THESIS

CATEGORIZATION AND REPRESENTATION OF FUNCTIONAL DECOMPOSITION BY EXPERTS

by

Paul W. Melançon

September 2008

Thesis Advisor:	Gary O. Langford
Second Reader:	John Osmundson

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis
4. TITLE AND SUBTITLE Categorization and Representation of Functional Decomposition by Experts		**5. FUNDING NUMBERS**
6. AUTHOR(S) Paul W. Melançon		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Undersea Warfare Center, Division Newport, Rhode Island		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The objective of this thesis is to investigate different approaches to identifying system functions. The approaches that are described are standard functional decomposition process, Unified Modeling Language (UML), System Modeling Language (SySML), and Integration Definition for Function Modeling (IDEF0). A discussion is presented on advantages and limitations of describing and using functions by means of graphical formatting. Improving system functionality by effective decomposition is vital to robust system development. However, not one of these approaches presents the best method for complete functional identification. While each has its benefits and should be considered during functional analysis, a good decomposition has proper interrogation of the functions by means of coupling and cohesion of the functionality as well as identifying functional overlap and underlap. Standard functional decomposition works best as the first step in laying out system functionality. Rigor and completeness are improved when followed up by UML, SySML, or even IDEF0. Value and risk of each function can and should be identified as a way of posing a series of questions that measure and analyze the appropriateness of the functional decomposition. Combining these different approaches can help lead to a more complete functional decomposition and therefore reduce the risk to system development.

14. SUBJECT TERMS functional decomposition, Function, function coupling, function cohesion, function overlap, function underlap, value of function, risk of function, complete functional decomposition			**15. NUMBER OF PAGES** 89
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

CATEGORIZATION AND REPRESENTATION OF FUNCTIONAL DECOMPOSITION BY EXPERTS

Paul W. Melançon
Civilian, Department of Defense
B.S., University of Massachusetts, Dartmouth, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
September 2008

Author: Paul W. Melançon

Approved by: Lecturer, Gary O. Langford
Thesis Advisor

John Osmundson
Second Reader

David H. Olwell
Chairman, Department of Systems Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The objective of this thesis is to investigate different approaches to identifying system functions. The approaches that are described are standard functional decomposition process, Unified Modeling Language (UML), System Modeling Language (SySML), and Integration Definition for Function Modeling (IDEF0). A discussion is presented on advantages and limitations of describing and using functions by means of graphical formatting. Improving system functionality by effective decomposition is vital to robust system development. However, not one of these approaches presents the best method for complete functional identification. While each has its benefits and should be considered during functional analysis, a good decomposition has proper interrogation of the functions by means of coupling and cohesion of the functionality as well as identifying functional overlap and underlap. Standard functional decomposition works best as the first step in laying out system functionality. Rigor and completeness are improved when followed up by UML, SySML, or even IDEF0. Value and risk of each function can and should be identified as a way of posing a series of questions that measure and analyze the appropriateness of the functional decomposition. Combining these different approaches can help lead to a more complete functional decomposition and therefore reduce the risk to system development.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
  A. BACKGROUND 1
  B. RESEARCH OBJECTIVE 9
  C. DEFINING COMPLEXITY, SYSTEM, AND FUNCTION 10
    1. Complexity 10
    2. Definition of System 11
    3. The Term Function, Explained 14
  D. A FRAMEWORK METHODOLOGY 18

II. KEY FACTORS RELATED TO COMPLETE FUNCTIONAL DECOMPOSITIONS 23
  A. COUPLING AND COHESION 23
  B. DEALING WITH OVERLAPPING AND UNDER-LAPPING OF FUNCTIONS 27
  C. FAILURE MODE ANALYSIS, AN INTERROGATION TECHNIQUE OF FUNCTIONS 29

III. DIFFERENT APPROACHES TO FUNCTIONAL DECOMPOSITION 33
  A. UNIFIED MODELING LANGUAGE (UML) 34
    1. Activity Diagram 36
    2. Sequence Diagram 37
    3. Class Diagram 38
    4. Collaboration Diagram 39
    5. Statechart Diagram 40
    6. Use Case Diagrams 43
  B. SYSTEM MODELING LANGUAGE (SYSML): 45
    1. The New Diagram; Requirement Diagram 47
    2. Differences between UML and SysML 51
  C. INTEGRATION DEFINITION FOR FUNCTION MODELING (IDEF0): 54

IV. CONCLUSION 61
  A. FUTURE WORK 62

LIST OF REFERENCES 67

INITIAL DISTRIBUTION LIST 71

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1 Typical Functional Breakdown 3
Figure 2 Typical Functional Decomposition (From Langford, 2006) 5
Figure 3 Functional Decomposition of Household Lighting System 6
Figure 4 Systems Engineering Methodology (From MIL Standard 499B Model, 1994) 7
Figure 5 Process Context Diagram 14
Figure 6 Example of an N-Squared (N^2) Chart 27
Figure 7 Example of not mitigating failure mode (From Donald A. Norman, 2005) 31
Figure 8 Typical Activity Diagram (showing order processing) 36
Figure 9 Typical Sequence Diagram (From Bell, 2003) 38
Figure 10 Typical Class Diagram (From TogetherSoft, Inc, 2001) 39
Figure 11 Typical Collaboration Diagram (From TogetherSoft, Inc, 2001) 40
Figure 12 Typical Statechart Diagram (showing states of a hybrid SUV) (From OMG SySML, 2007) 41
Figure 13 Use Case Diagram example 44
Figure 14 SySML Diagram Types (From INCOSE Handbook) 46
Figure 15 Graphical Nodes Included in Requirements Diagram (From OMG SysML, 2007) 48
Figure 16 Graphical Paths Included in Requirements Diagrams (From OMG SysML, 2007) 49
Figure 17 Inputs and Outputs into the functional block (From Blanchard & Fabrycky, 1998) 54
Figure 18 IDEF0 sample diagram 56
Figure 19 IDEF0 Modified with Risk Attributes (From Langford, 2008) 65

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1 Perceived Systems Engineering Limitations of UML V1.x (Friedenthal & Burkhart, 2007) 35
Table 2 Comparison of UML and SySML (http://www.uml-forum.com). 53
Table 3 Suite of IDEF0 Methods (current and in development) 57

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

This thesis compares and contrasts several approaches to identifying functions using Functional Decomposition, Unified Modeling Language (UML), System Modeling Language (SySML), and Integration Definition for Function Modeling (IDEF0). Each of these approaches is described and depicted graphically, explaining how each handles system functionality. Benefits and limitations of each approach are explained.

The driving force behind this analysis into identifying system functions was to expose differences and key factors that lead to effective decomposition of functionality. Good functional decomposition has great influence on the success of system development against schedule, cost, and performance and quality requirements. Defining which approach should be used in a particular development effort seems impractical since there are too many subjective ways to manage development. However, key factors that help guide towards more complete functional decompositions are explained which, if followed, could reduce the risk associated with incomplete decompositions.

The findings are such that each of the approaches has benefits in identifying system functions, but none alone is best suited for complete identification. Using the 'standard' functional decomposition approach, breaking the functionality down into manageable chunks, i.e., sub-functions, seems to be the best starting point before combining other approaches. Keeping in mind the key factors that interrogate the functions such as coupling and cohesion, overlapping and underlapping conditions, and failure analysis can lead to better decompositions. Expanding on the interrogation of the functions with respect to value and risk, e.g., such as the application of the Systems Engineering Value Equation with Risk (SEVER) equation, can result in more complete functional decompositions.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank the Naval Postgraduate School for the outstanding education that enhanced my career and new found enthusiasm in the Systems Engineering discipline. Professor Gary O. Langford and Professor Dr. John Osmundson, I thank for the support required to accomplish this task. I also would like to thank my organization who provided the financial backing to further my education.

Ultimately, it comes down to those dearest to my heart, my family. Thank you to my mother and father, my wife and son. I could not have done it without you.

Thank you!

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

Functional decomposition has been used in electrical engineering and software development. It has further evolved and has become a process for defining and understanding functionality and functional requirements of systems in the field of systems engineering.

This thesis presents the investigation of functional decomposition as it applies to the systems engineering process. The different ways or approaches of determining functional requirements, e.g., Functional Decomposition, Modeling languages, and Integration Definition Function Modeling (IDEF0) are compared.

It should be noted that some authors have criticized functional decomposition as being flawed and as contributing to systems that do not meet customer requirements [Cantor, 2003]. It is the intent of this thesis to begin a dissection of this criticism. Why does it appear that some people are inherently better at performing functional decomposition than others? Is it a difference in thinking, in manner of approach, or even in education? This question was explored theoretically by review and analysis of relevant research. Particular focus was given to three questions: What is the range of appropriateness of functional decomposition as a systems engineering tool? What are the limitations, bounds, and applications? Is there a better approach to capture functional requirements?

In general, decomposition is a notion founded in reductionism. Reductionism is an approach used to understand complex systems simply by reduction (a simplification or condensation). Reductionist thinking forms the basis for most modern science and axiomatic mathematics. The development of systems thinking promotes a holistic view rather than a reductionist's method. However, functional decomposition combines reductionism with systems thinking. The methods of the reductionist may lead to incomplete decompositions because these methods do not convey or acknowledge the relationships between the reduced system components. Systems engineering improves on this situation by viewing functions and their interfaces as the building blocks for the system. Parsing functions with their associated performances, quality, physical, informational and other views further improves the ability of systems engineering to better characterize the desired system (Langford, 2008).

The general notion of functional decomposition is to break apart (i.e., partition and objectify) the components of an object into it sub elements (Langford, 2006). The purpose of decomposition is to give precise meaning to the relationships between a whole and its parts. Decomposition specifies the structuring and distribution of these parts in terms of the transfer of information (i.e., energy) between the parts – specifically the elements of the parts). Systems engineers can and do use decomposition to obtain clarity in the understanding of the system design.

Functional decomposition is a widespread design technique applied to design problems in many fields, such as systems engineering, software development and electrical design (Coulston & Ford, 2004). It is well known in the field of systems engineering and software development, yet is often employed in an ad-hoc or haphazard fashion, leading to less than desired results (Coulston & Ford, 2004).

Functional decomposition is a fundamental tool of systems engineering. It maps functions to physical components (thereby ensuring that each function has an “owner”) (Langford, 2008). It maps functions to system requirements. By its intention, it ensures all necessary tasks are listed and no unnecessary tasks are requested. The process of performing the decomposition should begin with the top-level function (see Figure 1) and then proceed through the major subsystem (Langford, 2008). However, in practice, beginning at any level in a functional hierarchy, the process is to move through or decompose in a logically step-wise fashion. The functional subsystem level should be completed next and then advance to the hardware / software, if appropriate (Langford, 2008). At each level, one completes the activities of functional analysis, allocation, and synthesis before proceeding to next lower level (Langford, 2008).

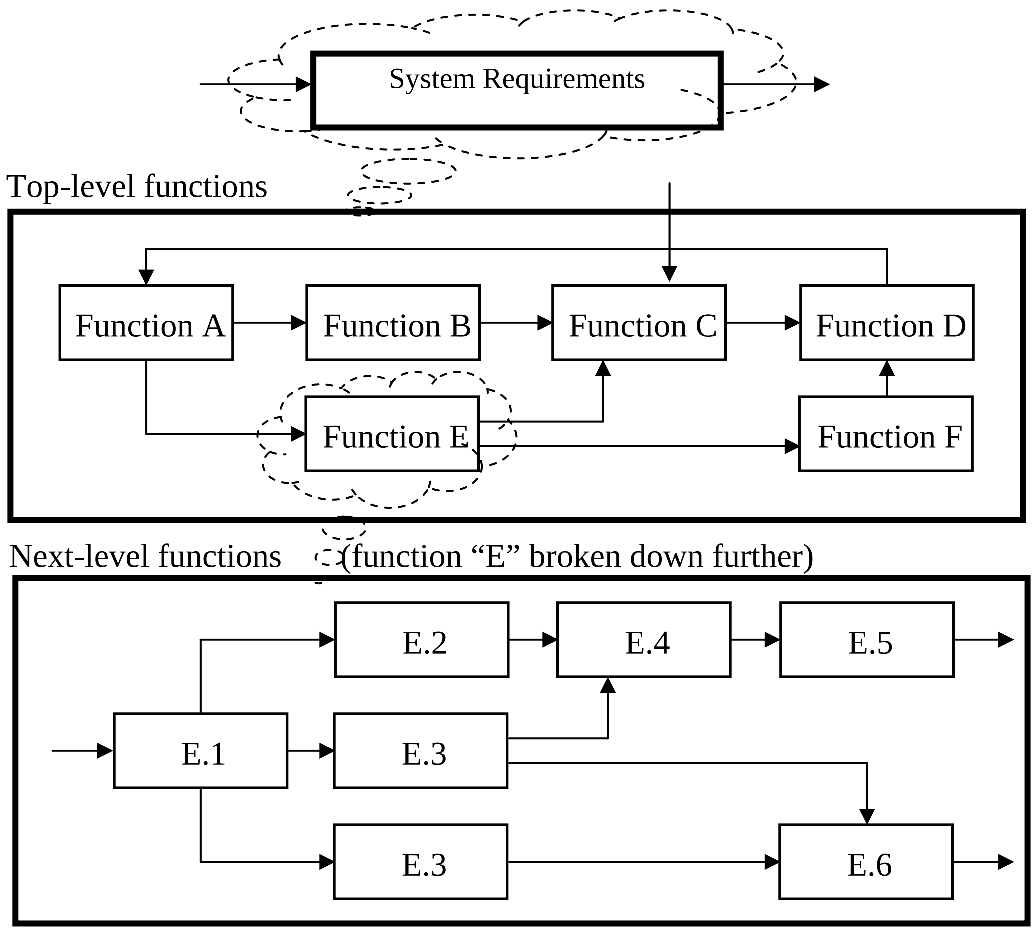


Figure 1 Typical Functional Breakdown

To some professionals, functional decomposition is something to avoid, as it has been rumored to be responsible for poorly designed, low quality systems (Cantor, 2003). Yet there are a plethora of successes using functional decomposition and it continues to be in widespread use, so, why do some consider there to be an issue? At first glance, the practitioner may not have the requisite skills and expertise to use functional decomposition effectively in system development. Therefore, the application of functional decomposition to derive requirements may be subjective and ill fated. With the interests in developing increasing complex systems, it is common to have tens of thousands of system requirements. Even the simplest of systems may have several thousand. For example, consider the simple task of withdrawing currency from an Automated Teller Machine (ATM). The general requirements of "ATM shall dispense currency," "ATM shall dispense currency in correct amount." and "ATM shall dispense currency when requested" can be expanded to include "REQUESTOR shall provide proper credentials," "ATM shall verify credentials," ATM shall authorizing disbursement," and "ATM shall disperse."

The typical functional decomposition results in a functional hierarchy diagram, a top to bottom parsing of general functions into their constituent parts. Higher levels of detail are found at the bottom. All functions and sub-functions are numerically designated to indicate kinship, (see Figure 2) (Langford, 2006). This depiction is an example of an event-structured functional decomposition. At the top of the hierarchy are the key function(s) that define the properties of the system required to complete the system objectives (Langford, 2007). The bottom of the hierarchy covers only limited objectives - a small set of the overall list of objectives. In that fashion, the top level function specifies the user need and the lower level functions specify specific systems needs (Langford, 2007). Some systems engineers have tried to generalize the hierarchy, short cut the methodology, and have stumbled because of a poorly defined set of terms that describe functions (Langford, 2007). The primary criteria for evaluating the worth and quality of a decomposed system are the numbers of interfaces and the type of information exchanged between system elements, i.e., the complexity of the decomposition (Langford, 2007). For example, complexity can be inferred from an entropic view (degree of uncertainty) of the decomposition (Langford, 2007).

