arrow that connects two or more vertical lines

One labeled vertical line represents the system of interest. Each vertical line represents an external system with which the system interacts (exchanges inputs

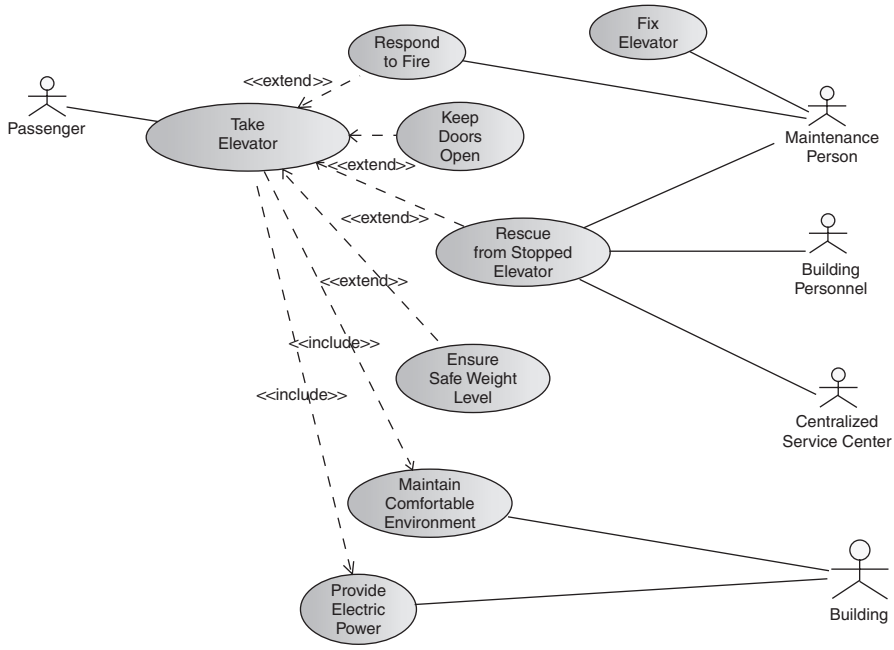


FIGURE 3.1 Exemplary use case diagram.

and outputs) during the use case. There must be at least two vertical lines. Time is assumed to go from the top to the bottom of the vertical lines. The labeled horizontal arrows represent the flow of items (information, energy, or physical entities) between the systems that the horizontal arrows connect. These items move in the direction of the arrow.

The syntax of sequence diagrams dictates that earlier flows in the use case appear above later flows, but time is not represented in appropriately scaled time units. Figure 3.2 provides a simple example for an elevator system in which a potential passenger calls an elevator to go up or down.

One contentious issue in sequence diagrams is what the labels of the horizontal arrows should represent. Many authors and practitioners label the arrows with the function being performed by the system of interest. In this book we adopt the convention of labeling the horizontal arrows with a name that represents the item being transferred from one system to another. The reason for this convention will be described in detail in Chapter 6 and is associated with the contention that functional requirements should be written about inputs and outputs rather than functions.

Finally, SysML provides a basic representation for defining requirements and a broad set of representations for relating requirements to other requirements and system concepts. Rather than detail this part of SysML, we will use the capabilities in CORE that were described in Chapter 2.

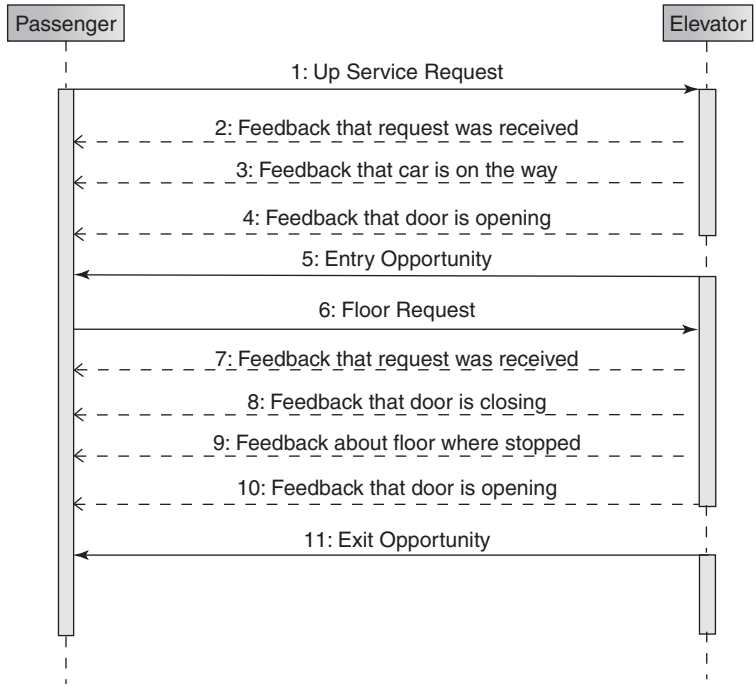


FIGURE 3.2 Exemplary sequence diagram.

3.5 STATIC BEHAVIORAL PROCESS MODELING WITH IDEF0

While IDEF0 was not included in SysML as a modeling technique, it will continue to be used throughout this book. First, it provides a very useful graphical representation of the interaction of the functional and physical elements of a system. IDEF0 is definitely not a sufficient modeling representation for the engineering of systems since it is *not* precise enough to define a unique dynamic representation of the system's design. In fact, it is not even a necessary modeling language since other languages have been successfully used for decades in its place. However IDEF0 has gained wide acceptance and standardization and also has been used successfully for decades as an approach to start the modeling process.

The IDEF acronym comes from the U.S. Air Force's Integrated Computer-Aided Manufacturing (ICAM) program that began in the 1970s. IDEF is a complex acronym that stands for ICAM Definition. The number, 0, is appended because this modeling technique was the first of many techniques developed as part of this program. More recently the U.S. Department of Commerce [National Institute of Standards and Technology (NIST)] has issued Federal Information Processing Standard (FIPS) Publication 183 [1993a] that

defines the IDEF0 language and renames the acronym, Integrated Definition for Function Modeling.

