Practical Model-Based Systems Engineering

Jose L. Fernandez • Carlos Hernandez

Practical Model-Based Systems Engineering

For a listing of recent titles in the
Technology Management and Professional Development Library,
turn to the back of this book.

Practical Model-Based Systems Engineering

Jose L. Fernandez
Carlos Hernandez

ARTECH
HOUSE
BOSTON | LONDON
artechhouse.com

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalog record for this book is available from the British Library.

ISBN-13: 978-1-63081-579-0

Cover design by John Gomes

685 Canton Street
Norwood, MA 02062

10 9 8 7 6 5 4 3 2 1

To María Eugenia, my best friend, my wife, and my everything,
and to my son Pablo; keep reaching for your dreams, they will come
through with hard work and perseverance
—JLF

This book is dedicated to Marjolein, because her love fuels my energy daily
—CH

Contents

Preface *xiii*

Acknowledgments *xv*

CHAPTER 1

Introduction 1

1.1 Book Goals and Readers 1
1.2 Book Content 2
1.3 Diverse Paths for Reading the Book 5
References 5

CHAPTER 2

Systems Engineering 7

2.1 Definition and Properties of a System 7
2.2 The System Life Cycle 9
2.3 Systems Engineering: A Discipline to Deal with Complexity 11
2.3.1 The Need for Systems Engineering 11
2.3.2 Key Tasks of Systems Engineering 12
2.4 System Development Alternatives 15
2.4.1 Sequential Approach 16
2.4.2 Incremental Approach 17
2.4.3 Evolutionary Approach 17
2.5 Summary 18
2.6 Questions and Exercises 19
References 19

CHAPTER 3

Model-Based Systems Engineering 21

3.1 Why Is Model-Based Engineering Necessary? 21
3.2 What Is Modeling in MBSE? 22
3.2.1 Why Are Models Used? 22
3.2.1 Models and Views 24
3.3 Modeling Languages, Methods, and Tools for MBSE 25
3.3.1 Modeling Languages 25

3.3.2 MBSE Methodologies 27
3.3.3 MBSE Tools 28
3.4 Summary 30
3.5 Questions and Exercises 31
References 31

CHAPTER 4

The ISE&PPOOA Process 33

4.1 Integrating Systems Engineering and Software Architecting 33
4.2 Challenges of the ISE&PPOOA Process 34
4.2.1 Implementing Nonfunctional Requirements 34
4.2.2 Dealing with Functional and Physical Interfaces 35
4.3 Conceptual Model for Systems Engineering with ISE&PPOOA 36
4.4 Dimensions and Main Steps of the ISE&PPOOA Process 38
4.4.1 The Systems Engineering Subprocess 39
4.4.2 The Software Architecting Subprocess 44
4.5 An Extension of the Process for Energy Efficiency Concerns 49
4.6 Summary 53
4.7 Questions and Exercises 53
References 54

CHAPTER 5

Functional Architecture 55

5.1 The Importance of Functional Architecture 55
5.1.1 Functional Architecture in Systems Engineering 55
5.1.2 What About Functional Architecture for Software Intensive Systems? 56
5.2 A Function is a Transformation 57
5.2.1 Main Concepts Related to the Functional Architectural Model 57
5.2.2 Functional Architecture Models 58
5.3 Modeling the Functional Hierarchy 63
5.3.1 Top-Down Approach for a Functional Hierarchy 64
5.3.2 Bottom-Up Approach for a Functional Hierarchy 65
5.4 Modeling the Functional Flows 65
5.5 Describing Functions and Functional Interfaces 66
5.6 Functional Requirements 67
5.7 Summary 68
5.8 Questions and Exercises 68
References 69

CHAPTER 6

Heuristics to Apply in the Engineering of Systems 71

6.1 Heuristics Framework 71
6.2 Systems Architecting Heuristics 75
6.2.1 Heuristics for Step 3: Functional Architecture of the ISE Process 75
6.2.2 Heuristics for Step 4: Physical Architecture of the ISE Process 75

6.3 Reliability and Maintainability Heuristics 77
6.3.1 Reliability and Maintainability Heuristics for Step 4: Physical Architecture of the ISE Process 78
6.3.2 Maintainability Heuristics for Software Architecting: PPOOA Process 78
6.3.3 Heuristics for Restricting the Visibility of Responsibilities 79
6.3.4 Heuristics for Preventing Unintended Effects 80
6.4 Efficiency Heuristics 80
6.4.1 Heuristics for Managing Demand 81
6.4.2 Heuristics for Arbitrating Demand 82
6.4.3 Heuristics for Managing Multiple Resources 83
6.5 Safety Heuristics 84
6.5.1 Heuristics for Hazard Avoidance 85
6.5.2 Heuristics for Hazard Reduction 85
6.5.3 Heuristics for Hazard Control 86
6.5.4 Heuristics for Mitigation of the Effects 87
6.6 Resilience Heuristics 87
6.6.1 Heuristics for Surviving a Threat 88
6.6.2 Heuristics for Adapting to a Threat 88
6.6.3 Heuristics to Degrade Gracefully 89
6.6.4 Heuristics to Act as a Whole in the Face of a Threat 89
6.7 Software Architecting Heuristics Using the PPOOA Framework 90
6.8 Summary 91
6.9 Questions and Exercises 92
References 92

CHAPTER 7

Physical Architecture 95

7.1 Physical Architecture in Systems Engineering 95
7.1.1 Main Concepts Related to the Physical Architectural Model 95
7.1.2 Functional, Physical, and Quality Trees in ISE&PPOOA 96
7.1.3 Other Architecture Models 96
7.2 Allocation and Modularity 98
7.2.1 Representation of the Modular Architecture 98
7.2.2 Allocation 100
7.3 Design Heuristics for Refining the Architecture 101
7.3.1 Control Monitor Pattern 101
7.3.2 Triple Modular Redundancy Pattern 102
7.4 Software Architecting with the PPOOA Framework 102
7.4.1 Domain Model 103
7.4.2 Software Components and the PPOOA Vocabulary 104
7.4.3 Coordination Mechanisms 106
7.4.4 Software Behavior and Causal Flow of Activities 108
7.5 Summary 109
7.6 Questions and Exercises 109
References 109

CHAPTER 8

Example of Application: Unmanned Aerial Vehicle-Electrical Subsystem 111

8.1 Example Overview, Needs, and Capabilities 112
8.1.1 Operational Scenarios and Use Cases 113
8.1.2 System Capabilities 115
8.2 Functional Architecture and System Requirements 116
8.2.1 Functional Architecture 116
8.2.2 System Requirements 119
8.3 Physical Architecture and Heuristics Applied 127
8.3.1 Heuristics Applied 128
8.3.2 Physical Architecture 130
8.4 Summary 131
References 135

CHAPTER 9

Example of Application: Collaborative Robot 137

9.1 Example Overview, Needs, and Capabilities 137
9.1.1 Identify Operational Scenarios 138
9.1.2 Capabilities and High-Level Functional Requirements 140
9.1.3 Quality Attributes and System NFRs 141
9.2 Functional Architecture and System Requirements 144
9.2.1 Functional Architecture 145
9.2.2 System Requirements 150
9.3 Physical Architecture and Heuristics Applied 154
9.3.1 Modular Architecture 155
9.3.2 Application of Heuristics to Refine the Physical Architecture 158
9.3.3 Representation of the Refined Physical Architecture 159
9.4 Software Architecture 161
9.4.1 Software Components 165
9.4.2 Casual Flows of Activities 167
9.4.3 Safety Heuristics 167
9.5 Summary 170
References 171

CHAPTER 10

Examples of Application: Energy Efficiency for the Steam Generation Process of a Coal Power Plant 173

10.1 Example Overview 173
10.2 Functional Architecture of the Steam Generation Process 176
10.3 Physical Architecture of the Steam Generation Subsystem 178
10.4 Equations and Correlations of the Matter and Energy Balances 179
10.5 Results 183
10.6 Summary 184
10.7 Questions and Exercises 184
References 185

CHAPTER 11

Trade-Off Analysis 187

11.1 Trade-Off and the Architecture Decision Process 187
11.2 Trade-Off Assessment Criteria and Utility Functions 189
11.3 A Trade-Off Subprocess to be Used with the ISE&PPOOA Process 190
11.4 Summary 191
References 192

CHAPTER 12

Other Topics of Interest and Next Steps 193

12.1 Agile Development 193
12.1.1 Principles and Misconceptions about Agility 193
12.1.2 Scalability in Agile Approaches 195
12.1.3 ISE&PPOOA Process and Agility 198
12.2 Architecture Evaluation and Model Checking 199
12.2.1 Architecture Evaluation 199
12.2.2 Diverse Practices for Architecture Evaluation 200
12.2.3 Model Assessment 202
12.3 Next Steps Recommended to the Reader 203
12.4 Summary 204
References 204

Appendix A SysML Notation 207

A.1 Use of SysML in the ISE&PPOOA Methodology 207
A.1.1 SysML Diagrams in ISE&PPOOA 207
A.2 SysML for the ISE&PPOOA Structural Perspective 209
A.2.1 Blocks and Block Definition Diagram 209
A.2.2 SysML Internal Block Diagram: Parts, Ports, Connectors, and Flows 210
A.3 SysML for the ISE&PPOOA Behavioral Perspective 211
A.3.1 Activity Nodes 212
A.3.2 Control Nodes 213
A.4 Other SysML Elements and Views in ISE&PPOOA 213
A.4.1 Allocation 213
A.4.2 Use Case Diagram 214
A.4.3 Constraint Blocks and Parametric Diagrams 215
A.4.4 Requirements 215
A.5 Complementing SysML: PPOOA Architecture Diagram 216
References 216

Appendix B Requirements Framework 217

B.1 Needs, Capabilities, and Requirements 217
B.2 Requirements Classification 218
B.3 Requirements Flowdown in Systems Development 222
B.4 Models and Requirements 224

B.5 Requirements Templates 228
B.6 Summary 229
References 230
About the Authors 231
Index 233

Preface

The main goal of this book is to provide systems engineers and practitioners with the analytic, design, and modeling tools of the Integrated Systems Engineering and Pipelines of Processes in Object-Oriented Architectures (ISE&PPOOA) methodology. This methodology integrates model-based systems and software engineering approaches for the development of complex products. A second goal is to facilitate the benefits of the model-based systems engineering (MBSE) approach and SysML standard notation to communicate and create a quality design, avoiding the burden and complexity of the full-blown MBSE. Therefore, we use a subset of SysML constructs. Our final goal is to provide sufficient examples and exercises for the readers to be able to apply ISE&PPOOA in the systems engineering activities of their own systems, making use of SysML diagrams for that purpose.

ISE&PPOOA is the result of more than 25 years of research and development.

PPOOA was created in the late 1990s as a software architecture framework for real-time systems. PPOOA proposes a vocabulary of building elements and how to use them in building a software architecture for a real-time system.

The seminal paper about PPOOA was published in 1998, titled "An Architectural Style for Object-Oriented Real-Time Systems" (5th International Conference on Software Reuse, Victoria, Canada, IEEE 1998). This paper describes the framework emphasizing the usage of coordination mechanisms and the architecting guidelines specified for the framework.

PPOOA was developed before UML standard publication. As UML popularity increased, the first author realized the importance of using UML notation. Then, partially funded by the European Union Fifth Framework Programme, we accomplished the CARTS IST project, where we developed a UML profile for real-time systems based on PPOOA, and an architecting process called PPOOA_Architecting Process (PPOOA_AP). PPOOA and PPOOA_AP were validated in autonomous robots and ground space systems developed by the industrial partners of the CARTS project (1999–2001).

In 2004, PPOOA was implemented in Microsoft Visio, which is a general drawing tool that provides mechanisms for implementing diverse engineering methods. This implementation offers the benefits of commercial CASE tools that already support UML notation, and supports the semantics and metamodel of PPOOA.

Predicting software performance at early phases (for example, after the architecting stage) of the software life cycle and evaluating it based on UML models is an attractive approach for saving expensive effort in system testing and fixing the

software later. The PPOOA CASE tool implemented a new module (Visio add on) that translated the PPOOA_UML architecture diagrams of a system to an XML file that could be read by Cheddar, a scheduling analyzer and simulation tool developed by the University of Brest (France). Later, the PPOOA tool was enhanced with new features as the use of intelligent agents for mentoring the software architects and early deadlock detection at the architecture stage of development.

Finally, the PPOOA software architecting process was extended to be used for complex products or systems that may be either software-intensive or not. The new MBSE process was called Integrated Systems Engineering and Pipelines of Processes in Object Oriented Architectures (ISE&PPOOA).

The ISE&PPOOA process was created as tool independent of but using a subset of the SysML notation standard.

The ISE part of the process includes the first steps of a systems engineering process applicable to any kind of system, not only the software-intensive ones. The ISE subprocess integrates traditional systems engineering best practices and MBSE.

The PPOOA part of the process supports the modeling of concurrency as earlier as possible in the software engineering part of the integrated process.

ISE&PPOOA provides a collection of guidelines or heuristics to help the engineers in the architecting of a system.

One of the main deliverables recommended by ISE&PPOOA is the functional architecture representing the functional hierarchy using the SysML block definition diagram. This diagram is complemented with activity diagrams for the main system functional flows. The N^2 chart is used as an interface diagram where the main functional interfaces are identified. A textual description of the system functions is also provided as part of the deliverable.

Another deliverable recommended by the process is the physical architecture, which represents the system decomposition into subsystems and parts using the SysML block definition diagram. This diagram is complemented with SysML internal block diagrams for each subsystem and activity and state diagrams as needed. A textual description of the system blocks is also provided. The heuristics used for the particular architecture solution are identified and documented.

The software subsystems architecture is described in PPOOA using two views supported by one or more diagrams using UML notation. One view is the static or structural view, and the other is the dynamic or behavioral view of the system. The system architecture diagram represents the system components and the composition and usage relations between them. Coordination mechanisms used as connectors are also represented. The system behavioral view is supported by UML/SysML activity diagrams, which represent an internal view of the flow of actions performed by the system in response to an event.

ISE&PPOOA/energy is the latest process of this MBSE method. ISE&PPOOA/energy is used to deal with energy efficiency issues in process plants.

Acknowledgments

The authors wish to acknowledge the many engineers and organizations that have contributed to the model-based approach, SysML, and system engineering in general. This book would not have been the same without their previous work and ideas.

The authors especially thank the reviewers of this book for their valuable feedback.

The authors also thank the Artech House staff, particularly Soraya Nair, Aileen Storry, and Kathryn Klotz, for being supportive when we needed encouragement.

Jose Fernandez would like to thank the following individuals and organizations:

- The SEI/Carnegie Mellon University for the opportunity they gave me to work on the taxonomy of coordination mechanisms for real-time systems.
- The European Commission, who through the project CARTS funded part of my research on software architectures for real-time systems.
- The INCOSE MBSE focus group and the OMG for giving me the opportunity to present ISE&PPOOA on the OMG MBSE wiki.
- The engineers and Universidad Politecnica de Madrid professors and students, who collaborated in the research and development of ISE&PPOOA with publications, tools, and examples of application. I would like to mention Juan A. de la Puente, Juan C. Dueñas, Silvia Palanca, Miguel A Aranda, Juan C. Martínez, Antonio Monzón, Bill J. Mason, Jean Barroso, Eduardo Esteban, Laura Sanz, Javier Carracedo, Gloria Mármol, Enrique Martin, Agatha Puigdueta, Noelia Delgado, Mario García, Patricio Gómez, Guillermo Moreno, Ignacio Cantón, Rubén Sancho, Fátima Cadahia, Alfonso García, Borja Martínez, Monica Diez, and Darío Nicolás.

This book would not have been possible without the contribution of an excellent team of people, including Carlos as coauthor, and Juan, Alberto, and Leticia who have been essential in the development of some of the book's examples.

Finally, I want to thank my family for their patience and understanding for the time that I was not able to dedicate to them.

Carlos Hernandez would like to thank the following individuals and organizations:

- Prof. Martijn Wisse, the Delft University of Technology, and the European Commission, who through the projects Factory-in-a-day and ROSIN have been supporting my research on the design of advanced robotic applications.
- The colleagues at the Autonomous System Laboratory of the Universidad Politectica de Madrid: Ricardo Sanz, Julia Bermejo, Manuel Rodriguez, Guadalupe Sanchez, and Ignacio Lopez, for infinite hours of providing ideas and discussion about systems, functions, and the ontology and epistemology of engineering.
- The colleagues in Delft, Mukunda Bhaharatheesha, and Gijs van der Hoorn. The example of the robotic application wouldn't have been possible without countless hours developing robotic applications and invaluable hours discussing their design.

I have to thank my family and friends, and specially my beloved Marjolein, for their continuous support and patience.

Finally, my contribution to this book would not have been possible without the priceless mentorship and friendship of Jose Luis, from whom I have the pleasure to learn every day new lessons on engineering, passion, hard work, and integrity.

CHAPTER 1

Introduction

This chapter informs the book reader about the book itself. That is: the goals of the book, the book audience, the book content, and diverse paths for reading and using the book.

1.1 Book Goals and Readers

The main goal of the book is to provide systems engineers and practitioners with the analytic, design, and modeling tools of the ISE&PPOOA methodology. As defined by Estefan, "A Model Based Systems Engineering (MBSE) methodology can be characterized as the collection of related processes, methods, and tools used to support the discipline of systems engineering in a model-based or model-driven context" [1].

ISE&PPOOA methodology integrates model-based systems and software engineering approaches for the development of complex products, such as the examples presented in the book that illustrate the application of the methodology.

A second goal is to facilitate the benefits of the MBSE approach and Systems Modeling Language (SysML) standard notation to communicate and create a quality design, avoiding the burden and complexity of the full-blown MBSE. So, we use a subset of SysML constructs as it is explained in Appendix A. For the software elements of the system we use an extension of Unified Modeling Language (UML) notation that was developed as part of the PPOOA architectural framework. The diagrams of the system model are complemented with textual descriptions mainly in tabular form that help to understand them better.

Our final goal is to provide sufficient examples and exercises that clarify the SysML model diagrams used, the ISE&PPOOA process application, and the work done in defining a system.

When we offer a MBSE methodology to the audience, we are proposing a way of thinking consistently to solve an engineering problem, where identifying the functions and quality attributes of the product to be developed is a main issue to synthesize the solution.

We can say that ISE&PPOOA is a requirements-driven, model-based systems engineering approach where the main outcomes are the functional and physical architectures of the product, system, or service to be developed. In some cases, the product, system, or service is software intensive, such as the collaborative robot example of Chapter 9, in other cases we applied the methodology to a subsystem that

is not software intensive, such as the electrical subsystem of an unmanned aerial vehicle presented in Chapter 8.

One of the main impediments to use a methodology in an organization is that people may think that its use is an obstacle for their creativity and the agility of the development team. We think it is not the case. Our experience with undergraduate and graduate students in academia and systems engineering practitioners in industry, is that the methodology helps steer their work toward the important project outcomes, but creativity and the knowledge of the application domain are still necessary skills and competencies of the project team individuals.

Creativity is part of the ISE&PPOOA methodology because instead of the rigid rules of a formal procedure it promotes the use of heuristics (see Chapter 6) to build physical architecture from the functional architecture following the well-known Sullivan's principle for architecting buildings: "Form follows function."

Are all the buildings with the same functionality equal? No; it is obvious. The same is true for systems. Commercial aircraft with the same functions may be either similar or different; for example, the function "control cabin and flight deck environment" is implemented differently by Boeing and Airbus as commercial aircraft manufacturers.

Currently, agility is a main concern as well. As we explain in Chapter 12, an agile approach to development is compatible with the ISE&PPOOA methodology presented in the book, so ISE&PPOOA can be used with some of the well-known scalable agile approaches summarized and referenced in Chapter 12.

The book is useful for several audiences: graduate and undergraduate students of systems engineering may use it as an instruction text; experienced systems engineers trying to apply MBSE can learn how to think to develop system models; and specialty engineers, such as mechanical, electrical, software, safety, and logistic engineers, can use the book as an approachable guide to MBSE that will help them understand the information captured in the system models.

An interesting audience group for the book are process plants engineers interested in energy efficiency that can use ISE&PPOOA/Energy (see Chapter 10) to holistically create a system model of the process plant with the level of detail needed to assess the process plant mass and energy balances.

1.2 Book Content

Besides this introductory chapter, the book has four parts and two appendixes.

Part I, "Foundations," consisting of Chapters 2 and 3, describes classical systems engineering, and model-based systems engineering as the new paradigm of systems engineering promoting the use of models instead of plain documents.

- Chapter 2, "Systems Engineering," describes traditional systems engineering as the framework that combines diverse engineering specialties to develop a complex product. A complex product may be considered as a system that has a life cycle. System development can be performed using alternative approaches summarized in this chapter.

- Chapter 3, "Model-Based Systems Engineering," provides a practical overview of MBSE, explaining its benefits (why), introducing basic concepts about modeling (what), and presenting the main aspects for MBSE (how): the modeling language, the methodology, and ending with a brief practical discussion of the tools for MBSE.

Part II, "The Methodology," consisting of Chapters 4–7, describes the ISE&PPOOA conceptual model and process, how to create the functional architecture, and promotes the use of heuristics to create a physical architecture allocating the functions identified, and implementing in the refined architecture the nonfunctional requirements previously specified.

- Chapter 4, "The ISE&PPOOA Process," describes the ISE&PPOOA methodology that promotes a MBSE approach that integrates systems engineering and software architecting. The main concerns of the methodology are how to deal with functional allocation, nonfunctional requirements implementation, and interfaces specification and design. The methodology is described in this chapter by its conceptual model and the definition of its process main steps and deliverables. An extension for energy efficiency assessment named ISE&PPOOA/energy is presented as well.
- Chapter 5, "Functional Architecture," describes the characteristics and modeling aspects of the system functional architecture, which is the core model deliverable of the ISE&PPOOA methodology. The chapter emphasizes that functional architecture is a representation of the system behavior that is independent of technology solutions.
- Chapter 6, "Heuristics to Apply in the Engineering of Systems," presents a collection of general heuristics and quality attributes heuristics to be applied to develop the system solution based on the main concerns specified by the requirements, particularly the nonfunctional ones.
- Chapter 7, "Physical Architecture," deals with the creation of the physical architecture of an engineered system. We discuss the building blocks used in the physical architecture. Then, we explain the allocation of the functions identified in the functional architecture to the physical building blocks, with the goal of modularity. Applying the selected heuristics, the refined physical architecture is obtained. The physical architecture is then explained, focusing on the logical and physical connectors that link the building blocks. Finally, the domain model is presented as the bridge from the system physical architecture to the software subsystem architecture, and the role of the software components in the architecture is discussed.

Part III, "Examples," consisting of Chapters 8–10, illustrate the application of ISE&PPOOA to three different examples. The first example is related to the aerospace domain and unmanned aerial vehicles (UAVs), where the electrical subsystem of a real fixed-wing UAV is modeled. The second example is a collaborative robot where the modeling of system behavior and software architecture are the main concerns. The third one is a realistic example of energy efficiency assessment of a

coal power plant. ISE&PPOOA/energy is used here to model the coal power plant to the level of detail necessary to solve the balances of mass and energy.

- Chapter 8, "Unmanned Aerial Vehicle-Electric Subsystem," illustrates how the ISE&PPOOA methodology is used to engineer the Seeker UAS by Aurea Avionics, a lightweight fixed-wing UAV designed to support and cover intelligence, surveillance, and reconnaissance (ISR) missions. This example describes the electrical subsystem design, which is not software intensive and, in this case, shows some distinctive characteristics compared to other subsystems.
- Chapter 9, "Collaborative Robot," the mission dimension of a collaborative robot application is discussed, applying ISE&PPOOA to identify the operational scenarios for the robot and specify the system capabilities and high-level functional requirements. The functional architecture is obtained by iteratively identifying the functions needed to realize the capabilities and decomposing them into subfunctions. Then it describes the allocation of the functions to the building elements of the robotic system and the refinement of the resulting physical architecture by applying the heuristics selected. Flexible robot solutions for manufacturing are software intensive, so this chapter presents how the PPOOA architecting subprocess can be applied to obtain the software architecture for the collaborative robot, which is refined using the PPOOA software-related heuristics.
- Chapter 10, "Energy Efficiency for the Steam Generation Process of a Coal Power Plant," describes an application example of the ISE&PPOOA/Energy systems approach, which combined with the equations of mass and energy balances, allows to evaluate the efficiency of an industrial facility or part of it, with the level of appropriate detail, adapting this level of analysis to the process data, equations, graphs, tables, and other correlations available for the particular industrial facility. In this example, the steam generation of a coal power plant is analyzed, choosing one of its 350-MW groups.

Part IV, "Other topics of interest," deals with some topics that we consider of interest besides the core of the book (Parts II and III). These topics are trade-off analysis, agile development, model checking, and some recommendations regarding how to apply this MBSE approach in an organization.

- Chapter 11, "Trade-Off Analysis," describes analytical trade-off analysis as a complex and interesting issue for systems engineers to explore the solutions space and recommends its use as complementary with the heuristics approach proposed by the ISE&PPOOA methodology.
- Chapter 12, "Other Topics and Next Steps," since agile development is growing and in particular has a large market adoption for software development. The chapter shows that the ISE&PPOOA methodology can be integrated and used in an agile project. Architecture evaluation and particularly model checking is still a research issue but is useful when the tools supporting it are available. The last section of this chapter is a recommendation to the readers

Table 1.1 Book Reading Paths

Systems Engineering Students	*Experienced Systems Engineers Interested in MBSE*	*Specialty Engineers*	*Process Plants Engineers*
Chapter 2	Chapter 3	Chapter 3	Chapter 2
Chapter 3	Appendix A	Appendix A	Chapter 3
Appendix A	Chapter 4	Chapter 4	Appendix A
Chapter 4	Chapter 6	Chapter 5	Chapter 4
Chapter 5	Chapter 7	Chapter 7	Chapter 5
Appendix B	Chapter 11	Chapter 8	Chapter 7
Chapter 6	Chapter 8	Chapter 9	Chapter 10
Chapter 7	Chapter 9		
Chapter 11	Chapter 12		
Chapter 8			
Chapter 9			
Chapter 12			

of the next steps to follow to apply the ISE&PPOOA methodology to use MBSE in their organizations.