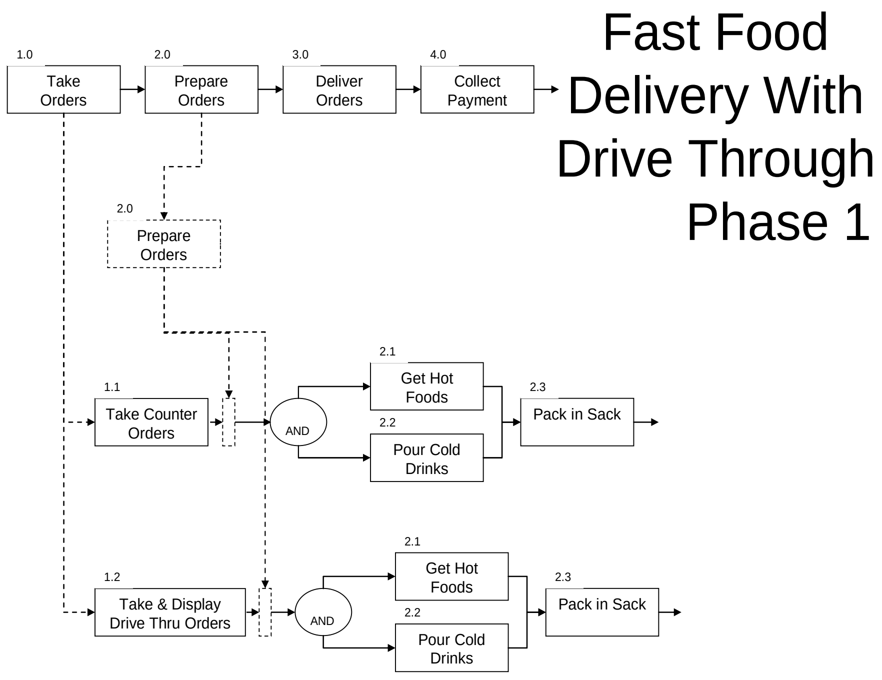


Figure 2 Typical Functional Decomposition (From Langford, 2006)

Figure 3 presents another graphical look at a typical hierarchical functional decomposition. Typically, functional decompositions are performed by first indentifying the top level physical constraints (boundary conditions). In Figure 3, "Functional decomposition of Household Lighting System," the boundary condition would be the physical limitation of the exterior walls of the house. There are interacting systems that would exist outside the boundary such as power lines and power stations (i.e., external systems) but here the primary focus is on the system of lighting within the house. The exterior of the house is the system boundary. The house also consists of a plumbing system, heating system, etc. A next step in functional decomposition could be to identify the top level functional descriptions of the physical items such as "provide power source." The physical aspect would be the power distribution system internal to the system boundary. Another example of a function is to "provide user control," the physical embodiment of the on/off light switch. From the physical decomposition diagram, the top level functional description is defined by parsing the interface requirements into a conjugate sets. The interface requirements are defined with consistency to instance the higher level inputs and outputs. This ‘nesting’ of hierarchical functions and their associated input/output processes and activities allows groupings (i.e., aggregation) of like and delineation of dislike interfaces between functions. At each level of decomposition the input and output requirements are matched to the functional description (Langford, 2006). A graphical method of systems engineering methodology showing the hierarchical system’s analysis (e.g., functional decomposition) is illustrated in Figure 3.

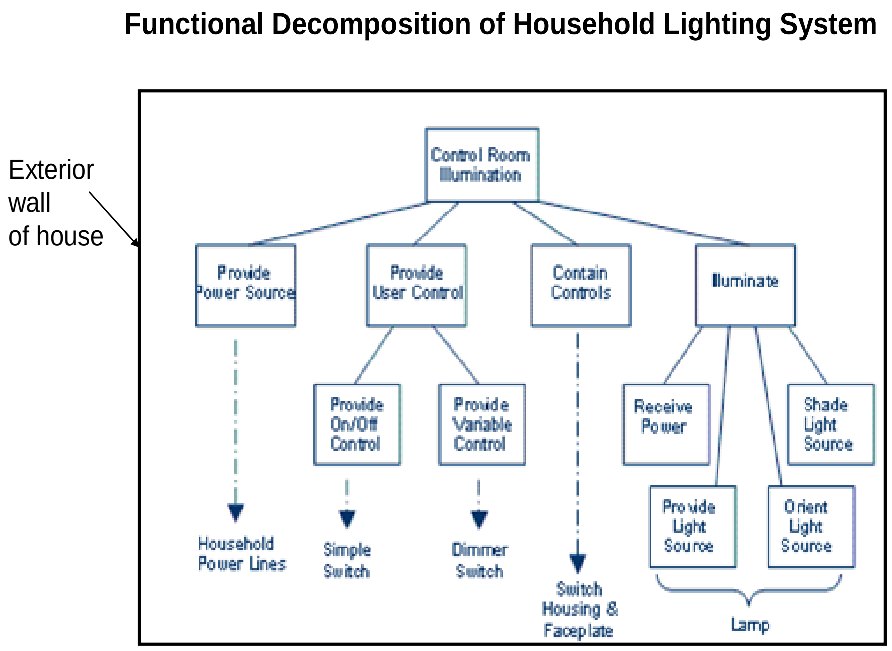


Figure 3   Functional Decomposition of Household Lighting System

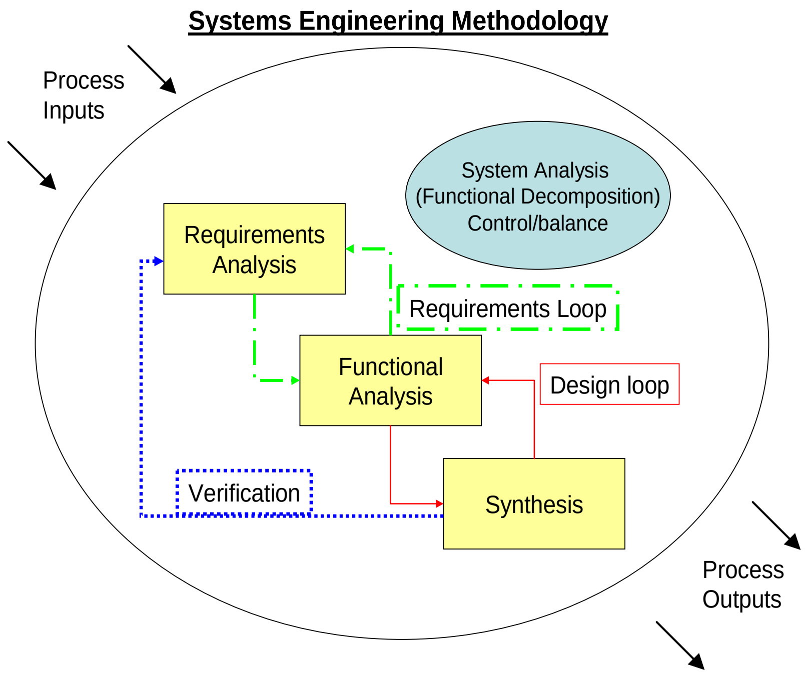


Figure 4 Systems Engineering Methodology
(From MIL Standard 499B Model, 2006)

There are many types of decomposition with different bases (e.g., functional, physical, informational). In general, the actions of separating distinct functionality into defined components that have well-defined interfaces are one of the essential ingredients of functional decomposition. Fundamentally, there are two types of decomposition: part/whole, and generalization/specialization. A discussion of decomposition theory can be traced back as far as 1776, and maybe further. Adam Smith, "An Inquiry into the Nature and Causes of The Wealth of Nations" (9 March 1776) stated:

Metals cannot only be kept with as little loss as any other commodity, scarce any thing being less perishable than they are, but they can likewise, without any loss, be divided into any number of parts, as by fusion those parts can easily be re-united again; a quality which no other equally durable commodities possess, and which, more than any other quality, renders them fit to be the instruments of commerce and circulation.

The decomposition of material elements that form the particular metal are broken down into sub-elements and reunited without overlapping or under-lapping issues (explained further in next section). A simple example of an underlap condition can be explained such as taking a block of wood (i.e., particular system) and decomposing it into smaller manageable parts by means of sawing, one would experience under-lapping due to the thickness of the saw cut (a loss of relationship between the parts). A reassembly of the parts back into the system as a whole would result with a smaller block of wood due to the loss of material from the saw cut.

An early publication that refers to the functional decomposition process was produced by G. Boole, *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities* London, 1854. Boole's writings are mentioned in much American and Russian research on the decomposition topic (Perkowski & Grygiel, 1995).

Additional influential work on decomposition was published by R.L. Ashenhurst, first in 1952, and later often described as the Ashenhurst Decomposition in 1957, a widely acclaimed paper on "The decomposition of Switching Functions" (Perkowski & Grygiel, 1995). The main idea of the Ashenhurst's decomposition, later modified by Curtis in 1962, was to decompose functions into simpler units of logic. This was done by reducing various cofactors in the corresponding representation of each unit, thereby compressing a larger number into a smaller number. Typically, this reduction was achieved by grouping redundant functions into a logical structure that services multiple other structures. The Ashenhurst-Curtis decomposition was appropriately characterized as recursive top-down reduction from the system whole to the constituent parts. A top-down approach to functional decomposition broke the main function into sub-functions, a hierarchical approach to better understanding of system complexity. The top-down approach allows logical, well-organized thought and orderly development of systems. However, premature binding of temporal relations can be a risk (Langford, 2008).

A second, popular work on the decomposition approach proposed by Dietmeyer, 1971, is qualified as a compositional type, which is described as bottom-up starting from defined parts, building through intermediate levels, until all output functions are realized.

In contrast to the Ashenhurst-Curtis decomposition, Dietmeyer's paradigm is particularly useful when the problem is defined to solicit a solution from a quantifiable and available set of well-quantified parts. An example might be to solve a problem by putting marbles into various boxes. The boxes and the marbles are givens, while the solutions are described as juxtapositions (put together to suggest a link or thread between them) of boxes and marbles. The Dietmeyer technique is premised on building blocks that are determined after a search of partition matrices for a classic decomposition property. The result is a predefined collection of modules that are used to design a function subject to constraints (Jozwiak et al, 1995).

These two techniques are adequately designed to address a wide range of different designs or paradigms. The Ashenhurst-Curtis Decomposition (top down hierarchal approach) is more suited to the open-ended design, while the Dietmeyer Decomposition (bottom up approach) applies appropriately to the self-constrained design. In addition, the Dietmeyer Decomposition lends itself more naturally to reversible logic designs, with the Ashenhurst-Curtis Decomposition being a special case of the Dietmeyer Decomposition for same. A like-minded approach was outlined in (Fang & Wojcik, 1998).

B. RESEARCH OBJECTIVE

The purpose of this thesis is to discuss and compare a few of the popular methods of deriving functional requirements – traditional Functional Decomposition, Unified Modeling Language (UML), Systems Modeling Language (SysML) and Integration Definition Function Modeling (IDEF0) – within the context of discovering improvements and limitations.

Research questions that were addressed include:

1. What factors contribute to incomplete functional decompositions?
2. What is the range of appropriateness of Functional Decomposition, as a systems engineering tool?
3. Are there other ways/approaches to performing the decomposition?

4. What are the limits of these different ways/approaches to functional decomposition?
5. Is one way/approach to determining requirements better than another?

One of the suggested critical aspects of improving system engineering is to transition systems engineering from a document centric perspective to an approach based on graphical modeling (Friedenthal & Burkhart, 2007). For some people, graphical views allow for better understanding, and traceability of system elements and their functionality.

Functional decomposition needs to be studied further. Incomplete decompositions can impact the success of systems development leading to poorly designed systems that are over cost, behind schedule, inadequately provided functionality and performance, or inadaptable to change. In addition, incomplete decompositions can lead system engineers to improper requirements and poorly defined architectures. Proper understanding of the approaches to decomposing system functions will help minimize the risk of not properly meeting the goal of the system. Further study may shed light on how to better develop less costly systems.

C. DEFINING COMPLEXITY, SYSTEM, AND FUNCTION

1. Complexity

The complexity and functionality of the cell phone has greatly increased in the past several years. However, does that mean the "system" is more complex or is it just that the heuristics are something we do not understand and are therefore deemed complex? At one time the "need" was to contact another person without the inconvenience of stopping what you were doing and traveling to a fixed location from which to place a call. Today the phone has mobility and juxtaposition with the caller, with an evolving set of functions that include placing and receiving calls, sending and receiving text messages, sending and receiving emails, "surfing" the Internet, verifying the stock market prices, taking digital images, and so on. Complexity is defined as the number and types of Worth Transfer Functions between stakeholders of a system, or likewise between system elements (Langford & Huynh, 2007). Therefore, increasing the number of functions increases system complexity. Heuristics, if not understood, may cause complexity in the system structure. Comprehensive and complicated are terms commonly used to describe systems that have given way to understanding of the system's heuristics. However, complexity is referred to that which is usually not understood.

Functional decomposition is the widely practiced methodology that deals with system complexity, focusing on intelligently partitioning the system into smaller, more definable pieces. An improvement over the standard functional decomposition is described in "A Methodology for Managing Complexity" (Langford & Huynh, 2007) in which Value Transfer Functions between stakeholders and the number of stakeholders is directly related to complexity. The paper outlines measures that lead to the understanding of schedule uncertainties and the sensitivities between the Work Breakdown Structure and the schedule. A large number of interacting elements or sub-systems can be difficult to understand. For example, the lamp in a room is a system and considered by most systems engineers to be a basic system. The function of the lamp is 'to light'. If one were to understand a broad view system (including power plant that provides the electricity to the lamp) that perspective would be describe as complicated and comprehensive, even if the heuristics were understood (Langford, 2008). Managing complexity is accomplished by defining tasks whose outcomes flow together, creating a successful system that includes all the various interactions and relationships. In these cases, functional decomposition is used to decompose the different elements or tasks of the system into more manageable parts, thereby allowing the overall system behavior to be understood as a straightforward composite of the behavior of its many elements (Langford, 2007). Functional decomposition is a convenient means to divide the problem into meaningful, yet understandable parts.

2. Definition of System

A system is defined as an assemblage or combination of elements or parts forming a complex or unitary whole, such as a transportation system (Blanchard & Fabrycky, 1998). Systems contain elements, which are interacting interdependent (or temporary) sets of variables that maintain certain functions, behaviors, and performance relations (Langford, 2006). Elements within the system are defined as functions, processes, technology, users, products, or services (Langford, 2006). Every element has a lifecycle. A lifecycle is the event-phased course of developmental changes that occur from concept to the termination of the element's use) (Langford, 2006).

Systems consist of many interfaces, e.g., physical and functional (Langford, 2006). The physical interface comprises the things we encounter everyday, such as cell phones, automobiles, shoes, etc. The functional interfaces are sometimes less obvious, but equally important as the physical interface. The functional interface goes hand-in-hand with the performance characteristics of the system. Consider a shoe made from steel rather than leather or cloth. The resultant performance difference between steel, leather, or cloth would be shoes that may wear extremely well in the case of steel, but be rather uncomfortable and difficult in which to walk. One can readily envision multiple trade-offs to include changes in temperature, size, weight, susceptibility to temperature, water, and different terrains. The manner and means of dealing with the physical and functional interfaces are central to the systems engineering process.

Systems have emergent properties that are described as having many system entities, operating in the same environment, resulting in a complex system (Langford, 2008). Emergent properties affect how the system capabilities are defined, which in turn may indicate how to meet the intended needs of the stakeholder (Langford, 2008). As described by (Blanchard & Fabrycky, 1998), "Systems are composed of components, attributes, and relationships." Components form the basis for system operations; therefore, the functions they perform characterize the system. Improper decomposition or identification of system functionality can lead to systems with missing elements and therefore do not meet stakeholder needs (Langford, 2008). Therefore, good decomposition results in better system requirements, i.e., those requirements that are verifiable and validated.

A system needs to have limitations imposed to relegate the problem to definable and implementable tasks. Boundaries and constraints define such limitations (Langford, 2006). A complete description of a system includes all of its domains and all of the elements contained within these domains. Otherwise, the boundaries are too constraining, and the alternative solutions to the problem may reside outside the imposed system boundary. The system will not meet the user's need (Langford, 2006). Determining a system boundary can be a challenge! How can you be sure that what has been defined is appropriate? Good analysis of the consequences of said boundaries and communication of the system boundary to stakeholders is important to making the final determination. As discussed in (Brown, Cantor, & Mott, 2006), (Cantor & Roose, 2005), (Dockerill, 1999), (Long, 2008), good communication can be represented by a standard, intuitive graphical language that is easily understood among all stakeholders, customers and system engineers.

The success of a development process can be measured by its value and value can be determined by the ratio of performance to investment. Successful systems are defined as satisfying the needs – are at or under budget, delivered on time, have requisite functionalities and performances, and do not result in unexpected losses (Langford, 2007). An analysis that indicates the stage of development, the next steps to be accomplished, defined ends and deliverables, the budget and schedules, and the conditions for success/failure are essential to developing a successful system (Langford, 2006). Generally speaking, this process is called systems engineering. Systems engineering helps keep track of stakeholders/customers and their needs; the product and specifications; production and operational support; and the program management and organization.

The process of thinking, reasoning, and structuring facts and relationships to provide clear and unambiguous direction to managers and developers, as well as accounting for progress and risks, is a strategy based on iterative, top-down, and hierarchical decomposition of system functions to derive requirements. By this process, the analyses and studies objectify the basis for, and sometimes suggest, methods to support key decisions (Langford, 2006). By following this top-down development process, one can reduce design risk by attacking the most difficult design area(s) first, throughout its total hierarchy, during the start of the development.

System elements display functions, behaviors, performance, and quality (Langford, 2006). These elements, or the constituent sub-elements of elements, have associated lifecycles, which means the elements are impacted by the developmental changes that occur from initial concept to final termination. The inability of certain system elements which are not adaptable to change or upgrades can influence and impact project success. This is sometimes found to be the case as technology evolves and the result is an increase in lifecycle costs. Such non-adaptive systems might be terminated earlier than originally scheduled. A decomposition of functions may not always expose all the requirements since no theory of building systems is finitely describable consistent, or complete (Langford, 2006).

3. The Term Function, Explained

Keeping in mind the applicability of functional decomposition to the systems engineer, the term function is defined loosely as a property of the system that is required to achieve a system objective (Langford, 2007). Functions are implemented as processes, with inputs and outputs, or as activities, without inputs and outputs. Inputs and outputs represent the context for the attributed function. Below is a Process Context Diagram.