The roots of IDEF0 can be traced to the structured analysis and design technique (SADT), developed and tested by Doug Ross at SofTech, Inc. from 1969 to 1973.

A sample of the modeling languages developed as part of the IDEF family follows.

IDEF0: a major subset of SADT; focus is a functional or process model of a system

IDEF1: focus is an informational model of the information needed to support the functions of a system

IDEF1X: focus is a semantic data model using relational theory and an entity–relationship modeling technique

IDEF2: focus is a dynamic model of the system

IDEF3: focus is both a process and object state-transition model of the system

3.5.1 IDEF0 Semantics or Elements

An IDEF0 model is comprised of two or more IDEF0 pages. The two semantical elements of an IDEF0 page are functions and flows of material, energy, or information.

A *function* or activity is represented by a box and described by a verb-noun phrase and numbered to provide context within the model (see Figure 3.3). A function in this context is a transformation that turns inputs into outputs.

Inputs to be transformed into outputs enter the function box from the left, controls that guide the transformation process enter from the top, mechanisms (physical resources that perform the function) enter from the bottom, and outputs leave from the right.

A *flow* of *material, energy* or *data* is represented by an arrow or arc that is labeled by a noun phrase (see Figure 3.4). The label is a noun phrase and represents a set or collection of elements defined by the noun phrase. The label is connected to the arrow by an attached line, unless the arc leaves the page, in which case the label is placed on the appropriate edge of the page.

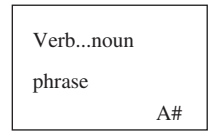


FIGURE 3.3 Syntax for an IDEF0 function.

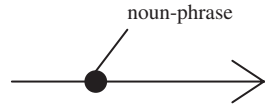


FIGURE 3.4 Syntax for an IDEF0 flow of material or data.

3.5.2 IDEF0 Diagram Syntax

An IDEF0 model has a purpose and viewpoint and is comprised of two or more pages, each page being a syntactical element of the model. The IDEF0 model:

- Answers definitive questions about the transformation of inputs into outputs by the system.
- Establishes the boundary of the system on the context page. This boundary is explicated, if needed, as a meta description.
- Has one viewpoint; the viewpoint is the vantage or perspective from which the system is observed.
- Is a coordinated set of diagrams, using both a graphical language and natural language.

The A-0 page is the context diagram, which defines the inputs, controls, outputs, and mechanisms (ICOMs) for the single, top-level function, labeled A0. The context page establishes the boundaries of the system or organization being modeled by defining the inputs and controls entering from external systems and the outputs being produced for external systems.

Other pages in the IDEF0 model represent a decomposition of a function on a higher page, with the exception of the external system diagram page, which is described later. The number of subfunctions for any IDEF0 function is limited to six, or possibly seven, for purposes of a readable display on a page. The decomposition of a parent function preserves the inputs, controls, outputs, and mechanisms of the parent. There can be no more, no less, and no differences. Every function must have a control. An input is optional. Functional boxes are usually placed diagonally on the page with the more control-oriented functions being on the top left and the functions responsible for producing the major outputs being on the bottom right. Arcs are decomposable, just as functions are. Feedback is modeled by having an output from a higher numbered function on a page flow upstream as a control, input, or mechanism to a lower numbered function.

Arc decomposition and joining are necessary to minimize the number of arcs on the upper pages of a model, enhancing the readability or communicability. Arc decomposition and joining are handled by branching and joining, respectively. The labeling conventions for joins and branches are shown in Figure 3.5. If an arc is labeled before a branch and not labeled after the arc branches into

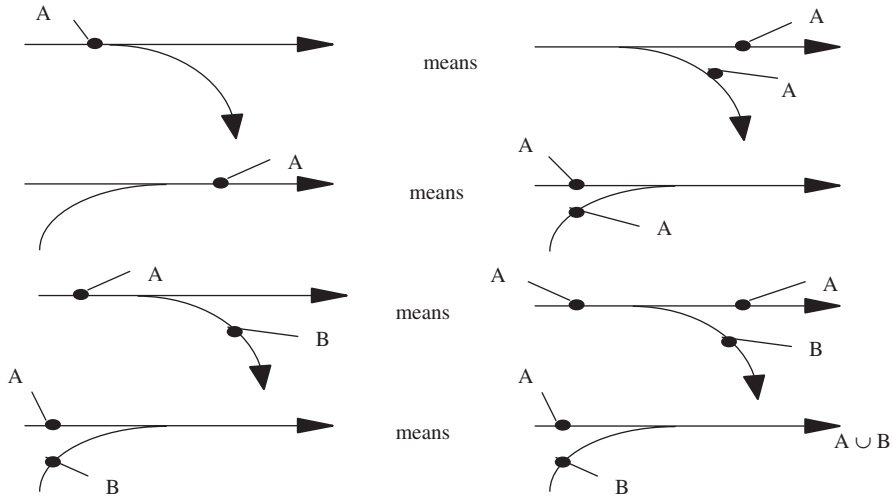


FIGURE 3.5 Labeling conventions for branches and joins.

two or more segments (as shown on the first of four examples in Figure 3.5), then the arc before the branch carries on after the branch. Similarly, if an arc is labeled after two or more arcs join (see the second example in Figure 3.5), then the label after the join also applies to the arcs before the join. If the label before a branch (after a join) does not apply to one or more of the arcs after the branch (before the join) then the arcs that deviate must have their own label. These labels of the exception branches have to be subsets of the labels before the branch (after the join), as shown in the bottom two examples of Figure 3.5.

Three different types of feedback are possible within an IDEF0 page: control, input, and mechanism. (The general topic of feedback will be discussed in more detail in Chapter 7.) Feedback in an IDEF0 diagram enables data or physical resources to be sent against the flow, down and to the right, so that closed-loop control can be used to improve key performance issues. The semantical protocols for showing these three types of feedback are shown in Figure 3.6. A control arc indicating feedback must go up and over the functions involved, coming down on the function for which it is a control. Input feedback is indicated by an arc that goes down and under the functions involved, coming up and into the function for which it is an input from the left. Finally, mechanism feedback must also be achieved by an arc that goes down and under the function for which it is a mechanism.