Appendix A, "SysML Notation," describes how a subset of SysML constructs are used by the ISE&PPOOA methodology. It is not intended as a complete description of the SysML standard.

Appendix B, "Requirements Framework," is a brief and practical guide on requirements classification, specification, and flowdown, and how it is done in the ISE&PPOOA methodology.

1.3 Diverse Paths for Reading the Book

You can read this book from cover to cover, but we recommend other reading options as well, these reading paths are based on your previous experience in systems engineering and your expectations on how to use the MBSE approach presented in the book.

We propose diverse paths to read the book considering the book audience identified in Section 1.1 (see Table 1.1).

References

[1] Estefan, J.A., "Survey of Model-Based Systems Engineering (MBSE) Methodologies". INCOSE-TD-2007-003-01 Version/Revision: B, Seattle, WA: International Council on Systems Engineering (INCOSE), 2008.

CHAPTER 2

Systems Engineering

Systems engineering is the framework that combines diverse engineering specialties to develop a complex product. A complex product may be considered as a system that has a life cycle. System development can be performed using alternative approaches. So this chapter describes what is a system, its properties and life cycle, the key tasks of systems engineering, and the system development alternatives.

2.1 Definition and Properties of a System

The realizations of current engineering are increasingly large and complex. The size is obvious in large civil works and transport infrastructure, whether roads, railways, airports, harbors, and others. The trend of increasing size in the aerospace sector where recent developments, such as the Airbus A380 or the Boeing 747-8, are illustrative examples. But size is not the only factor that makes a system difficult or not difficult to achieve. The complexity related to internal and external interactions, and the need to combine various engineering disciplines and specialties make the engineering and development of smaller systems, such as a UAV, a car, a printer, or a robot for surgery, more difficult every day, with the aggravating circumstance that due to the market demands development and production are getting shorter.

Therefore it is important, before defining what we mean by systems engineering, to know what we mean by a system and to identify the main characteristics of a system that are frequently independent of its size.

There are many definitions of system in the literature that are coincident in approaching the system as a whole, the so-called holistic view not found in other disciplines of engineering. For illustrative purposes we will consider some of the most popular definitions of a system.

The International Council on Systems Engineering (INCOSE) [1] defines a system as "an integrated set of elements, subsystems, or assemblies that accomplish a defined objective. These elements include products, processes, people, information, techniques, facilities, services, and other support elements."

NASA [2] provides two definitions for system as:

1. "The combination of elements that function together to produce the capability to meet a need. The elements include all hardware, software, equipment, facilities, personnel, processes, and procedures needed for this purpose."

2. "The end product (which performs operational functions) and enabling products (which provide life cycle support services to the operational end products) that make up a system."

The International Standards Organization (ISO) [3] defines a system as "a combination of interacting elements organized to achieve one or more stated purposes."

From the above definitions we realize that a system is a combination of interacting elements functioning together to achieve a purpose. A system is not only a product; it may include people, processes, facilities, and other support elements.

What are the properties that characterize a system? First we will identify the properties of the current systems and after we will consider the new properties that are required in new systems.

- A system has a hierarchical structure consisting of a number of major interacting elements, generally called parts, which themselves may be composed of more simple functional entities or simple parts. For example IEEE Std. 1220 [4] considers a system as a containment of a product and the processes to develop, test, manufacture, and support it. The product is decomposed into subsystems. Subsystems may be decomposed into assemblies that may contain components and subcomponents as shown in Figure 2.1. In the ISE&PPOOA approach (see Chapter 4) we consider a part as a composite of simple parts. computer hardware configuration items and computer software configuration items are examples of composite parts.
- A system is a whole whose properties are not only its parts properties but are also the consequence of the interaction of its parts. This property is called emergence, and it may appear as observable at the different system levels as described by Hitchins [5]. Emergence may be related to behavior, functionality, or quality attributes, for example system reliability emerging from the configuration of its parts.
- A system operates in a context consisting of the environment and other systems. Context may be considered as external users, external systems, natural environment, threats, and resources [6].
- Besides its functionality and performance, a system has some quality attributes dependent of its use for example maintainable, reliable, interoperable, and others.
- A system has a life cycle from its preliminary and concept stages to its retirement.

All systems can have emergent properties that may or may not be predictable by engineers. One example of emergent property is resilience (see Chapter 6). Resilience is a critical system property that is not meaningful at the part level but is meaningful at the whole system level.

Twenty-first century systems are not single systems as the example of Figure 2.1; they are interconnected and part of bigger systems. Some of the new systems, for example mobile robots, are required to be autonomous, self-repairing, or adaptive to a change in the environment. So, as proposed by a group of INCOSE fellows the future system is to be adaptive, resilient, and part of a system of systems,

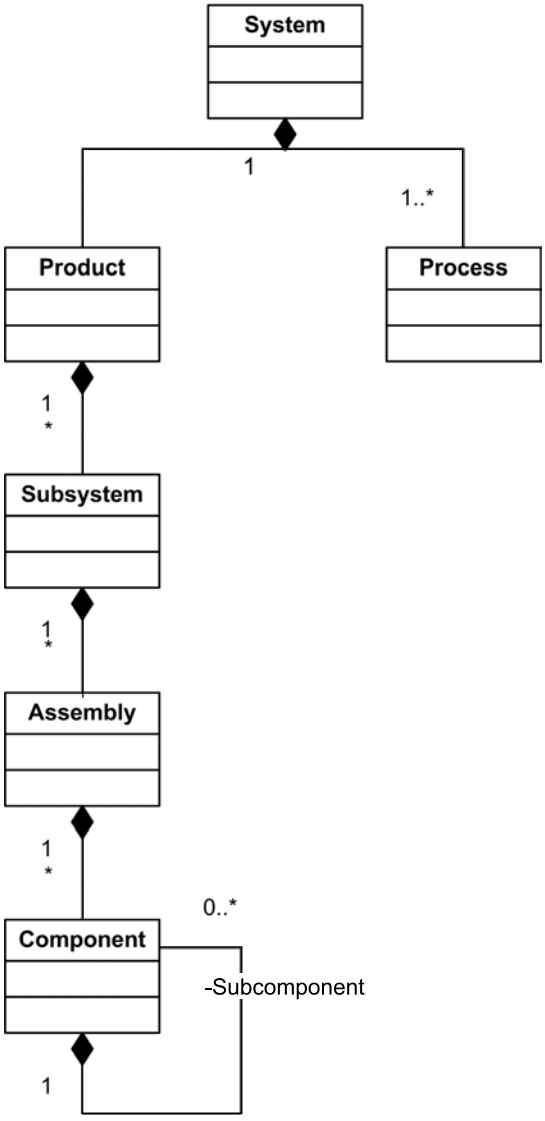


Figure 2.1 System hierarchy based on IEEE Std. 1220 [4].

including products, services, and enterprises, and integrating technological, social, and environmental elements [7].

2.2 The System Life Cycle

The first edition of ISO 15288:2002 [8] proposes the system life cycle from concept stage to retirement stage, which is represented in Figure 2.2 and summarized here.

In the concept stage, the need for a new system of interest is recognized. Diverse tasks related to analysis, feasibility evaluations, cost estimations, trade-off studies, and experimentation and demonstration may be performed. One or more solutions may be identified and evaluated.

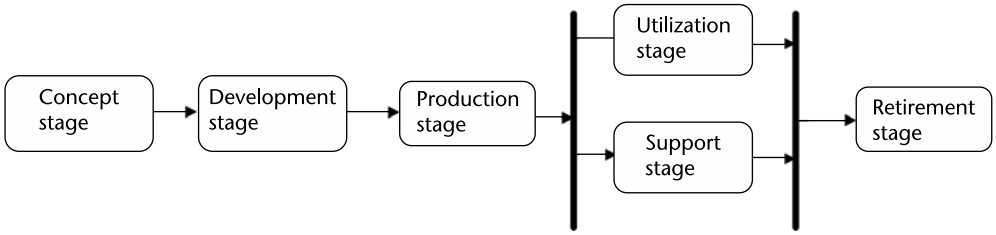


Figure 2.2 System life cycle.

Typical outputs of the concept stage are stakeholder requirements, concept of operations, feasibility assessments, risk register, and outlines of design solutions.

The development stage requires sufficient system requirements and design solutions to transform them into a feasible product. Interfaces are specified, designed, fabricated, integrated, and tested as applicable. Requirements for production, training, and support facilities are specified.

Typical outputs of the development stage are the system architecture, production plans, operating instructions, training manuals for operators, maintenance procedures, cost estimates for the following stages, risk register updates, and definition of the enabling processes required in the following stages.

The production stage begins with the approval to produce the system. The system may be unique or may be mass-produced. During this stage, the product may be enhanced or redesigned.

The main output is the product produced, but deliverables related to product transfer and quality issues are outputs as well.

The utilization stage begins after system installation and transition to use. The system is operated at the operational sites. During this stage the system can evolve giving rise to diverse configurations. This stage ends when the system is taken out of service.

The main outputs are the deployed system, the monitoring of performance and cost reports, and plan and exit criteria.

The support stage begins with the maintenance and logistics provisions. Process related to operating the support system and services are included.

The outcomes of the support stage include trained maintenance and support personnel, product maintenance, and logistic support.

In Figure 2.2, the utilization stage and support stage are shown in parallel. ISO 15288 [8] points out that these two stages may overlap with the production stage as well.

The retirement stage provides for the removal of the system and its operational and support services.

The main outcomes are related to the decommissioning of the system, including disposal, refurbishing, or recycling in accordance with applicable laws and regulations. Waste removal and operational staff reallocation are the main issues.

2.3 Systems Engineering: A Discipline to Deal with Complexity

Systems engineering as a discipline has as its main goal to deal with complexity, considering complex systems those with many interactions either external or internal to their constituent parts. Therefore, dealing with complexity means dealing with different parts either mechanical, electrical, or software intensive that interact in diverse ways.

There are diverse definitions of systems engineering in the literature. Here for the sake of brevity and understandability we consider NASA, INCOSE, and IEEE definitions.

NASA handbook defines systems engineering as: "methodical, multi-disciplinary approach for the design, realization, technical management, operations, and retirement of a system" [2].

NASA handbook also considers systems engineering: "a discipline that combines art and science for developing an operable system capable of meeting requirements within often opposed constraints" [2].

INCOSE handbook defines systems engineering as: "an interdisciplinary approach and means to enable the realization of successful systems" [1].

The INCOSE systems engineering approach focuses on defining customer needs and required functionality early in the development cycle, specifying the requirements and then proceeding with the design synthesis and validation of the system as a whole [1].

As part of its abstract, the IEEE Std. 1220 has the following description of systems engineering: "The interdisciplinary tasks, which are required throughout a system's life cycle to transform customer needs, requirements, and constraints into a system solution" [4].

The systemic approach that is part of common sense in humans is older than systems engineering. Aristotle defines the part and whole theory where the whole is greater than the sum of its parts. St. Paul, based on Hellenistic tradition, in his first letter to the Corinthians: 12, 12-27, describes the nascent Church as a body containing parts that are equally important for its mission. Von Bertalanffy's contribution to systems theory is fundamental. He considers the living organism as an open system having a steady state [9].

The modern origins of systems engineering can be traced to 1937 when the British created a multidisciplinary team to analyze their air defense system. Table 2.1 summarizes some of the main milestones in the evolution of systems engineering as a discipline.

2.3.1 The Need for Systems Engineering

Systems engineering may be helpful in diverse projects since a good understanding of the problem (analysis) and architecting the solution (synthesis) is part of engineering activity. This produces a better understanding of the project scope with savings in cost and time outweighing the costs of implementing the systems engineering processes and tools in the organization.

Table 2.1 Milestones of SE as a Discipline

1940	The term systems engineering is used for the first time by Bell Telephone Laboratories
1939–1945	NIKE project development for the U.S. antiaircraft missile system
1951–1980	SAGE project for air-defense system
1956	Systems analysis by RAND corporation
1962	Publication of *A Methodology for Systems Engineering* by A.D. Hall. Ed Van Nostrand
1990	INCOSE is founded
2006	SysML Standard release
2008	Harmonization of systems engineering concepts on ISO/IEC/IEEE 15288:2008

Complexity is the differential factor system where the use of systems engineering is especially needed. Complexity is not only a technical issue; Kossiakoff et al. [10] identify the characteristics of a system or product whose development, test, and application require the application of systems engineering best practices. These are:

- It is an engineered product that satisfies a need;
- The product contains interacting and diverse components that is, for example, a mechatronic product containing mechanical components, electronic components, and software components;
- Advanced technologies are used involving some development risks.

Examples of this kind of systems are commercial aircraft, complex agricultural machinery, or an electric power plant.

Today more products may need a systems engineering approach; for example, the so called smart products that integrate information and communication technologies. As described by Shamieh [11], smart products development requires addressing the following issues:

- Multiple technical fields are used in the development of a smart product;
- Software is a critical part of the smart product;
- Software needs to be integrated with hardware;
- The system or smart product interacts with other systems;
- Regulatory issues are important;
- Project complexity is an issue.

Examples of smart products are smart cars, autonomous robots, and drones.

2.3.2 Key Tasks of Systems Engineering

Eisner [12] proposes a variety of tasks for traditional systems engineering and its management. He calls these complex tasks the thirty elements of systems engineering. We use his list and describe briefly its elements in the Table 2.2 below using systems engineering standards as well [3, 4, 13], and The *Guide to the Systems*

Table 2.2 Systems Engineering and Management Tasks

Identification of Needs, Goals, and Objectives	Establish with the user or customer that the statements of needs, goals, and objectives are correct.
Engineering the Mission	Analysis of the intended mission of the system. As mentioned by ISO 15288-2008 [3], the ConOps is used to describe the user organization(s), mission(s), and organizational objectives from an integrated systems point of view.
Analysis and Allocation of Requirements	As part of requirements engineering, their analysis and allocation is a main issue.
Functional Analysis and Functional Decomposition	Identify the system functions, and then represent their hierarchy and the main functional flows.
Solution Design and Synthesis	As described by ISO 15288-2008 [3], the tasks here are to define and represent the alternative solution architectures.
Analysis and Evaluation of Alternatives	Consider previous task. Here the tasks are the analysis and evaluation of the alternative solution architectures.
Measurement of Technical Performance	As described by ANSI/EIA 632, identify technical performance measures that will be used to determine the success of the system, or some parts of it thereof, and that will receive management focus and be tracked using technical performance measurement (TPM) procedures [13].
Life Cycle Cost Management	System life cycle costs consists of three main categories: • Research, development, test, and evaluation; • Acquisition or procurement; • Operations and maintenance.
Risk Analysis	Cost, schedule, and technical risks are identified, analyzed, treated, and monitored.
Concurrent Engineering	Concurrent engineering is considered by Eisner as a contribution rather than a replacement for systems engineering [12]. IEEE Std. 1220 recommends that concurrent engineering should integrate product and process development to ensure that the product(s) are producible, usable, and supportable [4].
Development Specifications	As part of the system design and requirements flowdown, detailed specifications at either the system, subsystem, or component level are elaborated.
Hardware, Software, and Human Engineering	Eisner proposes these three dimensions: hardware, software, and human with the subsystems design [12].
Interface management	The management of the system's external and internal interfaces is a critical issue. Managing includes interface classification, definition, and interface development control.
Computer Tool Evaluation and Utilization	Currently, computer tools are used for system modeling, simulation, and evaluation. Tailoring and maintaining the tools used in the system development is an issue.
Technical Data Management and Documentation	Eisner defines two main tasks related to technical data management and documentation. One is to manage the large amount of data generated by the development project. This data may come in many forms. The other is an effective production of system documentation [12].
Integrated Logistics Support (ILS)	ILS considers the integration of support into the system design, developing support requirements, acquiring the required support, and providing the support during the system's operational stage.

Table 2.2 (continued)

Reliability, Availability, and Maintainability (RAM)	As mentioned in the *SEBoK*, RAM are inherent product or system attributes that should be considered throughout the development life cycle [14]. Reliability engineering during the system development seeks to increase system robustness through heuristics and design patterns such as redundancy, diversity, built-in tests, advanced diagnostics, and modularity, to enable rapid physical replacement. Increased maintainability implies shorter system repair times. System RAM characteristics are continuously evaluated as the design progresses. Once a system is deployed, its reliability and availability should be monitored.
Integration	Integration is the activity that brings together system parts into more complex or larger parts. As these larger parts are completed, they are tested. The system obtained is consistent with the architecture design. Nonconformances due to integration actions are recorded.
Verification and Validation	Verification confirms that the specified design requirements are fulfilled by the system. According to ISO 152888-2008, verification is planned and performed, ensuring the associated facilities, equipment, and operators are prepared to conduct it, and analyzing and recording discrepancies and corrective actions information [3]. Validation provides objective evidence that the services provided by a system, when in use, comply with stakeholders' requirements, achieving its intended use in its intended operational environment. Validation is planned and performed to demonstrate conformance of services to stakeholders´ requirements.
Test and Evaluation	Eisner considers test and evaluation task as the physical confirmation of the performance of the overall system. Two contexts are used for testing and evaluation purposes: one for the full-scale development of the system, and the other deploying the system in an operational environment [12].
Quality Assurance and Management	The purpose of this task is to assure that the products, services, and implementations delivered meet quality objectives and satisfy customer needs.
Configuration Management	The purpose of configuration management is ensuring system integrity and visibility. Configuration management involves • Configuration item identification; • Configuration control ensuring that all changes are recorded, evaluated, approved, incorporated, and verified; • Configuration status accounting; • Configuration audits.
Specialty Engineering	Specialty engineering encompasses engineering topics that may need to be explored as part of the system engineering effort. Some examples of specialty engineering include • Manufacturability; • Electromagnetic compatibility and interference; • Environmental impact; • Human factors; • Safety and health.
Preplanned Product Improvement	Eisner considers this task for identifying new paths and as a means to enhance the system beyond current contractual arrangement [12].

Table 2.2 (continued)

Training	Training is basically the task oriented to system operators and maintenance technicians. Sometimes it may be necessary to create special versions of the product for training operators.
Production and Deployment	Production and deployment is a stage of the system life cycle described in Section 2.2. The production stage begins with approval to produce the system. The system may be unique or may be mass-produced. During this stage, the product may be enhanced or redesigned. Related to deployment, ANSI/EIA 632 recommends considering deployment plans and schedules, deployment policies and procedures, mass/volume mockups, packaging materials, special storage facilities and sites, special handling equipment, special transportation equipment and facilities, installation procedures, installation brackets and cables, special transportation equipment deployment instructions, ship alteration drawings, site layout drawings, and installation personnel [13].
Operations and Maintenance	These are the tasks related to the utilization and support stages described in Section 2.2. From a systems engineering perspective it is important to implement a continuous measurement of a system´s performance.
Operations Evaluation and Reengineering	The performance data obtained in the task described above may be inputs for the identification of improvement areas for reengineering of the system.
System Disposal	The retirement stage described in Section 2.2, includes the tasks for safe disposal of the system assets. Disposal tasks include handling of hazardous items, cost of disposal, and approval.
Systems Engineering Management	Systems engineering management is about managing the resources and assets allocated to perform systems engineering, here in the context of a system development project. It includes to a lesser degree tasks related to planning, monitoring, and control and decision management. Management is applied to the previous 29 tasks.

Engineering Body of Knowledge (SEBoK) [14]. It is obvious that these complex tasks are dependent, may be split into subtasks, and overlapping between them can occur as well.

2.4 System Development Alternatives

There are several alternatives that can be used for the system development stages. Here we will describe the three main ones: sequential, incremental, and evolutionary. Other varieties of these three alternatives are possible, but for the sake of brevity, are not explained here. Although we use the term system development, we include production, utilization, and support as well.

Before explaining system development alternatives, it is important to understand that system life is split into stages and the stages represent different states of the system during this life cycle (see Section 2.2). Stages may overlap in time. As described in the *SEBoK*, the term phase is different. Phase refers to the steps of the program or project managing the life of the system [14]. In program or project management phases, milestones and decision gates are used for time management. Phases usually do not overlap.

There are two main issues relevant to the selection of a system development alternative

1. The maturity and completion of requirements;
2. Organizational factors impacting which process is appropriate for a given system.

Ill-defined or immature requirements are a barrier for a good definition of the system and the project scope, so more flexible alternatives dealing with these issues are needed.

In hierarchical organizations, a function-oriented organizational approach may be a barrier for implementing alternatives where requirements change and continuous delivery are the main system development drivers.

Next, we will describe the three main alternatives: sequential, incremental, and evolutionary.

2.4.1 Sequential Approach

In the sequential alternative, also known as waterfall, the development activities are performed in order, with minor overlap and with little or no iteration between them. Sequential alternative is shown in Figure 2.3.

The main characteristics of the sequential alternative are

- User needs are determined from the beginning;
- Requirements are defined;
- Work downstream should not begin until upstream uncertainties are resolved and major reviews (decision gates) have been satisfied;
- The full system is designed, built, and tested for its delivery at one point in time;
- Its strengths are that it is efficient and easy to verify;
- Its weakness is that it is difficult to cope with change and emerging requirements.

It is important to note that there are variations to the sequential model, such as the sequential version of the Vee model involving a sequential progression of plans, specifications, and products that are baselined and put under configuration management [14]. The Vee model represents system evolution from the perspective of decomposition (left side of the Vee) and integration activities (right side of the Vee).

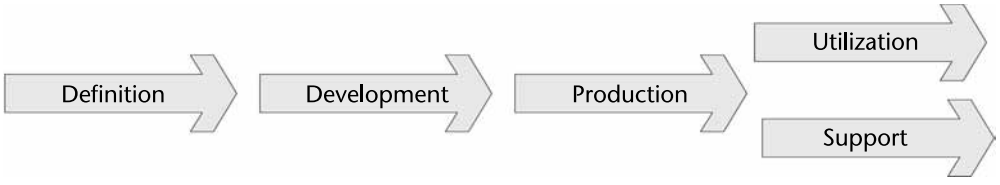


Figure 2.3 Sequential alternative for systems development.

2.4.2 Incremental Approach

In the incremental approach, user needs and architecture are defined at the beginning but the system is delivered in a set of increments or builds where the first increment incorporates a part of the total planned system capabilities, the next increment adds more planned capabilities, and so on, until the entire system is complete. The incremental approach is represented in Figure 2.4 below where we can see that definition is complete at the beginning but the following stages are based on the increments 1, 2, 3, and *n*.

The main characteristics of the incremental alternative are

- Initial stages produce specifications;
- The incremental approach implementation requires a sound build strategy based on the system architecture and the requirements to be implemented in each increment or build;
- Builds may include either integration of new system components or upgrading existing system components;
- Interface management is a big integration issue;
- Its strength is the scalability when the solution is stable;
- Its weaknesses are the difficulties with emergent requirements or rapid change architecture.

2.4.3 Evolutionary Approach

In the evolutionary approach the system also is developed in increments or builds but differs from the incremental approach in user needs definition and requirements completion at the initial stages. User needs and requirements are refined in each succeeding build. The evolutionary approach is represented in Figure 2.5 where the definition and development for the next increment or build are being performed in parallel with the production, utilization, and support of the current increment or build.

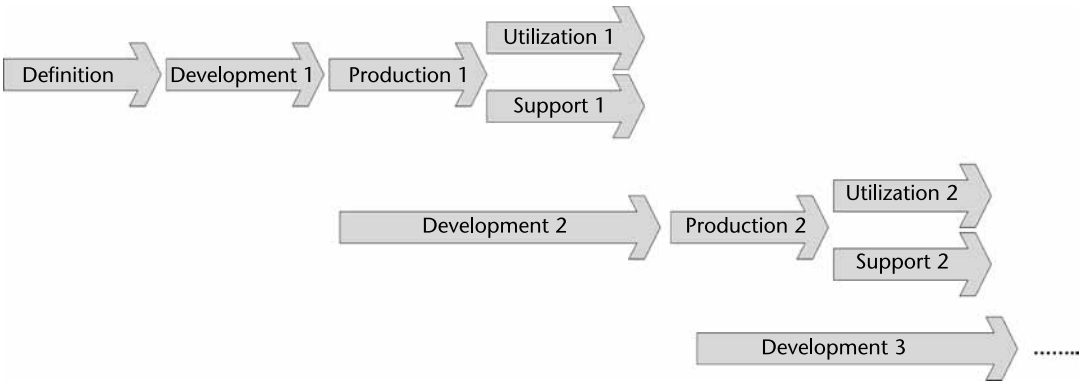


Figure 2.4 Incremental alternative for system development.

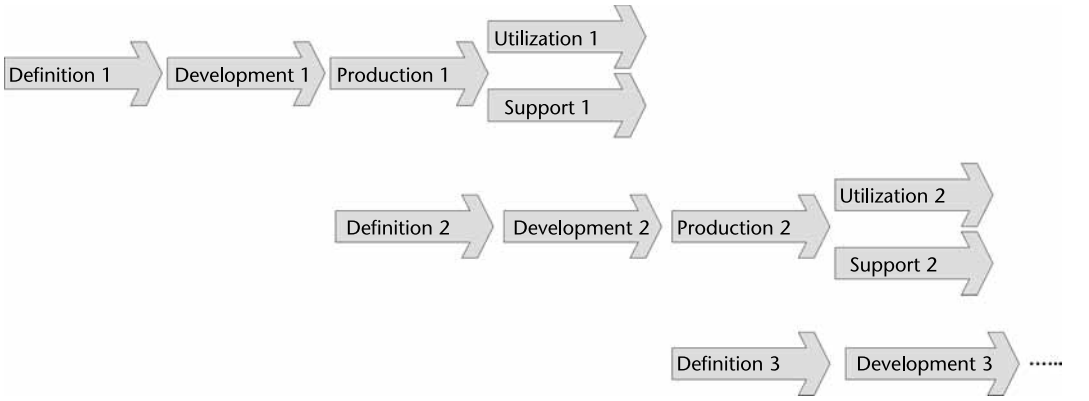


Figure 2.5 Evolutionary alternative for systems development.

It is important to note that the *SEBoK* describes two additional variants of the evolutionary approach described above. These variants are the evolutionary-opportunistic where increments definitions and development overlap, and evolutionary-concurrent where the plans and specifications for the next increment or build are being rebaselined concurrently with the development of the current increment or build and the production, utilization, and support of the previous increment or build [14].