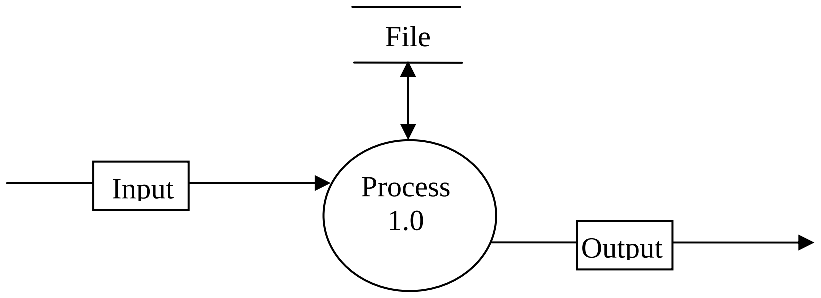


Figure 5 Process Context Diagram

Inputs are shown as text within an arrow pointing towards the process. Outputs are shown as text within a box with an arrow pointing away from the process. Processes are shown as text within a circle.

As described in “A Methodology for Managing Complexity” (Langford & Huynh), value of the developing or developed system is defined within the context of Value Engineering, as a ratio of performance to investment. Value per function is specified in terms of the performances of these functions. Worth is defined as the Value multiplied by the quantification of quality of the functions and their performance(s), i.e., the losses incurred due to the performance of functions (Langford & Huynh). Therefore, the system shall provide a function with a specified performance and a delimited level of quality (Langford & Huynh, 2007).

A function requires at least one input and at least one output to ultimately enact or realize the desires of the user. The result(s) of a function is its output(s). Decomposing functions exposes the required interfaces and connections as well as the boundaries of the beginning and ending of the domain of the function. During the decomposition process, if it is found that a function has no input or output, an incomplete decomposition has occurred. If the inputs or outputs are not apparent then incompleteness has been identified and the boundary between functions must be investigated and defined (Langford, 2007).

Distinctions between behavior and function, as well as function and purpose, are sometimes blurred to systems engineers (Bell, 2004). Definitions of function vary from researcher to researcher (Bell, 2004). Though these definitions are differently worded, they may be similar in meaning. Particular attention needs to be placed on the scope of the definitions and the resultant implications of scope (Langford, 2008). According to (Chittaro & Kumar, 1998) the operational definition is where function is a relation between input and output and the purpose, where function is described as a relation between user’s goal and the component’s (or system’s) behavior. For example, the function ‘to float’ is the behavior required of the boat (as a system). The behavior is defined as the system performance. The purpose of the boat may be to get across the river, but the performance ‘to float’ satisfies the need to get across the river and not sink before that task is accomplished. How well a boat performs the function ‘to move in water’ embodies the measures of the performance ‘to float associated with that function’.

Distinguishing between the four classes of knowledge (Chittaro & Kumar, 1998) structure, behavior, function and purpose, it is described (Bell, 2004) that structure is what is internal to the system, i.e., the function 'to move in water' and the performance 'to float' are structural consequences of the interfaces between the boat and the water. Behavior is what happens inside the system, i.e., does the system meet the intended goal? Function and performance are measured as what happens at the surface/boundary of the system, i.e., the boundary between the boat and the water. The purpose is the goal that is satisfied outside the system, i.e., the boat traverses the river without sinking.

For every function (from the highest to the lowest levels of decomposition), there is at least one performance requirement and at least one quality requirement (Langford, 2007). For every performance requirement, there must be at least one quality requirement (Langford, 2007). Functions are identified as properties of a system requiring achievement of a system objective, while performance is a measure that qualifies the fulfillment of those functions (Langford, 2007).

Functions are grouped in a logical form to meet both functional and supplementary needs. (Cantor & Roose, 2006) These interactions, or threads between each system function, must be described adequately to allow the intended system's purpose to be met. These functions are allocated to physical "owners," e.g., hardware, software, etc. The collaboration between each function is then evaluated and portrayed to determine the extant relationships. This analysis is referred to as a Functional Thread Analysis (FTA) (Langford, 2008). Some apply the Use Case methodology defined in the Unified Modeling Language (UML) to define and illustrates these relationships.

The community of functional reliance has applied knowledge of system relationships, and therefore of system functions, to deduce the system behaviors (Langford, 2008). This supported diagnosis (Sticklen et al., 1989) and Failure Modes and Effects Analysis (FMEA) (Hawkins & Woollons, 1998). In this regard, the system function can be suggested (and perhaps defined) by relationships between its lower level subfunctions. In contrast, a "top down" system perspective expresses functions that can support the design and architecture processes (Iwasaki et al., 1993). By using the example supported by functional refinement of the design process in (Gero, 1990), the hierarchical association of the system's functions can congruently be associated with constituent characteristics (Langford, 2008). The functional levels (or views) can be interpreted as behaviors. These views express the input and output models for analysis of design and its various interpretations (Snooke & Price, 1998). By this approach of "functional labeling," system behaviors are tracked by the system's functions. The set of behaviors are mapped to the desired functionality and are often delineated as process outputs. Some authors refer to these outputs as vectors, or system states, or "goal states." Applying such nomenclature, the function "headlight NOT lit" associates inputs and outputs through the related system functionality with a system state which represents that the headlight is off. The use of goal states is discussed in (Snooke & Price, 1998). Goal states' approaches associate functions with failed outputs, and lend themselves to failed output reporting, which facilitates fault analysis and diagnosis.

The intended results are a functional description language (Langford, 2008) that deals with functional (input and output) dependencies. This is most useful when viewing the system as a totality, without visibility into its internal workings explained in (Bell, 2004). A more formal definition of function can be explained through a model of elements, e, functions, F, and triggers, G, which results in the achievement of an external effect E (Langford, 2008). Functions are built up of subfunctions and activities. Subfunctions and activities are the elemental units of the System (Langford, 2008). Some authors (Chittaro & Kumar, 1998) pertain to objects, O, functions, F, and triggers, T. Chittaro and Kumar refer to external triggers that permit the system to achieve goals. In fact, system functions are enacted only through internal triggers (Langford, 2008). However, both definitions are consistently modeled as relationships between functions, goals, and behaviors. (Chandrasekaran & Josephson, 1997) viewed these models as merging the purpose and operation as representations of system functions. In total, this representative view of functions that are triggered by internal (or external) triggers is unlike other perspectives on functions. The distinguishing difference is formed by both the system behaviors and purpose. As such, rigorously defining functions in this manner, supports the view that element, trigger, and effects are inexplicitly related (Langford, 2008). This representation determines: (1) how the system achieves the function, (2) the trigger that stimulates the function, and (3) the function's effect. The idea that a function can have multiple triggers and resultant effects is conveniently incorporated in those decompositions that are comprised of subfunctions. Derived from this descriptive state (with multiple possible states) is the conclusion that some system enactments may be without the fulfillment of some functions. The result is a system behavior that may not be predictable based merely on a decomposition of functionality, but instead may achieve altered states of enactment (sometimes with emergent properties) that are wholly unpredictable (Langford, 2008).

If functions are decomposed into subsidiary functions, it may be necessary to relate the system effectiveness to the various states of system and subsystem functionalities. This decomposition schema can result in several states of functions and sub-functions each indicating a different or variant scenario (Langford, 2008).

D. A FRAMEWORK METHODOLOGY

The term 'framework' for discussion, refers to a means of organizing and reporting the performance, impacts, and effectiveness of an enterprise (Langford, 2008). Representations of frameworks are a graphical means to display the impacts of functional analyses. Frameworks are based on a broad appreciation of the kind, purpose, and content for assessing and portraying impacts and effectiveness.

Another implication of the objective of decomposition is to define a whole in terms of parts (i.e., partition and objectify) that have the least-complexity with the most architecturally effective design. One way to think about architecture is through various perspectives of the result of a system (Langford, 2008). One can posit a framework of views, or enterprise framework(s) that are comprised of nine architectural views (physical, operational, functional, performance, quality, information, energy, profit, and temporal) (Langford, 2008). These architectural views are based on long-standing determinates of design – independence, aggregation, form, relationship, attribute, pattern, and juxtaposition (Langford, 2008), which dictate the many factors that relate to the problem and the solution.

The two primary interfaces in all systems are physical and functional, which is a consequence of defining the term "system." These interfaces are viewed as two sub-divided areas: internal and external. Internal interfaces encompass the partitioning and design elements. The external interfaces span differences between partitions and interactions with the non-system related environment(s). Partitioned boundaries are selected carefully to minimize the number of requirements with cross partition interfaces (Kanefsky et al., 1999). Protocols, processes, and activities between activities and interfaces are therefore improved.

Partitioning is the process of dividing a system into constituent parts, thereby defining smaller tasks as opposed to one all encompassing task. This division serves to expose redundancies in the system design; specifically issues related to physical, operational, functional, performance, and quality. Partitioning functions allow the system engineer to measure, interrogate and objectify the system (Langford, 2008). Partitioning, much like decomposition, is formulated by both the process and the intent of the partitioning. In part this process depends on is the division of a large number of small tasks or a small number of large tasks. A good partition (or complete decomposition) allows flexibility to small changes and robustness in terms of scalability to meet additional needs. When the designer (or architect) first focuses on the movement of energy (or energy equivalents) through the proto architecture, the partitioning technique is termed *domain decomposition* (Langford, 2008). Where, as the alternative approach, *functional decomposition*, deals first with the functions to be performed, and then deals with the energy issues (i.e., architecture). These two types of decomposition represent different, but complementary, ways of thinking about structuring a problem. For instance, if the design can be divided into disjointed, but non-overlapping parts, (Langford, 2008) the partition function is complete.

In the previous section, the decomposition technique presented by Ashenhurst-Curtis laid the groundwork for the top-down approach to decomposition. This technique pushes the partitioning process to completion prior to defining the system. An iterative process, much like the functional decomposition generally referred to in systems engineering, is the practical application of top-down (and bottom-up) approaches.

Reconciliation of the two up/down approaches brings closure of this iterative process. Usually closure is achieved after the system has been implemented (Langford, 2008). This all leads to the chief criticism of the Ashenhurst-Curtis decomposition.

Each partition matrix must be examined to determine if it possesses the specific decomposition property that is essential to its part in fulfilling the top-level and horizontal-level functions and not to another part. By comparison, these partition matrixes reveal the overlaps and under-laps of the parts when aggregated and summed to make the whole. Since only a very small number of decompositions will have the requisite set of specific attributes, the challenge is to group attributes to reflect the desired degree of uniqueness or omnipresence that satisfy the system requirements (Langford, 2008).

The analysis framework will organize the presentation and facilitate discussions about the functions and performance requirements. Understanding of the overall design is encouraged by the constituent relations and functions, transformations (e.g., combining and composing), and comparing properties of classes of functions. This framework will formulate and describe the appropriate relationships, and extend and apply generalized notions to the specifics of the uses and actions of the system to form operational scenarios. The framework should also consist of or be integrated with applicable rules and best practice principles. A convenient way to organize the structure of the framework is to facilitate decision-making. One of the purposes of a framework is to support and simplify decision-making processes (Langford, 2008).

Frameworks are built within the context of a set of like-kind scenarios, having at least a commonality that distinguishes one framework from another framework. The measures may be similar for a variety of frameworks, and perhaps there is a grouping of like-kind frameworks with similar (or same) measures. The metrics may be different for different frameworks, but alternatively, the metrics may be the same for different frameworks (Langford, 2008).

The foundation of the framework could be a model of a process or a set of processes, a function or a set of functions. There could be a ‘model’ framework, an ‘application’ framework, a ‘domain’ framework (when aggregated it represents the ‘system’ framework), and a ‘support’ framework. The framework could be modular so that new frameworks could be proposed. Frameworks could be considered from different perspectives, much the same as the points of view of stakeholders. There must also be the consideration of scope of framework (Langford, 2008). The ‘application’ framework could comprise the set of applications that are broadly applicable to multiple frameworks. The ‘domain’ framework would contain all the functionality for the groupings of bounded set of elements. The boundary may be physical or intellectual. Generally, the boundary will be of a convenience that permits aggregation in one form, or another of a generally accepted grouping (Langford, 2008). The ‘support’ framework could provide the system functionality that underlies all other frameworks. The ‘model’ framework would be representations of any other domain(s) abstracted to facilitate understanding without loss of clarity.

THIS PAGE INTENTIONALLY LEFT BLANK

II. KEY FACTORS RELATED TO COMPLETE FUNCTIONAL DECOMPOSITIONS

A. COUPLING AND COHESION

As mentioned in theChapter I, some view functional decomposition as flawed, possibly because many have used the process incorrectly. The processes of functional decomposition, along with emphasis on key aspects that lead to insufficient decompositions, need to be explored. When done properly, the process of functional decomposition defines the 'logic' of the system. At a fundamental level, functional decomposition is the basis of all science, all structure, and all thought (Langford, 2008).

The struggle to perform functional decomposition (from the perspectives of the unskilled or unknowledgeable) centers on two questions: (1) how does one determine if enough information has been collected about the functions? and (2) when enough is enough during the decomposition process, i.e., how far does one go with decomposition? The following questions should be investigated: how to define the specific tasks of the functions; how active should stakeholders be during development of functional hierarchies; how are tasks performed; where are these tasks performed; what is the timing and sequencing of each task; what is the nature of inputs and outputs for each task; who are users of the outputs of each task; what is required of each task; is each task necessary; what policies apply to the work; what rules are key drivers; which regulations apply to the functionalities; what controls are required to be applied to the function; and what equipment is used to enact the function. Consideration, and perhaps answers, to these and other questions, helps to make the determination as to whether enough information was collected about the function (Langford, 2008). If a function cannot be allocated to a component of the system, further decomposition of this function is necessary to determine the level whereby the function can be allocated to the proper component. In some cases, the functional hierarchy may not be mature enough with the resultant requirements unallocated as a single entity. Therefore, the performance decomposition would need postponement until the functions are clearly defined. A usual criterion for completion of functional decomposition is to continue the process until the functional requirement is clear, realizable, and allocatable in hardware, software, and/or manual operations (Langford, 2007). The objective of generally decomposing the system into its hierarchical components helps the analyst better appreciate and deal with over stated and under stated functionality. Assessing risk involves partitioning functions by the appropriate measures of performance, the lifecycle costs, and the losses incurred as a result of the observed performance. Determining when decomposition is considered “complete” is therefore warranted.

Coupling and cohesion of system function is a means of performing an interrogation of “completeness” of the decomposition process, i.e., how far to go with decomposition. “Completeness” is defined as a reasonable level of verification that what has been conducted thus far for decomposition has sufficiently identified all necessary parts, elements, or steps to effectively and properly define the system. Total “completeness” is not a goal, rather completeness of functional decompositions means no better decomposition is likely given the knowledge and skills.

Functional Analysis (the method of identifying, characterizing, and arranging the levels and domains of system functionalities) is a development of the architectural determinants which evaluate the degree of coupling, cohesion, and connectivity of functions and sub-functions (Langford, 2006). Functional analysis can be used to verify that the intended state of a system suffices to improve the system performance. Functional decomposition is performed to determine what the system is supposed to (and likely to) do. By this method, the current state of the system can be defined as well as the future desired state.

Coupling is a measure of interdependence between sub-functions. Low coupling is defined as that change in a module that affects very few other modules (Langford, 2008). Functions that carry a higher coupling pose risk, as they are prone to creating a ripple of changes that involve other modules. The higher coupling factors also make the impacted modules difficult to understand by themselves and stand-alone testability of the module becomes an issue. An increase in dependant modules this leads to an issue with reusability as more modules are impacted.

Coupling and cohesion are related. When a low coupling exists, a high cohesion will result. When the modules are independent (without responsibilities to each other), the cohesion is increased. Cohesion is defined as the similarity of functions performed. High cohesion aggregates as many like tasks as is convenient, up to the limitations of physical and other system properties. Connectivity is defined as the reference that relates one module to another. By circumstance, lower connectivity implies a smaller number of interfaces.

If a function consists of only a single output variable with a precise task, it is considered cohesive. At high levels of cohesion the functions perform "tasks," like "turn on" or "turn off," but a higher level of cohesion such as "turn on light," "turn off light" is considered an improvement. The more the function is focused, the more it is cohesive. A high cohesion function is simpler to understand, having to do only a single task, while a low cohesion function will be difficult to follow due to the many different tasks it executes. In addition, a high cohesion function is easier to reuse because of the limits on tasking, and therefore easier to extend. High cohesion maximizes reusability and extendibility. A simple test to determine low cohesion: no simply name describes the function.

Functions that perform several activities are considered non-cohesive. Instead of one monolithic function that performs many activities, it is preferable to have several smaller functions each performing a single activity (Lakhotia & Deprez, 2008). Cohesive functions also reduce the complexity of the system, and aid in better understanding, communication and more complete functional decompositions (Langford, 2008).

Coupling and cohesion are explained in terms of elements (like modules, classes, or frames) that are linked in some way (e.g., by function calls) (Langford, 2008). The degree of dependence within such an element is called cohesion, and the degree of interdependence between these elements is called coupling (Langford, 2008). In general, low coupling and high cohesion are indicators of minimal interfaces and good modularization (Kramer & Kaindl, 2004).

Another tool that aids in functional decomposition is the function matrix or N-Squared (N^2) chart (Long, 2008), shown in Figure 6. Functions are indicated on the diagonal of a matrix of functions as well as inputs/outputs. Outputs are indicated horizontally, while inputs are indicated vertically. Non-functional entities are defined as interfaces with only an input and output. These are linked functions. The N-Squared Chart allows for a graphical view of functional inputs and outputs and their dependences, thereby allowing for visual verification of coupling and cohesion.