A major difficulty with IDEF0 models is determining whether an item should be an input or control. The primary distinction is that inputs are items that are transformed or consumed in the functional process associated with the production of its outputs. Controls, on the other hand, are not transformed or consumed, but rather are information or instructions that guide the functional

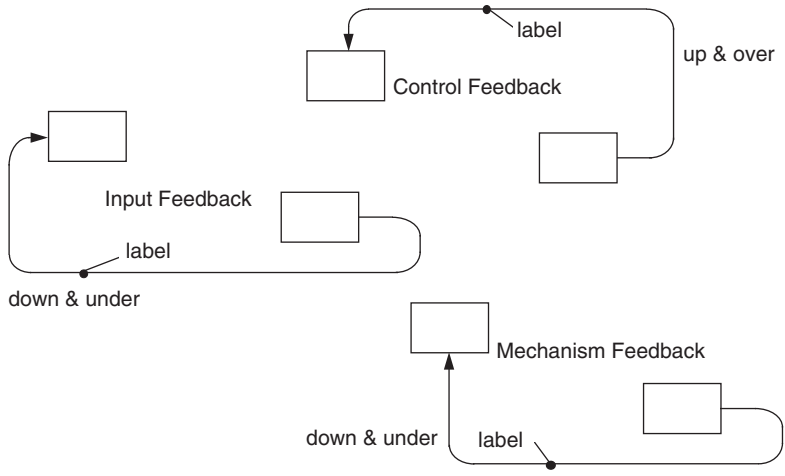


FIGURE 3.6 Feedback semantics within an IDEF0 page.

process. Typical examples of controls are a blueprint and recipe instructions (e.g., bake at 375°F for one hour, use a 9.5- by 12-inch baking pan). Nonetheless, there are many times when it is very difficult to determine whether an item is an input or control. In these cases, the decision is the author's, with the provision that every function must have at least one control while inputs are optional.

Readers of an IDEF0 model are often surprised to see a function with a control and output, but no input. This seems to suggest a counterexample to the conservation of mass and energy in physics. Remember though that outputs of a function in an IDEF0 model do not have to have mass or energy but can be information. A common example of a function that can produce an output without an input is a function that produces a time mark for other parts of the system. This function receives a control whenever the time mark is needed and uses its timekeeping resources to produce the time mark as an output.

3.5.3 IDEF0 Model Syntax

An IDEF0 model is a functional decomposition of the top-level, or A0, function. The decomposition is a hierarchy, as shown in Figure 3.7. The function numbers are shown on the right and the corresponding IDEF0 page numbers are shown on the left.

The function that is being decomposed is the parent, while the functions decomposing it are called its children. The node numbering process defines the tree. The node numbering convention as shown in Figure 3.7 is summarized in Table 3.3.

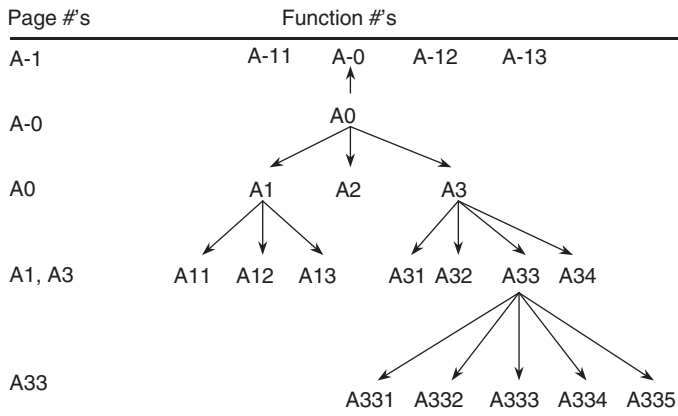


FIGURE 3.7 IDEF0 functional decomposition.

As an example of this decomposition process, the A0 page, shown in Figure 3.8, defines the decomposition of the A0 function by three functions in this case. Note there are two inputs, three controls, three outputs, and one mechanism for the function A0; each of these ICOMs is given a generic label to emphasize the conservation of ICOMs. On the decomposition of A0 into Al, A2, and A3, there are again two external inputs, three external controls, three external outputs, and an external mechanism. Note that I1, C2, C3, and M1 branch on this A0 page. In addition, the joining of outputs from Al and A2 produces 02. A number of internal items are produced, some of which branch and join.

IDEF0 models can also address the interaction of the system with other systems. This interaction is modeled on the A-1 page, which takes the A0 function and places it in context with other systems or organizations. This representation is often critical to understand the relationship of the system being addressed to the system's outside world and establishing the origination of inputs and controls and the destination of outputs.

An IDEF0 model also has a data dictionary. An IDEF0 model should have a glossary page that defines the special words and acronyms in the labels and

TABLE 3.3 IDEF0 Page Hierarchy

Page Number(s)	Page Content
A–1	Ancestor or External System Diagram
A–0	Context or System Function Diagram (contains A0)
A0	Level 0 Diagram with first tier functions specified
A1, A2, …	Level 1 Diagrams with second tier functions specified
A11, A12, …, A21, …	Level 2 Diagrams with third tier functions specified
…	…