The main characteristics of the evolutionary approach are

- Requirements are unclear from the beginning or their change rate is high;
- The customers wish to keep the system solution open for new technology;
- Continuous integration, verification, and validation of the evolving system;
- Its strength is adaptability to change;
- Its weakness is the scalability and systems engineering gaps for large systems.

The scalable spiral uses a risk approach for software intensive systems development based in the observe, orient, decide, and accept loop [15], and scaled agile approaches such as Scaled Agile Framework (SAFe) considering multiple Agile Release Trains and system suppliers [16], are variants of the evolutionary approach.

2.5 Summary

This chapter summarized the systems engineering discipline that is treated in diverse books [10–12,17] and handbooks [1, 2]. Important aspects are the need for systems engineering, the system life cycle, the traditional systems engineering tasks, and system development alternatives.

The topics described in this chapter are useful as well when applying a MBSE approach such as the ISE&PPOOA method described in this book. The main issue is that instead of using a document paradigm, a model paradigm approach is used.

2.6 Questions and Exercises

1 Identify the main functions of configuration management.
2. Compare two system development alternatives you think you can apply in your development projects.
3. What is emergence in a system? Give an example of emergence in a natural or man-made system.
4. What do we mean by interfaces management?
5. What is a smart product?

References

[1] Walden, D. D., et al., *Systems Engineering Handbook – A Guide for System Life Cycle Processes and Activitie*s, INCOSE-TP-2003-02-04. Hoboken, NJ: John Wiley & Sons, 2015.

[2] NASA, *Systems Engineering Handbook*, *NASA SP-2016-6105 Rev2*, Washington, DC: NASA, 2016.

[3] ISO, ISO/IEC/IEEE 15288:2008 *Systems and Software Engineering–System Life Cycle Processes, 2nd ed.*, Geneva: International Standards Organization, 2008.

[4] IEE,. *IEEE 1220 Standard for Application and Management of the Systems Engineering Process,* New York: Institute of Electrical and Electronic Engineers, 2005.

[5] Hitchins, D. K., *Advanced Systems Thinking, Engineering, and Management*, Norwood, MA: Artech House, 2003.

[6] Sillitto, H., *Architecting Systems: Concepts, Principles and Practices*, UK: College Publications, 2014.

[7] Sillitto, H., et al., "A Fresh Look at Systems Engineering–What Is It, How Should It Work?" Proceedings 28 *The Annual INCOSE International Symposium*, Washington, DC: July 7–12, 2018.

[8] ISO, ISO/IEC/IEEE *15288:2002 Systems and Software Engineering–System Life Cycle Processes, 1st ed.,* Geneva: International Standards Organization, 2002.

[9] Von Bertalanffy, L., *General System Theory,* New York: George Braziller, 1969.

[10] Kossiakoff, A., et al., *Systems Engineering Principles and Practice,* Hoboken, NJ: John Wiley & Sons, 2011.

[11] Shamieh, C., *Systems Engineering for Dummies,* Indianapolis, IN: John Wiley & Sons, 2012.

[12] Eisner, H., *Essentials of Project and Systems Engineering Management, 2nd ed.*, New York: John Wiley & Sons, 2002.

[13] ANSI /EIA, ANSI/EIA Std. 632, *Processes for Engineering a System*, Arlington, VA: Electronic Industries Alliance, 1999.

[14] BKCASE Editorial Board, 2017, *The Guide to the Systems Engineering Body of Knowledge (SEBoK),*R. J. Cloutier (editor in chief). Hoboken, NJ: The Trustees of the Stevens Institute of Technology.

[15] Boehm, B., and Lane J. A., "21st Century Processes for Acquiring 21st Century Software-Intensive Systems of Systems," *Cross Talk*., May 2006, pp 4–9.

[16] Leffingweel, D., et al., *SAFe Reference Guide,, Boston:* Pearson Education, Addison-Wesley, 2018.

[17] Blanchard, B. S. and Fabrycky, W. J., *Systems Engineering and Analysis*, Fifth Edition, Essex, England: Pearson Education, 2014.

CHAPTER 3

Model-Based Systems Engineering

This chapter provides a practical overview of MBSE, explaining its benefits (why), introducing basic concepts about modeling (what), and presenting the main aspects for MBSE (how): the modeling language and the method, and ending with a practical discussion of the tools for MBSE. In the first section, we present the MBSE approach by comparing it to the traditional, document-based approach to systems engineering, describing its benefits. Then, we take a step back to understand what modeling is in MBSE, what are the uses of models, and the role of modeling views. In the third section, we present the main aspects of MBSE, the modeling language, the method and the tools, and briefly comment on the existing alternatives for these three aspects.

3.1 Why Is Model-Based Engineering Necessary?

The traditional approach to systems engineering is document-based. During the life cycle of the system, multiple (hundreds!) of documents are created by the different stakeholders to capture the decisions and results of the engineering activities. Engineering is an iterative activity; in other words, changes are introduced in the design of the system when flaws are detected, and the diversity of stakeholders (different concerns, expertise, etc.) require a diversity of formats and corresponding documents to present the same information (i.e., copies). Therefore, the management of that disjointed documentation becomes critical if we need to ensure that the information is consistent, complete, and valid. Friedenthal et al. [1] provide a good summary of all the activities involved in the documentation-based approach to systems engineering. This document-centric approach is too expensive [2]: it is inefficient and time-consuming (redundancy updating the same information in multiple documents, graphics, and spreadsheets), making it difficult to maintain and reuse the design information, spread among multiple artifacts, and worse, it is prone to errors (for example, when missing to rename a system's component in all the documents where it is referred to) that can result in quality issues or major flaws in the delivered system.

MBSE addresses all these shortcomings, and it is defined as "the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases" [3]. Whereas in the traditional

approach models of the system are certainly used to support the different activities in the engineering process, the key difference is that in MBSE the primary and central artifact resulting from those activities is an integrated, consistent, and coherent system model.

MBSE has been widely accepted by industry as the solution to the problems described above with a traditional, document-centric approach to systems engineering. According to a survey in 2014 in France [4], MBSE companies in that country claim to have MBSE implemented, although with a very heterogeneous level of maturity.

MBSE has many potential benefits: it improves quality by reducing ambiguities and increasing design integrity and traceability, it increases productivity, automatizing parts of the process (e.g., document generation) and reducing errors, and it enhances knowledge transfer and reuse, among other things (see [1] for a detailed list). Moreover, metrics based on modeling can be defined to help assess the design quality and progress [1]. This requires having a good model in the first place, and then quality criteria can be established to determine how well the model meets its purpose (see Section 3.2).

3.2 What Is Modeling in MBSE?

In general, a model is a representation of entities that may be realized in the physical world. In systems engineering, a model is an abstraction of a system of interest that is developed for a purpose. More concretely, a model should address specific stakeholder concerns/needs and have a clear usefulness to engineering the system [5], which is what modeling means in MBSE.

3.2.1 Why Are Models Used?

The purpose for modeling a system must be defined in terms of how the various stakeholders intend to use the model across the system's life cycle. For example, the model can be used in the concept phase to represent a system concept, to specify and validate requirements by the requirements engineers by domain engineers to perform trade-off analysis of alternative implementations, or to estimate the system costs. Friedenthal et al. [1] provide a comprehensive list of model uses.

In MBSE, quality criteria can be established to assess that a model meets its purpose (e.g., understandability, completeness, or consistency) [1]. Friedenthal et al. [1] defines model validation as "the process for determining the extent to which the model accurately represents the domain of interest (e.g., the system and its environment) to meet the model's intended use". This aspect is treated in Chapter 12.

Let us now discuss two main uses of models in systems engineering: communication and simulation.

Model to communicate. The most basic requirement for a model in systems engineering is that it serves as a form of communication between the engineers and stakeholders involved in developing a system. The systems engineering team must be able to communicate the problem and the potential solutions, from gathering the requirements from a diversity of system users and owners, to specifying technical architectures that will cover complex capabilities [6]. From this perspective,

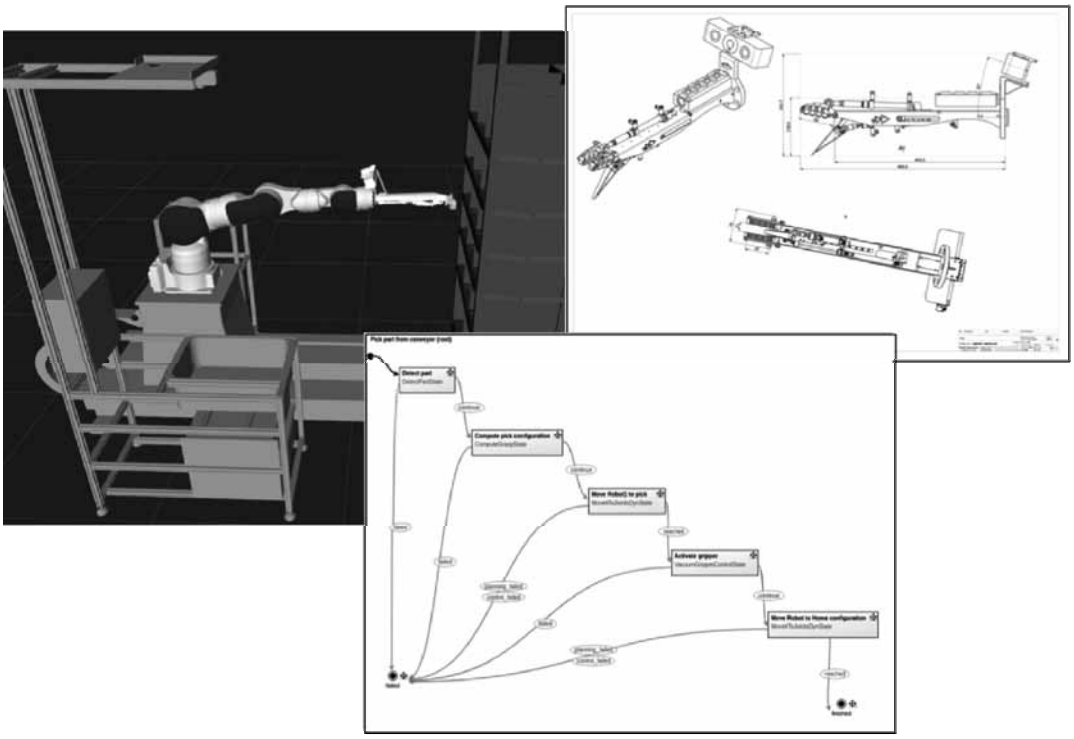


Figure 3.1 Different models involved in a robotic pick and place application. Clockwise from the left: CAD model of the robot cell, mechanical drawings of the robot gripper, and a state machine to define the high-level behavior.

a model is referred to as a *declarative model* when it communicates information about the system to humans. MBSE improves communication among the development team and other stakeholders thanks to the possibility of presenting and integrating views of the system from multiple perspectives, which provides a shared understanding of the system. Communication is precisely one of the main focuses and advantages of the ISE&PPOOA method for MBSE presented in this book.

Model to simulate. Another key advantage of MBSE is that the model is intended to be interpreted by machines or computer programs to simulate the system that it specifies [1]. The model is thus referred to as an *executive model* because it contains all the information necessary for a machine to execute it. For the model to be executable the modeling language needs to have semantics precise enough to allow it. In the case of the Systems Modeling Language (SysML) modeling language, which will be discussed in the next section, the standard has been extended with additional specifications to provide for the semantics needed for simulation.

Another example of model simulation is the graphical language used in Vitech's model-based systems engineering framework [6], which permits simulation through a discrete event simulator.

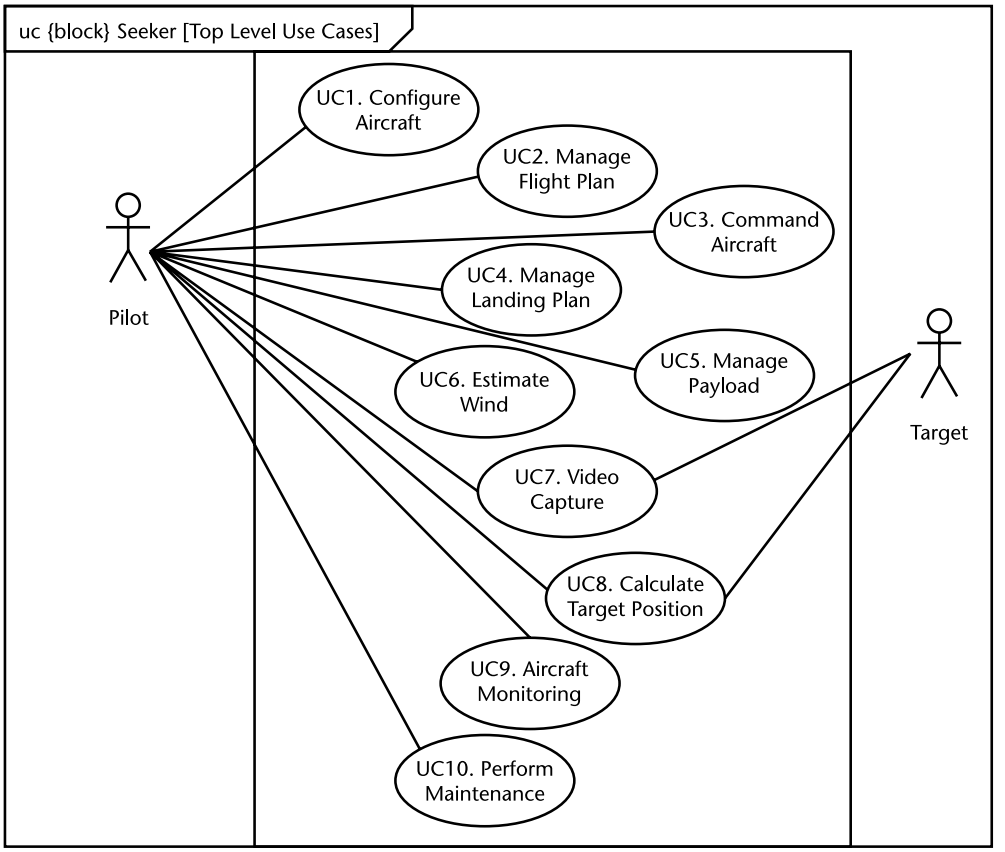


Figure 3.2 Example of model used to communicate: use case diagram to define the operational scenarios of an unmanned aerial vehicle (this example is further discussed in Chapter 8).

3.2.1 Models and Views

Views and viewpoints are fundamental in MBSE. Models of engineered systems are in general complex specifications so extensive that no single engineer can comprehend all its aspects. Moreover, different engineers in the development team and stakeholders involved in the system life cycle have different needs and backgrounds when examining the specifications in the system's model. In systems engineering, a *viewpoint* is a partitioning or restriction of concerns in a system. A viewpoint helps to manage complexity by addressing only those aspects relevant to associated concerns, so different viewpoints are used to address, for example, structural and behavioral aspects (see Figure 3.3). Other examples of viewpoints are safety, operational, manufacturing, or security.

ISO/IEC/IEEE 42010 [10] formalizes views and viewpoints. A viewpoint is a specification for an individual view, it "frame one or more concerns held by the stakeholders about the system of interest" and provides the conventions, rules, and languages for constructing, presenting, and analyzing views. Therefore, a *view* is a representation of a whole system from the perspective of a viewpoint and it specifies the model content that is presented to the stakeholder [1].

Models have views that are represented in *diagrams,* but the model is unique. In the MBSE model-centric approach, the data is captured once and represented

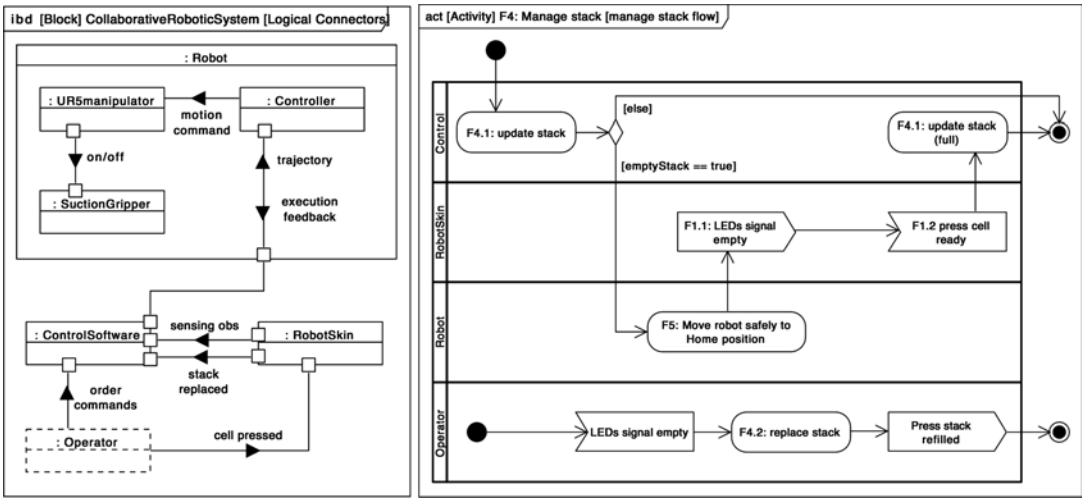


Figure 3.3 Two SysML diagrams showing two different viewpoints of the model of robotic collaborative application: (a) structural, in the Internal Block Diagram (IBD), and (b) behavioral in the activity diagram (ACT).

many times based on the defined viewpoint of the system description [5]. It is important to note that while a view can be constructed and be part of the system model, artifacts potentially produced from that view are not. For example, a document generated from a view is not part of the system model, while the view itself is [1].

3.3 Modeling Languages, Methods, and Tools for MBSE

Delligatti [2] clearly summarizes as the three pillars of MBSE all the elements that an engineering team needs to know to apply MBSE: a modeling *tool* to perform all the tasks in the processes prescribed by a *methodology* to design the system and capture this design in a central, integrated model that is expressed in a standard *modeling language*. In the following, we examine these three fundamental elements and provide a summary overview of current alternatives for each of them. However, it is not the authors' intention here to provide a complete survey of MBSE languages, methods, and tools. The references in this chapter are intended to provide the reader with additional resources where they can expand the summary information conveyed here.

3.3.1 Modeling Languages

SysML is the modeling language most widely used among MBSE practitioners and the one (with some extensions) employed by the ISE&PPOOA methodology that is used throughout this book. However, MBSE does not necessarily imply SysML, it is independent of it, and as proof we also present two other languages for MBSE.

3.3.1.1 SysML

The Object Management Group (OMG) defines theSysML [11] as "a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying

complex systems that may include hardware, software, information, personnel, procedures, and facilities." SysML is a graphical language that defines diagrams to represent system requirements, behavior, structure, and constraints on system properties (see Figure 3.4).

SysML originated as a joint initiative of the INCOSE and the OMG, which departed from the UML language for software systems[1], to create a modeling language for systems engineering applications, providing better semantics and new elements and diagrams to represent the essential aspects of a system's design and support the needs of the systems engineering activities.

In relation to the communication use of models, the MBSE language SysML includes specific Viewpoint and View elements that are consistent with the ISO-42010 [10] standard for visualizing SysML models. In SysML, Viewpoint is a specification of the conventions and rules for producing artifacts that offer customized presentations of information contained in a SysML model [1].

For the simulation use of the models, SysML incorporates Foundational UML (fUML) [7], which specifies foundational execution semantics for it, and the Precise Semantics of UML Composite Structures [8], which extends those semantics to the composite structures. The OMG has also adopted a complementary specification to fUML called the Action Language for Foundational UML or Alf [9], which is useful when describing the detailed behavior of activities. More details about model execution in SysML can be found in [1].

We include in Appendix A a summary of the SysML notation and diagrams used throughout this book, but there are many excellent references and resources available about SysML [2, 11] and how it can be used in a MBSE approach [1, 15], and these provide the reader with more in-depth discussion about the usage of SysML.

As already mentioned, despite SysML being the leading language, MBSE does not necessarily imply SysML. There are other modeling languages in other design domains, such as mechanical, software, and electronics, which are better suited for related activities in the system's life cycle. Moreover, there are alternative languages to SysML for model-based systems engineering that offer interesting properties,

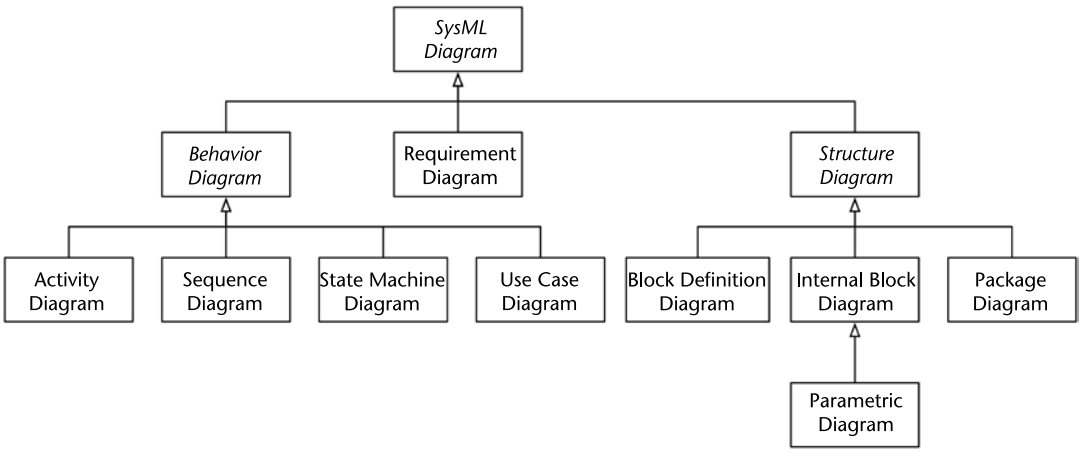


Figure 3.4 Taxonomy of the different diagrams in SysML.

1. SysML is defined as an extension of a subset of UML.

and here we mention two: Object Process Methodology (OPM) and Vitech's System Definition Language (SDL).

3.3.1.2 OPM

OPM is a conceptual modeling language and methodology for capturing knowledge and designing systems, that was created and developed by Dov Dori [12]. OPM has been adopted by ISO as standard ISO/PAS 19450. OPM is based on an ontology to specify the function, structure, and behavior of systems in multiple domains using formalized syntax and semantics. The language-building elements to describe any system are *object* (i.e., a thing that exists or may exist, physically or logically) and *process,* which transforms an object. Objects are in *states* at each point in time. OPM models can be expressed both graphically in Object-Process Diagrams (OPD) and verbally/textually in Object-Process Language (OPL). OPM's minimal[2] ontology addresses the challenge of reducing the complexity of the representation while maintaining the required accuracy and detail in the system's model.

3.3.1.3 Vitech

In *A Primer for Model-Based Systems Engineering* [6] Long and Scott describe the SDL that is used in Vitech's MBSE tools. Vitech's approach to MBSE is a more traditional one to systems engineering. The SDL is focused on being clear, unambiguous, and easily understandable (even simple mapping to common language is provided), avoiding the complexities of specialty language of domain experts.

3.3.2 MBSE Methodologies

A modeling language provides a means to encode the design information and description of the solution in a model. However, it does not specify how the design solution is created and modeled. A modeling method or methodology is needed to provide the road map or guidelines determining that the modeling efforts process a result that fulfills the intended purpose.

Friedenthal et al. [1] defines a method as "a set of related activities, techniques, conventions, representations, and artifacts that implement one or more processes and is generally supported by a set of tools." The OMG stands by the Estefan et al. characterization of a MBSE methodology "as the collection of related processes, methods, and tools used to support the discipline of systems engineering in a 'model-based' or 'model-driven' context" [13]. A particular method can thus be considered MBSE if it "implements all or part of the systems engineering process and produces a system model as one of its primary artifacts" [1].

The use of a suitable methodology when building a system with MBSE is as crucial as a recipe when cooking. The methodological pillar is still a wide-open challenge in MBSE (e.g., a survey in France has pointed out that most of the MBSE application is done without a formalized process or when formalized with a cus-

2. The OPM authors describe OPM's ontology as minimal because it can specify a large variety of domains using a reduced number of concepts.

tom process [4]), whereas "methods" is one of the main objectives the participants would like to see achieved in an MBSE working group.

The OMG maintains a listing of leading MBSE methodologies [14], including the ISE&PPOOA that is the main subject of this book, departing from the results of a survey in 2008 [13]. In Table 3.1 we present some of these MBSE methodologies, summarizing the process and views they support and the language they use. As mentioned before, MBSE can be integrated with a specific scope in the development process of particular kinds of systems, as exemplified by the Agile Architecture Framework and the State Analysis methodologies (Table 3.2).

3.3.3 MBSE Tools

Finally, an engineering team trained in a modeling language and a methodology still needs appropriate tools to apply MBSE efficiently and effectively to design a system.

The ISE&PPOOA methodology and the SysML language are specifications independent of particular implementations in the form of tools for MBSE. There are many tools available for MBSE (see Table 3.3), and we strongly encourage the interested reader to conduct an assessment to choose the tool better suited to her/his type of projects or products developed.[3] For example, some tools implement a full-blown MBSE approach, including some of the advanced features enabled by the implementation of the complete SysML specification, such as model validation, model exchange through the XML Metadata Interchange (XMI) standard, and so on, whereas other tools provide easier diagramming and model documentation capabilities, which is especially useful for lightweight approaches to MBSE, such as the one promoted by ISE&PPOOA, which focuses on the engineering process and key viewpoints. Here we do not try to make a complete listing of MBSE tools, but to provide some references as a starting point for the interested reader to explore the diversity of options out there.

Tools with a long tradition implementing the full-blown MBSE approach with SysML are those of IBM and NoMagic. These tools are complex, and enforce the OMG SysML standard, but precisely this fact and their support of other related standards allow them to offer powerful features for integration with requirement management tools (e.g., DOORS) or model simulation and validation. For example, MagicDraw from NoMagic has plug-ins available for architecture frameworks (DoDAF2, MODAF) or a simulation toolkit. The Cameo Systems Modeler is a new tool with selected and optimized capabilities from MagicDraw for systems engineering. IBM Rational Rhapsody has powerful support for state machine diagram execution and simulation, and plug-ins for integration of SysML parametric diagrams with other popular modeling tools such as Simulink. PTC Integrity Modeler includes similar features and it also incorporates a reviewer feature to improve quality and track the progress of your design. Capella is another tool with powerful MBSE support compatible with most engineering processes and based on its own methodology and approach to MBSE. Other tools, such as Visual Paradigm,[4] offer

3. The SysML online forum provides a good discussion of available tools at https://sysmlforum.com/sysml-tools/.
4. The diagrams in this book have been created with Visual Paradigm v15.1 and Microsoft Visio with the stencils provided by Pavel Hruby (http://softwarestencils.com/sysml/).