The functional decomposition along with functional flow diagrams is used to portray what the system is supposed to do. The functional flow block diagram shows the function and sequence of functions. This validation method displays the current state, i.e., effectiveness, of the system in meeting its desired goal. From here, the performance of the system can be altered or changed. The behaviors of the functions are captured in a behavioral analysis chart that indicates functional juxtapositions, input and output ports (or connectivity's), sequences, controls, and data / and data flows. The timeline diagram shows functions, sequences, and timing. Whereas the data flow diagram shows data/data flows and control flows.

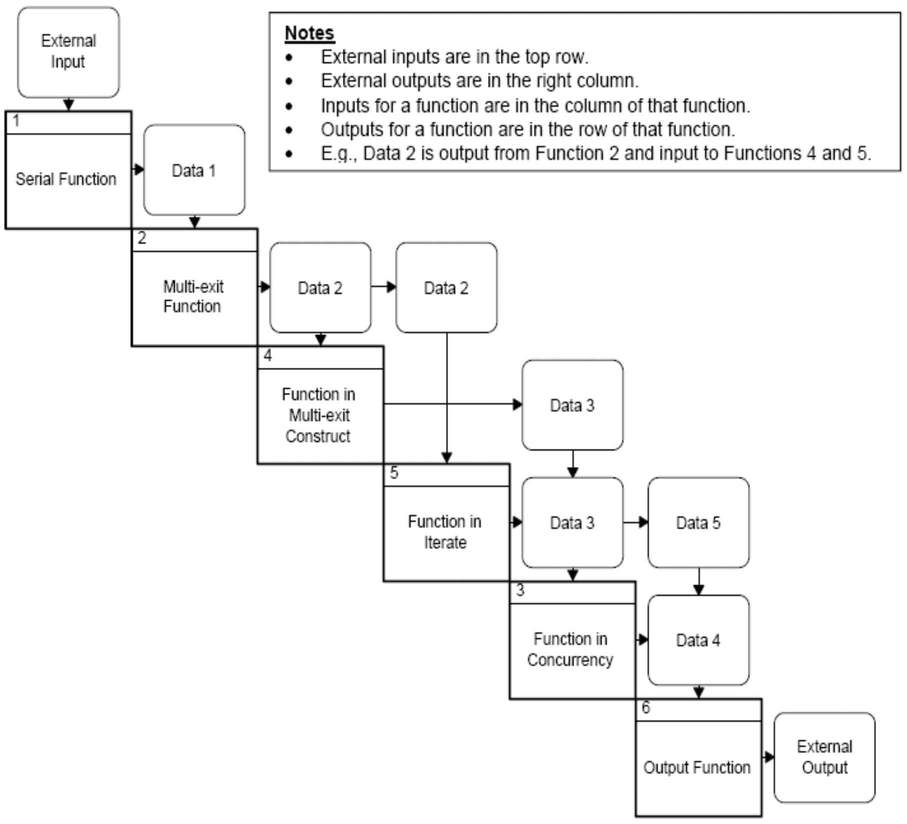

External Input
Notes
External inputs are in the top row.
External outputs are in the right column.
Inputs for a function are in the column of that function.
Outputs for a function are in the row of that function.
E.g., Data 2 is output from Function 2 and input to Functions 4 and 5.
1
Serial Function
Data 1
2
Multi-exit Function
Data 2
Data 2
4
Function in Multi-exit Construct
Data 3
5
Function in Iterate
Data 3
Data 5
3
Function in Concurrency
Data 4
6
Output Function
External Output

Figure 6 Example of an N-Squared (N^2) Chart

B. DEALING WITH OVERLAPPING AND UNDER-LAPPING OF FUNCTIONS

Good functional decompositions consist of no parts having overlaps with other parts, and no under-laps with logically adjacent parts (Langford, 2008). Dealing with these overlapping and under-lapping issues requires analyses of (1) scenarios (e.g., Use Cases), (2) data flow diagrams, and (3) N^2 charts. All these are facilitated by, and enacted through, functional decomposition. Diagrammatic tools capture behaviors, i.e., activities, sequences, collaborations, and statecharts. These diagrams can augment and enlighten various processes (see UML and SysML sections).

Overlap of system functionality is defined as functions that are duplicated and not independent. The overlap of functions leads to conflicts in the execution of system tasks (Langford, 2008). Overlap is not the same as redundancy, redundancy to increase reliability.

Underlap conditions can be detected by analyzing the functional decomposition to determine if applying scenarios exercise all functions (Langford, 2008). Scenarios can be enacted by the use of functional decomposition by itself or behavior diagrams, activity diagrams, sequence diagrams, and Use Cases. Functions that are unaccounted for in the functional decomposition (and need to be invoked) are considered underlaps. This underlap issue is corrected by adding a function or a set of functions and rerunning the scenario through the functional decomposition. Sometimes the notion of simply adding a new function does not adequately address the issue of underlap. Since the newly added function will have interfaces to other system functions, those new interfaces may not seamlessly integrate with these other functions (Langford, 2008). Consequently, some functions will need to change in scope (inputs, outputs, mechanisms, and processes) to better align the processes and actions with the new structure of functionality. Once the underlap condition is accommodated and corrected, the range of scenarios should be broadened to continue testing the efficacy of the functional decomposition. Additional new functions may be necessary, and if sufficient, the functional decomposition will be complete, through the first phase of analysis.

The next phase of this analysis centers on interactions. Specifically, consider the actions of one function of the system with either the extended system or with an external system. Newly added function(s) need to satisfy the underlap condition and may result in changes in the interactions with external systems. The result might be a different interface requirement between the developing system and the external system(s). Making adjustments in both the total system functionality as well as in the interaction(s) with other system(s) may be required in order to satisfy the goal for correcting an underlap condition (Langford, 2008).

C. FAILURE MODE ANALYSIS, AN INTERROGATION TECHNIQUE OF FUNCTIONS

Failure mode analysis is a method of finding possible faults in a system and reviewing these faults as to the consequence on the system. A definition given by the British Standards (1991) describes failure mode analysis as:

> A method of reliability analysis intended to identify failures which have consequences affecting the functioning of a system.

Failure mode analysis is often criticized as being effective in the design due to the amount of effort it usually takes, therefore resulting in exceeding the design/development phase (Hawkins & Woollons, 1998). Mitigating failure modes due to system functionality have proven to be effective in system design by requiring a deep understanding of the engineering system, functional diagrams, i.e., graphical understanding (Hawkins & Woollons, 1998). There are two main areas of this analysis (Aerospace, 1979). The first is the hardware approach, which deals with the behavioral changes that occur from the failure. This greatly affects the effectiveness on the design since it is performed once hardware has been designed. Changes to hardware because of this analysis can be costly and affect the program schedule. This is not to say that it should not be conducted, however, more attention to the second approach will mitigate the impact to the program from the hardware perspective. The second approach focuses on the functional aspect. The analysis of failures at a functional level will greatly benefit the program since it is performed early on in the initial design stages, i.e., during functional decomposition (Langford, 2008). If the functional approach yields the most benefit early in the design phase then why is it not widely practiced? As mentioned before, the effort to conduct a failure mode analysis can take time and this deters most developers due to program time constraints. However, completion of programs does not necessarily mean they were successful. More and more it is apparent through Department of Defense case studies and GAO reports that programs are continuing to be behind schedule, over budget, or both. These problems with schedule and budgets have resulted from a plethora of “required” design changes, often ascribed to low technology maturity, not meeting user or customer needs, not properly addressed human interfaces, etc. By decomposing system functionality using mitigating failure modes, one can develop a significant set of scenarios that far outstrip the Use Case design technique (Langford, 2008). This interrogation technique of system functionality, if not used, can lead to undiscovered functional issues, i.e., overlaps and underlap of functions. Such activities can be improved upon by differentiating between relevant actor inspired Cases. Relevant Cases typify the expected outputs and the expected failures that ensure. Systems can be designed with functions and their associated structures to provide the means for achieving complete system functionality (Langford, 2008).

Disassembling the system's functionality with respect to behavior, failure, etc., is important to the success of the system in meeting its goal, as well as to the completeness of the functional decomposition. Functional decomposition, when *improperly* conducted, can leave out relationships between elements, and therefore result in loss of value and purpose of smaller bits of system functionality. Various definitions and constructs should be imposed during the development process so all measures and means of accounting for the various attributes of the problem are explored and exhausted (Langford, 2006). By following a more rigorous approach to functional decomposition, the systems developer may generally concluded that no essential element was overlooked, and further that the problem was not overly specified, which could lead to unnecessary program costs.

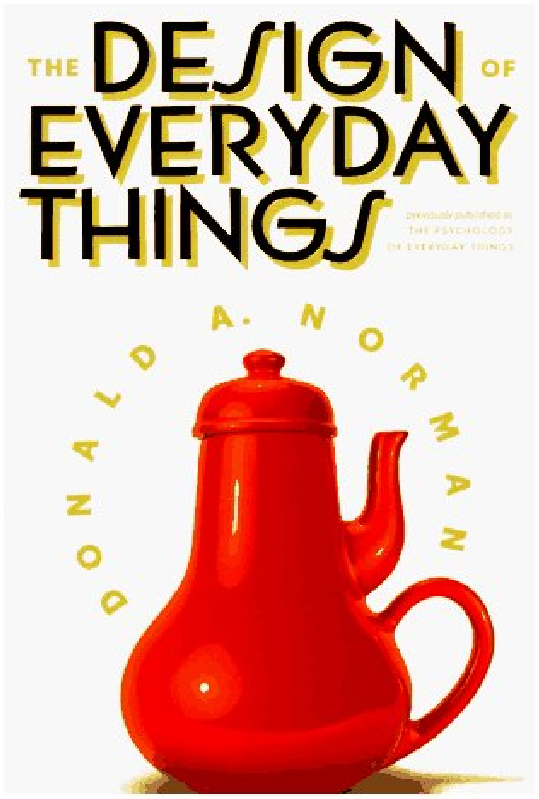

THE DESIGN OF EVERYDAY THINGS
previously published as THE PSYCHOLOGY OF EVERYDAY THINGS
DONALD A. NORMAN

Figure 7 Example of not mitigating failure mode (From Donald A. Norman, 2005)

The process of mitigating failures implies that a complete failure analysis is performed for every function, to better understand the ramifications and extent of the possible failure modes for and inherent in each function. The behavior(s) identified can be categorized into catastrophic or inconsequential, implying some effects (or modes) are below the threshold of interest, and therefore do not need to be carried forward in subsequent analyses.

Appropriate and mature processes exist to perform a failure mode and effects analysis (FMEA). Some of these processes have been developed into extensive models that focus on the functional aspects (Hawkins & Woollons, 1998) of the differences between calculated behavior and expected behavior. These facilitate the derivation of descriptive failure modes and consequences.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DIFFERENT APPROACHES TO FUNCTIONAL DECOMPOSITION

With the concern that functional decomposition may lead to poorly developed systems (Cantor, 2003), two questions arise: Are different alternatives to functional decomposition available, and what are the consequences of using them?

The desire to transition Systems Engineering from a document centered process model to a modeling approach is suggested as critical to improving Systems Engineering (Friedenthal, Burkhart). System engineers have used many different types of graphical means to modeling projects, including hand drawings on the traditional white board, but until the development of the Systems Modeling Language (SysML) there had not been a systems engineering standard modeling language, which had been recognized by the International Council on Systems Engineering (INCOSE, 2007).

A standard modeling language allows the system engineer to communicate system requirements and design specifications among other engineers. Modeling languages such as the Unified Modeling Language (UML) and Systems Modeling Language (SysML) provide such means of effective communication. For example, we know that while driving our vehicle we will come across a red eight sided sign. This sign instructs the driver to stop their vehicle, even without the word “stop” printed on the sign. The symbol has been known for the action to stop and take caution. The tools within the language help dissect the system in order to verify requirements, functionality, behavior, etc. which allows early identification of design issues and system effectiveness to meeting customer goals.

Viewing systems in a graphical form such as behavior diagrams, functional flow diagrams and N-Squared (N^2) Charts are effective means for the systems engineer to understand and present the functional and data flow characteristics of their systems (Long). These graphical forms (or representations) provide a valuable set of tools and methods for the systems engineer. They support and allow for the decomposition of the functional and data models into a natural hierarchical structure. The processes used to perform any decomposition are always subjective from one expert to another. For example, the expert's opinion, knowledge or even behaviors influence their decisions, which ultimately affect the outcome of the decomposition. The following tools and methods discussed below result in similar decompositions when deriving the intended goal of the functional relationship and the behavior of each system element. However, these tools and methods may benefit some decomposition compared to others. This is dependant on the intent of the decomposition, the structure of the elements, and the skill of the systems engineer. For example, UML was originally developed for software developers and electrical engineers, but systems engineers to help communicate the key fundamental threads that characterize the entire system also utilize it. The following sections examine and compare the basic functional decomposition method.

A. UNIFIED MODELING LANGUAGE (UML)

UML is a means to communicate with stakeholders the ideas concerning system development. UML provides a defined method of communication that consists of specific graphical format for systems and software engineers (Fowler & Scott, 2004). This graphical format seems to aid in understanding complexity and de-convolving the twists of interactions and relationships that mire some product developments. But is this simply that the population of UML users does not adequately understand how to use the tools (Grossman et al., 2004)? We find that more and more this graphical method of communication is becoming increasingly important as systems become more complex. For others, UML is used as a formal mechanism for requirements definition and design (Fowler & Scott, 2004).

Based on surveys of knowledgeable users of UML users, it appears the technology is poorly defined and lacks maturity (Grossman et al., 2004) in more than a few selected areas of application. The limitation of current UML (its inability to model continuous behavior and to deal with performance) hinders a wider application beyond software dominated parts (Volvo, 2002). There is additionally a lack of empirical evidence to support that UML leads to greater performance in system development (Grossman et al., 2004) or that it enforce the issues with usage, i.e., UML is considered complex therefore difficult to learn, inconsistent, and incomplete. UML is continually evolving, e.g., note the recent introduction of UML 2.0 that aspires to address some of the major issues. Table 1 lists the key perceived limitations of earlier versions of UML 1.x. UML 2.0 was created to offset some of these limitations and become more of a systems engineering asset than just a software asset, such as the inclusion of requirements constructs.

Perceived Limitations of UML V1.x
Continuous time behavior
Decision tree
Hierarchical modeling of behavior and structure
Input/output flow (i.e., data, mass, energy)
Parametric models and integration with other analysis models (i.e., performance, reliability, safety...)
Performance and physical characteristics (including probabilities)
Physical interfaces and connections
Problem definition and causal analysis
Requirements constructs
System, subsystem, and component representations
Terminology harmonization
Verification and validation models and constructs

Table 1 Perceived Systems Engineering Limitations of UML V1.x (From Friedenthal & Burkhart, 2007)

UML is not a tool, but rather more of a format that controls how engineers, stakeholders, customers, etc. communicate by means of diagrams such as various types of diagrams to depict Use Cases, classes, states, activities, composite structures, interaction overviews, sequences, collaborations, components, deployments, and timings.

Use Case diagrams offer a view of a system through a functional description that is enacted by events. That description includes the actors, who are internal or external triggers. The diagram activities, sequences, collaborations, composite structures, interactions, and statecharts represent the behaviors of the system. For example, an activity is represented by diagrams of control flow(s) between activities.