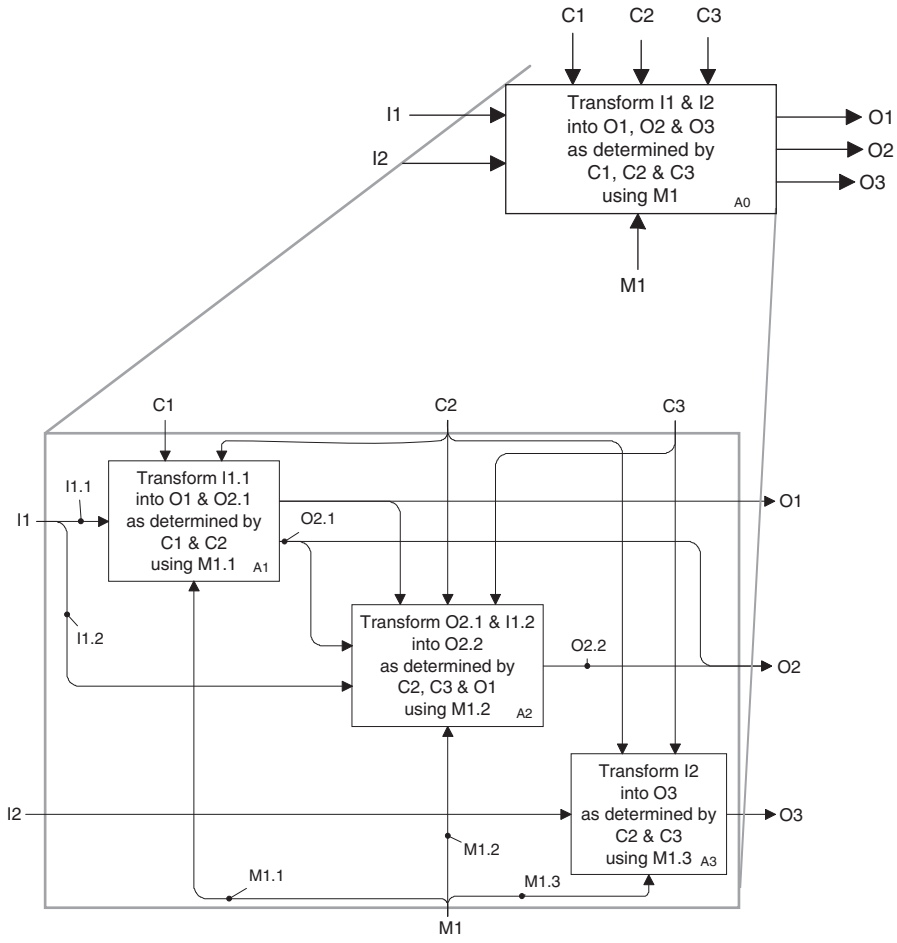


FIGURE 3.8 Functional decomposition in an IDEF0 model. Showing the preservation of inputs, controls, outputs, and mechanisms.

functions of the model. The data dictionary defines the arc decompositions. These decompositions reflect the arc branches and joins in the model. The dictionary also describes which functions use/produce which data elements.

3.5.4 IDEF0 Advanced Concepts

Advanced concepts to be discussed in this section are loops, tunneling, functional activation rules, exit rules, and call arrows.

IDEF0 allows the use of loops to show memory storage and feedback (see Figure 3.9). A loop is showing that there is feedback involved in the

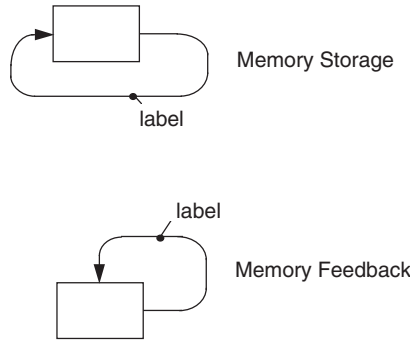


FIGURE 3.9 Memory semantics in IDEF0.

decomposition of the function shown with the loop. Usually the loop is not needed because the feedback will be seen on the decomposition. If the function is not going to be decomposed, it may be wise to show the loop. There are very few instances in which a loop is appropriately shown.

Tunneling is a technique within IDEF0 to hide an input, control, output, or mechanism in part of the model. The use of parentheses around either the head or tail of an arrow depicts a tunnel in IDEF0. Parentheses around the head of an arrow that is entering a functional box indicates that the input, control, output, or mechanism associated with that arrow will not be seen on the decomposition of that function; that is, the ICOM is going underground and may or may not reappear. If the ICOM does reappear, it will have parentheses around its tail to depict that it is exiting the ground. The rationale for tunneling is that certain ICOMs are not particularly relevant for understanding the functional model at specific levels of detail and therefore should not clutter up these pages of the model.

Each function is activated when sufficient inputs and controls are present to produce the relevant outputs, given those inputs and controls. This functional activation is typically defined as a set of rules. A rule is a set of "if …, then …"statements, or pre-conditions and post-conditions. Boolean algebra is used to specify these rules. These activation rules are embedded in each function; a "for exposition only," or EEO page, is often used to articulate the activation rules of a particular function or sets of functions.

For each function there are one or more exit criteria that determine when the function has completed its execution. Typically, the exit criterion is associated with the production of one or more outputs. If more than one output may be produced by a given function, then it is critical to state the exit criteria.

The final advanced concept is that of a call arrow. A call arrow is an arrow that breaks all of the rules of ICOMs that have been presented so far and is seldom used in the author's experience. The call arrow exits the bottom of an activity's box and points toward the bottom of the page; see FIPS Publication

183 [1993a] for an example. The label attached at the end of the call arrow signifies another box that may be part of the IDEF0 model, or part of another IDEF0 model. The call arrow is indicating that there is no decomposition of the activity from which the call arrow is exiting, but that there is a decomposition of the activity at the box associated with the label of the call arrow. The advantage of the call arrow is that fewer pages need to be part of the IDEF0 model if several of the boxes have the same decomposition.

3.5.5 Systems Engineering Use of IDEF0 Models

A major emphasis in this book is the development of a functional architecture for the system that defines what functions the system must perform to transform the system's inputs into its outputs. An IDEF0 model, minus the mechanisms, can be used to define the *functional architecture*.

As part of the development of the allocated architecture the system's functions are allocated to the system's components and CIs. This allocation of functions is captured by adding the mechanisms to the functional architecture, producing a description of the *allocated architecture*.

3.6 DYNAMIC BEHAVIORAL PROCESS MODELING WITH EFFBDS

Function flow block diagrams (FFBDs) were traditionally used in conjunction with N^2 diagrams as the original approach to functional decomposition in systems engineering. (In this book we are substituting IDEF0 for N^2 diagrams; N^2 diagrams are covered in Chapter 12 for the interested reader.) Later FFBDs were extended and enhanced to become EFFBDs. The extended FFBDs added more types of dynamic control logic. The enhanced FFBDs included some items into the models for better explication and understanding. This section first presents the full set of control logic of EFFBDs. Then shows how the items will be added.