Table 3.1 Summary of Some of the MBSE Methodologies

OOSEM [1]	*Process*
The Object-Oriented Systems Engineering Method (OOSEM) is a top-down, scenario driven approach that leverages object-oriented concepts and other modeling techniques to help architect systems. Language: SysML Views: • *Structural:* BDD and IBD diagrams • *Behavioral:* activity diagrams • *Other:* requirement diagrams	1. Analyze stakeholder needs 2. Define system requirements 3. Define logical architecture 4. Optimize and evaluate alternatives 5. Synthesize physical architecture 6. Integrate and verify the system 7. Manage requirements traceability
SysMod [15,16]	*Process*
SysMod is a service-oriented approach where engineers first identify the system services. It is a pragmatic approach to model a system from analysis to design. Language: SysML Views: • *Structural:* BDD and IBD diagrams • *Behavioral:* sequence and state machine diagrams • *Other:* use cases diagrams	1. Determine requirements 2. Model the system context 3. Model use cases 4. Model domain knowledge 5. Create glossary 6. Realize use cases.
Harmony [18]	*Process*
Service request driven approach including the ease of passing off information to software engineering. An Agile version of Harmony developed by Bruce Powell Douglass is referenced in Chapter 12. Language: SysML and UML Views: • *Structural:* BDD and IBD diagrams • *Behavioral:* Sequence and state machine diagrams • *Other:* Use cases diagrams, activity diagrams	1. Requirements capture 2. Definition of the system use cases 3. System functional analysis 4. System architectural design 5. Subsystem architectural design
OPM [12]	*Process*
The Object Process Methodology by Dov Dori [12] OPM can be used to formally specify the function, structure, and behavior of artificial and natural systems in a large variety of domains. Language: OPM Views: • *Structural:* Object-Process Diagram • *Behavioral:* Object-Process Diagram	1. Identify the system function (process) – top-level OPD 2. Identifying the system's beneficiaries—added as object element to the OPD 3. Identify the relation between the process and the object as a transformation of the object or the state of the object 4. Proceed iteratively to refine the model by modeling the subprocesses for the process and the constituent objects for the object

more basic support of the SysML standard, but can be a reasonable choice to draw SysML-compliant diagrams. There are also free open-source tools, such as Papyrus (https://www.eclipse.org/papyrus). Although it is not mature enough to compete with commercial tools, it offers very interesting extensibility and customization capabilities thanks to the Eclipse Modeling framework it is based on.

Tool complexity has been identified as one of the three main challenges in MBSE, limiting its applicability in organizations [4]. It is important to emphasize

Table 3.2 Two Domain Specific MBSE Methodologies

Agile Architecture Framework [19]	*Process*
Elicitation of needs, specification of requirements, and illustration of the overall design of Command and Control (C^2) systems. Language: SysML Views: • *Behavioral:* Activity diagrams • *Other:* Use cases diagrams	1. Create operational system external view 2. Create operational system internal view 3. Create system external view 4. Create system internal view 5. Create technical system external view 6. Create technical system internal view
JPL State Analysis (SA) [20, 21]	*Process*
The State Analysis methodology defines a process for identifying and modeling the states of the physical system under control and the needed control capabilities in the context of Mission Data System (MDS) Control Architecture. Language: UML and mathematical representations, also SysML profile available Views: • *Structural:* BDD and IBD diagrams • *Behavioral:* Parametric diagrams • *Other*: Use cases	1. Identify needs for controlling the system 2. Identify state variables that capture the control needs 3. Define state models for the identified state variables–may uncover additional state variables 4. Identify measurements needed to estimate the state variables 5. Define measurement models for the identified measurements–may uncover additional state variables 6. Identify commands needed to control the state variables 7. Define command models for the identified commands 8. Iterate Until the Scope of the Mission Has Been Covered

Table 3.3 General Overview of Tools for MBSE

SysML compliance	*Diagraming*	*MBSE (nonSysML)*
Cameo, MagicDraw	Visual Paradigm	Vitech CORE
IBM Rhapsody	Visio	Capella
PTC Integrity Modeler		

that each engineering team and project have their special needs, and those needs should be carefully taken into account when choosing the MBSE tool they plan to use.

3.4 Summary

This chapter presented the MBSE approach to systems engineering. Its benefits and advantages over the traditional document-centric approach were discussed. The different uses of modeling and the important role of views and viewpoints in MBSE are analyzed. Finally, the three main elements for MBSE: language, methodology, and tools were discussed, and existing alternatives for the three of them were reviewed, in addition to a summary discussion of SysML, the main language for MBSE and the one used in this book. How to be agile using MBSE methods and tools is explained in Chapter 12.

3.5 Questions and Exercises

1. What is the difference between the document-centric approach to systems engineering and MBSE?
2. What are the three main elements that an engineering team needs to know to apply MBSE?
3. Which role do views and viewpoints fulfill in MBSE?

References

[1] Friedenthal, S., A. Moore, and R. Steiner, *A Practical Guide to SysML,* San Francisco: Morgan Kaufmann, 2015.

[2] Delligatti, L., *SysML Distilled: A Brief Guide to the Systems Modeling Language*, Boston: Addison-Wesley, 2014.

[3] International Council on Systems Engineering (INCOSE), *Systems Engineering Vision 2020*, Version 2.03, TP-2004-004-02, September 2007.

[4] Ferrogalini, M., D. Lesens, *2014 MBSE French Survey*, http://www.omgwiki.org/MBSE/doku.php?id=mbse:afis#mbse_french_survey.

[5] NASA, *Expanded Guidance for Systems Engineering, Volume 2: Crosscutting Topics, Special Topics, and Appendices*, technical report, National Aeronautics and Space Administration, 2016.

[6] Long, D., and Z. Scott, "A Primer for Model-Based Systems Engineering,"Vitech Corporation, 2011.

[7] Object Management Group, *Semantics of a Foundational Subset for Executable UML Models (FUML)*, http://www.omg.org/spec/FUML/.

[8] Object Management Group, *Precise Semantics of UML Composite Structures*, http://www.omg.org/spec/PSCS/.

[9] Object Management Group, *Action Language for Foundational UML (ALF),* http://www.omg.org/spec/ALF/.

[10] ISO/IEC/IEEE 42010:2011, *Systems and Software Engineering—Architectural Description,* The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) in collaboration with the Institute of Electrical and Electronic Engineers (IEEE), 2011.

[11] OMG, *Systems Modeling Language*, http://www.omgsysml.org/.

[12] Dori, D., *Model-Based Systems Engineering with OPM and SysML*, New York: Springer, 2016.

[13] Estefan, J. A., *Survey of Model-Based Systems Engineering (MBSE) Methodologies*, Rev. B, INCOSE Technical Publication, Document No. INCOSE-TD-2007-003-01, International Council on Systems Engineering, San Diego, CA: June 10, 2008.

[14] OMG MBSE Wiki, Methodology and Metrics. http://www.omgwiki.org/MBSE/doku.php?id=mbse:methodology.

[15] Weilkiens, *T., SYSMOD - The Systems Modeling Toolbox*, 2nd edition, MBSE4U Booklet Series, 2016.

[16] Weilkiens, T., J. Lamm, S. Roth, and M. Walker, *Model-Based System Architecture*, Hoboken, NJ: John Wiley & Sons, 2016.

[18] Hoffmann, H. P., *SysML-Based Systems Engineering Using a Model-Driven Development Approach*, white paper, Rational IBM, 2008, http://www.ccose.org/media/upload/SysML_based_systems_engineering-08.pdf.

[19] Hallberg, N., R. Andersson, and C. Ölvander, "Agile Architecture Framework for Model Driven Development of C2 Systems," *Systems Engineering*, Vol. 13, No. 2, 2010, pp. 175–185.

[20] Ingham, M. D., et al., "Engineering Complex Embedded Systems with State Analysis and the Mission Data System," *Journal of Aerospace Computing, Information, and Communication*, Vol. 2, No. 12, 2005, pp. 507–536.

[21] Wagner, A., et al., "An Ontology for State Analysis: Formalizing the mapping to SysML," *Proc. IEEE Aerospace Conference*, Big Sky, MT, 2012.

CHAPTER 4

The ISE&PPOOA Process

The core of a MBSE approach is the process because it helps the engineers to think about how to specify the problem and find a solution to be implemented as a complex product. This chapter describes the ISE&PPOOA process that promotes a MBSE approach that integrates systems engineering and software architecting. The main concerns of the process are how to deal with functional allocation, nonfunctional requirements implementation, and interfaces specification and design. The ISE&PPOOA process is described in this chapter by its conceptual model and the definition of its main steps and deliverables.

4.1 Integrating Systems Engineering and Software Architecting

Traditional systems engineering portrays system development as a top-down process based on the functional paradigm, where a function is considered as a transformation that consumes something and generates or transforms it into another thing. Physics conservation laws of mass, energy, and momentum are an important issue in engineering and in traditional systems engineering, but they are not the issues in software-intensive systems.

The use of software is increasing in aircraft, cars, medical devices, and home appliances. This means that more of the system's functionality is performed by software. This situation has changed the design paradigm from a functional to an object or component oriented approach at the software level. This change can create difficulties in the design of a system when it combines software with mechanical, electrical, or electronic parts.

Current engineering projects give documents and product deliverables as a general result, but there is an increasing level of product complexity due the integration of diverse building elements such as mechanical, electronics, and software parts that interact in complex ways. The difficulty, size, and multidisciplinary dimensions of these complex products make the application of systems engineering best practices more advisable, some of these practices are supported by the use of models, so it is called MBSE as described in Chapter 3.

The above issues are addressed by an integrated systems and software engineering process that combines best practices of traditional systems engineering with model-based approaches and the PPOOA architectural framework for component-based software engineering. In particular the methodological approach described

here deals with concerns regarding system functional decomposition and functional flow modeling, nonfunctional requirements implementation, interfaces specification, and software architectures implementing multiple threads of execution.

This chapter presents the integrated systems and software engineering process called ISE&PPOOA. The first author extended PPOOA, a software architecture framework based on software components and connectors he developed in the late nineties [1]. This extension maintains the usefulness of systems engineering functional decomposition, but uses the object-oriented analysis and design for the software subsystems. The structural views of the system and its software subsystems are different. They are represented using SysML structure diagrams for the system and extended UML class diagrams for its software subsystems.

The semantic gap between the system and software domains is supported by a bridge between the identified system functions and the software components based on the elaboration of a domain model, using UML class diagrams, of each software subsystem to be developed.

The use of design heuristics or tactics (see Chapter 6) to implement nonfunctional requirements is another main contribution of the ISE&PPOOA process presented here.

4.2 Challenges of the ISE&PPOOA Process

4.2.1 Implementing Nonfunctional Requirements

In general, nonfunctional requirements are those that specify criteria that can be used to judge the development or usage of a system rather than specific behaviors. This should be contrasted with functional requirements that define specific behavior or functions. Nonfunctional requirements act to constrain the solution space. Alternative names of nonfunctional requirements are quality attribute requirements or nonbehavioral requirements.

The engineering of nonfunctional requirements should consider the following issues:

- Before the proper selection and definition of nonfunctional requirements, it is required to clearly state the context of application for those requirements. This context is usually known as the quality framework. This framework is in fact a model, frequently represented as a tree, that considers a set of quality factors and subfactors that can be assessed through a set of criteria. These criteria may be quantified through a set of metrics [2].
- In decomposing a nonfunctional requirement, the systems engineer can choose to decompose its type (security, reliability, etc.) based on the quality framework, or its topic, which is where in the system, the nonfunctional requirement is applicable in either the whole or one of its parts. The soft goals approach provides examples where two nonfunctional requirements share the same topic (for example, customer account), but address different quality factors (for example, performance and security) [3].
- It is possible and should be taken into account that some nonfunctional requirements may be affected either positively or negatively at the same time;

for example, those related to system security and performance. These trade-offs are very important because they have an impact on the solution system architecture.

The collection of heuristics presented in Chapter 6 helps to identify architecture and design alternatives that are likely to meet nonfunctional requirements related to diverse quality attributes. These heuristics are neither new nor revolutionary. They are collected from the systems literature and tailored to be applied in the ISE&PPOOA process. Some of them represent design decisions currently made by system architects.

4.2.2 Dealing with Functional and Physical Interfaces

Missing or incorrect interfaces are a major cause of project costs overruns and system failures. It is important to consider that potential interfaces are not only those related to data, signals, or commands, but also to mass and energy transfers related to physical elements not implemented as software components.

A combination of MBSE SysML diagrams, text, and tables (N^2 charts) to describe system external and internal interfaces is used in the ISE&PPOOA methodological approach to MBSE. The rationale for this approach is summarized below.

As the systems engineer proceeds into system design he or she must consider the internal interfaces between subsystems. For both external and internal interfaces, early documentation is important to avoid rework and problems in later development.

Currently SysML block diagrams are used to represent interfaces, augmenting them with tool-generated textual descriptions of the interfaces using tabular forms or SysML requirements notation. When a description and the model element it relates to are shown in the same diagram, the relation may be depicted directly. SysML provides either the trace, refine, or satisfy relationship between the requirement and the related model element.

Systems engineers created N^2 charts to depict the items (data, energy, or mass) that are input and output of the system entities in the system architecture. An N^2 chart is a table with N+1 rows and N+1 columns. It records entities and interchanges between them. Entities are entered on the leading diagonal, while interchanges or interfaces appear in the other cells as appropriate. External inputs can optionally be shown in the row above the first entity on the diagonal, and external outputs can be shown in the right-hand column. In Table 4.1 the convention IC inputs in columns is used, so for example entities B and C receive inputs from entity A, entity D receives an input from entity C, and entity B receives an input from entity C. External inputs to the entities A, B, C, and D are shown in the first row of Table

Table 4.1 N^2 Chart

↓	↓	↓	↓	
Entity A	→↓	→↓		→
	Entity B			→
	↑←	Entity C	→↓	→
			Entity D	→

4.1. External output from the entities A, B, C, and D are shown as arrows in the right-hand column. These entities may represent either system functions or system physical blocks depending on whether we are developing the functional interfaces or the physical interfaces. Blank cells represent no interface between these entities.

The use of an N^2 chart for functional and/or physical architecture supplements the SysML diagrams due to the following benefits of using the N^2 chart:

- N^2 charts are very compact, allowing the overview of even the most complex systems.
- The interfaces tend to occur in pairs, potentially forming simple, reactive causal loops [4].
- N^2 charts and Design Structure Matrix (DSM)[1] are very helpful tools to allocate functions to subsystems or system parts so that there is minimal interaction among them [6]. The clustering of functions modifies the order of functions so the interactions among the functions are grouped close to the diagonal.

4.3 Conceptual Model for Systems Engineering with ISE&PPOOA

The aim of the proposed conceptual model represented in Figure 4.1 is to express the meaning of terms and concepts used in the ISE subprocess for systems engineering and to find the correct relationships between different concepts. The conceptual model may be considered as an ontology that attempts to clarify the meaning of various, usually ambiguous terms, and ensure that problems with different interpretations of the terms and concepts cannot occur when using the ISE&PPOOA process. The conceptual model is represented using a UML class diagram.

The main concepts and the relationships between them are represented in Figure 4.1. The conceptual model of the terms used in the PPOOA subprocess for software architecting was described in a previous paper [7].

The following list defines the concepts represented in Figure 4.1:

- *System:* A combination of interacting parts organized to achieve one or more stated purposes.
- *Part:* A building element of the system. Parts may be composite parts (e.g., a subsystem) containing other parts. A part depends on other parts.
- *Physical interface:* The description of the physical dependencies between system parts.
- *Environment:* All of the entities influencing a particular system. Systems do not operate alone. They interoperate in the natural and sociopolitical environments as well as with other systems external to themselves.

1. A DSM is a square matrix, representing linkages between the constitutive elements of a system, organization, or process. DSM matrices are categorized to building elements-based or architecture DSM, team-based or organization DSM, activity-based or schedule DSM, and parameter-based DSM [5].

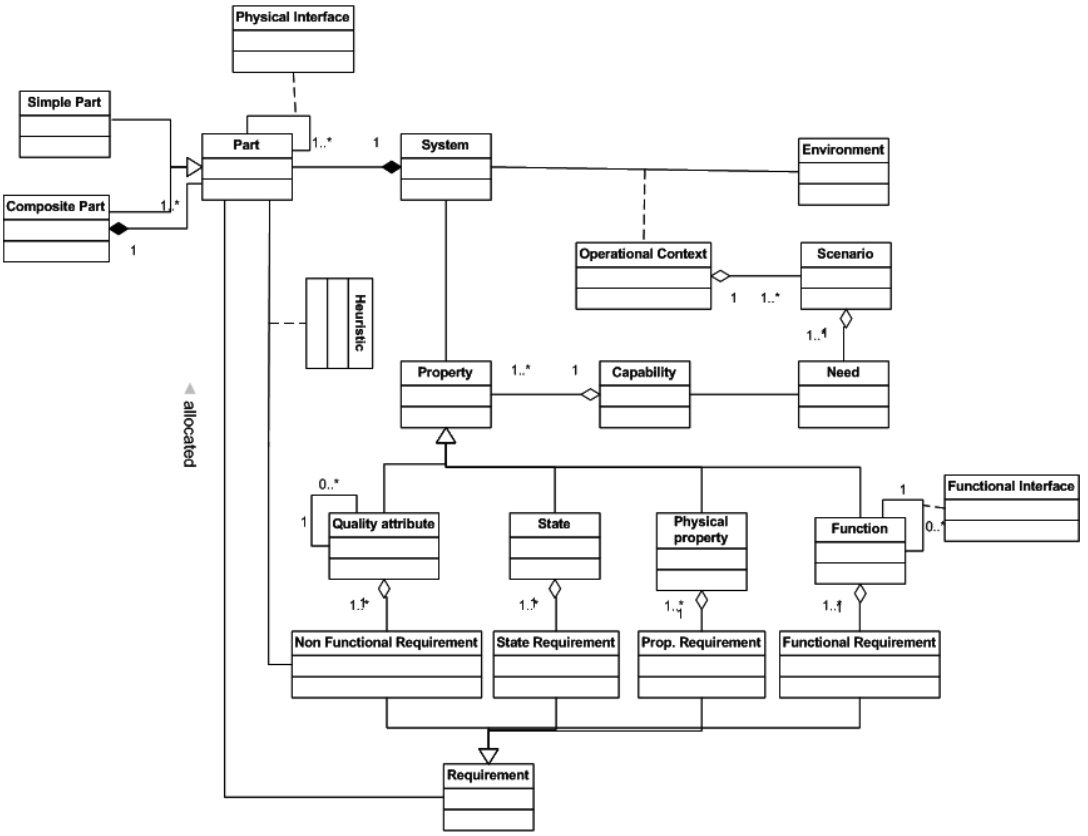


Figure 4.1 Conceptual model of ISE&PPOOA for systems engineering.

- *Operational context:* Abstract model of the operations of a specific system or group of systems.
- *Scenario:* Instance of how the system is used in specific circumstances. The system intended behaviors are described by the operational scenarios, where the preconditions, postconditions, and steps of each scenario are identified.
- *Need:* Here needs are defined in the answer to the question "What problem are we trying to solve with the new system operating in a particular environment?" Some authors call them operational needs or user needs.
- *Capability:* The ability to perform an effect under specified standards and conditions through combinations of means and ways to perform a set of tasks.
- *Property:* Observable, measurable, and reproducible characteristic of a system.
- *Function:* A transformation performed by the system that consumes mass, energy, or signals and generates new ones or transforms them.
- *Functional interface:* The description of the functional dependencies between system functions.
- *State:* The condition of a system defined by its current condition/configuration and the functionality provided.

- *Quality attribute:* A high-level aspect or characteristics of a system or part related mainly how it is to be developed or used.
- *Physical property:* Here physical properties are described by example so they are properties of mass, shape, color, temperature, and so on.
- *Requirement:* The statement of a property that a system or one of its parts should exhibit.

The conceptual model also represents some important relationships between the concepts by various lines and symbols. The following symbols represent relationships in the UML notation:

- Association is the relationship between two concepts that need to know each other. Represented as a line.
- Composition is the relationship between an element and its parts. Represented as a black diamond.
- Aggregation can occur when a concept is a collection or container of other concepts, but where the contained concepts do not have a strong life cycle dependency on the container. If the container is destroyed, its contents are not. Represented as a hollow diamond.
- Specialization is used when a concept is specialized in more specific ones. Represented as a hollow triangle.

For the sake of understandability, the relationships represented in Figure 4.1 are summarized below.

A system has parts that may be either simple or composite parts. A system interacts with the environment. These interactions are described by an operational context that models the interactions as a set of scenarios.

Based on the operational context and scenarios, the engineer translates the set of specific needs into a set of system capabilities that should be solution-independent. Each capability is a container of system properties that may be either system quality attributes, physical properties, states, or functions.

In contrast to functional requirements that are allocated to system parts, nonfunctional requirements implementation is essentially different. Nonfunctional requirements may be met by the application of design heuristics. For this reason a specific association is depicted between the nonfunctional requirement concept and the part concept as shown in Figure 4.1.

4.4 Dimensions and Main Steps of the ISE&PPOOA Process

The ISE&PPOOA process can be envisioned as the assembly of the three dimensions (left blocks in Figure 4.2) of software-intensive systems design. Each dimension has associated project deliverables, mainly models but complemented with textual and tabular representations.

The sections above describe the foundations of the ISE&PPOOA process presented in this section. The ISE&PPOOA process is an integrated systems and

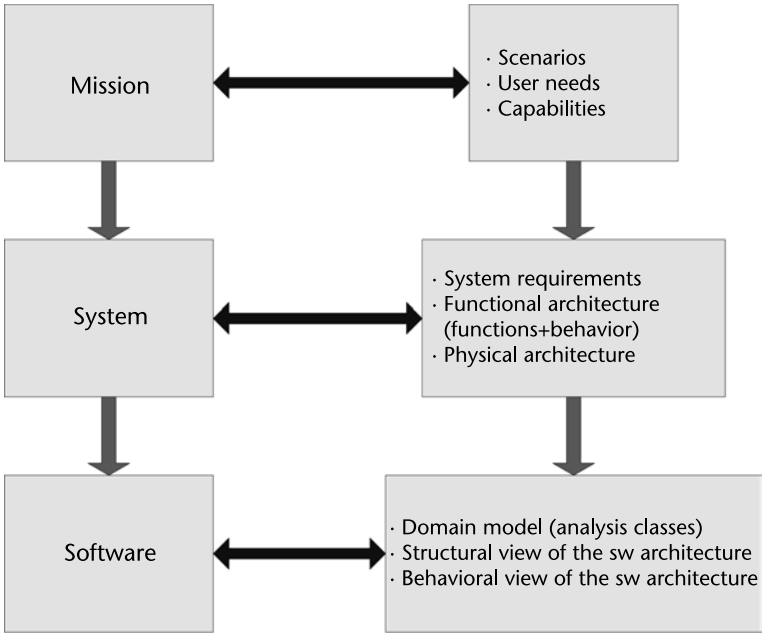


Figure 4.2 The three dimensions of the ISE&PPOOA process.

software engineering process. Traditional functional-based systems engineering is combined with MBSE using the functional paradigm to represent system behavior. This approach allows its integration with component-based software engineering using the PPOOA software architecting process.

4.4.1 The Systems Engineering Subprocess

The systems engineering part of the ISE&PPOOA process (shown in Figure 4.3 as an activity diagram), is described here in more detail. The software engineering part of the process is shown in Figure 4.4. It is described in a previous paper as well [8].

An important result, besides requirements, of the systems engineering subprocess of the ISE&PPOOA process is the creation of the functional and physical architectures of a system, identifying the constituent subsystems and their interfaces. The system may have subsystems that are software intensive and/or nonsoftware-intensive where physics conservation laws of mass, energy, and momentum are an important issue that should be considered when representing the system views.

The process presented in Figure 4.3 has four groups of steps that are performed sequentially for groups 1, 2, 3, and 4. Step 2 is split into two parallel or concurrent steps labeled 2a and 2b. Step groups 3 and 4 are executed iteratively as seen in Figure 4.3. For the sake of simplicity, Figure 4.3 does not include the inputs and outputs from each activity. The goal of each step, the final or intermediate deliverables produced, and profiles recommended to achieve it are described below.

1: Identify operational scenarios,

Goal: Identify the operational context of the system and describe its operational scenarios for the different modes of operation.