1. Activity Diagram

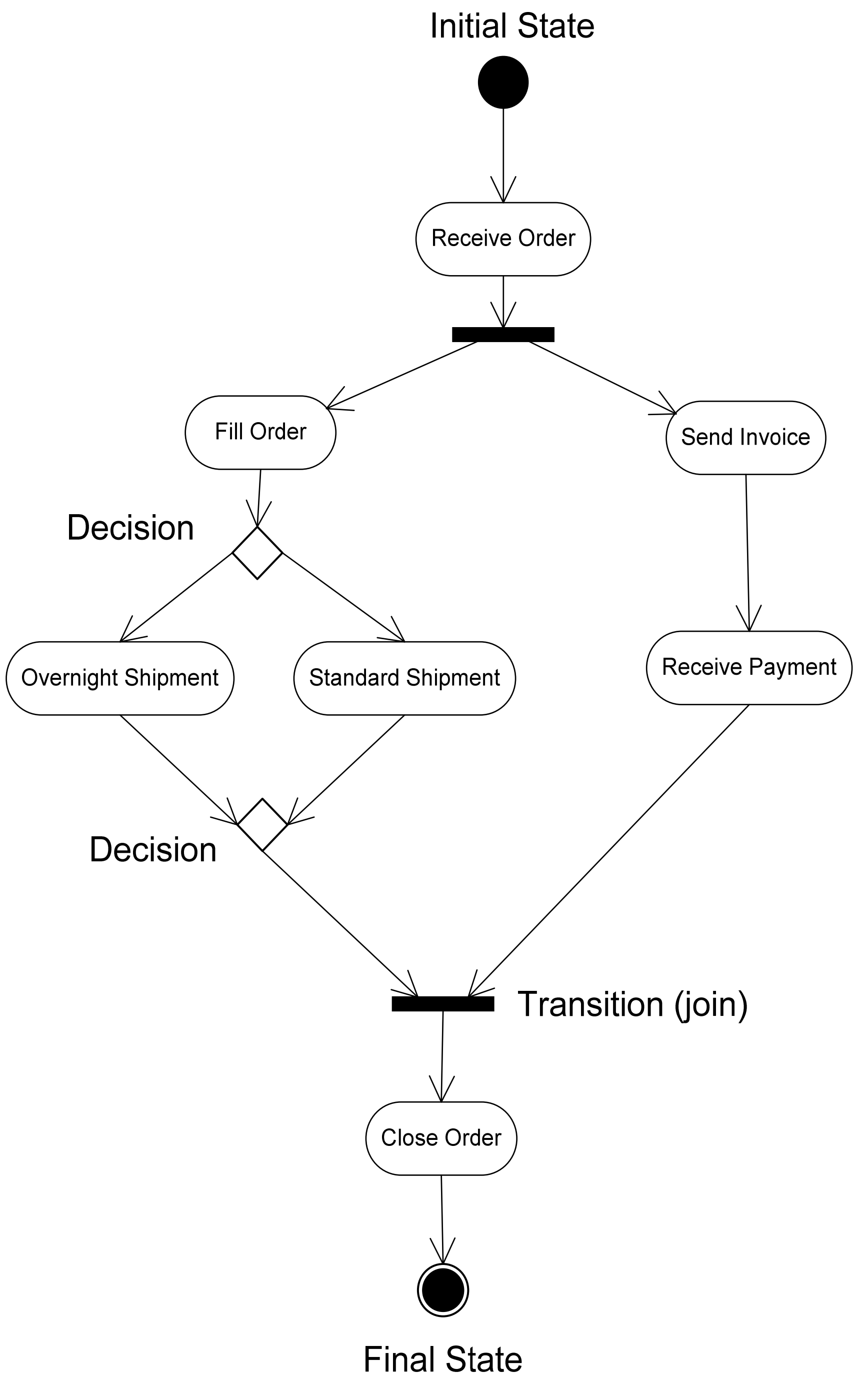


Figure 8 Typical Activity Diagram (showing order processing)

Activity diagrams lay out the procedural flow of the activities or actions of a high-level activity. Use Cases exist in most development projects, and concomitantly, activity diagrams should exists that portray activity diagrams that enlighten the Use Cases at a more detailed level. Activity diagrams do not need to be always combined with Use Cases. They can be used independently and beneficially for business level functions, an example of which is modeling of online procurements.

Activity diagrams model workflows that comprise a system. These diagrams are used along with other views, typically interactions, and states. Activity diagrams can also be used to analyze Use Cases. In this instance, actions are described in terms of their local interactions along with their associated timing. Activity diagrams give neither detail about objects and their behaviors nor objects and their collaborations (Fowler & Scott, 2004).

2. Sequence Diagram

Sequence diagrams describe the succession of interactions between objects. Horizontal arrows represent messages or logic between objects. Similar to the format used in N-Squared Chart messages begin in the top left and proceed down in a step formation. Sequence diagrams organize and display the requirements that would be expressed in Use Cases. Sequence diagrams document objects and their interactions. These diagrams are useful for system architects and designer, and further, prove useful as a means to create continuity between project teams and individuals in (Bell, 2003).

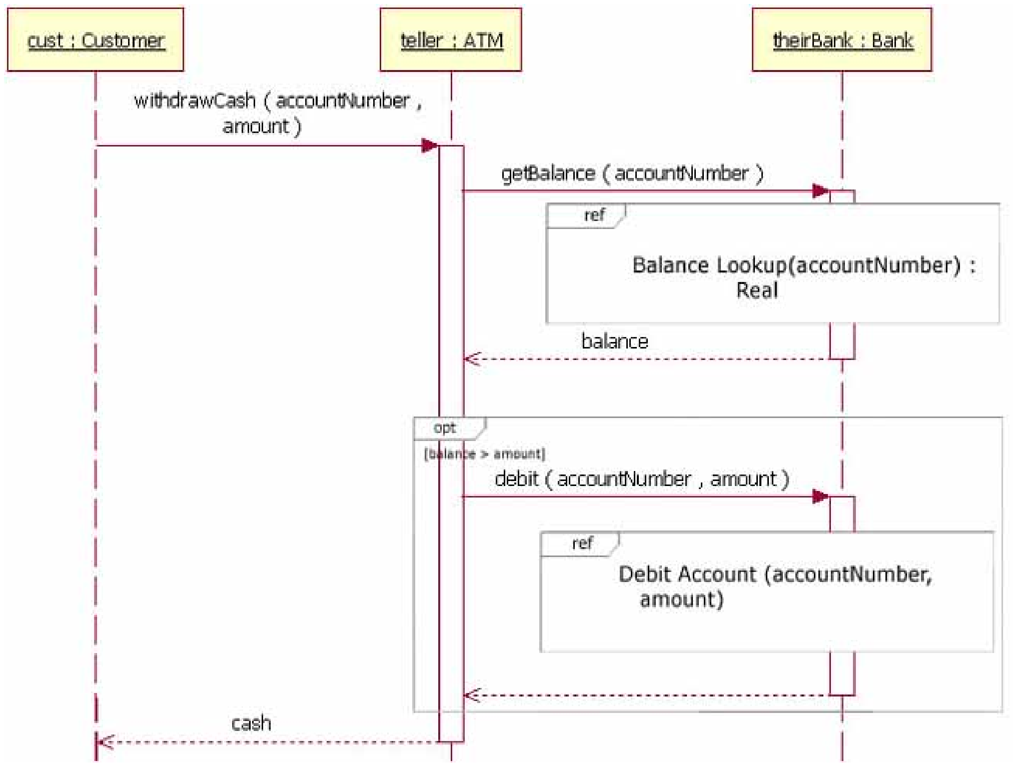

cust : Customer
teller : ATM
theirBank : Bank
withdrawCash (accountNumber , amount)
getBalance (accountNumber)
ref
Balance Lookup(accountNumber) : Real
balance
opt
[balance > amount]
debit (accountNumber , amount)
ref
Debit Account (accountNumber, amount)
cash

Figure 9 Typical Sequence Diagram (From Bell, 2003)

3. Class Diagram

Class diagrams provide the basic notation used in all other structure diagrams in UML. Class diagrams portray static structures. These diagrams focus on classifiers (Bell, 2004). Class diagrams are particularly useful when building business operational models or military organizational models (Langford, 2007).

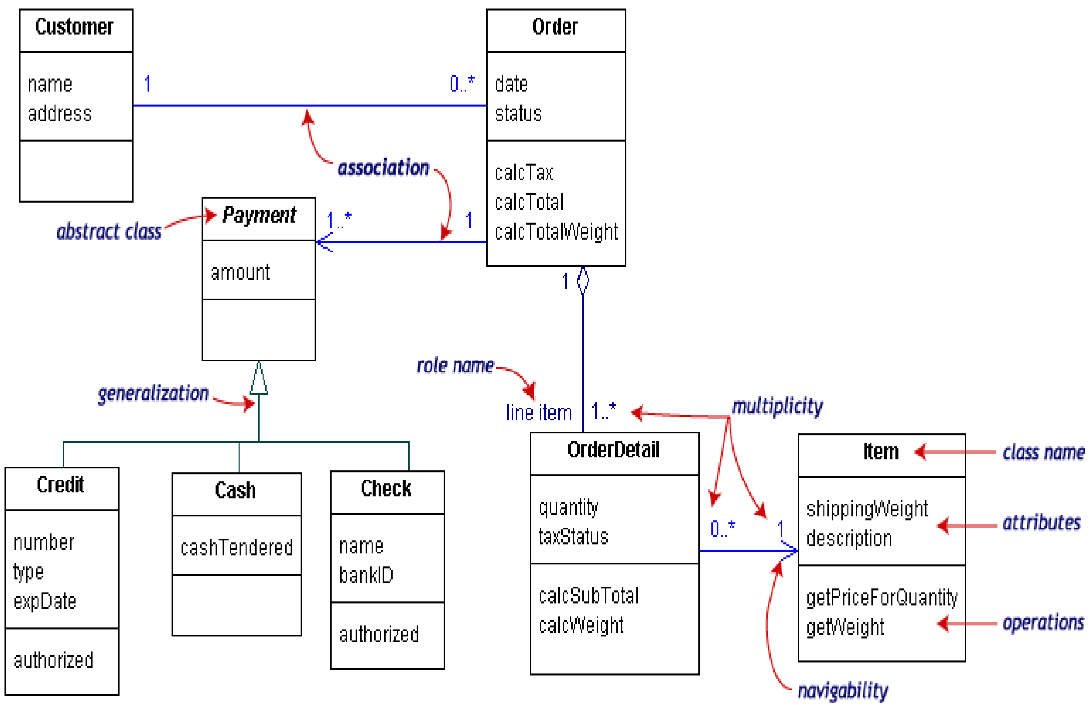


Figure 10 Typical Class Diagram (From TogetherSoft, Inc, 2001)

4. Collaboration Diagram

Collaboration diagrams focus on the behavior of objects external to, but interactive with, the system. These diagrams are similar to the sequence diagrams, as they include the same information, but are attributed to show the collaboration between asynchronous messages. Collaboration diagrams represent objects by icons and their message sharing as labeled arrows (Borysowich, 2007).

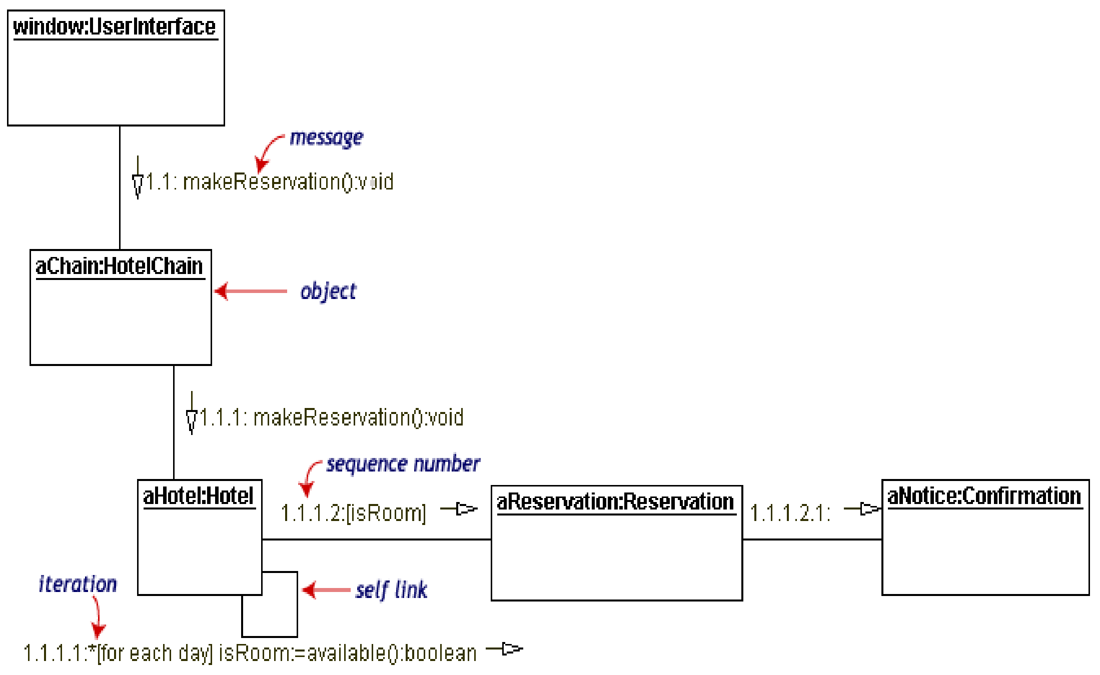


Figure 11 Typical Collaboration Diagram (From TogetherSoft, Inc, 2001)

5. Statechart Diagram

A statechart diagram describes the state of the object. The statechart diagram also shows how the objects are affected during actions. Typically, statecharts are used to describe behavior of classes, but can also be used to identify proper behavior of entities such as actors and Use Cases in the Use Case diagram. It is another way to identify behavior that in turn can be used to investigate the function.

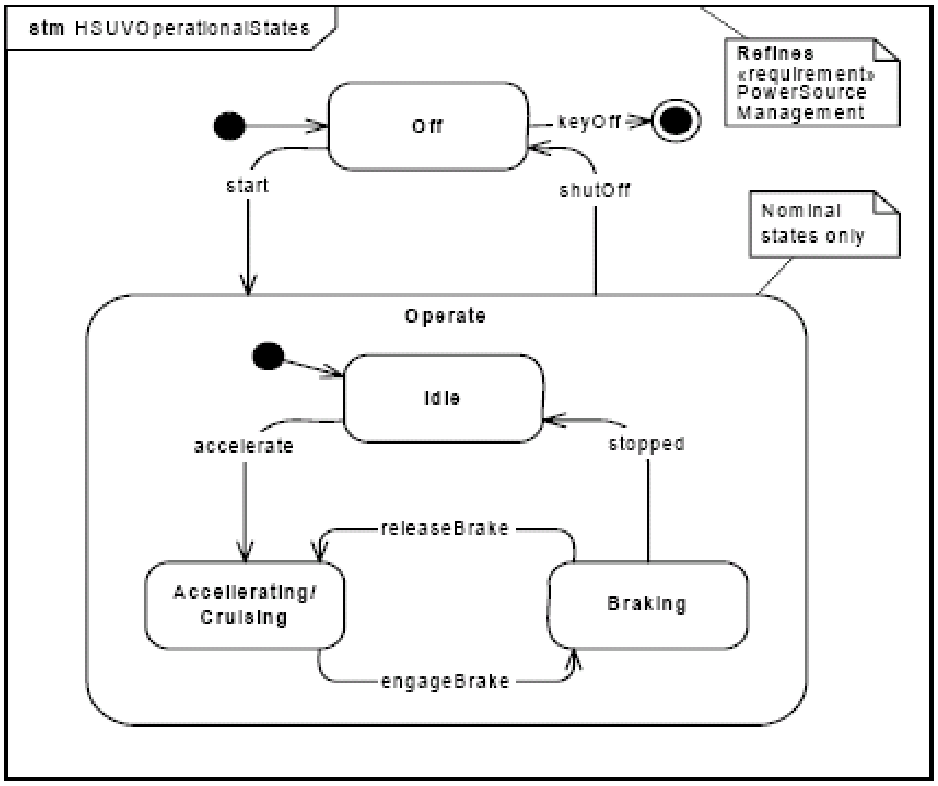


Figure 12 Typical Statechart Diagram (showing states of a hybrid SUV) (From OMG SySML, 2007)

A statechart diagram describes the transitions of an object from one state to another in response to events, and the actions that occur within a state (Friedenthal & Burkhart, 2007).

Controls and flows of control can be diagrammed as sequences of events or states in which object interact. Through the passing of messages statecharts depict objects in transition from one state to another. Changes are responsive to events and actions – all occurring within a state. Software systems are represented in UML by various diagrams: object, component, sequence, state, deployment, and timing, to mention a few. These diagrams cover the types of classes, operations, attributes (collectively referred to as class diagrams); the objects (object diagrams and structures); partitioning of classes among components (component diagrams); and how components are staged and executed (deployment and timing diagrams).

The Unified Modeling Language was developed as an industry standard for modeling software intensive systems. It allows the designer to visualize, specify, and document the artifacts of the software (Bell, 2003). UML employs Use Cases, which according to various critics of functional decomposition is the best means for capturing and documenting requirements. Use Cases define the interaction of the user or actor to the system itself sometimes via the initiator of the interaction. Use Cases are intended to help build a sound understanding of the system being designed by decomposing its behavior into components and interactions. Use Cases were expected to address the issues with UML regarding non-traceable requirements but did not since the system design requirements are usually only traced to Use Cases and not the design (Leffingwell & Widrig, 2002). UML 2.0 provides extension to the previous versions, i.e., UML 1.x. According to (Kobryn, 2004) UML 2.0 provides support for representing structural behavior in a hierarchical decomposition of the behavior allowing for diagrams that are understandable and contain complex behavior descriptions. Also UML 2.0 now facilitates viewing the same element in multiple perspectives. The result is an improvement in understanding the extractions (or models) of a system (Herzog & Pandikow, 2005), and (Kobryn, 2004).

UML methods provide diagrams, and visual graphics. When used within system design or methodology UML allows better understanding of a system under development (Bell, 2003). According to a survey conducted by (Grossman et al., 2004) most respondents concluded that UML provides benefits for understanding the communication aspects via graphical notation. However, there is yet no consensus as to whether UML makes any real difference in the performance of the development task. Perhaps a lack of adequate understanding of UML is responsible. On the other hand, perhaps the use of UML has result in no real difference (Grossman et al., 2004).

UML 2.0, as with UML 1.x, is based on two basic categories of diagrams - Diagrams of structures and diagrams of behavior. Structure diagrams show the static nature of the modeled system being. They include classes, components, and objects. Behavioral diagrams indicate dynamic behaviors between objects and diagrammed sequences. Behavioral diagrams present activities, Use Cases, and sequence diagrams.