An EFFBD model contains all of the information in an IDEF0 model plus sufficient information to create a unique discrete event simulation of the dynamic behavior of the system. This is quite an added benefit over the IDEF0 model, but it also requires additional sophistication to create. The view adopted here is that the IDEF0 model is a stepping stone to the completed EFFBD model for beginning systems engineers. Many experienced systems engineers can skip the IDEF0 model and create the EFFBD directly. However there are many other experienced systems engineers who view the IDEF0 modeling process as an important learning and communication process for the stakeholders.

An EFFBD model has pages just as an IDEF0 model does. In fact, one could take an IDEF0 model, add control logic to each page, and end up with an EFFBD model. So the EFFBD model provides a hierarchical decomposition of the system's functions with a control structure that dictates the order in which

the functions can be executed at each level of the decomposition. The control structure and arrival sequence of "triggers" (special control inputs) determines this order. This makes the syntax and semantics of an EFFBD model identical to that of an IDEF0 model.

The only semantical difference between an IDEF0 and EFFBD page is that the EFFBD has control symbols and lines that are not present in IDEF0. These control symbols and lines will be the main emphasis of this section.

In the *original,* or *basic*, FFBD syntax there were four types of control structure that were allowed: series, concurrent, selection, and multiple-exit function. A set of functions defined in a series control structure (see Figure 3.10) must all be executed in that order. In fact, the second function cannot begin until the first function is finished, and so on. (Note that in the diagrams shown in this chapter the two nodes at each end with missing center panels on the top and bottom of the functional rectangles are functions that are outside of the decomposition of the system function.) Control passes from left to right in FFBDs along the arc shown from outside (depicted by a function in a box with broken top and bottom lines) and activates the first function. When the first function has been completed (i.e., the function's exit criterion has been satisfied), control passes out of the right face of the function and into the second function, and so on. (Note that the little solid squares in the upper left corner of functions 1 and 2 are a software construct of CORE that indicate the function has been further decomposed.)

The concurrent structure (Figure 3.11) allows multiple functions to be working in parallel, thus this structure is sometimes called "parallel." However, the concurrent structure should not be confused with the concepts of parallel in electric circuits or redundant systems. Essentially control is activated on all lines *exiting* the first AND node and control cannot be closed at the second AND node until all functions on each control line entering this second AND are completed. This control structure is almost always appropriate for the external systems diagram; the external systems typically act concurrently with each other and the system in which we are interested. The concurrent control structure is also common for the first level functional decomposition of the system function.

A selection structure and a multiple-exit function achieve essentially the same purpose: the possibility of activating one of several functions. The multiple-exit function (see Figure 3.12) achieves this by having a function placed at the fork of the selection process to make the selection explicit; this is the preferred approach to the selection structure.

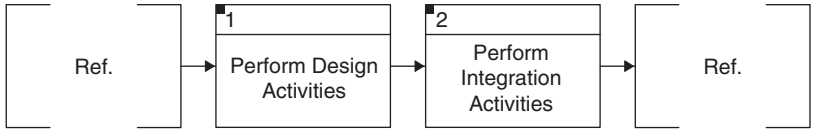


FIGURE 3.10 A series function flow block diagram.

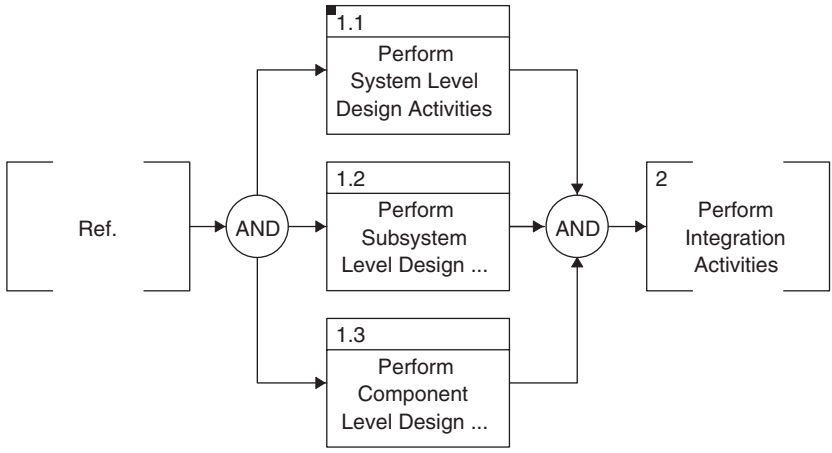


FIGURE 3.11 A concurrent control structure in an FFBD.

When the selection function has been completed, one of the two or more emanating control lines is activated. Each control line can have zero, one, two, or more functions on it. Additional control structures, such as concurrent, can be placed on any of these exiting control lines. Once all of the functions on the activated line have finished execution, control passes through the closing OR node. Each exit criterion for the control lines exiting the multiple-exit function appears has a label on the control line. (Note there is an exit criterion for every function with only one exit but the exit criterion is not commonly shown on the exiting control line. The engineer may add a label for this purpose if desired.)

For the selection construct, which is an exclusive or, the first OR node passes control to one of the exiting control lines in a manner that is unspecified on the diagram. This control line stays active until the set of functions on that control line are completed; control then passes through the second OR node. Figure 3.9 would be a selection construct if the AND nodes were OR nodes. Since the passing of control at the first OR node is not defined on the diagram, the author strongly recommends the use of a multiple-exit function instead of the selection control construct.

Additional control structures have been added to FFBDs to form what are called enhanced FFBDs: iteration, looping, and replication. See Sidebar 3.2 for a comparison of FFBD control constructs to structured programming.

Looping (Figure 3.12) is the repetition of a set of functions, based upon a specific criterion. The loop control structure begins with an LP control node and ends with a second LP node, as shown in Figure 3.12. The exit criterion for a loop is shown on the line that closes the two LP nodes. In the loop structure it is possible to exit the loop if the appropriate criterion has been satisfied.