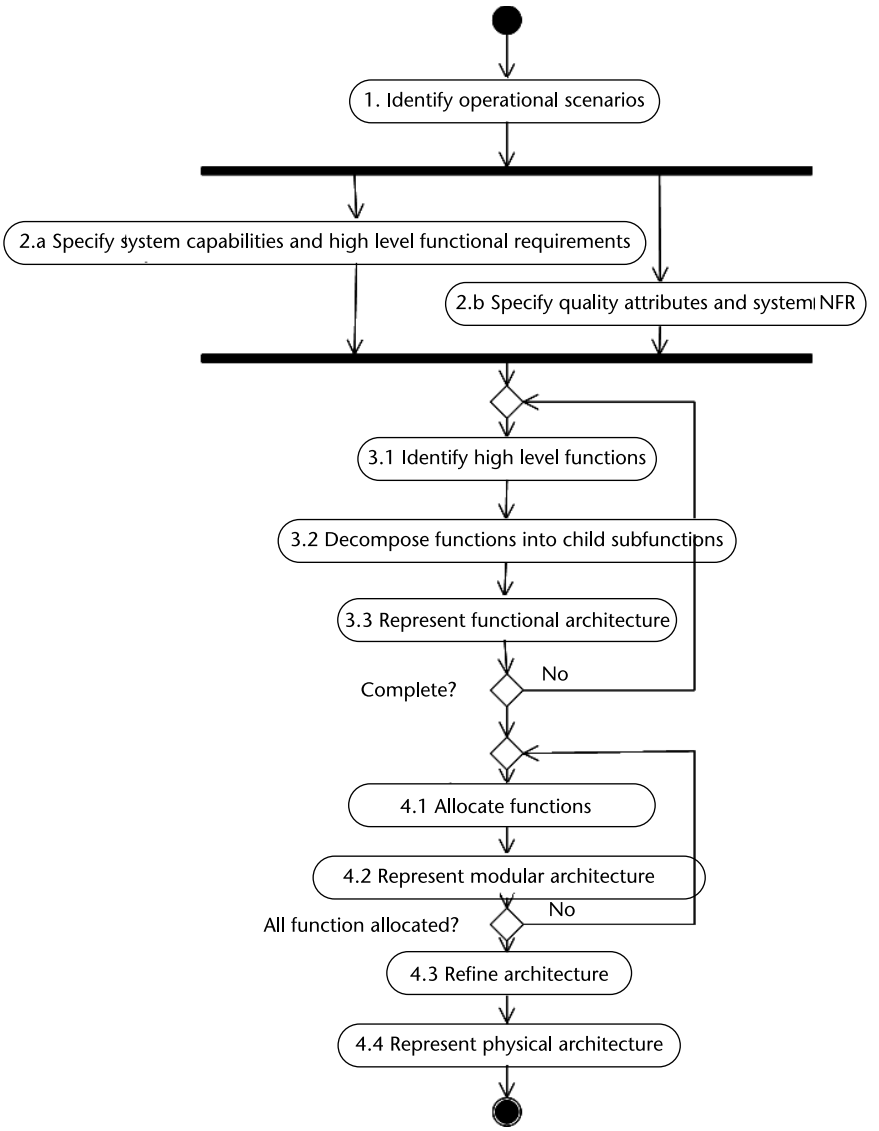


Figure 4.3 ISE Subprocess for systems engineering.

Deliverable: The system intended behaviors are described by the operational scenarios, where additionally to the preconditions, postconditions, and steps of each scenario, the needs are identified. These needs are the inputs for the later identification of the system capabilities and quality attributes in the following steps of the subprocess. Illustrative examples can be found in Chapter 8, Section 8.1, and Chapter 9, Section 9.1.

Profiles: Operational concept experts, requirements engineers, future system users, and other project stakeholders.

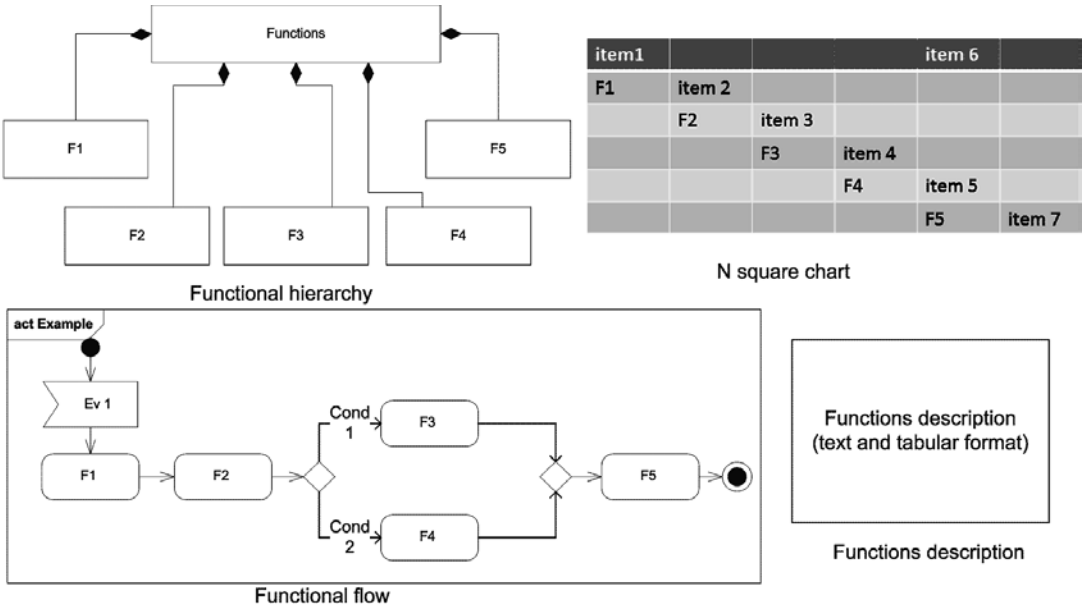


Figure 4.4 Functional architecture deliverables.

2a: Specify system capabilities and high-level functional requirements,

Goal: Transform scenarios and needs into a set of system capabilities and high-level system requirements.

Deliverable: The deliverable is the representation of capabilities with a hierarchical decomposition using the block definition diagram of SysML. System functional requirements specified in natural language but based on the hierarchical decomposition are obtained. System measures of effectiveness (MoEs) can be identified in this step. Illustrative examples can be found in Chapter 8, Table 8.2, and Chapter 9, Figure 9.1 and Table 9.4.

Profiles: Domain experts supported by systems engineers, customers, and other project stakeholders

2b: Specify quality attributes and system nonfunctional requirements (NFRs).

Goal: Transform scenarios and needs into a set of quality attributes for example reliability, availability, security, and others including the associated nonfunctional requirements.

Deliverable: In decomposing a nonfunctional requirement, the systems engineer can chose to decompose its type (security, reliability, etc.) based on a selected quality framework, or its topic, considering if it applies to the whole system or one of its parts. It is possible and should be taken into account that other nonfunctional requirements may be affected either positively or negatively at the same time. These trade-offs have an impact on the following architecting steps of the process. Illustrative examples can be found in Chapter 8, Subsection 8.2.2 and Chapter 9, Figure 9.2 and Subsections 9.1.3 and 9.2.2.

Profiles: Systems engineers with the collaboration of customers and experts in some quality domains for example security or safety.

3.1: Identify high-level functions.

Goal: Find the top-level functions of the system. Top-level functions are those functions used to organize the system functionality. They may be identified by previous knowledge of systems with similar capabilities or analyzing the main inputs and outputs of the system to be developed; that is, identifying what is called the system context in the systems engineering community.

Deliverable: The output is the top level of the functional hierarchy using a SysML block definition diagram. Illustrative examples are in Chapter 8, Figure 8.4, and Chapter 10, Figure 10.3.

Profiles: Systems engineers with the collaboration of customers and users.

3.2: Decompose functions into child subfunctions.

Goal: Build the functional hierarchy identifying the subfunctions of each high-level function of the system and continuing until the granularity of the subfunctions or children functions is appropriate for the allocation step (step 4.1), Therefore, step 3.2 is part of an iterative process of building the functional hierarchy.

Deliverable: The deliverable is the functional hierarchy using a SysML block definition diagram. Illustrative examples can be found in Chapter 8, Figures 8.5, 8.6, and 8.7, and in Chapter 9, Figure 9.9.

Profiles: Systems engineers with the collaboration of customers and users.

3.3: Represent functional architecture.

Goal: Represent the functional architecture identifying the functional hierarchy, functional flows or behavior, and functional interfaces.

Deliverable: The deliverable is the functional architecture representing the functional hierarchy using a SysML block definition diagram. This diagram is complemented with activity diagrams for the main system functional flows to represent behavior. An N^2 chart is used as an interface description where the main functional interfaces are identified. A textual description of the system functions is provided as well; see Figure 4.4 as a simple example of this deliverable content. Illustrative examples can be found in Chapter 8, Section 8.2, and Chapter 9, Section 9.2

Profiles: Systems engineers with the collaboration of customers and users.

4.1: Allocate functions.

Goal: Identify the building elements of the solution that implement each of the functions represented in step 3.3.

Deliverable: The building elements or physical components of the solution are identified. The heuristics of modularity described in Chapter 6 are applied here, choosing the solution elements so that they are as independent as possible; that is, solution elements with low external complexity (low coupling) and high internal complexity (high functional cohesion). Allocation may be represented in tabular

form or at the system blocks represented using SysML notation that are part of the deliverable of step 4.2. Illustrative examples can be found in Chapter 9, Figure 9.13.

Profiles: Systems architects.

4.2: Represent modular architecture.

Goal: The selection of the solution is based mainly on the clustering of functions to obtain a modular architecture.

Deliverable: The deliverable is the first version of the physical architecture. The modular architecture is represented by the system decomposition into subsystems and parts using a SysML block definition diagram. This diagram is complemented with SysML internal block diagrams representing the system physical blocks with either logical or physical connectors for each subsystem identified and activity and state diagrams for behavioral description as needed. A textual description of the system blocks is provided as well.

Profiles: Systems architects.

4.3: Refine architecture.

Goal: The modular architecture of the previous step is refined considering the implementation of nonfunctional or quality attributes requirements. Design heuristics are used for taking into account the system nonfunctional requirements. So the heuristics are the means of satisfying a nonfunctional requirement by manipulating some aspect of a quality attribute model through design decisions. Currently the heuristics presented in Chapter 6 are related to maintainability, safety, efficiency, and those related to resilience.

Deliverable: Each used heuristic may be documented with the template presented in Table 4.2. The heuristics collection is an asset that will be updated with the experience of the projects closed.

Trade studies may be performed to select the preferred physical architecture that optimizes the measures of effectiveness that may be defined in step 2a. See Chapter 11 for trade studies that may be used in the ISE&PPOOA approach to complement the heuristics application.

Profiles: Systems architects with the collaboration of customers and experts in some quality domains; for example, security or safety.

Table 4.2 Template for Project Heuristic Description

Aim	Main purpose of the heuristic used
Description	A description of the heuristic including the means used to meet the stated aim
Rationale	Claims for this heuristic that can assist in achieving the quality goal that is either maintainability, safety, efficiency, resilience, or other
Impact	Impact for the use of this heuristic regarding any assumption, cost, and other system or project issues
Side Effects	The side effects on other quality attributes
Patterns	Any architectural patterns that implement this heuristic
Related Heuristics	Alternative or complementary heuristics

4.4: Represent system physical architecture.

Goal: Represent the final architecture of the solution or refined physical architecture.

Deliverable: The deliverable is the refined physical architectures obtained after the application of the design heuristics. The refined physical architecture is represented by the system decomposition into subsystems and parts using a SysML block definition diagram. This diagram is complemented with SysML internal block diagrams representing the system physical blocks with either logical or physical connectors for each subsystem identified, and activity and state diagrams for behavioral description as needed. A tabular description of the system parts may be provided as well. See Figure 4.5 as a simple representation of the physical architecture deliverable content. Illustrative examples of the physical architecture are shown in Chapter 8, Section 8.3, and Chapter 9, Section 9.3.

Profiles: Systems architects with the collaboration of customers and experts in some quality domains; for example, security or safety.

4.4.2 The Software Architecting Subprocess

The systems engineering subprocess described above is applied for each nonsoftware-intensive subsystem and its parts. For the software-intensive subsystems, the PPOOA subprocess, represented as an activity diagram in Figure 4.6, is used.

The integration between the systems engineering modeling process ISE and the PPOOA software architecture modeling process is achieved by using a responsibility-driven software analysis approach supported by class responsibility collaborator (CRC) cards, a technique proposed by the object-oriented community [9]. The

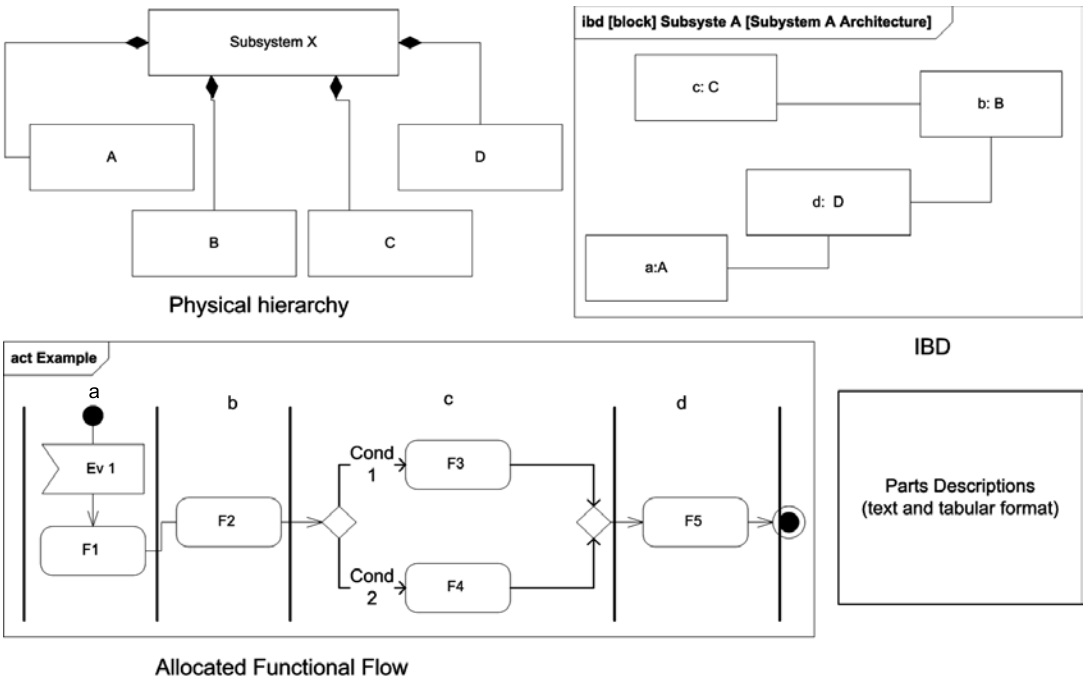


Figure 4.5 Physical architecture deliverables.

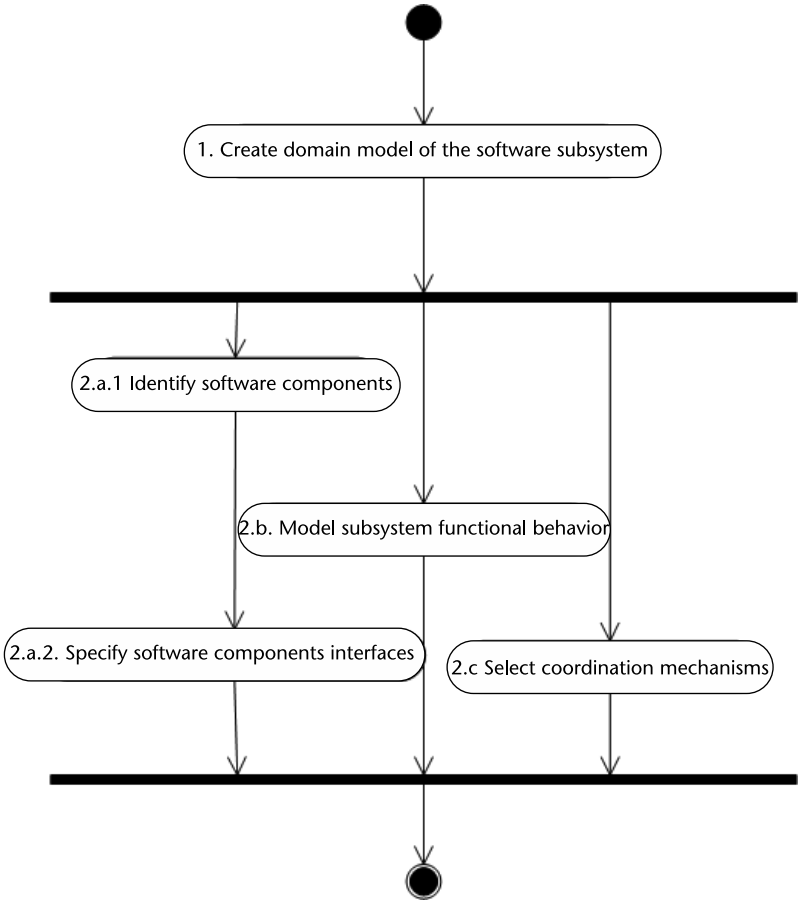


Figure 4.6 PPOOA subprocess.

guiding principle of responsibility driven modeling is that the central issues surrounding how a software subsystem is partitioned can be captured by asking what the responsibility of each part has toward the subsystem. This question manifests issues of function, function distribution, communication, locality of control, robustness with respect to change, and so on.

PPOOA is more than a process, it is an architecture framework oriented to design the software of real-time systems. PPOOA uses two viewpoints: structural, using UML class diagrams extended with PPOOA stereotypes, and behavioral, supported by UML activity diagrams.

The PPOOA vocabulary of building elements is composed of components and coordination mechanisms. A component is a conceptual computation entity that performs some responsibility and may provide and require interfaces to other components. Generally, it may be decomposed into small granularity parts. A coordination mechanism provides the capabilities to synchronize or communicate components of the software architecture. Synchronization is the blocking of a software process until some specified condition is met. Communication is the transfer of information between the components of the software architecture [7].

The building elements supported by the PPOOA architecture framework are

- *Algorithmic component:* These are elements of the software architecture that perform calculations or transform data from one type to another but separated from its structural abstraction. Data classification components, Unix filters, or data processing algorithms typically represent them. The algorithm component in PPOOA is similar to the utility defined in the UML metamodel but the algorithmic component may have instances.
- *Domain component:* This is an element of the software architecture that responds directly to the modeled problem. This component does not depend on hardware or user interfaces. An instance of a domain component corresponds to a software object in traditional object-oriented design.
- *Process:* This is a building element of the software architecture that implements an activity or group of activities that can be executed at the same time as other processes. Its execution can be scheduled. The PPOOA architecture framework supports two different types of software processes: periodic and aperiodic.
- *Structure:* A structure is a component that denotes an object or class of objects characterized as an abstract state machine or an abstract data type. Examples are stack, queue, list, ring, and others.
- *Controller object:* The controller object is responsible for initiating and managing directly a group of activities that can be repetitive, alternative, or parallel. These activities can be executed depending on a set of events or conditions.
- Coordination mechanisms supported by PPOOA are semaphore, bounded buffer, mailbox, transporter, and rendezvous that are implemented by popular real-time operating systems.

PPOOA as an architecting process is essentially an iterative process that is split into major steps and minor steps. Minor steps are not represented in Figure 4.6. Major steps are those that follow the identification of the software components of the software subsystem, their interfaces, and the main concurrent flows of activities that the software subsystem implements.

The crucial step here is step 1 of Figure 4.6, the creation of the domain model of the software subsystem. A domain model yields a more precise specification of software requirements than project team has in the results from earlier system requirements. It is described using more formalism than textual descriptions; for example, UML class diagrams, and can be used to reason about the internal workings of the software subsystem. A domain model is an essential input when the subsystem is shaped in software architecture, design, and implementation.

CRC cards (see Table 4.3) are index cards, one for each domain model class, on which the responsibilities of the class are briefly documented and a list of classes collaborated with to achieve those responsibilities. Using a CRC card keeps the complexity at a minimum. It focuses the designer on the essentials of the class and prevents him/her from getting into its details and inner workings at a time when such detail is probably counterproductive. It also forces the architect to refrain from giving the class too many responsibilities.

Table 4.3 Template for CRC Card

Class Name
Software class identity
Class Responsibilities
• What does the class know? • What does the class do?
Class Collaboration
• Other classes the class is collaborating to achieve its responsibilities

Text-based CRC cards are limited in their ability to record collaborations in a comprehensive way. Graphical representations of the domain model using UML diagrams better illustrate the connections between objects and the overall context.

The software architecting part of the process presented in Figure 4.6 has four main steps that are performed in parallel and iteratively. For the sake of simplicity, Figure 4.6 only represents the process steps, avoiding the representation of the deliverables that are input to some step and output from it.

The goal of each step, the deliverables produced and profiles required to achieve it, are summarized below.

Step 2a.1: Identify software components.

Goal: Create the initial set of the system conceptual components. A software conceptual component is a computation entity that performs an assigned responsibility, provides interfaces to other components, and may require some interfaces from other components. Examples of types of components provided by PPOOA framework are given above.

Deliverable: The structural view of the architecture is represented by a PPOOA architecture diagram that is used instead of the UML component diagram to describe the software architecture, but it maintains some similarities with the UML class diagram. The architecture diagram focuses on software components and coordination mechanisms representation and the dependence relationships between them. Composition relations between components are represented as well. Two types of interactions are represented: synchronous and asynchronous. The latter is implemented in the software architecture by the usage of coordination mechanisms represented as UML stereotypes and provided by the PPOOA framework as described above.

Figure 4.7 shows an example of using PPOOA for the structural view of the architecture of a software controller for three robotic arms. Work orders are decomposed into subwork orders that ultimately result in communicating microcoordinates to an axis computer that contains the device drivers responsible for actual movement of the robotic arms.

The three planner processes presented in the figure carry out planning activities: primarily, they receive work orders and create plans. Each plan produced by each planner process results in a sequence of subwork orders that are asynchronously passed to the generation of microcoordinates process. Each process planner-generation of microcoordinates pair is associated with its own robot arm. Five buf-

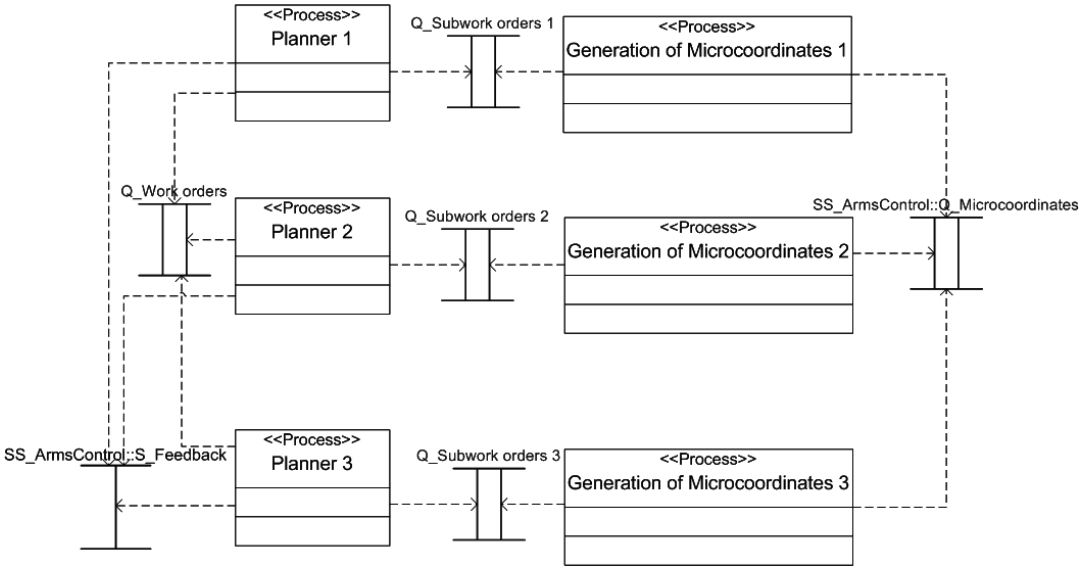


Figure 4.7 Structural view of the planner subsystem.

fers and one semaphore used for coordination purposes are represented in Figure 4.7 as well.

PPOOA building elements allow a complete representation of the static view of the software architecture including processes, passive components, and the buffers or queues used to communicate software processes.

Chapter 9, Figure 9.20, shows an illustrative architecture diagram with the software components identified for the allocation of the software functions of a collaborative robot.

Profiles: Software architect and software engineers.

Step 2a.2: Specify software components interfaces.

Goal: Group the identified component operations in interfaces that are meaningful to the architecture solution.

Deliverable: Component interfaces are described in textual and tabular format. A component interface is more important for an architectural perspective than the way the interface is realized or implemented. It should be possible to replace one component with another of an equivalent interface without affecting the architecture.

Profiles: Software engineers.

Step 2.b: Model subsystem functional behavior.

Goal: Describe the dynamic or behavioral view of the software subsystem architecture. Describing the static structure of a system can reveal what it contains and how its elements are related, but it does not explain how these elements cooperate to provide the software subsystem functionality.

Deliverable: Behavior view is complementary to structure. To represent system behavior this methodological approach proposes to model it using UML/SysML activity diagrams as used in the systems engineering subprocess of ISE&PPOOA. Each event response is represented by a causal flow of activities (CFA) modeled as an activity diagram. A CFA is a cause-effect chain of activities that cuts across different building elements of the software architecture. This chain progresses through time and executes the activities provided by the software architecture components until it gets to an ending point.

Figure 4.8 is an example of a CFA representing one causal flow of the software controller for the three robotic arms whose structural view was presented in Figure 4.7. The activities allocation using the UML concept of partition (swimlanes) is an important contribution of the PPOOA architecture framework for the representation of the mapping of the CFA activities to the components and coordination mechanisms of the system architecture. This allocation is a critical issue in the engineering process since it permits the assessment of the different design decisions.

Chapter 9, Figures 9.21, 9.22, and 9.23 show some CFAs examples of the software behavior of a collaborative robot.

Profiles: Software architect, software, and systems engineers.

Step 2c: Select coordination mechanisms.

Goal: Identify the more suitable coordination mechanism (semaphore, buffer, or other) to implement asynchronous interaction between the software components of the software architecture.

Deliverable: Coordination mechanisms are used for implementing asynchronous interaction between software components. They are represented by stereotyped icons in the structural view and swimlanes in the behavioral view of the software architecture. Figure 4.7 shows the use of one semaphore and five buffers. Chapter 9, Figure 9.20 shows the coordination mechanisms used in the software architecture of the collaborative robot presented as an example.

Profiles: Software engineers coordinated by the software architect.

4.5 An Extension of the Process for Energy Efficiency Concerns

The ISE&PPOOA/energy process combines model-based systems engineering with the matter and energy balance concerns used for energy efficiency involving several activities integrated in a single engineering process, such as the definition of a context for the efficiency analysis, the modeling of the system, either a complete plant, or one of its processes to be analyzed, and the resolution of the matter and energy balance equations at the appropriate level of abstraction represented by the SysML block definition diagrams.

This approach allows a study of alternatives that can range from the tuning of operating parameters to replacing equipment for a more efficient one but that is equivalent in functionality and interfaces with other elements of the plant, which is reflected in the SysML internal block definition diagrams obtained by the application of model-based systems engineering.