6. Use Case Diagrams

Use Case diagrams provide descriptions of system functionality, including actors. Actors are external to the system and include domains such as the environment. The Use Cases represent usage(s) of the system, i.e., the subject, which correspond to the basic functionalities that the system and actors support (Friedenthal & Burkhart, 2007). The associations between the actors and the Use Case represent the communications between the actors and the processes and activities that will accomplish the functionality (OMG SysML, 2007). Bottom line: Use Cases can be used to capture the functional requirements of the system.

Functionality is represented in Use Case Diagrams in a top-down fashion. Use Case Diagrams represent behavior as relationships, which are rather different from Functional Flow Diagrams that represent behavior in a linear fashion, captured in a time-framed way. However, as with functional decomposition, the Use Case Diagram process begins by identifying the top level system functionality layout of the Use Case diagram. This top level is a description of what the system is to do, but not how it will do it. In addition, as with the functional decomposition process, further decomposition of system functionality is created by the Use Cases that were used during the top-level decomposition.

Use Cases are generally neither definitive nor complete when tying to understand failure analysis. The diagrams can quickly become confusing with overlapping functionality. In this case, sequence or flow diagrams provide a better way to represent failure modes and branching conditions. Sequence diagrams are used to address the exception behavior, the “what if” function. The sequence diagram is another tool to help understand the systems functionality. If the failure analysis is simple, i.e., pass or fail then only two different Use Case ovals are needed and at this point are only extensions of the original Use Case.

Figure 13 shows a typical Use Case Diagram. The typical Use Case diagram shows how to communicate the high level functions of the system and the system's scope (Bell, 2003). In Figure 13, the functions are portrayed graphically, such as; the commander views the target statistics and the topology of the target area.

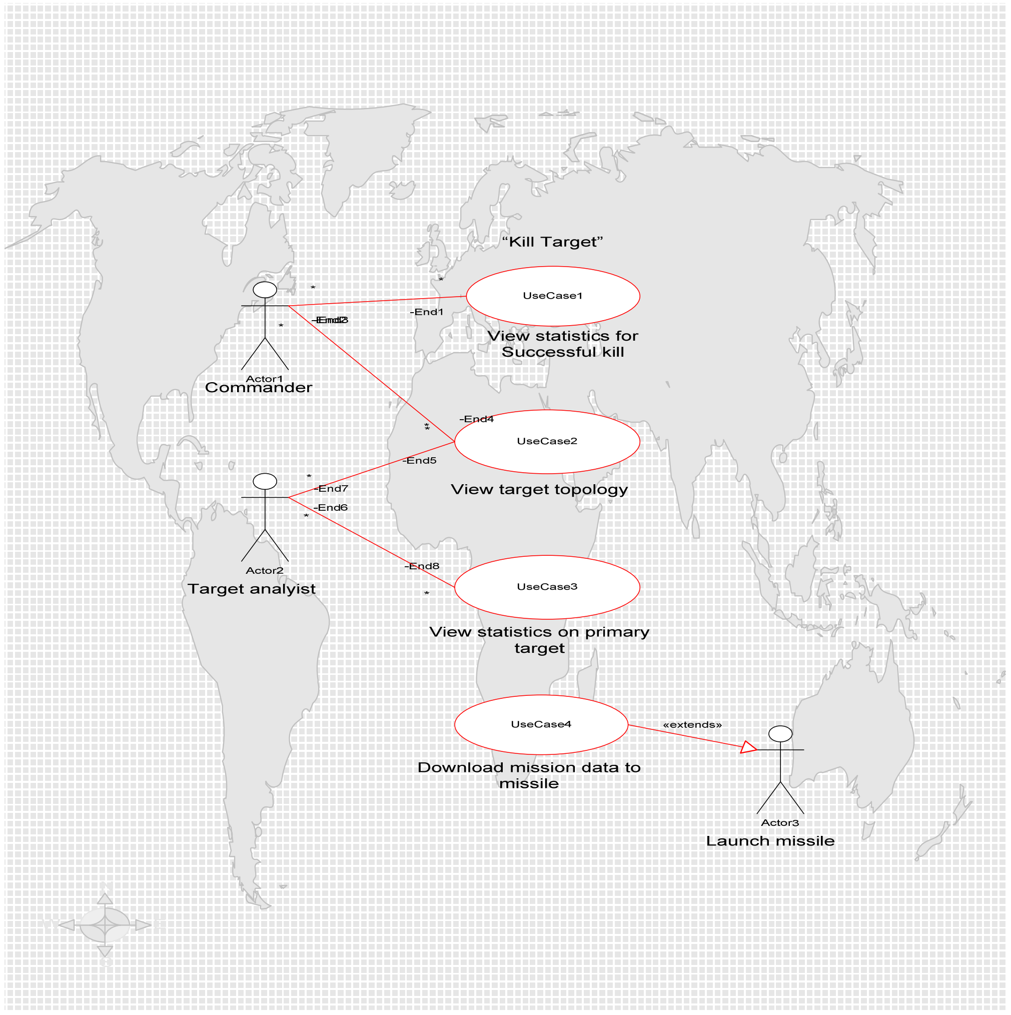


Figure 13 Use Case Diagram example

Clean visual description, depicted in Figure 13 allows the system engineer as well as the other stakeholders to see if more or less functionality is required in the system.

"Use Cases capture who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals. A complete set of Use Cases should specify all the different, key ways to use the system. Therefore, these Use Cases define the behaviors required of the system, and they serve to bound the scope of the system" (Malan & Bredemeyer, 2001). Use Cases can provide benefit to the product definition work, which in turn is relevant to defining the system architecture by a subset of Use Cases for each of the products (Malan & Bredemeyer, 2001).

According to (Malan & Bredemeyer, 2001), Use Cases do not offer a means to reflect commonality/variability across products in a product line or family. Such remarks and sentiment appear in other surveys and questioner's, such as (Grossman et al., 2004) and UML for Systems Engineering Request for Information (SE DSIG RFI 1) by the Object Management Group. In the article from (Malan & Bredemeyer, 2001) it is noted, "Many teams are not able to decide on the appropriate level of abstraction to which to take the Use Cases, and therefore experience an uncontrolled and time-consuming proliferation in their Use Cases."

UML focuses on a narrow view of the development rather than a broader, more systematic view. This narrow view causes the UML model of the system to stand alone in isolation from other domains, stakeholders, and systems. The limited involvement of stakeholders, customer and systems engineer gives a false design mentality that "we need only provide the functions the customer wants" (Gotterbarn, 2008).

UML does not have an explicit way of connecting the abstract description of processes, resources, and structures, in addition to details of behaviors and structures of objects (Kim et al., 2002).

B. SYSTEM MODELING LANGUAGE (SYSML):

The development of SysML is a joint initiative of OMG and the International Council on Systems Engineering (INCOSE, 2007). SysML was developed to assist the systems engineer with the specification, analysis, design, verification and validation of a broad range of complex systems which are not necessarily software based (Vanderperren & Dehaene, 2005), like UML. SysML is a modeling language used to represent systems and product architectures, as well as their behavior and functionality (Balmelli & IBM, 2008). Unlike UML, SysML does address the requirements traceability needed by systems engineers by linking the requirements to the design. Linkage is accomplished through requirements diagrams within the SysML environment. SysML is derived from UML, however, with changes geared to the systems engineer. The structural layout of SysML is shown in Figure 14.

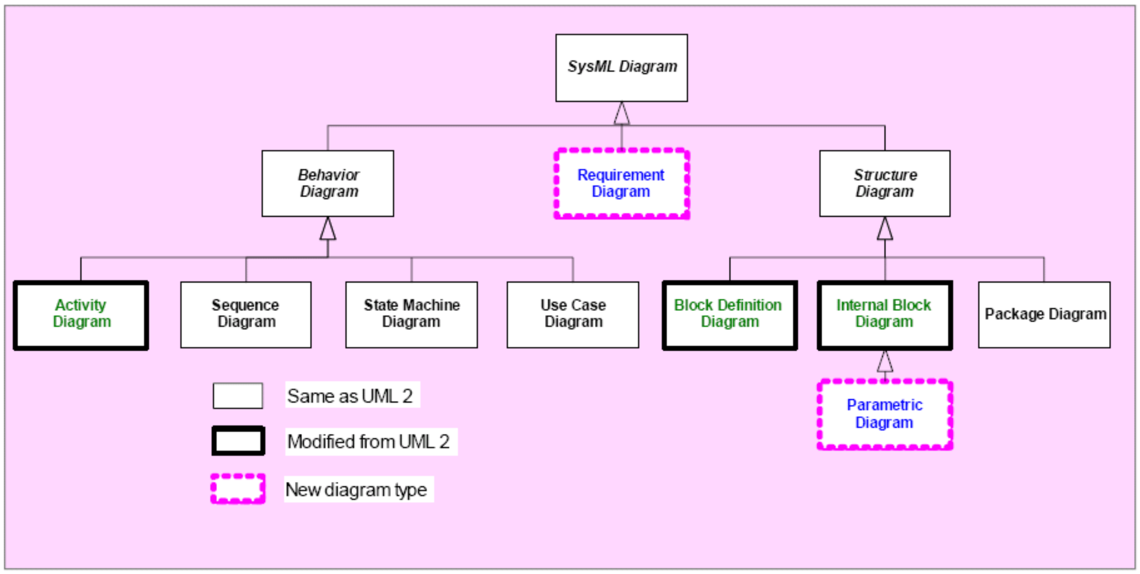


Figure 14 SySML Diagram Types (From INCOSE Handbook)

Within this diagram tree, there is the behavior diagram that addresses Use Cases. The Use Case diagrams provide descriptions of system requirements as previously mentioned in the UML section.

The attempts to extend UML by tools or modeling ultimately created difficulties when trying to integrate the different viewpoints and achieve traceability (Hause, Thom, & Moore, 2008). “SysML was developed to address this issue by providing a standard modeling language among all system engineers to analyze, specify, design, and verify complex systems, intended to enhance systems quality, improve the ability to exchange systems engineering information amongst tools, and help bridge the semantic gap between systems, software, and other engineering disciplines” (Hause, Thom, & Moore, 2008). SysML uses additional tools as compared with UML. The tools consist of models and techniques for analyzing, verifying models and decision trees.

Unlike SysML, UML is biased towards its use with software development, and so not easily adaptable to systems engineering (http://www.uml-forum.com). SysML reduces the size of UML and extends its semantics to requirements and parametric constraints (http://www.uml-forum.com) (see also Figure 14). SysML provides these model requirements and parametric constraints to support some of the most important systems engineering processes, requirements engineering and performance analysis.

1. The New Diagram; Requirement Diagram

The requirement diagram is added to SysML to benefit the systems engineer during defining requirements. The requirement diagram was brought into the modeling language to support the system engineer by listing requirements based on text and textual attributes (e.g., id, text statement, and criticality). This diagram also formulates requirements decomposition into its sub elements, has traceability between derived and formative requirements, allows inspection into elemental components that can satisfies various requirement(s) and provides a means to verification of requirements by test cases (OMG SysML, 2007).

The requirement diagram can only display requirements, packages, other classifiers, test cases, and rationale (OMG SysML, 2007). SysML represents the requirements as elements of a system model. Requirements are inherent to the system architecture. SysML represents textually drafted requirements (e.g., functional, performance, quality), and their relations (Balmelli & IBM, 2006). Generally, the requirements diagram is another means to requirements traceability. Requirement derivation and traceability can be performed by many methods including other diagrams within the UML or SysML format (OMG SysML, 2007). Functional decomposition and IDEF0 provide a means as well. Add in tools such as DOORS help the process as well. Figure 15 and Figure 16 display how SysML graphically represents the requirements in nodes and paths as outlined in the OMG SysML specification.

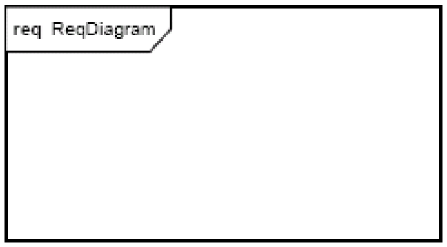
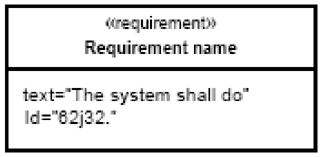
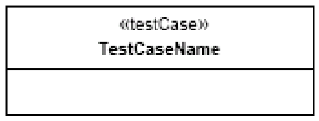
Node Name	Concrete Syntax	Abstract Syntax Reference
Requirement Diagram		SysML::Requirements:: Requirement, SysML:: ModelElements::Package
Requirement	 «requirement» Requirement name text="The system shall do" Id="62j32."	SysML::Requirements:: Requirement
TestCase		SysML::Requirements:: TestCase

Figure 15 Graphical Nodes Included in Requirements Diagram (From OMG SysML, 2007)

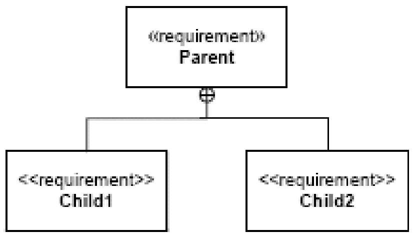
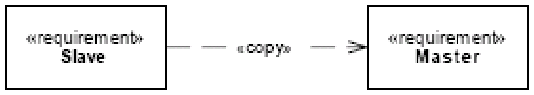
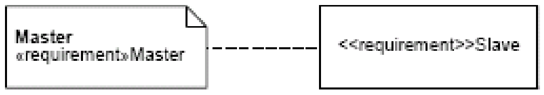
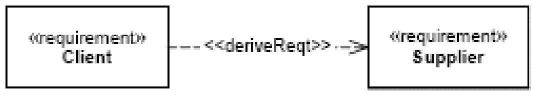
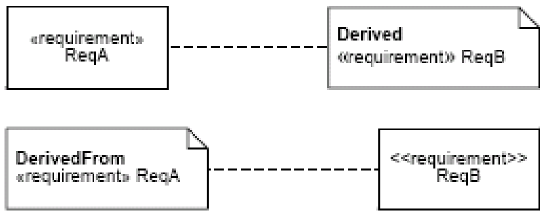
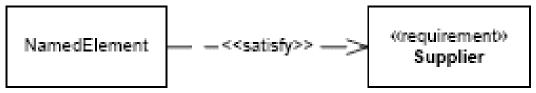
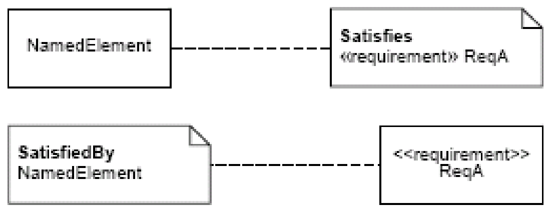
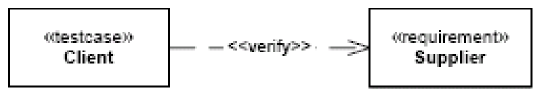
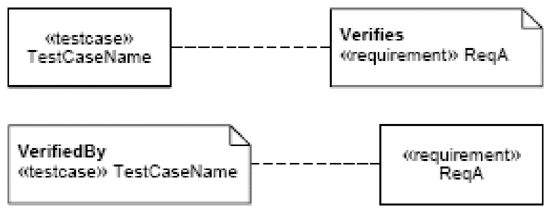
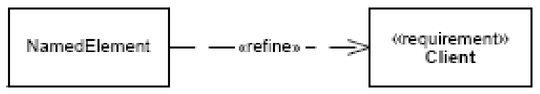
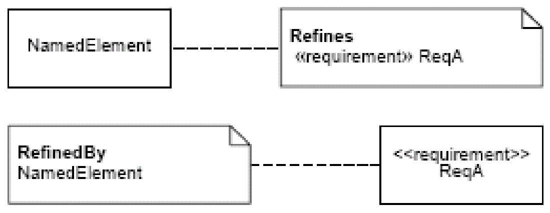
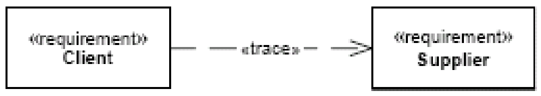
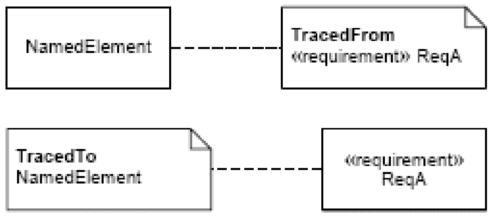
Path Type	Concrete Syntax	Abstract Syntax Reference
Requirement containment relationship	 «requirement» Parent <<requirement>> Child1 <<requirement>> Child2	UML4SysML:: NestedClassifier
CopyDependency		SysML::Requirements:: Copy
MasterCallout		SysML::Requirements:: Copy
Derive Dependency		SysML::Requirements:: DeriveReqt
DeriveCallout	 «requirement» ReqA Derived «requirement» ReqB DerivedFrom «requirement» ReqA <<requirement>> ReqB	SysML::Requirements:: DeriveReqt
Satisfy Dependency		SysML::Requirements:: Satisfy

Figure 16 Graphical Paths Included in Requirements Diagrams (From OMG SysML, 2007)

New requirements are produced, and subsequently decomposed, during requirements analysis. These are notionally associated with the formative requirements, which were initially conceived or handed-down (Balmelli & IBM, 2006) (see graphical notation above).