SIDEBAR 3.2: STRUCTURED PROGRAMMING AND FFBD CONSTRUCTS

These constructs are quite analogous to those of structured programming, which began in the late 1950s and early 1960s with people such as Bohm, Dijkstra, Jacopini, and Warnier [De Marco, 1979]. Initially, the goal of structured programming was to define programming control structures that enhanced readability and improved testing. However, the goal evolved to define the control structures that would enable proving the correctness of an algorithm. While correctness proofs are still a goal, it was clear to these early investigators that program simplicity was critical. An intermediate goal to a correctness proof became the identification of the minimum set of logical constructs that would be sufficient to write any program. Bohm and Jacopini [1966] showed that only two constructs are necessary beyond the obvious series processing construct: "if-then-else" and "do-while." The if-then-else construct is the equivalent of the multi-exit function in FFBDs for situations in which a function does not need to be repeated. For repetitive activities that fit within if-then-else, the looping control structure is used. The iteration control structure is the same as the do-while programming construct. The other FFBD control structures are needed for implementation-peculiar issues of a system: Concurrent structures represent multiple resources of the system performing different functions simultaneously, and replication represents multiple resources performing the same function simultaneously.

Iteration is the repetition of a set of functions, as often as needed to satisfy some domain set; this domain set must be defined based upon a number or an interval. The iteration control structure begins with an IT control node and ends with a second IT node, see Figure 3.12. The domain set for the iterative repetition is shown on the line closing the two IT nodes.

Finally, replication is the repetition of the same function concurrently using identical resources. This repetition is shown using the stacked paper icon; the reader can see an example of this in the section of Chapter 12 on behavior diagrams. This control structure is appropriate for certain physical designs and some functional architectures.

In general FFBDs and EFFBDs do not show the inputs and outputs for functions. However, the SysML examples of EFFBDs do show at least a subset of the most important inputs and outputs, bringing the diagrams closer to IDEF0 diagrams. Remember, IDEF0 has no way to capture the dynamic information that EFFBDs do.

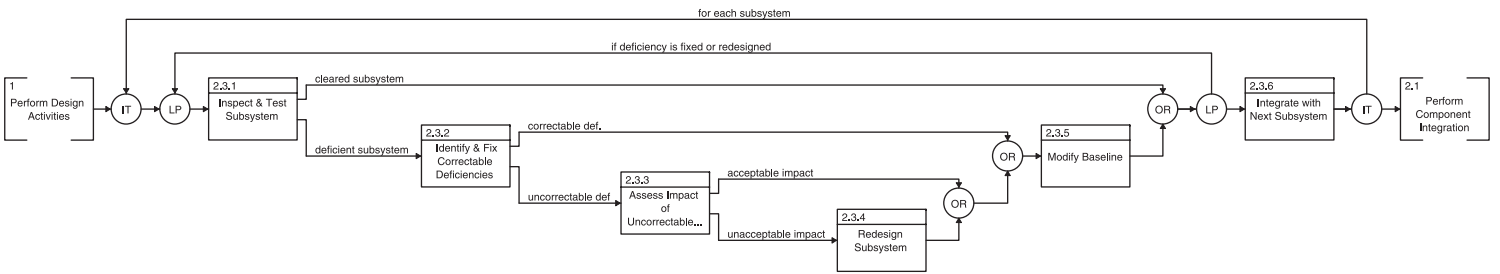


FIGURE 3.12 Selection and multiple-exit functions in an FFBD.

3.7 STRUCTURAL MODELING OF THE SYSTEM'S COMPONENTS

Systems engineers have been using block diagrams since the beginning of systems engineering. However there has been no standardization of how to construct these block diagrams, no uniform syntax and semantics. SysML has provided a much needed syntax and semantics. A block is some element within the spectrum from meta-system down to configuration item (CI). Each element represents a set of resources (people, hardware, software, etc.) that can be used to perform one or more functions as inputs are transformed into outputs. The purpose of the block diagram is to display which blocks are connected to others based on either a hierarchical relationship or on a peer to peer basis. *Block definition diagrams* represent hierarchical relationships such as how one block is composed of several other blocks. *Internal block diagrams* show which blocks within a higher level block are connected to each other via interfaces.

The semantics for the block definition diagram include a labeled rectangle to define blocks, a labeled connector with a diamond on one end and an arrow head on the other to show the hierarchical relationships. Figure 3.13 shows these two syntactic elements. Note the full SysML semantics [Friedenthal et al., 2008] includes many other elements, but these two are the basic ones that will be used later in Chapter 8. Figure 3.14 shows the syntax of a block definition diagram for the elevator system and its subsystems that was discussed in Chapter 2.

The semantics of an internal block diagram (see Figure 3.15) include a labeled rectangle for the specific blocks that compose the higher level block that is the subject of the diagram, small unlabeled blocks on the boundary of the larger labeled blocks to define the connection between the block and the interface to another block, and unlabeled lines to show the interfaces or ports

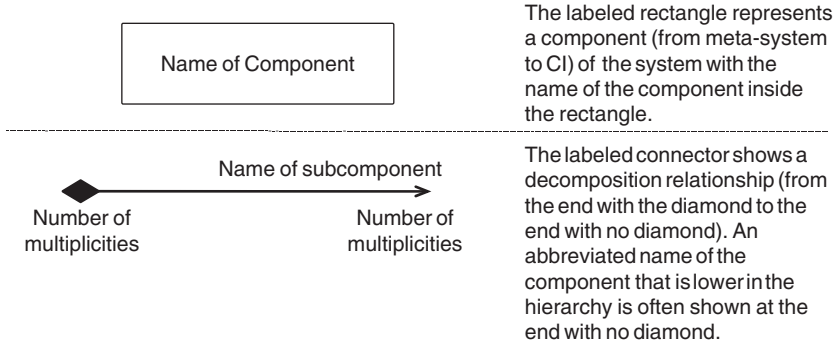


FIGURE 3.13 Semantic elements of a block definition diagram.