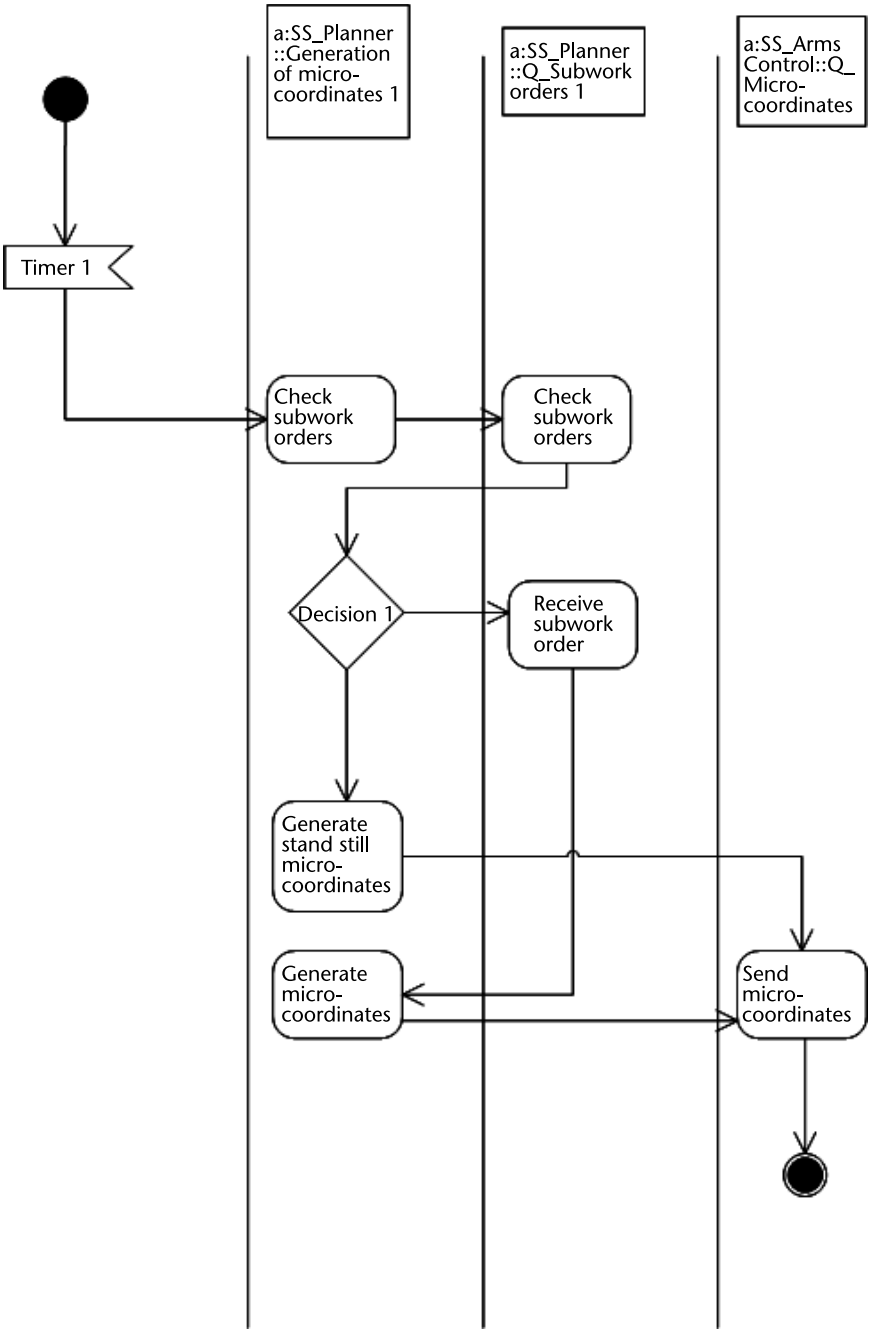


Figure 4.8 CFA execute subwork order.

The plant models are built with SysML standard notation that supports their wide understanding. The process used allows to describe the plant with the detail level needed for an efficiency analysis that is more rigorous than sketches but less detailed than the engineering deliverables for building the plant.

Therfore, the application of MBSE for energy efficiency of process plants requires the usage of structural diagrams to represent the industrial plant and its energy-related processes. The SysML block definition diagram is used here to represent the plant and its constituent parts hierarchically. The SysML internal block diagrams are used here to show the internal structure of a complex block of the plant that is its parts, their interfaces, as well as the connectors between them.

Another diagram, which is supported by SysML standard notation and proposed here, is the parametric diagram. Parametric diagrams are used to show relationships or equations and value properties of the block that make up the industrial plant.

The steps of this method, which we call ISE&PPOOA/energy, are shown in Figure 4.9 and described below, taking into account that, similar to a reengineering project, it is being applied to an already built system.

- *Step 1*. Identify the context and boundaries of the system to be analyzed;
- *Step 2*. Obtain the functional and physical architecture diagrams of the system of interest (the complete plant or one of its processes), using SysML notation and according to the ISE&PPOOA process described in Section 4.4 and Figure 4.3;
- *Step 3*. Identify the main flows of matter and energy between identifiable system blocks;
- *Step 4*. Detail the equations, graphs, tables, and correlations that determine the unknown values of the major flows of matter and energy;
- *Step 5*. Determine according to the degrees of freedom of the system to evaluate, if it is possible to solve it;
- *Step 6*. Solve the equations and correlations with the necessary computer tools.

Sections 10.2 and 10.3 of Chapter 10 are illustrative examples of the functional and physical architecture of the steam generation process of a coal power plant.

Briefly, it may be said that the above steps involve the identification of the system (either a complete plant or one of its processes) to be analyzed and the representation of the system architecture with system modeling techniques, in particular using the block definition diagrams and the internal block diagrams supported by the SysML standard. Steps 3 and 4 are performed by identifying the main flows of matter and energy using the information of interfaces and connectors found in the internal block diagrams that model the physical architecture of the system. For each flow what is known and not known is determined.

In step 4 the equations, equalities, and correlations are identified to obtain the unknown values in each one of the flows. Also, if necessary, process restrictions can be associated with some of the variables such as maximum temperature or pressure values. SysML constraint blocks are used to show how the properties associated with the flows of matter and energy are constrained [10].

A constraint block encapsulates a constraint on the properties of a physical block of the industrial plant, so constraint blocks are used here to define the

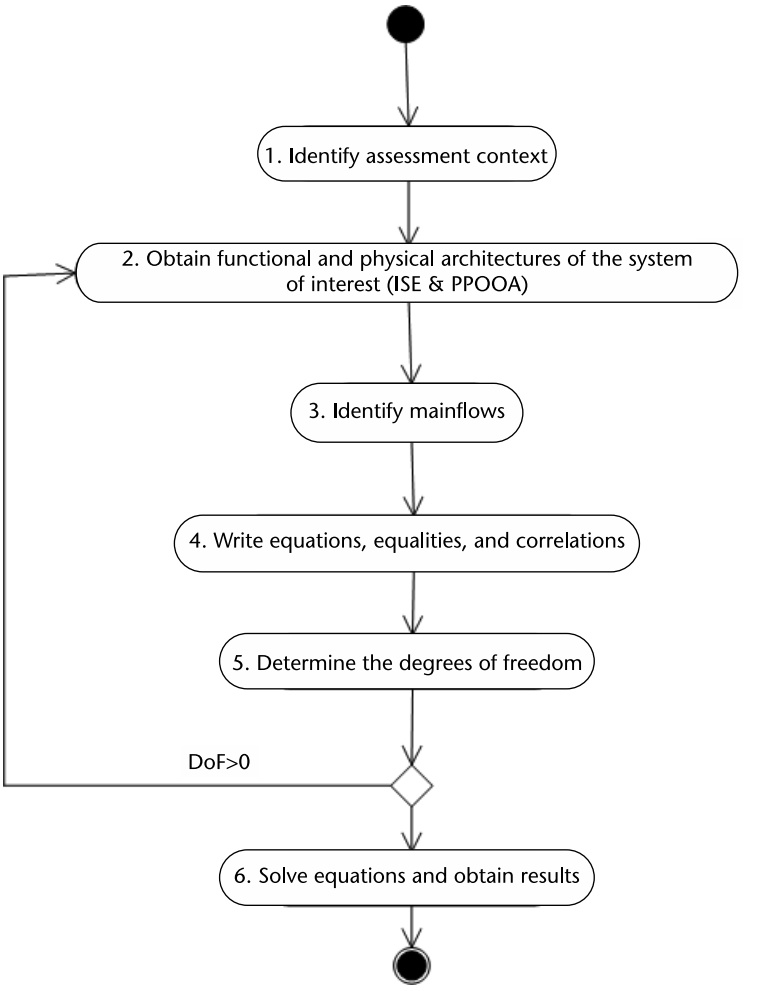


Figure 4.9 ISE&PPOOA/energy engineering process.

equations, equalities, and correlations related to the flows of matter and energy. Engineers can compose complex constraint blocks from existing constraint blocks on a block definition diagram. These diagrams are used to define constraint blocks in a similar way to which they are used to define the physical blocks of the plant.

Step 5 establishes whether the number of equations and correlations identified in step 4 are sufficient to determine the dependent variables. In order to have a unique solution, the analyzed system should have zero degrees of freedom; that is, the number of variables, including the quantities of matter and energy balances, should be equal to the number of equations combined with any other correlation used. If the number of degrees of freedom is positive, the system cannot be solved, forcing back to step 2 and performing logical groupings of physical blocks (see plant equipment) to reduce the number of dependent variables. The systemic approach based on hierarchies and blocks is very useful for this as will be seen in the example of the steam generation process of a coal power plant in Chapter 10.

Finally in step 6 the equations and correlations are solved using the appropriate computer tools.

4.6 Summary

This chapter presented the ISE&PPOOA process. This process integrates MBSE with software architecting so it can be applied to software-intensive systems or any system containing software subsystems.

The foundations of the ISE&PPOOA process discussed in this chapter are:

- The functional architecture that is independent of the technical solutions and is represented by hierarchies of functions, functional flows, and functional interfaces.
- Dealing with nonfunctional requirements that are implemented using design heuristics.
- The physical architecture that is represented by hierarchies of building elements and SysML internal block diagrams. The physical architecture is created and refined in several steps from a modular architecture to a refined architecture implementing the design heuristics that meet the nonfunctional system requirements.
- The bridge from the systems engineering model to the software architecture is the domain model of the software subsystem. The domain model identifies the particular software subsystem classes and describes them using the CRC template.
- PPOOA architecture framework is used to model the software architecture.

The ISE&PPOOA process is extended to deal with energy efficiency issues related to industrial plants. This energy efficiency engineering process is called ISE&PPOOA/energy and is based on the combination of the systems engineering models with the equations and correlations representing the balances of mass and energy.

4.7 Questions and Exercises

1. What is the difference between a need and a requirement?
2. What do we mean by a modular architecture?
3. What do we need to use heuristics?
4. What is a domain model?
5. What is a software coordination mechanism in the PPOOA architecture framework?
6. What happens if the degrees of freedom in an energy efficiency problem are greater than zero?

References

[1] Fernandez, J. L., "An Architectural Style for Object-Oriented Real-Time Systems," *Proc. 5th International Conference on Software Reuse*, Victoria, Canada, June 2–5, 1998.

[2] Firesmith, D., "Using Quality Models to Engineer Quality Requirements," *Journal of Object Technology 2*, Vol. 25, 2003, pp. 67-75.

[3] Chung, L., B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering,* Norwell, MA: Kluwer Academic Publishers, 2000.

[4] Hitchins, D. K., *Advanced Systems Thinking, Engineering, and Management*, Norwood, MA: Artech House, 2003.

[5] Eppinger S. D., and T. R. Browning, *Design Structure Matrix Methods and Applications*, Cambridge, MA: MIT Press, 2012.

[6] Bustna, T., and J.Z. Ben-Asher, "How Many Systems Are There? - Using the N^2 Method for Systems Partitioning," *Systems Engineering,* Vol, 8, No. 2, 2005, pp. 109–118.

[7] Fernandez-Sanchez, J. L., and A. Monzon, "Une Extension d`UML pour les Architectures à base de Composants Temps Réel," *Genie Logiciel*, Vol. 60, 2002, pp. 10–17.

[8] Fernandez-Sanchez, J. L., and B. J. Mason, "A Process for Architecting Real-Time Systems," *Proc. 15th International Conference on Software & Systems Engineering and their Applications*, Paris, France, December 3–5, 2002.

[9] Beck, K., and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *Proc. OOPSLA Conference on Object-Oriented Programming Systems, Languages and Applications*, New Orleans, LA:, October 3–6, 1989.

[10] Friedenthal, S., A. Moore, and R. Steiner, *A Practical Guide to SysML, The Systems Modeling Language*, Burlington, MA: Morgan Kaufmann, 2008.

CHAPTER 5

Functional Architecture

This chapter describes the system functional architecture as the most critical model deliverable of the ISE&PPOOA methodology. Functional architecture is a representation of the system behavior (what the system does) that is independent of technology solutions so it is more stable in time that the physical architecture described in Chapter 7. The functional model is the source for obtaining new functional requirements for the system's levels as well.

5.1 The Importance of Functional Architecture

5.1.1 Functional Architecture in Systems Engineering

Traditional systems engineering portrays system development as a top-down process relying on the functional paradigm, where the required system functions (what the system does) are allocated to the physical elements (the parts) that perform them.

This allocation of functions to physical elements, including human beings as well, may produce diverse solutions but these diverse solutions should always implement the functionalities previously identified. This approach was not new since it was proposed by the architect Sullivan for building architecting, where he recommended the following prescription "form follows function." Therefore, following Sullivan, the design of buildings should be based on the functionality they were intended to satisfy.

The functional analysis approach has been applied to the development of many defense systems since 1950 and is still in practice by diverse companies in aerospace, automotive, and ship design applications where functional models are used to obtain functional requirements and to facilitate requirements flowdown.

Following the above approach requires the functional architecture of the system as the first architecture to be modeled before modeling the solution as a physical architecture. This functional architecture may be represented by diverse diagrams and textual descriptions that are described below.

Two main models are the basis of MBSE: the functional model, here named the functional architecture, and the structural model, here named the physical model, as described in Chapter 7. Some object-oriented system modeling approaches promote the system structure identification and modeling before the functional and behavioral modeling. These approaches may miss results of the form follows function

architecting prescription making the traceability of the functional requirements to the physical components of the system difficult.

As defended by Carson and Sheeley, having a functional architecture of the system enables the validation of its functionality and performance with respect to the operational needs, the decomposition and allocation of subfunctions to physical entities of the system architecture, and the identification and operation of interfaces [1].

The functional architecture that is a representation of the problem space (what the system does), should be at the highest levels of its hierarchy, independent of the technical solutions. This characteristic makes the functional architecture of the system more permanent than the physical architecture that is more dependent on the technology evolution. Therefore, the functional architecture of a system can be reused for similar systems of the same product family or systems with similar missions or in the same application domain, saving development time and money.

Functional architecture is a necessary input for establishing the completeness of system performance requirements. A performance requirement quantifies the extent to which a system or one of its parts performs a particular functionality. Functional architecture may be used as an input for identifying and classifying potential system safety hazards as well.

Functional architecture represents the core model for the development of mechatronic systems, which are systems with mechanical, electronic, and software parts, such as those presented as examples in this book (see Chapters 8 and 9). For those mechatronic systems the functional architecture can be considered as the frame that integrates the diverse engineering specialties involved in the development of a mechatronic system. First functionalities are identified and modeled and later a decision is made about how to allocate and implement the functions using diverse technologies.

5.1.2 What About Functional Architecture for Software Intensive Systems?

Software functional modeling was developed in the second half of the twentieth century and has been applied extensively in a wide range of applications from aerospace to information systems.

One of the most popular functional modeling approaches is structured analysis and design technique (SADT). SADT is a methodology for describing systems as a hierarchy of functions. It is a structured analysis modeling language that uses two types of diagrams: functional models and data models. It was developed in the late 1960s by Douglas T. Ross and was formalized and published as IDEF0 in 1981.

In the late 1980s there was a change of the software design paradigm from the functional-oriented to the component or object-oriented. This change of design paradigm facilitates software reuse, but has important consequences when behavioral aspects are frequently represented using use cases or message diagrams instead of functional architecture. A UML-SysML sequence diagram is a form of interaction diagram that shows objects as lifelines running down the diagram, with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline (see Appendix A). Sequence diagrams are good at showing which objects communicate with which other objects and what messages trigger those communications, but they do not represent system functions.

5.2 A Function is a Transformation

5.2.1 Main Concepts Related to the Functional Architectural Model

Some of the main concepts related to the modeling of a functional architecture were presented in the conceptual model of the ISE&PPOOA method described in Chapter 4; other concepts more specific to functional architecture and borrowed from the literature are summarized in Table 5.1 and described below.

The first concept to consider is the function. As described in Chapter 4, in the ISE&PPOOA method a function is considered as a transformation of inputs (material, energy, or data) into outputs (material, energy, or data). This definition of the function as a transformation allows the functional analysis and the modeling of the functional architecture for any system or process to be either physical or an information system. Other sources, for example the *INCOSE Handbook,* defines a more general function as a characteristic task, action or activity that must be performed to achieve a desired outcome [3].

In the SysML standard, functions are defined as activities, so a SysML activity is the transformation of items as inputs and providing items as outputs. An item is an entity that may flow through a system, whether physical or information. Therefore, an item may be physical (matter or energy), data, or a software objects [4].

Activity usage, called an action, is how an activity is used in the definition of another enclosing activity. Activity instance is the particular execution of an activity. The determination of when activities perform their transformation is defined as control. Control includes starting as well as stopping a transformation or function [4].

The sending and receiving of signals is one mechanism for communicating between activities executing in the context of different system blocks and for handling events such as time-outs. Signals are sometimes used as an external control input to initiate an action within an activity that has already started [5].

We can consider two main kinds of activities: nonstreaming and streaming activities. Nonstreaming activities are those that accept input items and provide output items only when they start and finish. Streaming activities accept input items and provide output items anytime during their operation.

In an SysML standard, an activity parameter may be designated as streaming or nonstreaming, which affects the behavior of the corresponding activity parameter node. An activity parameter node for a nonstreaming input parameter may only accept

Table 5.1 Main Concepts Used in the Functional Architecture Modeling

Concept	*Source*
Function	ISE&PPOOA conceptual model (Chapter 4)
Activity	SysML standard
Action	SysML standard
Signal	SysML standard
Nonstreaming activity	SysML standard
Streaming activity	SysML standard
Condition	SysML standard and [6]
Event (time event, change event)	SysML standard

tokens prior to the start of activity execution, and the activity parameter node for a nonstreaming output parameter can only provide tokens once the activity has finished executing. This contrasts with a streaming parameter, where the corresponding activity parameter node can continue to accept streaming input tokens or produce streaming output tokens throughout the activity execution [5].

Streaming activities start when input items are available, continue to accept new input items, and provide output items for an indefinite period of time until a control activity terminates them.

Other main concepts related to functional architecture and system behavior are conditions, events, states, and modes.

A condition is a Boolean combination of logical expressions formed from attributes of objects within the system [6].

In the SysML standard an event has different meanings. A time event corresponds to an expiration of an (implicit) timer. In this case the action has a single output pin that outputs a token containing the time of the accepted event occurrence. A change event corresponds to a certain condition expression (often involving values of properties) being satisfied. In this case there is no output pin, but the action will generate a control token on all outgoing control flows when a change event has been accepted. A change event can also be related to the change in the value of a structural feature, for example flow property [5].

A state as described in Chapter 4 represents the condition of a system or a part defined by its current condition/configuration and the functionality provided. In ISE&PPOOA state diagrams are used to complement the behavioral description modeled by activity diagrams when states are used to identify diverse conditions/configuration where different functionality may be provided by the system.

For many authors a mode is synonymous of a state but others define a mode as a combination of conditions that may exist whatever the system and that has a widespread effect on the way the users perceive it [6].

5.2.2 Functional Architecture Models

This section summarizes the most common graphical representations used in functional architecture models but it is important to emphasize what we consider the main purpose of the functional architecture model. Which is to represent

- The hierarchy of all the functions the system must perform to meet its requirements;
- The main functional flows that define the system behavior;
- The functional interfaces.

Besides its graphical representation, each function is also described in terms of inputs, transformation, and outputs.

This section describes for illustrative purposes some common graphical representations used by industry for functional architecture modeling. As described by Long, the most common representations used before the SysML standard release were the functional flow block diagram (FFBD), data flow diagram (DFD), IDEF0

diagram, N^2 (N-Squared) chart, and enhanced functional flow block diagram (EFFBD) [7].

The FFBD was one of the first graphical representations used by systems engineers to model the main functional flows of a system. The FFBD shows only the control sequencing for the functions but not the flows of items between them. A set of control constructs are available to represent concurrency, alternatives, loops, and iterations. Figure 5.1 is a simple example of FFBD where functions F3, F4, and F5 are performed in parallel.

The DFD represents the system as a network of functions that accept and produce data. The DFD shows terminators or external entities as squares, the functions at the same level of hierarchy as bubbles and the data flows represented as arrows between them, external to them, and to and from stores. Stores are used to represent an item or set that is operated by a group of functions.

Each high-level bubble (parent function) can be decomposed into lower-tier bubbles (children functions) connected by the data that flow between them supporting the top-down modeling of the system functionality.

Flows can converge and diverge as well. They converge when separate flows join to a single flow. Similarly a single flow may diverge if subsets of the output flow of a function are inputs to two or more succeeding functions.

The notation for DFDs was proposed by De Marco [8] but other authors extend it to include control transformations and event flows that allow the modeling of reactive real-time systems [9].

Currently the DFDs are complemented with the specifications of the functions and the description of the stores and data items. Data items may be elemental or composite.

Figure 5.2 is an example of DFD with two terminators or external entities (E1 and E2), one store (S1), and four functions (F1, F2, F3, and F4), and several data flows represented as arrows connecting them.

During the 1970s, the U.S. Air Force Program for Integrated Computer Aided Manufacturing (ICAM) pursued the increase of manufacturing productivity through rigorous application of computer technology. As a result, the ICAM program developed a series of techniques known as the ICAM Definition (IDEF) techniques, which included the IDEF0, used to produce a functional model.

In 1991 the National Institute of Standards and Technology (NIST) received support from the U.S. Department of Defense to develop one or more Federal Information Processing Standards (FIPS) for modeling techniques [10].

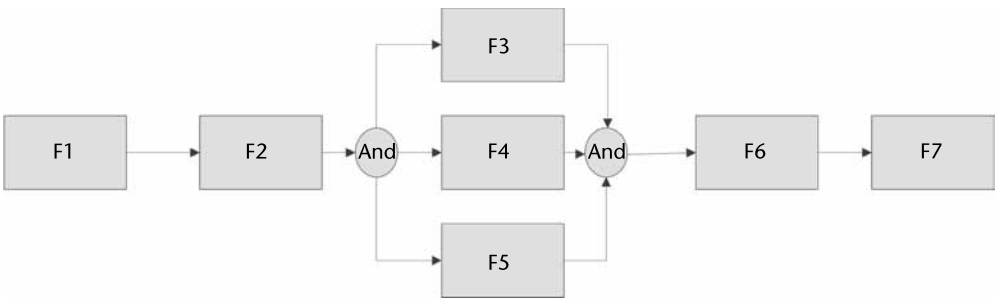


Figure 5.1 FFBD diagram.

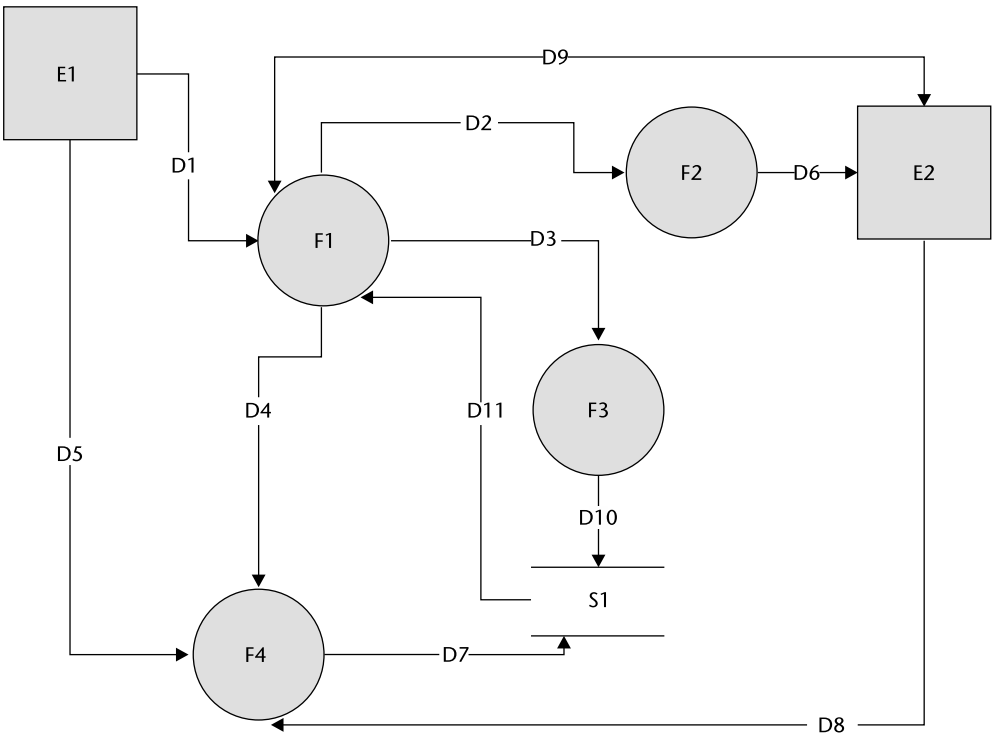


Figure 5.2 DFD diagram.

An IDEF0 functional model consists of a hierarchical series of diagrams, text, and glossary cross-referenced to each other. The two primary modeling elements are functions (represented on a diagram by boxes) and the data and objects that interrelate those functions (represented by arrows).

The basic notation used in IDEF0 is summarized here. A function is represented by a box, a function box has a standard box/arrow relationship where input arrows interface with the left side of a box, control arrows interface with the top side of a box, and output arrows interface with the right side of the box. Mechanisms or physical resources to perform the function are arrows pointing upward and connecting to the bottom side of the box. The mechanisms represent the system components where the function is allocated when the physical elements of the system are identified.

A diagram may have associated structured text, which is used to provide a concise description of the diagram. Text is used to describe flows and interbox connections to clarify the intent of items and functional patterns considered to be of significance. Text should not be used merely to describe, redundantly, the meaning of boxes and arrows [10].