Path Type	Concrete Syntax	Abstract Syntax Reference
SatisfyCallout	 NamedElement Satisfies «requirement» ReqA SatisfiedBy NamedElement <<requirement>> ReqA	SysML::Requirements:: Satisfy
Verify Dependency	 «testcase» Client <<verify>> «requirement» Supplier	SysML::Requirements:: Verify
VerifyCallout	 «testcase» TestCaseName Verifies «requirement» ReqA VerifiedBy «testcase» TestCaseName «requirement» ReqA	SysML::Requirements:: Verify
Refine Dependency		UML4SysML::Refine
RefineCallout	 NamedElement Refines «requirement» ReqA RefinedBy NamedElement <<requirement>> ReqA	UML4SysML::Refine

Graphical Paths included in Requirements Diagrams (From OMG SysML, 2007), *continued*

Path Type	Concrete Syntax	Abstract Syntax Reference
Trace Dependency		UML4SysML::Trace
TraceCallout	 NamedElement TracedFrom «requirement» ReqA TracedTo NamedElement «requirement» ReqA	UML4SysML::Trace

Graphical Paths included in Requirements Diagrams (From OMG SysML, 2007), *continued*

2. Differences between UML and SysML

What are the differences between these two modeling languages? The systems engineer that could influence decisions when utilizing one of these languages should know what advantages are best. SysML is a domain specific modeling language and UML is construed as a general purpose modeling language. UML has evolved to UML 2.0 on which SysML was built to allow the reuse of maturing notation and semantics (http://www.uml-forum.com). Both UML 2.0 and SysML provide the system engineer with means to derive system functionality.

What advantages does SysML offer the systems engineer? The following is a list provided by the UML Forum at http://www.uml-forum.com.

SysML expresses systems engineering semantics (interpretations of notations) better than UML. It reduces UML's software bias and adds two new diagram types for requirements management and performance analysis: requirement diagrams and parametric diagrams, respectively.

SysML is smaller and easier to learn than UML. Since SysML removes many software-centric and gratuitous constructs, the overall language is smaller as measured in diagram types (9 vs. 13) and total constructs.

SysML allocation tables support various kinds of allocations (e.g., requirement allocation, functional allocation, structural allocation) thereby facilitating automated verification and validation (V&V) and gap analysis.

SysML model management constructs support the specification of models, views, and viewpoints and are architecturally aligned with IEEE-Std-1471-2000 (IEEE Recommended Practice for Architectural Description of Software-Intensive Systems).

The following table, also provided by the UML Forum at http://www.uml-forum.com, compares SysML diagrams with their UML counterparts. The capability of each modeling language is listed directly in the column applicable to it. Where N/A is indicated the particular language does not support the category described.

SysML Diagram	Purpose	UML Diagram Analog
Activity diagram	Show system behavior as control and data flows. Useful for functional analysis. Compare Extended Functional Flow Block diagrams (EFFBDs), already commonly used among systems engineers.	Activity diagram
Block Definition diagram	Show system structure as components along with their properties, operations and relationships. Useful for system analysis and design.	Class diagram
Internal Block diagram	Show the internal structures of components, including their parts and connectors. Useful for system analysis and design.	Composite Structure diagram
Package diagram	Show how a model is organized into packages, views and viewpoints. Useful for model management.	Package diagram
Parametric diagram	Show parametric constraints between structural elements. Useful for performance and quantitative analysis.	N/A
Requirement diagram	Show system requirements and their relationships with other elements. Useful for requirements engineering.	N/A
Sequence diagram	Show system behavior as interactions between system components. Useful	Sequence diagram

	for system analysis and design.	
State Machine diagram	Show system behavior as sequences of states that a component or interaction experience in response to events. Useful for system design and simulation/code generation.	State Machine diagram
Use Case diagram	Show system functional requirements as transactions that are meaningful to system users. Useful for specifying functional requirements. (Note potential overlap with Requirement diagrams.)	Use Case diagram
Allocation tables* *dynamically derived tables, not really a diagram type	Show various kinds of allocations (e.g., requirement allocation, functional allocation, structural allocation). Useful for facilitating automated verification and validation (V&V) and gap analysis.	N/A
N/A		Component diagram
N/A		Communication diagram
N/A		Deployment diagram
N/A		Interaction overview diagram
N/A		Object diagram
N/A		Timing diagram

Table 2 Comparison of UML and SysML (http://www.uml-forum.com).

Use Cases, as shown in the above table, are also implemented using SysML and have not been modified. In this respect, the UML or SysML formats are the same.

An alternative to beginning with functional decomposition is to decompose a system into objects. This is the approach used in applying SysML. First is to start with an object decomposition that then leads to another way of identifying functions using the principles of UML (Osmundson, 2007).

C. INTEGRATION DEFINITION FOR FUNCTION MODELING (IDEF0):

Integration Definition for Function Modeling (IDEF0) is a modeling method that defines process and data flows. "IDEF0 is useful in conducting systems analysis and design at all levels, for system composed of people, machines, materials, computers and information of all varieties" (IEEE Std 1320.1-1998). The IDEF0 graphical model is laid out in a hierarchical arrangement of boxes or diagrams. Each box represents a prime function and the arrows into and out of represents the data that interacts with the particular function. The format of the IDEF0 function box is very similar to the description of "function" as depicted by (Blanchard & Fabrycky, 1998). The IDEF0 defines function as "a set of activities that takes certain inputs and, by means of some mechanism, and subject to certain controls, transforms the inputs to outputs" (Kim et al., 2002).

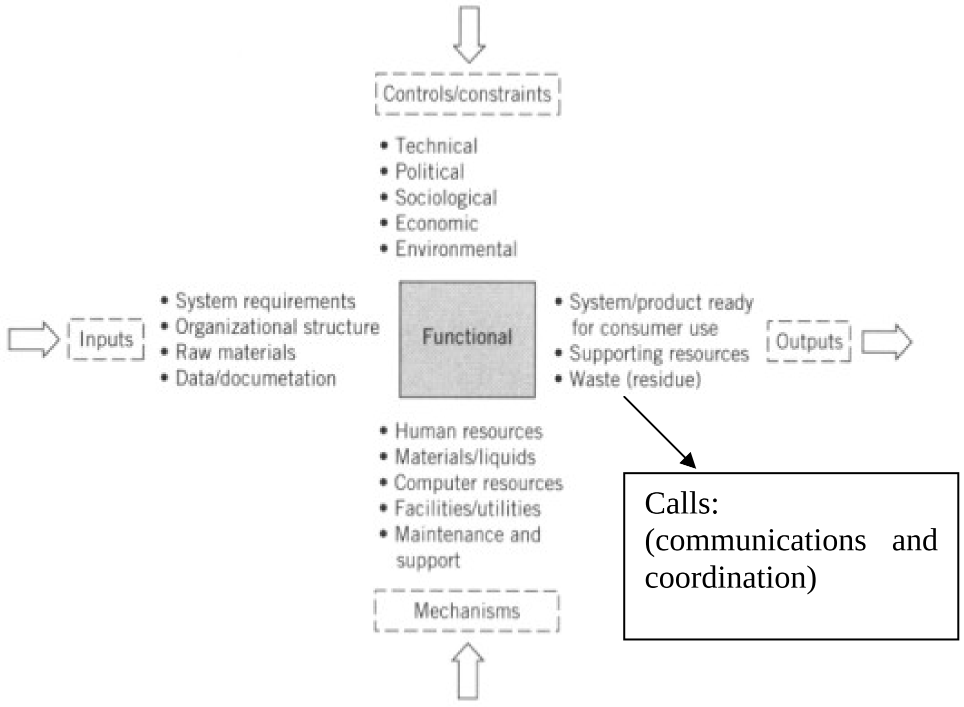


Figure 17 Inputs and Outputs into the functional block (From Blanchard & Fabrycky, 1998)

In Figure 17 the functional box is an activity, process, or transformation. This function is identified by a verb or verb phrase that describes what the function must accomplish. The inputs into the function box are transformed by the function itself into an output. The controls/constraints from the top are transformed by the function, creating the output. The mechanisms enable the sharing of detail between other models or with a model (Osmundson, 2007).

IDEF0 is structured to make the understanding of a process easier since it organizes the structures process in the same manner as the user may think about the process, i.e., sequences (FIPS, 1993). IDEF0 is very similar to functional decomposition as it is based on breaking down a process into sub-processes to make it easier to handle complex systems. Functional decomposition is usually the beginning of the IDEF process by which it gives a systematic way of dealing with levels of complexity (Kim et al., 2002). This breakdown also follows the similar functional flow diagrams (FFD) as the IDEF0 is arranged in a descending sequence with left to right flow, making it possible to easily follow how each sub-process interacts with another. IDEF0 has a similar methodology to that of functional decomposition, as it is hierarchical in nature. The model is laid out in it's as-is condition in a top down fashion, but allowing for an analysis to be conducted in a bottom up approach.

The hierarchical layout of system functionality gives the designer the ability to view the system from a "current" viewpoint. The IDEF0 diagram allows for a bottom up analysis of the system functionality. Much like the cohesion process mentioned earlier, similar or closely related activities are grouped together resulting in a more appropriate hierarchal structure depicting the functional architecture. This process, as with functional decomposition and Use Cases, is recursive until the desired level of detail in the hierarchy is developed.

The IDEF0 method appears to be one of the best for a side-by-side comparison with the functional decomposition method as they work well with each other (Kim et al., 2002). The IDEF0 represents the mechanism (usually the system components to which the function is allocated) which performs the function. Figure 18 is an example of an IDEF0 diagram.

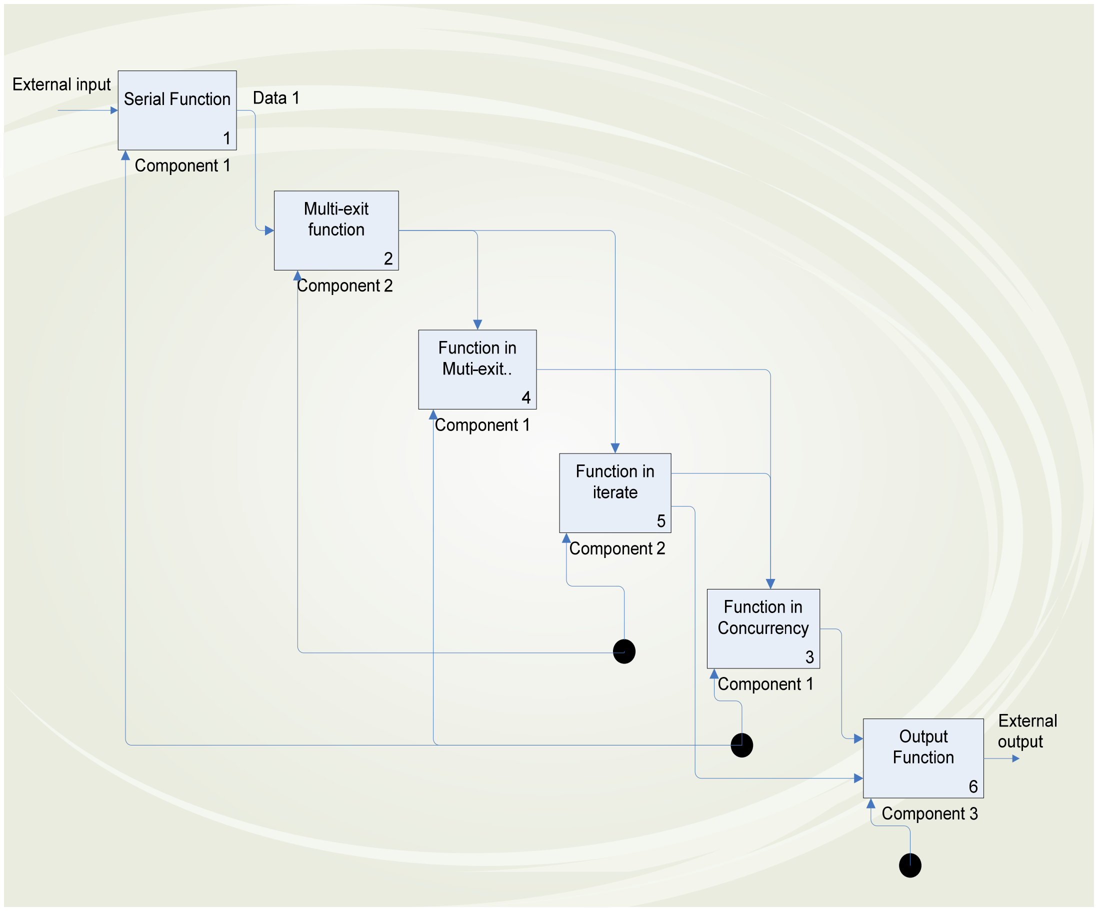


Figure 18 IDEF0 sample diagram

As displayed in Figure 18, data into each function icon occurs or enters on the left side. Control inputs enter the function icon at the top. Outputs of each function icon exit on the right side. The mechanism or system components, as mentioned earlier, are allocated to the function and enter each function icon from the bottom.

“IDEF0 is used to analyze and assist the modeler in identifying the functionality that is to be performed. Analysis as to how the system performs these functions is conducted which leads to identification of what the system does right and what the system does wrong. Thus, IDEF0 models are often created as one of the first tasks of a system development effort” (Knowledge Based Systems, 2008). Similar to other means of identifying system functions, i.e., the functional decomposition process or Use Cases, IDEF0 presents the functional depiction in a graphical form. A graphical form seems easier to understand than merely words in a document because of the relationship it shows between functions.

IDEF is not alone and in fact, IDEF is not just for function modeling. IDEF0 is a family of methods ranging from IDEF0 to IDEF14, each providing a perspective into the domain of study.

IDEF0	Function Modeling
IDEF1	Information Modeling
IDEF1X	Data Modeling
IDEF2	Simulation Model Design
IDEF3	Process Description Capture
IDEF4	Object-Oriented Design
IDEF5	Ontology Description Capture
IDEF6	Design Rationale Capture
IDEF8	User Interface Modeling
IDEF9	Scenario-Driven IS Design
IDEF10	Implementation Architecture Modeling
IDEF11	Information Artifact Modeling
IDEF12	Organization Modeling
IDEF13	Three Schema Mapping Design
IDEF14	Network Design

Table 3 Suite of IDEF0 Methods (current and in development)

The IDEF0 method models the system functions and relationships with other functions. However, IDEF0 notations are only conceptual models and therefore not effective for the generation of implementation schemata (Kim et al., 2002). This is not to mean that IDEF0 is not very useful for system development. In fact, IDEF0 models provide value to understanding system functionality and functional requirements, when combined or mapped with object oriented models as with the generation of computer executable systems outline in (Kim et al., 2002). Developing an IDEF0 diagram and following functional understanding with Use Cases used in UML and SysML one can address key constituents that IDEF0 alone cannot. The consideration of using another

IDEF0 method such as IDEF3 for behavioral aspects still has its limitations as the description of interactions between organizations are limited and “the lack of any clear distinction between material flow and information flow lead to semantic constraints that limit the use of all IDEF models” (Kim et al., 2002).

Note, the IDEF0 diagram is similar to the N-Squared chart mentioned before. The input into the function box of the IDEF0 diagram allows the specification of control, but does not have the ability to characterize the control in terms of constructs such as triggering (Long, 2008). Other types of diagrams found in the UML and SysML formats such as behavior diagrams allow this ability. IDEF0 diagrams also allow the explicit representation of functional allocation (Long, 2008) i.e., the particular system component that performs the function.

“A problem that has been described with IDEF0 models (Knowledge Based Systems, 2008) is that IDEF0 can be confused with describing the sequence of events within the system activity. Getting around this issue is not impossible. The systems engineer may layout the system activities in a typical left to right sequence when decomposing.” These sentiments are typical of the normal way one might think when problem solving. It is etched into our thinking, especially when reading or writing, that we must work the flow from left to right. The risk with the IDEF0 model not being structured in an activity-sequencing format is that other developers of a team may mistake this and attempt to add interpretation (Knowledge Based Systems, 2008)

“IDEF0 has been very effective when detailing the system activities for function modeling” (Knowledge Based Systems, 2008), (Kim, et al., 2002). The IDEF0 process continues with further decomposing of these activities into greater detail until the system is described in enough detail to understand that it meets the intended need or goal. However, one of the observed problems with IDEF0 models is “that they often are so concise that they are understandable only if the reader is a domain expert or has participated in the model development” (Knowledge Based Systems, 2008).

IDEF0 models can be suitably constructed to describe system complexity. The first such description was for manufacturing. IDEF0 models may posit many perspectives (Kim et al., 2002). The modeling of functional behaviors in the system are not handled well in IDEF format (Kim et al., 2002) so it makes sense to utilize other means of filling in the gaps as UML does allowing system structures, components, and computer program packages to be designed, developed, and reused (Kim, et al., 2002). It appears logical that multiple viewpoints, such as IDEF0, Use Cases (both in UML and SysML), Behavioral diagrams (both in UML and SysML), Requirements Diagrams (SysML) and the process of decomposing system functionality with functional decomposition and the key factors contributing to completer decompositions combine well to effectively derive good system requirements. This works well as an interrogation technique covering many views of the system.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CONCLUSION

In summary, it is apparent from research that the process of coming up with functional requirements via the functional decomposition method or using other approaches such as UML, SysML and IDEF0 is feasible but varies dependant on the experience of the system engineer and type of system, i.e., software or hardware.