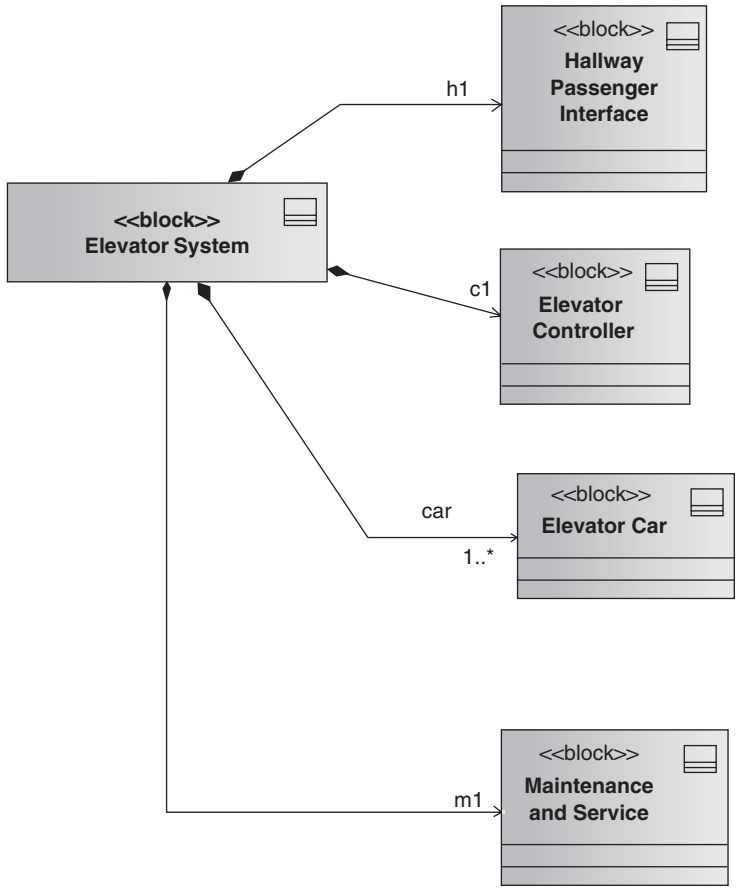


FIGURE 3.14 Exemplary block definition diagram (syntax) for an elevator.

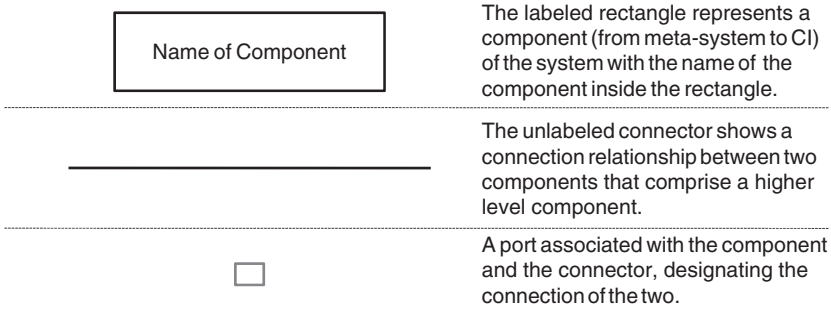


FIGURE 3.15 Semantic elements of the internal block diagram.

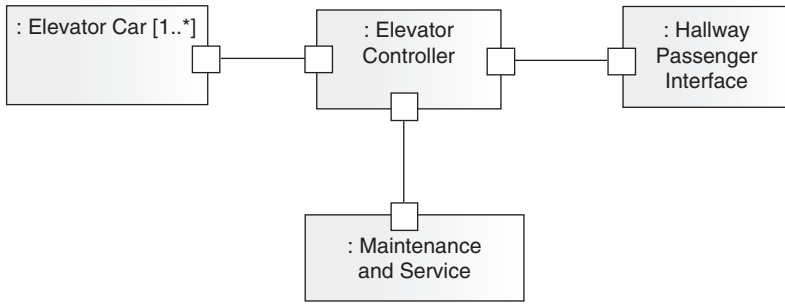


FIGURE 3.16 Exemplary internal block diagram for subsystems of an elevator system.

that connect blocks. Again, there are more elements of the semantics for an internal block diagram but these will suffice for an introduction. Figure 3.16 shows an internal block diagram showing the interface connections among the subsystems of the elevator.

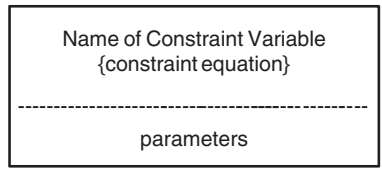
3.8 REQUIREMENTS MODELING

SysML also includes diagrams for requirements modeling. These diagrams show the requirements taxonomy being used by the systems engineering team. Far too many systems engineering teams do not have a requirements taxonomy so this feature of SysML should dramatically improve the practice of systems engineering. Chapter 6 of this book covers one possible requirements taxonomy.

In addition, SysML includes diagrams for showing the relationships established by the systems engineering team between each requirements and specific system functions, components, items (inputs and outputs of functions), and interfaces. Establishing these kinds of relationships was covered in the previous chapter as part of learning how to use CORE so it will not be repeated here.

3.9 PERFORMANCE MODELING

SysML uses a combination of block definition and parametric diagrams to enable the systems engineer to define performance and trade off models for use as part of the design process. The semantics of the block definition diagrams for performance modeling is not quite the same as that for block diagrams, see Figure 3.17. A rectangle, called a constraint block, is used to define each major variable for which an equation or constraint is defined. Besides the name of the variable appearing in the rectangle, the constraint equation appears inside the delimiters – {…}. In addition, a list of parameters used in the equation with their mathematical abbreviations is shown in the rectangle below a separating



The labeled rectangle represents a constraint variable for use in defining the equations in the parametric diagram.

The unlabeled connector shows a decomposition relationship (from the end with the diamond to the end with no diamond).

FIGURE 3.17 Semantic elements for the block definition diagram used for performance modeling.

line. The same sort of connecting line is used to show decomposition as in the block diagram case. Multiplicities are not needed. Figure 3.18 shows an example of a partial fundamental objectives hierarchy for a hypothetical elevator system.