Figure 5.3 is an example of IDEF0 diagram where we have four functions, two of which are functions labeled F2 and F3 and are executed in parallel and using the same mechanism M2. Inputs, outputs, and controls are represented as well. An IDEF0 model with no mechanisms may be considered as a functional architecture model before function allocation to physical elements.

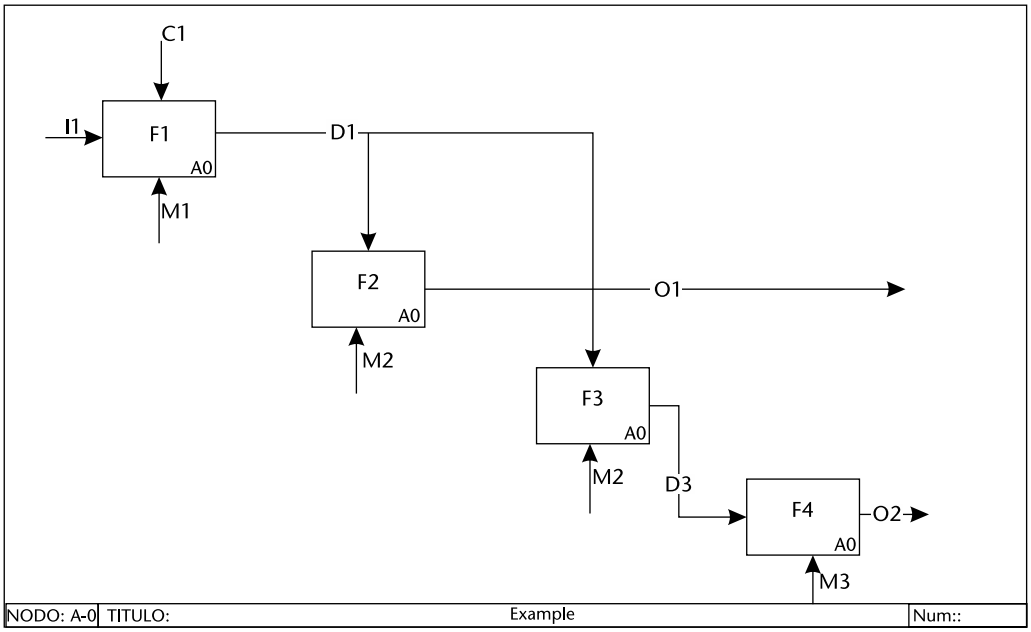


Figure 5.3 IDEF0 diagram.

The EFFBD represents control flow, similar to the FFBD but additionally it represents the data flows in the same diagram and therefore inputs and outputs to the functions are represented as well. A function is enabled by the completion of the functions preceding it in the control sequence, similarly to a FFBD, and triggered, if any data input to it is identified as a trigger, before it can execute [7]. A correspondence between constructs in the EFFBD and UML-SysML activity diagrams can be established [4].

The SysML activity diagram (See Appendix A) defines the actions in the activity along with the flow of input/output and control between them, and so an activity decomposes into a set of actions that describe how the activity executes and transforms its inputs to outputs. Activities are based on token-flow semantics related to Petri nets.

There are several different categories of actions in SysML including send signal, accept event, and wait time actions. The action symbol varies depending on the type of action, but typically it is a rectangle with round corners. A call behavior action is a specialized action that invokes another behavior when it becomes enabled. Call behavior actions allow the decomposition of a higher-level behavior into a set of lower-level behaviors. Call behavior actions can be used to factor out a common chunk of functionality that appears in multiple places and instead define it in a separate behavior that is simply invoked multiple times [11].

Control nodes control the execution of an activity along paths other than a simple sequence of actions. Control nodes can direct the flow of control tokens as well as object tokens within an activity. There are seven kinds of control nodes: initial nodes, activity final nodes, flow final nodes, decision nodes, merge nodes, fork nodes, and join nodes.

By default, actions and activities consume their input object tokens only at the moment they begin executing. Similarly, they deliver their output object tokens

only at the moment they finish executing. In Section 5.2, we referred to this as nonstreaming behavior.

An activity may have multiple inputs and multiple outputs called parameters.

The pin symbols are small boxes located on the outside surface of the action symbol and may contain arrows indicating whether the pin is an input or output.

Tokens hold the values of inputs, outputs, and control that flow from one action to another. An action processes tokens placed on its pins. A pin acts as a buffer where input and output tokens to an action can be stored prior to or during execution; tokens on input pins are consumed, processed by the action, and placed on output pins for other actions to accept [5].

Object flows are used to route input/output tokens that represent information and/or physical items (matter or energy) between object nodes. Activity parameter nodes and pins are two examples of object nodes.

ISE&PPOOA uses a simplified version of the SysML activity diagram to represent only the functional flows of the system that is the main sequences of system responses to events or periodic timers.

Figure 5.4 represents how a SysML activity diagram is used in ISE&PPOOA. It represents any activity with diverse control nodes (initial node, fork and join nodes, decision and merge nodes, and activity final node), one wait for event action and seven nonstreaming behavior actions. Actions a2 and a3 are executed in parallel, and actions a5 and a6 are alternative actions that execute depending on conditions 1 and 2. Allocation is not presented here because it is a mere functional flow represented before functional allocation. When the functional allocation is achieved, the activity diagram allows its representation using partitions also termed colloquially as swimlanes.

In Chapter 4, we proposed the graphical, tabular, and textual representations used in the ISE&PPOOA methodology for the functional architecture models that are the functional hierarchy using a SysML block definition diagram, complemented with activity diagrams for the main system functional flows to represent system responses or behavior. The N^2 chart is used as an interface description where the main functional interfaces are identified. Alternatively, the SysML activity diagram allows the representation of the items either sent or received by an activity, but for the sake of simplicity and understandability we recommend using the N^2 chart for the functional interfaces description instead of representing the items in the activity

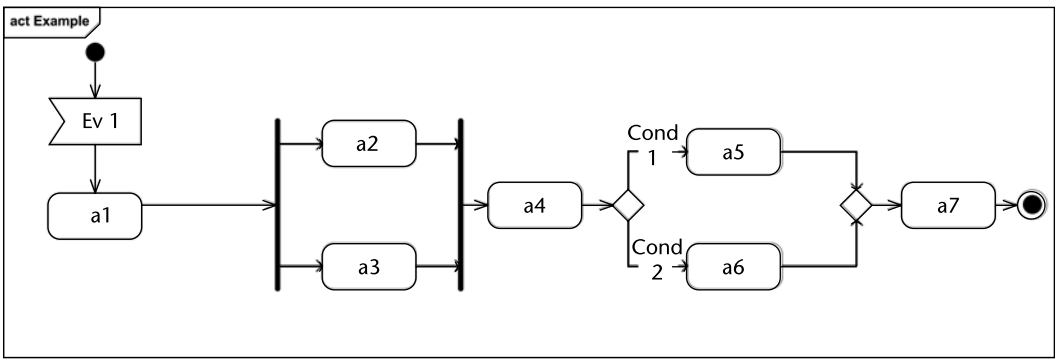


Figure 5.4 Activity diagram.

diagrams. A tabular textual description of each system function should be provided as well (see Table 5.2).

For the software architecture of the software intensive subsystems, PPOOA framework also proposes the use of UML activity diagrams for the representation of the software intensive subsystems behavior, which is the chain of actions in response to either an external, internal event, or a time event. This chain of actions is termed in PPOOA as a CFA, and it represents a cause-effect chain of actions that cuts across different building elements of the software architecture. This chain progresses through time and executes the actions allocated to the software architecture components until it gets to an ending point [12].

5.3 Modeling the Functional Hierarchy

The functional hierarchy or functional breakdown structure, shown in Figure 5.5, is the outcome of steps 3.1 and 3.2 of the ISE&PPOOA process described in Chapter 4. Summarizing this hierarchy represents the top-level functions of the system and their subfunctions or children functions.

The modeling of the functional hierarchy may produce diverse results depending on the approach and the criteria used to develop it. In any case, the goal is to produce a robust functional hierarchy with the following characteristics:

- Full coverage of the system functionality;
- High cohesion;
- Consistency between the functional hierarchy and the functional flows representing behavior.

The functional hierarchy should be taken only to a level of detail appropriate to the issues being addressed in this phase of development. A practical assessment of full coverage is that there is no system output without the functionality to produce it.

High cohesion is a more complex issue. As stated by Stevens et al. [13], cohesion measures the degree to which functions are related to one another. Cohesion can be

Table 5.2 Tabular Format for Function Description

Function name:
Label:
Description:
Inputs:
Outputs:
Parent function:
Children functions:

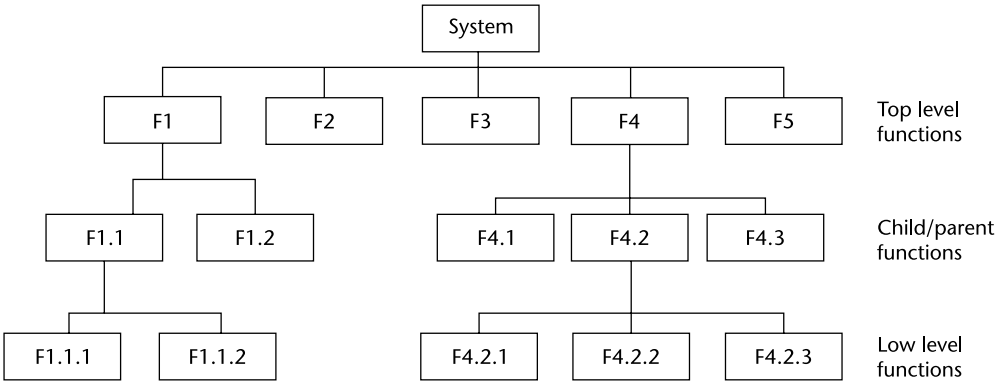


Figure 5.5 Functional hierarchy.

- Functional when a group of functions perform similar or related transformations;
- Sequential when the output of one function becomes the input of the next function;
- Communicational when the functions share the same data;
- Procedural when there is a chain of functions that follow one another as part of a functional flow;
- Temporal when functions occur simultaneously;
- Logical when functions perform a range of similar activities.

Consistency between functional hierarchy and the functional flows means that the activities and actions represented in the functional flows must be present in the functional hierarchy.

There are two approaches to be applied with the ISE&PPOOA methodology for modeling the functional hierarchy of a system; one is top-down and the other is bottom-up. In some cases we can combine both as well depending on our understanding of the system and experience.

5.3.1 Top-Down Approach for a Functional Hierarchy

When using the top-down approach for building a functional hierarchy, the engineer selects the system capabilities and high-level functional requirements obtained in the step 2.a of the ISE&PPOOA process (see Chapter 4) and transform them into the list of high-level functions, following step 3.1 of the ISE&PPOOA process (see Chapter 4).

It is important to notice that sometimes a capability is transformed into a group of functions, properties or quality attributes. Think, for example, of a case of the long endurance capability of an UAV. Long endurance capability is transformed into aerodynamic properties of the wing, system functions such as health monitoring, and quality attributes such as system reliability. In the case of a collaborative robot, a capability is harmless. Harmless is transformed into physical properties of

the robotic arm, system safety as a quality attribute, and contingencies management as a system function.

Often it is useful to use a context diagram of the system identifying its main inputs and outputs for helping in the identification of the functions that receive such inputs and produce such outputs.

Another useful strategy is to use taxonomies of functions in similar domains of application to that of the system to be developed; for example, the INCOSE taxonomy of functions for a commercial aircraft [14] or the taxonomies of functions for industrial processes: branch, channel, connect, control magnitude, convert, provision, signal, and support [15].

Every function must be decomposed when additional insight into the production of its outputs is needed. When a parent function is decomposed into child functions or subfunctions, the decomposition process must conserve all of the inputs and outputs of the parent function. This interface consistency is enforced in the diagrams representing the parent–children hierarchies. Cohesion—specifically functional cohesion—as described above should be guaranteed.

5.3.2 Bottom-Up Approach for a Functional Hierarchy

When using the bottom-up approach for building the functional hierarchy, the engineers select the system operational scenarios obtained in the step 1 of the ISE&PPOOA process (see Chapter 4) and use them as an input to model the preliminary system functional flows (activity diagrams for the main system responses to external, internal or time events), following step 3.3 of the ISE&PPOOA process (see Chapter 4) and the approach explained in Section 5.4.

Since these activities and actions are usually in the third, fourth, or even lower levels of the functional hierarchy, they are combined into parent activities (functions) meeting the cohesion properties described above. When cohesive child functions are combined into a parent function, the engineers can represent the functional hierarchy tree.

ISE&PPOOA allows as well a combination of the top-down and bottom-up approaches in this combined approach where steps 3.1, 3.2, and 3.3 of the ISE&PPOOA process (see Chapter 4) are applied iteratively.

An assessment of full functional coverage of the functional tree or functional breakdown structure is a main issue. The functional tree describes what the system does when the lowest level functions are one step away from being implementation specific.

5.4 Modeling the Functional Flows

The ISE&PPOOA methodology proposes modeling the functional flows as simplified activity diagrams. Simplified activity diagrams are those that show the flow of activities/actions, but do not represent tokens or the items that flow between object nodes. For the item flow we propose to use a tabular form, the N^2 chart, which provides a compact view of the functional interfaces that we describe in the next section.

The functional flows are modeled as system responses to either external, internal, or time events. The modeling of the functional flows is part of step 3.3 of the ISE&PPOOA methodology (see Chapter 4), but it can be applied iteratively with step 3.2 as we propose in Section 5.3.

When producing the activity diagrams representing the functional flows (behavior of the system), it is important to maintain consistency with the functional hierarchy and the functional interfaces represented in the N^2 charts. We recommend following these guidelines:

- Construct activity diagrams in several levels of detail, creating them as least at the system level and subsystem level.
- Before constructing the activity diagrams that represent the functional flows, identify the dependencies between functions. The most common dependency between functions is the output-input dependency when an item produced by one function is used as an input by the next function to produce its output. In contrast, function parallelism is the result of input item flows that are independent.
- Think about the activation criteria for each function (activity).
- In some cases(see Chapter 9 example) a flow of functions (activities) need to be interrupted by some exceptional event that will abort the normal execution of the activation. The activity diagrams in SysML allow the use of interruptible regions.

5.5 Describing Functions and Functional Interfaces

There are several ways to describe functions, mainly textually, tabular formats, as pseudocode, or as a combination of simple unit operations. The latter is useful for the description of transformations in chemical industry processes.

NASA Systems Engineering Handbook recommends that each function of the system is described in terms of inputs, outputs, failure modes, consequence of failure, and interface requirements [16].

In the case of information systems and particularly for real-time systems, Ward and Mellor build the function specification by using a Program Design Language (PDL), which is a textual representation of the function transformation logic with a rigorous syntax but not executable by a computer [9].

In the examples of development with the ISE&PPOOA method we recommend using a rigorous textual description in tabular form where the inputs and outputs of the function are identified. Failure modes are an optional issue. Table 5.2 shows the tabular format we used in UAV projects for functions description.

ISE&PPOOA methodology proposes N^2 charts to depict the items (data, energy, or mass) that are input and output of the functions in the functional architecture. N^2 chart is a table with N+1 rows and N+1 columns. It records the functions, doing better at the same hierarchical level, and the items that flow between them. Functions are entered on the leading diagonal, while interchanges or interfaces appear in the other cells as appropriate. External inputs can optionally be shown in

the row above the first entity on the diagonal and external outputs can be shown in the right-hand column.

In Table 5.3, the convention IC (inputs in columns) is used, so for example in the Table 5.3 functions B and C receive inputs from function A, function D receives an input from function C, and function B receives an input from function C. External inputs to the functions A, B, C, and D are shown in the first row of Table 5.3. External output from the entities A, B, C, and D are shown as arrows in the right-hand column. Blank cells represent no flow of items between these functions.

The use of an N^2 chart for functional architecture supplements the SysML diagrams due to the following benefits of using the N^2 chart (some of which were detailed in Chapter 4):

- N^2 charts are very compact, allowing the overview of even the most complex systems.
- The interfaces tend to occur in pairs, potentially forming simple, reactive causal loops [17].
- The N^2 chart is a very helpful tool to allocate functions to subsystems or system parts such that there is minimal interaction among them. The clustering of functions modifies the order of functions so the interactions among the functions are grouped close to the diagonal.
- The N^2 chart is very useful for detecting some common mistakes: functions that produce outputs without any inputs, functions that produce no outputs, and the violation of the conservation law of inputs and outputs between a parent function and their children.

5.6 Functional Requirements

The functional architecture is developed through an iterative process that identifies the functions to be provided by the system and grouped together following the cohesion criteria described above.

This functional hierarchy can be used as a framework to organize the system functional requirements so they are readable and robust in case of change. The resulting organization of the system functional requirements automatically traces the lower-level requirements to the higher-level requirements.

Therefore, the ISE&PPOOA method recommends that functional requirements are specified relying on the children functions outputs identified in the functional architecture. The detailed level functional system requirements are presented in

Table 5.3 N^2 Chart for Functional Interfaces

↓	↓	↓	↓	
Function A	→↓	→↓		→
	Function B			→
	↑←	Function C	→↓	→
			Function D	→

the lower-level functions of the system. At least each function is associated with one functional requirement and each functional requirement is associated with a function.

The use of templates for specifying functional requirements is highly recommended. For example, the standard ISO/IEC/IEEE 29148 provides this template for a functional requirement:

> When <condition clause>, the <subject clause> shall <action verb clause><object clause> <constraint of action> [18].

The subject here is the function and the object is one of its outputs, if there are many. A condition clause indicates a circumstance or event under which the requirement applies so it may define the state, mode, or control that enables the function.

Constraint of action with numerical values can be used when functional performance requirements are specified; this may be done either before or after the functional allocation to physical elements (step 4.1 of the ISE&PPOOA process). When done before, measures of performance are associated directly with functions rather than with the physical element that implements the function.

5.7 Summary

This chapter presented how to develop the functional architecture of a system using the ISE&PPOOA methodology. This approach integrates MBSE with classical and well-known functional analysis techniques.

The functional architecture representation has three main constituents. The first is the functional hierarchy using a SysML block definition diagram. This diagram is complemented with activity diagrams for the main system functional flows to represent behavior. The second is the N^2 chart, which is used as an interface description where the main functional interfaces are identified. The third is a textual description in tabular format of the system functions, should be provided as well.

The approach to obtain the functional architecture may be top-down, beginning from the top-level functions identification, or bottom-up, beginning from low-level functions and systems responses. Sometimes a combination of both approaches is the wisest approach.

Understanding the system functions is the key of requirements flow down. The functional hierarchy of the functional architecture is used as the framework to organize the system functional requirements so they are readable and robust in case of change. The resulting organization of the system functional requirements automatically traces the lower-level requirements to the higher-level requirements.

5.8 Questions and Exercises

1. What is the difference between a use case and a function of a system?
2. What is the difference between a function and an item?
3. What is the difference between the functional hierarchy and the functional flow representations?

4. Identify the top-level functions of a washing machine.
5. Create an N^2 chart of the washing machine functions identified above.
6. Identify the top-level functions of the electric parking brake of a car, including the capability for automatic release of the parking brake.

References

[1] Carson, R. S., and B. J. Sheeley, "Functional Architecture as the Core of Model-Based Systems Engineering," *Proc. INCOSE International Symposium*, Vol. 23, 2013, pp. 29–45.

[2] Fernandez, J. L., "An Architectural Style for Object-Oriented Real-Time Systems," *Proc. 5th International Conference on Software Reuse*, Victoria, Canada, June 2–5, 1998.

[3] Walden, D., et al., *Systems Engineering Handbook*, International Council on Systems Engineering (INCOSE), San Diego, CA, 2015.

[4] Bock, C., "SysML and UML 2 Support for Activity Modeling," *Systems Engineering*, Vol. 9, No. 2, 2006.

[5] Friedenthal, S., A. Moore, and R. Steiner, *A Practical Guide to SysML. The Systems Modeling Language*, Burlington, MA: Morgan Kaufmann, 2008.

[6] Wallace, R. H., J. E. Stockenberg, and R. N. Charette, *A Unified Methodology for Developing Systems*, Intertext Publications, McGraw-Hill, 1987.

[7] Long, J. E., "Relationships between Common Graphical Representations Used in Systems Engineering," *INCOSE Insight,* Vol. 21, No. 1, 2018.

[8] De Marco, T., *Structured Analysis and System Specification,* Upper Saddle River, NJ: Yourdon Press, 1978.

[9] Ward, P. T., and S. J. Mellor, *Structured Development for Real-Time Systems*, Upper Saddle River, NJ: Yourdon Press, Prentice Hall, 1985.

[10] NIST, FIPS Publication 183 released of IDEFØ, Computer Systems Laboratory of the National Institute of Standards and Technology (NIST), 1993. (Withdrawn by NIST 08 Sep 02).

[11] Delligatti, L., *SysML Distilled: A Brief Guide to the Systems Modeling Language*, Upper Saddle River, NJ: Addison Wesley, 2014.

[12] Fernandez, J. L., and E. Esteban, "Supporting Functional Allocation in Component-Based Architectures," *Proc. 18th International Conference on Software and Systems Engineering (ICSSEA),* Paris, France, December 2005.

[13] Stevens, R., et al., *Systems Engineering. Coping with Complexity,* Hemel, UK: Prentice Hall Europe, 1998.

[14] INCOSE, "Framework for the Application of Systems Engineering in the Commercial Aircraft Domain," International Council on Systems Engineering (INCOSE), San Diego, CA: 2000.

[15] Hirtz, J. M., et al., "Evolving Functional Basis for Engineering Design," *Proc. ASME Design Engineering Technical Conference*, Pittsburgh, PA, September 9–12, 2001.

[16] NASA, *NASA Systems Engineering Handbook*, NASA SP-2016-6105 Rev2, National Aeronautics and Space Administration, Washington D.C.: 2016.

[17] Hitchins, D. K., *Advanced Systems Thinking, Engineering and Management*, Norwood, MA: Artech House, 2003.

[18] ISO/IEC/IEEE, ISO/IEC/IEEE 29148:2011, *Systems and Software Engineering. Life Cycle Processes. Requirements Engineering*, International Standards Organization, Geneva (CH), IEC, Geneva (CH), and Institute of Electrical and Electronic Engineers, New York: 2011.

CHAPTER 6

Heuristics to Apply in the Engineering of Systems

ISE&PPOOA considers that systems architecting is still an art. Therefore, the methodology proposes a collection of general heuristics and quality attributes heuristics to be applied to refine the system solution based on the requirements, particularly the nonfunctional ones.

We present general systems and software architecting heuristics and those related to maintainability, efficiency, safety, and resilience quality attributes. These heuristics are collected from diverse sources referenced at the end of this chapter and tailored to be used in the ISE&PPOOA methodological approach. We conclude the chapter with heuristics relative to the PPOOA software architecture framework.

6.1 Heuristics Framework

A heuristic is a means of satisfying a nonfunctional or quality attribute requirement by manipulating some aspect of a quality attribute model through design decisions.

The definition has the following consequences:

- A heuristic bridges the quality attribute model related to the problem definition and the physical architecture of the system. It does so by specifying how the quality attribute requirements, also known as nonfunctional requirements, can be controlled through design decisions to meet them.
- A heuristic uses knowledge from various quality reasoning frameworks (such as performance, maintainability, safety, or resilience engineering) to facilitate the creation of quality attribute models that mirror the system architecture being designed.

Figure 6.1 shows the overall flow of the architecture development and how heuristics are applied to make some design decisions.

Quality models allow engineers to select the heuristics that offer guidance for making design decisions. The models also offer a reasoning framework for explaining how changes in the design decisions affect the quality attribute validation. Heuristics are part of this reasoning framework.

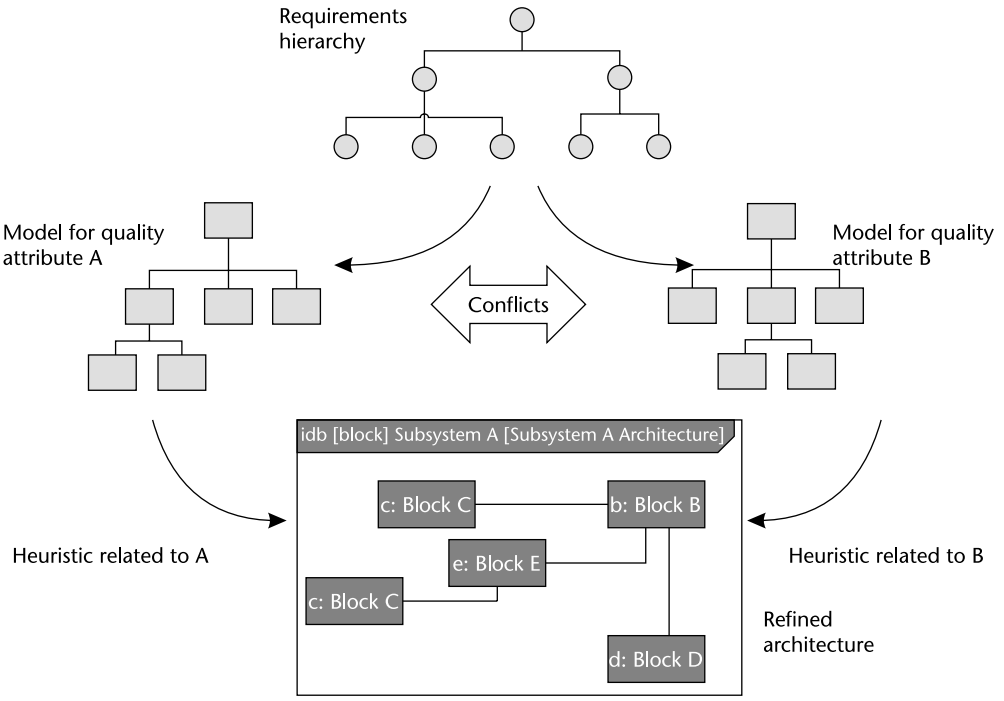


Figure 6.1 Heuristics and the architecting stage.

The quality model we proposed here is for refining the solution architecture when using ISE&PPOOA. It is shown in Figure 6.2 and described below.