From this research, it appears that the process of functional decomposition, while focusing on the key factors that lead to good decompositions, is the first stepping stone to deriving functional requirements. What graphical method to use to address behavior is considered subjective, as well as how the system accomplishes its task. The examination into the object and process-oriented views (object being UML and SysML) shows that combining approaches works well and depending on the system under development , i.e., software or hardware, different approaches and emphasis on one or the other is needed. For example, software development seems to favor mostly UML and functional decomposition alone. This multi view analysis of system functionality is a good way to view the functional requirements in different ways, helping to ensure proper functionality.

As shown in (Kim et al., 2002) and in (Doyle & Pennotti, 2005), object oriented models and process oriented models work well together. In (Doyle & Pennotti, 2005) their experiment showed that using UML/SySML Use Case Diagrams made it difficult to grasp the initial view of how a system would accomplish its task. The Use Case Diagrams show what the system might be expected to do but fails at easing the understanding on how. The approach that worked well in their experiment was the combination of functional decomposition and IDEF0. Remember the functional decomposition was the first step in the IDEF0 process so as mentioned earlier IDEF0 and functional decomposition go hand in hand. This format leads the experiment to better communication from a project management viewpoint.

In the article by (Kim et al., 2002), the combined use of UML and IDEF0 provided multiple views of requirements. Redundancy of the models is less desired since the result would likely lead to additional development cost. However, from their experience and others (Langford, 2008) it is believed that the increased quality of the modeling out weighs the additional cost. In fact, the hierarchical functional decomposition process (in process-orientated models) has shown in both these papers that it works much better than a single approach.

The practice of hierarchical functional decomposition is most often easier and takes less time than trying to force the hierarchy structure into Use Cases or considered as a hack functional decomposition (Langford, 2008). "Hack" functional decompositions are characterized by inattention to defined terms and mixing functions, with processes, with performance and with quality. The result is a mishmash of ill-defined states, modules, and entities which results in poorly defined inputs and outputs and dependencies that are either undiscovered or intentionally disregarded. Hierarchy and modularity permit reuse and replacement and therefore adaptability, standardization, scalability, and understanding. Decomposition can be valuable to present a simple view of the parts of a function (assuming that the top-level function can be broken into parts). If the structure of sub functions permits modularization, then the decomposition can be scaled and generally used more adaptively through interfaces with other modules. Simplicity is a way of gaining understanding about the nature of the top-level function, its relationships, its components, and how it can be used. Fundamentally, functional decomposition is the basis for all science, all structure, all thought. If done properly, logic is defined.

A. FUTURE WORK

In consideration of future work, a different approach to performing functional decomposition is discussed. The combination of multiple approaches to give different views of system functionality was shown to be best, but the functional decomposition must be good in order to be complete. It was mentioned that coupling and cohesion play an important role in good decompositions. The question is what else?

Systems Engineering Value Equation with Risk (SEVER) (Langford, 2008) is another way of performing an interrogation of the functional decomposition with good coupling and cohesion. It is a way to determine the gap that can exist between an existing product and a desired product. In terms of cohesion, it is important to position the functions within the system hierarchy so if changes are required, i.e., function is re-structured, it will not significantly affect the totality of design. The coupling and cohesion interrogation is conducted first, and then in the more detailed levels in the functional hierarchy, SEVER can be used to improve the functional decomposition. Now consider that a problem is discovered that exposes lifecycle inconsistencies with the product's requirements or proposed architecture. In effect, there are conflicts between the design objectives and the impacts of lifecycle considerations. The issue could be with part of the problem that will require an upgrade but at a high impact, i.e., redesign, then a simple way is to design this part of the product to be replaced earlier. The quality may be reduced driving the need to replace sooner but this will reduce the cost of the function. This situation makes sense for products are impacted by rapid technology changes when an east upgrade is required.

SEVER provides a way to analyze the value of each function. The fundamental equation from value engineering, $V_F = \frac{P}{I}$, where the value of the function is defined as the performance divided by the investment. For example, looking at three functions $F_{1.1}$, $F_{1.2}$ and $F_{1.3}$, let's say that $F_{1.1}$ has high value to the customer, but $F_{1.3}$ has high risk, and therefore not much value or performance. The customer indicates interest in $F_{1.3}$ but not at the high cost and risk. Combining $F_{1.2}$ and $F_{1.3}$ in a way that cost could be shared and performance improved is leveling the load of both functions. This is a means of using the value portion of SEVER to investigate the load leveling portion in the functional domain.

This moving of functions into the functional domain is a way of quantifying the coupling and cohesion within the functional decomposition. In some circumstances, coupling and cohesion will not provide any improvement in the decomposition. Value is another way of viewing the functional decomposition. It is a powerful tool that can be used in the first order to view the predominate value of the system. For example the function $F_{1.1}$ is most valuable to the system and function $F_{1.3}$ could essentially be eliminated, but the customer is willing to pay for $F_{1.3}$ functionality because it has some value to them but not worth as much as the cost of fielding $F_{1.3}$ alone. So at the first level of the functional decomposition one could design the interface so it fits with $F_{1.2}$ and $F_{1.3}$ but as a modular upgrade to the system. If an improvement in technology or a reduction in cost occurs then investment of the function $F_{1.3}$ goes down and value goes up.

There appears to be no easy way to address value in UML or SySML, but it can be done with IDEF0. IDEF0 has inputs, outputs, mechanisms and controls, as shown in Figure 17. SEVER can be incorporated as in input into the IDEF0 function block as a worth equation, where $R_F = \frac{P * Q * LOC}{I}$ (see Figure 19) where R is the risk, F is the function, P is the performance, Q is the quality, LOC is the likelihood of occurrence, and I is the investment (Langford, 2008). Worth is defined as Value (measured in units of performance) multiplied by Quality (measured in units of I). Worth multiplied by LOC is equal to R, risk.

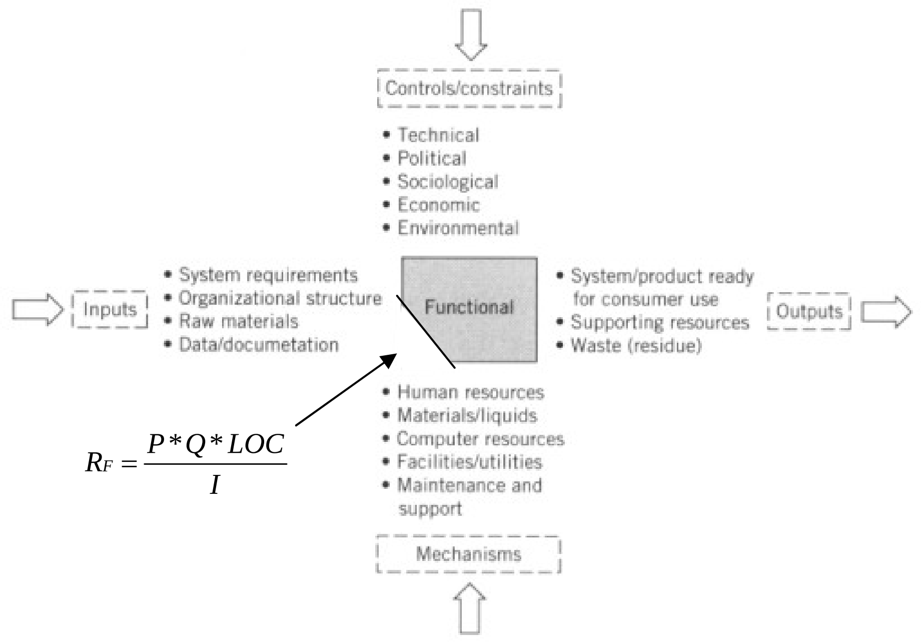


Figure 19 IDEF0 Modified with Risk Attributes (From Langford, 2008)

Looking at each function, and determining the risk of each function, defines the loss that may occur by using the initial functional decomposition. Moving around the value and risk of a function will increase the performance and reduce the cost thereby resulting in an improvement to the architecture/design. Whether IDEF0 structure is used or just a straight functional decomposition these tools, value of function and risk of function can be used further to refine the architecture /design. Worth presents another view of the impacts of functional decomposition. It is defined as $W = P_{/c/t} * \frac{P}{T} * \frac{Q}{P}$ where P is the performance, T is the total time in hours, Q is loss function (minimum loss - $L_n(X)$ and $L_n(X) = k(x-m)^n$ (standard loss function)) (Langford, 2008). The function $F_{1.3}$, which is of high risk, is going to cost a lot of money, which we knew when we looked at the value of the function. At the next level of detail, the cost is related to the number of people working the function, which could be for discussion, lines of code in software development. A further refinement using SEVER can be conducted on the functional decomposition to isolate problems that should be fixed by means of load leveling the functions. For example a module of code that could de defined as high risk could be moved from $F_{1.3}$ to $F_{1.2}$. Function $F_{1.2}$ may have a different skill set that would reduce the risk of developing the high risk module and reduce the cost of rework when kept in $F_{1.3}$. This is an improvement of the functional decomposition from a management point of view. Looking at the management of functional decomposition first from a top level of negotiation, a second level for design, and a third level for life cycle issues, is a way to look at the entire business process to see if the right thing will be done with the product.

LIST OF REFERENCES

Bahill, T.A., & Dean, F.F. (2007). *What is systems engineering?*A Consensus of Senior Systems Engineers. Retrieved December 5, 2007, from http://www.sie.arizona.edu/sysengr/whatis/whatis.html.

Balmelli, L., & IBM. (2006). *An overview of the systems modeling language for product and systems development -- Part 1: Requirements, use-case, and test-case modeling.* Retrieved January 12, 2008, from http://www.ibm.com/developerworks/rational/library/aug06/balmelli/.

Balmelli, L.D., Brown, M., Cantor, M., & Mott, M. (2006). *Model-driven systems development.*

Bell, J. (2004, July). *The role of functional decomposition*, Doc. ref. SD/TR/FR/10.

Bell, D. (2003, June). UML basics: *An introduction to the unified modeling language*, Staff, IBM.

Blanchard, B., & Fabrycky, W. (1998). *Systems engineering and analysis*, Third Edition. Prentice Hall.

Borysowich, C. (2007). (Chief Technology Tactician) *Overview of functional decomposition* Retrieved February 20, 2007, from http://blogs.ittoolbox.com/eai/implementation/archives/overview-of-functional-decomposition-14609.

British Standards, BS5760 (1991).

Cantor, M. (2003). *Thoughts on functional decomposition*, The Rational Edge.

Cantor, M., & Roose, G. (2005, December). *Hardware/software codevelopment using a model-driven systems development (MDSD) approach.*

Cantor, M., & Roose, G. (2006). *The six principles of systems engineering, IBM's rational rules developed over 10-Year period.*

Chittaro, L., & Kumar, A.N. (1998). *Reasoning about function and its applications to engineering*. Artificial Intelligence in Engineering 12 (4), 331.

Chandrasekaran, B., & Josephson, J.R. (1997, July). *Representing function as effect*, Proceedings of the Fifth International Workshop on Advances in Functional Modeling of Complex Technical Systems, Paris France.

Coulston, C., & Ford, R.M. (2004, October). *Teaching functional decomposition for the design of electrical and computer systems,* 34th ASEE/IEEE Frontiers in Education Conference.

Dedek, A., Suffolk University, & Lieberman, B., BioLogic Software Consulting (2006, June). *Qualifying use case diagram associations*.

Dietmeyer, D.L. (1971). *Logic design of digital systems,* Boston: Allyn and Bacon.

Dockerill, K. (1999).*The importance of animation with UML*, Proc. Ninth Int. Symp. INCOSE.

Dockerill, K. (2001, June). *UML for systems engineering*, Proc UML for Real-Time Systems Development, Newbury, UK, Adaxia.

Doyle, L., & Pennotti, M. (2005, March). *Systems engineering experience with UML on a complex system*, Department of Systems Engineering and Engineering Management, Stevens Institute of Technology.

Friedenthal, S.A., & Burkhart, R. (2007). *Extending UML from software to systems.*

Fang, K.Y., & Wojcik, A.S. (1998). *Modular decomposition of combinational multiple-valued circuits*, IEEE Trnas. On Comput., Vol. 37, No. 10, 1293-1301.

FIPS 183, (1993). Draft Federal Information Processing StandardsPublication 183.

Fowler, M., & Scott, K. (2004). UML Distilled Third Edition: *A brief guide to the standard object modeling language*, Addison-Wesley, Boston.

Gero, J. (1990). Design prototypes: *A knowledge representation schema for design.* AI Magazine, 11(4), 26–36.

Gotterbarn, D., UML: *Improving its risk reduction*, Auckland University of Technology, New Zealand.

Grossman, M., Aronson, J.E, & McCarthy, R.V. (2004). *Does UML make the grade*, Insights from the software development community.

Hause, M., Thom, F., & Moore, A. (2008). Inside SysML - *Artisan Software Tools.*

Hawkins, P.G., & Woollons, D.J. (1998). *Failure modes and effects analysis of complex engineering systems using functional models*. Artificial Intelligence in Engineering, 12(4), 375–397.

Herzog, E., & Pandikow, A. (2005). *SysML – an assessment*, Sweden.

Hitchins, K.D. (1997). *Systems thinking*, Retrieved February 16, 2008, from http://ourworld.compuserve.com/homepages/prof_Hitchins/.

Hitchins, K.D. (1992). *Putting systems to work.*

IEEE Std 1320.(1998). *IEEE standard for functional modeling language—syntax and semantics for IDEF0*.

Iwasaki, et al. (1993). *How things are intended to work capturing functional knowledge in device design*.

Jozwiak, et al. (1995). *Efficient decomposition of assigned sequential machines and Boolean functions for PLD implementations*, IEEE, 258-266.

Kanefsky, P., Nelson, V.A., & Ranger, M. (1999). *A systems engineering approach to engine cooling design*, SAE International, 44th Ray Buckendale Lecture.

Kim, C.H., Weston, R.H., Hodgson, A., & Lee, K.H. (2002). *The complementary use of IDEF and UML modeling approaches*, Computers in Industry, Elsevier.

Knowledge Based Systems, Inc., 1408 University Dr. East College Station, TX 77840, Retrieved January 20, 2008, from http://www.idef.com/idef0.html.

Kobryn, C. (2004). *UML 3 and the future of modeling*, Journal of Software and System Modeling, Vol. 3, No. 1, 4-8.

Langford, G. (2006). Course on Systems Engineering presented at the Naval Postgraduate School, Monterey CA.

Langford, G. (2007). Draft paper for the International Council on Systems Engineering (INCOSE) on Functional Analysis.

Langford, G. (2008). Personal communication.

Langford, G., & Huynh, T. (2007, September). *A methodology for managing complexity*, Systems Engineering Test and Evaluation Conference, Sydney, Australia, 24-27.

Larkin, J. (1983). *The role of problem representation in physics*. In Gentner, D. & Stevens, A. (Eds.), Mental models. Hillsdale, NJ: Erlbaum.

Leffingwell, D., & Widrig, D. (2002). *The role of requirements traceability in system development*, The Rational Edge.

Long, J.E. (2008). *Relationships between common graphical representations used in systems engineering*, Vitech Corporation.

Maier & Rechtin. (2002). *The art of systems architecting*, Second Edition, by CRC Press LLC.

Malan, R., Bredemeyer, D., & Bredemeyer Consulting. (2001). *Functional requirements and use cases*.

Meyer, B. (1997). *OOSC2: The use case principle*, Objected-Oriented Software Construction 2nd ed.

Naval Air Systems Command Systems Engineering Guide (2003, May).

Osmundson, J. (2007). (slides UML basics) Naval Postgraduate School.

OMG SysML. (2007, September). Ver. 1.0, OMG Available Specification.

Poort, E., & LogicaCMG. (2008). *Resolving requirement conflicts through non-functional decomposition*, Meander 901, 6825 MH Arnhem, The Netherlands; Peter H.N. de with LogicaCMG / Eindhoven Univ. of Technol., P.O. Box 513, 5600 MB Eindhoven.

Perkowski & Grygiel. (1995, November). *A survey of literature on functional decomposition*, Ver. VI.

Sembugamoorthy & Chandrasekaran. (1986). *Functional representation of devices and compilation of diagnostic problem solving systems*.

Snooke, N.A., & Price, C.J. (1998). *Hierarchical functional reasoning.* Knowledge-Based Systems, 11(5–6), 301–309.

Sticklen, Chandrasekaran, & Bond. (1989). *Functional reasoning for design and diagnosis*. In Proceedings Model Based Diagnosis International Workshop (DX-89).

TogetherSoft, Inc, (2001). Retrieved April 17, 2008, from http://www.it.lut.fi/kurssit/01-2/010758000/UML_index.html.

Vanderperren, Y., & Dehaene, W. (2005). *SysML and systems engineering applied to UML-based SoC design*, Proc. 2nd UML for SoC Design Workshop, 42nd DAC.

Volvo Car Corporation. (2002). Object Management Group's UML Systems Engineering (SE) Request for Information (RFI) Response.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Gary Langford
   Department of Systems Engineering
   Naval Postgraduate School
   Monterey, California

4. John Osmundson
   Naval Postgraduate School
   Monterey, California