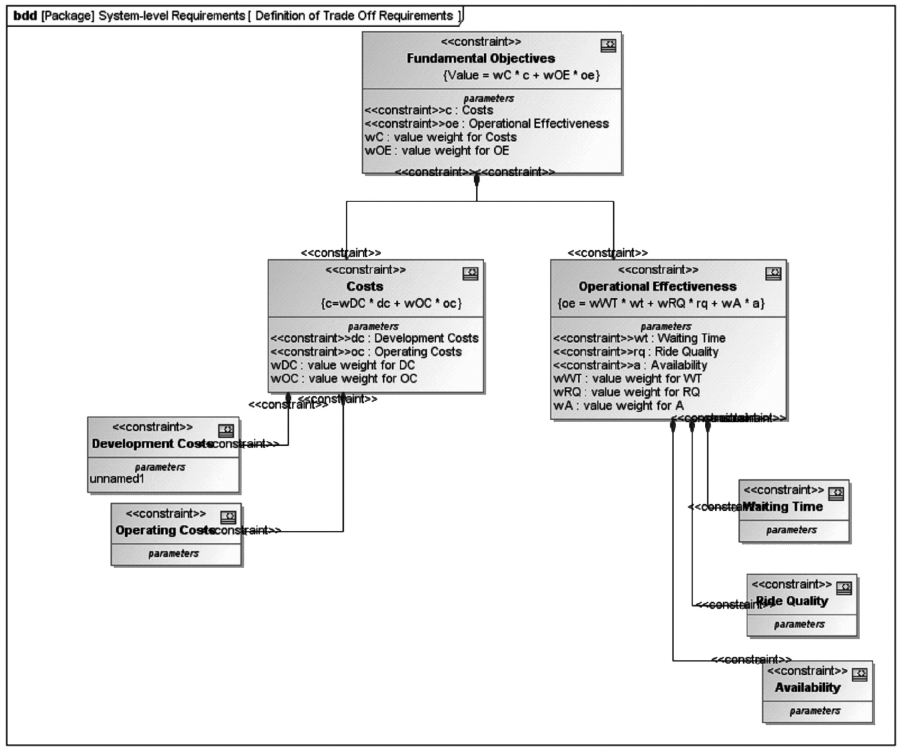


FIGURE 3.18 Exemplary block definition diagram for the fundamental objectives hierarchy of an elevator system.

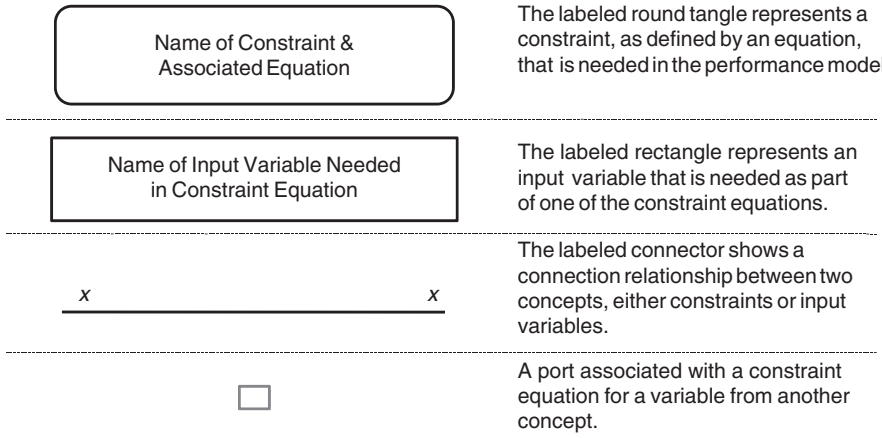


FIGURE 3.19 Semantic elements for the parametric diagram.

The second SysML diagram used as part of specifying a performance model is called a parametric diagram. The parametric diagram contains roundtangles for the variables with equations and rectangles for the input variables associated with those equations. Regular lines are used to connect the concepts in the roundtangles and rectangles. Finally, a small rectangle is used to show connecting ports for the roundtangles. These connecting ports are associated with variables being used in the equation. Figure 3.19 shows the semantics of the parametric diagram.

3.10 SUMMARY

The role of qualitative modeling in the engineering of systems is essential. This chapter introduced modeling, purposes of models, and categories of models and discussed how engineers use models in the engineering of a system. Models are used to answer questions for which better answers are needed than currently exist; each modeling technique has its own language of symbols and conventions for combining symbols into higher level concepts. A model is an abstraction of reality; models were characterized for the purposes of this book as mental, qualitative, quantitative, and physical. Each type of model has its advantages in terms of the types of questions that it answers best, as well as the development and operational costs for the model.

SysML's diagrams were introduced. The meta-system approaches of use case diagrams and sequence diagrams were described and illustrated for the elevator system that will be used throughout this book to illustrate the engineering of a system.

Next, IDEF0, a commonly used process modeling technique, was introduced and described in sufficient detail so that the reader should not only be able to

read an IDEF0 model authored by someone else but will be able with additional practice to develop IDEF0 models on her or his own. This process modeling technique was introduced here because this book concentrates on the methods to be used in the engineering of systems, and some process modeling technique is needed to describe these methods. IDEF0 has the advantage of being a good communication tool as well as having a standardized syntax and semantics that do not vary by organization and discipline.

Enhanced Function Flow Block Diagrams (EFFBDs) were described next as a way to capture the dynamic execution of functions within the system. EFFBDs have a general set of control structures that overlay the functional decomposition in an IDEF0 model to capture the unique dynamics envisioned within the system.

Next the block diagram semantics and syntax introduced by SysML were presented for both block definition diagrams and internal block diagrams. The former shows the decomposition of the physical architecture. The second shows the interface connections within a specific decomposition of a component.

Finally the new concept of parametric diagrams to define the performance modeling being done within the engineering of the system is presented.

PROBLEMS

3.1 Reproduce the IDEF0 diagrams of the process for engineering a system in Appendix B using CORE. You must pay attention to details of content as well as format. Both will be graded very carefully.

3.2 Create an FFBD diagram in CORE for each page of the IDEF0 model in Appendix B using CORE. Write a justification for the control logic of each diagram.

3.3 Describe at least three ways to estimate how much storage space would be needed if all of the emails sent during a 24 hour period from all of the people in the United States to anyone else in the United States were intercepted.