This quality model considers the quality characteristics and subcharacteristics we consider more useful for the type of applications where ISE&PPOOA can be used.

A more general quality model is proposed by ISO/IEC 25010 [1] where they describe quality characteristics, such as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability.

The quality model proposed here is oriented to the solution, either the system or software architecture refinement, so we grouped the heuristics we collected from diverse sources, referenced below, and labeled them based on the quality characteristic we consider most adequate to the applications we developed. The main quality characteristics we consider are reliability, maintainability, efficiency, safety, and resilience.

We recommend the reader to adopt and adapt this solution-oriented quality model with the quality attributes he or she consider for the systems developed by his or her organization. Frequently, maintainability is an issue for any product to be durable, and reliability is very important for the success of the mission. Efficiency is an issue particularly for reactive and real-time systems. Safety is a concern in some regulated domains such as aerospace, automotive, medical appliances, and robotics. Resilience is related in this quality model to survivability, adaptation, and graceful degradation, which are very important for some autonomous systems.

Reliability and maintainability are represented together. Reliability may be defined as the extent to which the system or one of its parts performs its functions

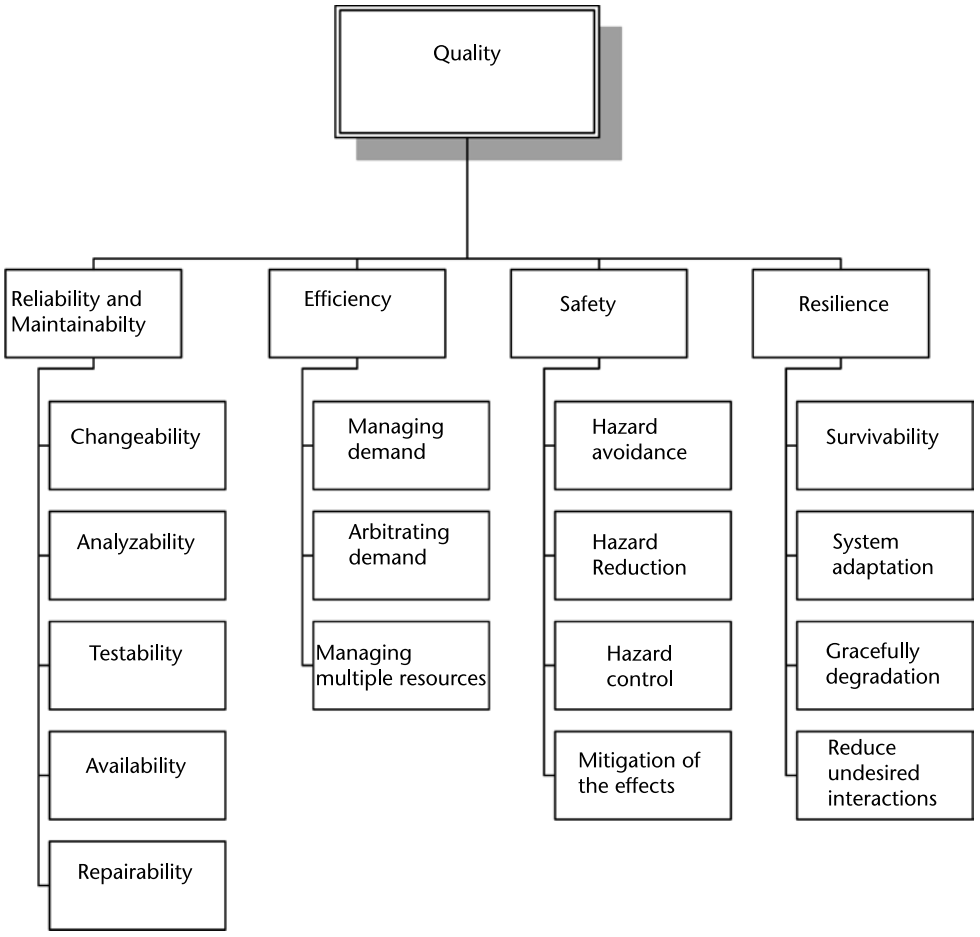


Figure 6.2 Quality model for heuristics classification.

without failure. Maintainability is the capability of the system or one of its parts to be modified. Modifications may include corrections, improvements, or adaptation of the element to changes in environment and in requirements. Reliability and maintainability break down into the following subcharacteristics:

- Changeability is the ability of the system or part to enable a specified modification to be implemented.
- Analyzability is the ability of the system or part to be diagnosed for defects or causes of failures.
- Testability is the ability of the system or part to be validated.
- Availability is defined by a measure of the probability of the system to be in an operable condition at the start of a mission initiated at random times.
- Repairability is the ability of a failed or damaged system or part to be restored to acceptable condition within an acceptable period of time. The MTTR defines how long it takes to repair the part on average.

Efficiency is the ability of the system to provide appropriate performance, relative to the amount of resources used, under stated conditions. Efficiency breaks down into the following subcharacteristics:

- *Managing demand*: To help to control the system resources demand;
- *Arbitrating demand*: To control preemption and waiting times when there are competing requests and contention for shared resources;
- *Managing multiple resources:* To enable multiple resources to be used efficiently to ensure that available resources are used when they are needed.

Safety is defined by Firesmith as the degree to which accidental harm is prevented, identified, reacted to, and adapted to [2]. Safety concerns are the identification and management of hazardous conditions with catastrophic consequences.

Safety is organized into four subcharacteristics.

- Failure avoidance;
- Hazard reduction;
- Hazard control;
- Mitigation the effects.

The term resilience has diverse meaning in such fields as psychology, ecology, and engineering. The U.S. Government defines resilience as the ability to adapt to changing conditions and prepare for, withstand, and rapidly recover from disruptions [3]. Jackson and Ferris apply resilience for engineered systems. These resilience engineered systems are able to restructure during the threat encounter withstanding the situation by retaining partial or full functionality [4]. Based on Jackson and Ferris, the following subcharacteristics are considered in the quality model we proposed here:

- Survivability is the ability of the system to deal with the threat disruption that it encounters;
- System adaptation is the degree to which the system changes itself to adapt to the threat;
- Graceful degradation is the ability of the system to maintain a limited functionality even when a large portion of it has been destroyed or rendered inoperative;
- Undesired interactions, also called hidden interactions. These interactions are caused when there is a lack of whole system design or system architecting principles as opposed to emphasizing component design.

Heuristics have specific implementations in the architecture. For example, while a performance engineering queuing model might only require an average execution time as input to the model, there are many sources of execution time that need to be derived from an architectural design. There is a many-to-one relationship between this model and the corresponding solution architectures.

Heuristics are neither absolute nor independent. The application of a heuristic may also require additional heuristics to be applied. For example, the application of the "break the dependency chain" heuristic (see Section 6.3) to insert an intermediary would likely require additional heuristics to isolate that intermediary responsibility.

This chapter does not attempt to identify the solutions that may be created from each heuristic since many concrete solutions may be created only by a single heuristic. Consequently, using architectural heuristics involves the identification of the possible mappings from an architectural model to a quality attribute model. Heuristics dependences are another issue that is related to the solution implementing them.

Below we present general systems and software architecting heuristics and those related to reliability and maintainability, efficiency, safety, and resilience quality attributes.

6.2 Systems Architecting Heuristics

These general systems architecting heuristics apply to functions allocation and functional requirements mainly to obtain a modular physical architecture. These heuristics are grouped by the architecture development step (see Chapter 4) where they apply. Systems architecting heuristics are either descriptive or prescriptive. Descriptive heuristics describe a situation; prescriptive heuristics give an architectural approach to one situation.

6.2.1 Heuristics for Step 3: Functional Architecture of the ISE Process

SA_Heu_1: Use functional hierarchy. Work on high-level functions first as it is recommended in the ISE&PPOOA process described in Chapter 4. The reason is that high-level functions are less likely to change [5].

SA_Heu_2: Limit the side effects of functions. Functional cohesion implies that a function does the transformation that its name and description implies and nothing else.

As explained in Chapter 5, high cohesion is a complex issue. Cohesion measures the degree to which functions are related to one another. Functional cohesion is when a group of functions perform similar or related transformations. The terms used for describing a function are very important and ambiguity should be avoided.

6.2.2 Heuristics for Step 4: Physical Architecture of the ISE Process

SA_Heu_3: Each hierarchical level of the system provides a context for the level below. Leave the details of specialties to the specialist. The level of detail required by the architect is only to the hierarchical depth of a building element or component critical to the system as a whole. But the architect must have access to that level and know about its criticality and status [6].

SA_Heu_4: Architecture refinement is complete when the engineer considers that the system can be built to the client's satisfaction [6]. The deliverable at the end of the architectural stage is not only the refined physical architecture but the acceptance criteria. ISE&PPOOA defines acceptance based on the complete functional allocation and how the nonfunctional requirements are implemented (see Chapter 4).

SA_Heu_5: The choice between possible physical architectures may depend on which set of drawbacks the client can handle best. If the results of trade-off analysis (see Chapter 11) are inconclusive, then the wrong selection criteria were used. Find out needs and system requirements, and then repeat the trade using those factors as the selection criteria [6].

SA_Heu_6: Group and allocate functions that are strongly related to each other, and separate functions that are unrelated. Many of the functions can be brought together to complement each other in the modular architecture. Obviously the more modularity, the more probable the overall success, which is more functional cohesion and less coupling. The use of clustering techniques and the N^2 chart are helpful to follow this heuristic (see Chapter 4).

SA_Heu_7: Promote subsystems implementation independency. Subsystem interfaces should be designed so that each subsystem can be implemented independently of the specific implementation of the subsystems to which it interfaces [6].

SA_Heu_8: Choose a solution configuration with minimal communications between the subsystems. Choose the building elements so that they are as independent as possible; that is, elements with low coupling and high functional cohesion [6].

SA_Heu_9: Physical architecture should follow functional architecture. Each low-level function should be allocated to one physical component. Except for nonfunctional requirements implementation, functional and physical architecting should match at the modular architecting step of the ISE&PPOOA process (see Chapter 4).

SA_Heu_10: Use defined criteria for system decomposition. Do not slice through system regions where high rates of exchange are required. The principles of minimum communications and proper partitioning are critical to system modifiability, testability, and fault isolation [6].

SA_Heu_11: The greatest leverage and danger in systems architecting is at the interfaces [6]. Special care should be given to interface design so that interface does not have to change when its associated interfacing elements change. Use guidelines for good quality interface specifications. Stakeholders must agree on the definitions in the Interface Control Document (ICD) and both interacting elements; for example, subsystems, need to include the corresponding interface requirements in their requirements document. The interface requirements should trace to each other, a common definition in the ICD, and a common parent [7]. Currently there are no standards on how to map ICD and SysML diagrams.

SA_Heu_12: Consider design margins. Design margins are used for different purposes: budget reserves, tolerances, performance capabilities, and safety factors. As the system becomes better understood the design margins are reduced [5]. Design margins are applied for example to weight or energy consumption. Design margins are allocated at the different levels of the system as well [8].

Table 6.1 summarizes the general systems architecting heuristics described above.

6.3 Reliability and Maintainability Heuristics

Here, reliability and maintainability are represented together. Reliability may be defined as the extent to which the system or one of its parts performs its functions without failure. Maintainability can be considered as the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers [1].

Maintainability is sometimes known as modifiability and may be the quality attribute most closely aligned to the system architecture [9].

Bass et al. classify the modifications to a system as extending or changing its capabilities, deleting unwanted capabilities, adapting to new operating environments, and restructuring it. Restructuring the system includes modularization and optimization [9].

Table 6.1 Summary of Systems Architecting Heuristics

View	*Category*	*Heuristic*
Functional architecture	Scoping	SA_Heu_1. Use functional hierarchy.
		SA_Heu_2. Limit the side effects of functions.
Physical architecture	Scoping and planning	SA_Heu_3. Each hierarchical level of the system provides a context for the level below.
		SA_Heu_4. Architecture refinement is complete when the engineer considers that the system can be built to the client's satisfaction.
	Selection of the solution physical architecture	SA_Heu_5. The choice between possible physical architectures may depend on which set of drawbacks the client can handle best.
	Structuring the physical architecture	SA_Heu_6. Group and allocate functions that are strongly related to each other, and separate functions that are unrelated.
		SA_Heu_7. Promote subsystems implementation independency.
		SA_Heu_8. Choose a solution configuration with minimal communications between the subsystems.
		SA_Heu_9. System architecture should follow functional architecture
	Decomposition and interfacing	SA_Heu_10. Use defined criteria for system decomposition.
		SA_Heu_11. The greatest leverage and danger in systems architecting is at the interfaces.
	Design margins	SA_Heu_12. Consider design margins.

The heuristics for affecting system maintainability are those to be applied at the system or subsystem level during step 4 of the ISE&PPOOA process.

The heuristics for affecting software maintainability are those applied to the PPOOA process for architecting software. These are organized into three categories: those that localize expected modifications, those that restrict the visibility of responsibilities, and those that prevent the ripple effect. Each category is described below.

6.3.1 Reliability and Maintainability Heuristics for Step 4: Physical Architecture of the ISE Process

Man_Heu_1: Change independence of system parts. Change independence means that changing one system part does not force changes in other parts of the system. Suh has a more complete definition of independence. He applies the concept of independence at the functional requirements level and applies the independence axiom for defining coupled design and uncoupled design. Uncoupled design is the design where each functional requirement can be satisfied independently by means of one design parameter [10].

Man_Heu_2: Identify system parts that are likely to change. For the system parts that are likely to change, put extra effort into designing their interfaces [5].

Man_Heu_3: Facilitate the system detection and isolation of faults. Self-diagnostic is a system capability that contributes to maintainability and safety as well [11].

Man_Heu_4: Facilitate functional and physical substitution. Physical substitution is facilitated by interfaces standardization that is connectors, pins, and mountings that allow physical interchangeability. If two physical parts are not designed to be functionally interchangeable, they should not be physically interchangeable [11].

Man_Heu_5: High failure parts should be more accessible. High failure parts should be more accessible than those that seldom fail [11]. This heuristic can be applied when location and size issues are considered during system architecture.

Man_Heu_6: Reliability of all parts in a composite part should be equal. The part with lowest reliability in a composite part impacts in its overall reliability [11].

6.3.2 Maintainability Heuristics for Software Architecting: PPOOA Process

The responsibilities that are allocated to software components greatly influence the cost of making a change. Depending on how the allocation is done during the software architecting stage, a specific change can affect either a single software component or multiple ones. The goal of this set of heuristics is to directly affect as few software components as possible with a single change by presenting guidelines for how responsibilities are allocated.

The heuristics for localizing expected modifications are given below.

Man_Heu_7: Maintain semantic coherence. Semantic coherence refers to the relationships between a software component's responsibilities. Semantically coherent responsibilities are related by what they do, carrying out the same or at least similar functions (functional cohesion). The goal here, ensuring that the software component's responsibilities all work together without excessive reliance on other software components, is achieved by choosing responsibilities that have some sort of semantic coherence. Doing so binds responsibilities that are likely to be affected by a change [12]. This heuristic may be applied during domain modeling of the software subsystems or the step 2.a.1 of the PPOOA process (see Chapter 4).

Man_Heu_8: Isolate the expected change. Separating the responsibilities that are likely to change from those that are unlikely to change separates software architecture into fixed and variant parts. This enables more design effort to be devoted to making a selected subset of software components as easy to change as possible [12].

Man_Heu_9: Raise the abstraction level. Raising the abstraction level, and thus making a software component more general, allows that component to use a broader range of functions based on input [12]. Raising the level of abstraction for a responsibility involves parametrization of its activities [13].

Man_Heu_10: Limit options. Limiting the set of possible modifications will reduce the variations that need to be considered in the design and simplify constructing a software subsystem suitable for modification [12].

Man_Heu_11: Abstract common services in the primary software components. Localize services that are used commonly by a variety of consumer software components [13].

6.3.3 Heuristics for Restricting the Visibility of Responsibilities

If a software component is affected by a change, it is important to know whether that change will become visible outside the software component. If it will, changes to other dependent components will most likely be required.

The heuristics for restricting visibility are given below.

Man_Heu_12: Hide information. This heuristic is based on the well-known information hiding software engineering paradigm [14].

Applying the heuristic divides the component's responsibilities into two categories: public and private. Public responsibilities are those that are visible from both inside and outside the software component. Private responsibilities are those that are visible only from inside the software components. Encapsulation acts by enforcing information hiding, thus making the public responsibilities visible through an interface [12].

Man_Heu_13: Maintain existing interfaces. This heuristic is based on keeping interfaces across a particular change during the software architecting stage. That is, the

software component developer will maintain the old identity, syntax, and semantics of an existing interface even if the modification changes it [12].

Man_Heu_14: Separate the interface from the implementation. This heuristic allows the realization of the implementation later in the development process than the interface specification [12].

6.3.4 Heuristics for Preventing Unintended Effects

An unintended effect from a modification is the necessity for making changes to software components that are not directly affected by that change. This necessity occurs because of a dependency between the component that is directly affected by the modification and another component that is dependent on it.

The heuristics for preventing unintended effects are given below.

Man_Heu_15:_Break the dependency chain. This heuristic refers to the use of an intermediary to keep one component from being dependent on another and therefore to break the dependency chain [12]. In the PPOOA architecture framework, this heuristic is implemented by the use of coordination mechanisms. As part of the development of the PPOOA architecture framework, 14 coordination mechanisms were analyzed and a taxonomy was developed [15]. Of those mechanisms, the following have been considered as part of the PPOOA framework:

- Bounded Buffer;
- General-Semaphore;
- Mailbox;
- Transporter;
- Rendezvous.

Man_Heu_16:_Make the data self-identifying. Tagging the data with identification information, such as sequence number, syntax descriptions, or their identity, will break the software components dependencies on either sequencing or syntax [12].

Man_Heu_17:_Limit communication paths. Restricting the other software components with which a given software component will communicate has the effect of ensuring that no dependency exists between the two components [12].

The heuristics for software architecting and maintainability are summarized in Table 6.2.

6.4 Efficiency Heuristics

The efficiency heuristics help to identify architecture and design alternatives that are likely to meet performance efficiency qualities that are related to time behavior, resource utilization, and capacity. These heuristics are neither new nor revolutionary. Some of them represent design decisions currently made by engineers. However, the heuristics described below generalize and summarize the knowledge and experience that performance engineers use in constructing software intensive systems.

Table 6.2 Maintainability Heuristics

Quality	*Category*	*Heuristic*
Maintainability	Localizing expected modifications	Man_Heu_7. Maintain semantic coherence.
		Man_Heu_8. Isolate the expected change.
		Man_Heu_9. Raise the abstraction level.
		Man_Heu_10. Limit options.
		Man_Heu_11. Abstract common services in the primary software components.
	Restricting the visibility of responsibilities	Man_Heu_12.Hide information.
		Man_Heu_13. Maintain existing interfaces.
		Man_Heu_14. Separate the interface from the implementation.
	Preventing unintended effects	Man_Heu_15. Break the dependency chain.
		Man_Heu_16. Make the data self-identifying.
		Man_Heu_17. Limit communication paths.

This section presents heuristics for efficiency-oriented design. They are grouped into three categories:

1. *Heuristics for managing demand*: These heuristics help to control the system resources demand;
2. *Heuristics for arbitrating demand*: These heuristics control preemption and waiting times when there are competing requests and contention for shared resources;
3. *Heuristics for managing multiple resources*: These heuristics enable multiple resources to be used efficiently to ensure that available resources are used when they are needed.

6.4.1 Heuristics for Managing Demand

These heuristics help design the solution considering the efficiency of the system by controlling the resources demand.

Eff_Heu_1: Quantify performance. This heuristic refers to the definition of specific, quantitative, measurable performance efficiency requirements at the system level of interest. Therefore, it is important to control how often the system events are generated and place a limit on how much execution time is used to respond to a particular event.

Performance efficiency requirements explicitly state the required performance efficiency rigorously enough so that the evaluator can quantitatively determine whether or not the architecture meets that requirement.

In the case of ISE&PPOOA, the flows of system activities representing its behavior are represented using activity diagrams where engineers can place a limit on how much execution time is used to respond to the event initiating the flow.

Eff_Heu_2: Instrument the system. Instrument the system as you build it to enable measurement and evaluation of workload scenarios, resource requirements, and efficiency requirements compliance [16]. This heuristic when applied to a software subsystem is

related to inserting code probes at key points to enable the measurement of its execution characteristics. The validator needs information on resource requirements for critical portions of code, not just the total for the software. To collect this data, the developers must insert code to call system-timing routines, and write key events and relevant data to files for later analysis.

Eff_Heu_3: Identify dominant workload. The heuristic purpose is to identify the dominant workload system parts and minimize their processing, focusing attention on the parts of the architecture that have the greatest impact on performance.

The heuristic is concerned with identifying the subset (the 20% or less) of the system components that will be used the most (80% or more) of the time. These frequently used components are the dominant workload. These dominant workload also cause a subset (=< 20%) of the operations in a software subsystem to be executed most (>= 80%), as well as the code within operations, and so on. Improvements made in these dominant workload functions thus have a significant impact on the overall performance of the system [16].

For ISE&PPOOA architectures involves identifying the critical causal flows of activities (CFAs). These are flows of activities that are critical to the operation of the system or that are important to responsiveness as seen by a user. Critical flows may also include those that relate to risks involving efficiency.

EFF_Heu_4: Fixing point. This heuristic related to software addresses that for better responsiveness, fixing should establish connections at the earliest point in time, so that retaining the connection is cost-effective. Fixing connects the desired action to the computer instructions used to accomplish that action or the desired result to a data used to produce it.

The fixing point is a point in time. The latest the fixing point is during execution, just before the instructions are to be executed. Dynamic binding in object-oriented languages or polymorphic function calls that cannot be resolved during compilation, exhibit late fixing. Fixing can also establish the connection at a several possible earlier times: earlier in the execution, at system initialization, during compilation, or even outside the software. In some cases, early fixing may reduce the flexibility of the design [16].

EFF_Heu_5: Processing versus frequency heuristic. This heuristic states that the product of processing times frequency should be minimized. This heuristic is concerned with the amount of work done in processing a request and the number of requests received. It seeks to make a trade-off between the two. It may be possible to reduce the number of requests by doing more work per request, or vice versa [16].

6.4.2 Heuristics for Arbitrating Demand

These heuristics control preemption and waiting times when there are competing requests and contention for shared resources

Eff_Heu_6: Use parallel processing when possible. If system requests can be processed in parallel, the blocked time can be reduced [9].

The parallelism may be real or apparent. In real parallelism, the execution threads execute simultaneously in different processors. In this case, the processing time is reduced by an amount proportional to the number of processors. In apparent parallelism, the threads are multiplexed on a single processor. The PPOOA framework supports apparent parallelism by the multiplexed execution of software processes components.

Eff_Heu_7. Use shared resources when possible. In general, a resource is any hardware or software that is accessible by the software components of a subsystem. This heuristic recommends sharing resources when possible to reduce contention delays. When exclusive access is required, it is important to minimize the sum of the waiting time and the servicing time [16].

Eff_Heu_8. Determine appropriate scheduling policy. This heuristic focuses on the policies to assign processing time to software processes. The appropriate software processes scheduling policy depends on the system´s performance requirements. Scheduling solutions include offline scheduling, time-based scheduling, semantic-importance-based scheduling, aperiodic servers, and fairness-based scheduling.

Eff_Heu_9. Use synchronization protocols. When more than one software process (thread of execution) needs to access a shared resource (such as a queue, structure, or domain component in a mutually exclusive manner), coordination mechanisms such as semaphores or mutexes are commonly used. Various protocols including first in, first out (FIFO), priority inheritance, and priority ceiling protocols are used by the real-time systems community to avoid problems such as priority inversion.

6.4.3 Heuristics for Managing Multiple Resources

These heuristics enable multiple resources to be used efficiently to ensure that available resources are used when they are needed.

Eff_Heu_10: Balance the computing load. The heuristic is balance the computing load when possible by processing conflicting loads at different times or in different hardware resources.

This heuristic is similar to the shared resources heuristic, they both address resource contention delay. The shared resources heuristic reduces the delay by minimizing waiting time and servicing time. This heuristic reduces the delay by reducing the number of software processes that need a hardware resource at a given time and by reducing the amount of resource that they need. Evaluating solution alternatives requires the use of performance engineering models to quantify resource contention delays for each alternative [16].

Eff_Heu_11_Locality. This heuristic promotes the creation of actions, functions, and results that are close to the physical computer resources used.

The types of locality are spatial, temporal (i.e., time), effectual (i.e., purpose or intent), and degree (i.e., intensity or size) [16].

In object-oriented architectures, such as those build with the PPOOA framework, it is important to keep related data and behavior together in the same software component. An object or software component instance should have most of the data that it needs to make a decision or perform an action. Software components that have very frequent interactions should be assigned to the same processor and should perhaps even be compiled and linked together.

Table 6.3 summarizes the efficiency heuristics described above.

6.5 Safety Heuristics

Safety is related to the absence of catastrophic consequences on health, property, and environment of an accidental harm, addressing the degree to which this accidental harm is prevented, identified, reacted to, and adapted to.

Here safety extends the Avizienis et al. taxonomy of dependable and secure computing [17] where a fault may cause an error that may lead to a failure, which in a safety context presented here, can cause harm.

Safety stands out as an emergent property of a system that should be considered within the dependability quality attribute. The dependability of a system is its ability to avoid service failures that are more frequent and more severe than is acceptable.

Security heuristics go hand in hand with safety and are applied to resist, detect, and recover from intentional attacks or malicious harm. In security the emphasis tends to be on data assets providing their confidentially, integrity, as well as availability.

Dependability is an integrating quality attribute that encompasses the following attributes:

- *Availability:* Readiness for correct service;
- *Reliability:* Continuity of correct service;
- *Safety:* System absence of catastrophic consequences on the humans and the environment;

Table 6.3 Efficiency Heuristics

Quality	*Category*	*Heuristic*
Efficiency	Managing demand	Eff_Heu_1. Quantify performance.
		Eff_Heu_2. Instrument the system.
		Eff_Heu_3. Identify dominant workload.
		Eff_Heu_4. Fixing point.
		Eff_Heu_5. Processing versus frequency heuristic.
	Arbitrating demand	Eff_Heu_6. Use parallel processing when possible.
		Eff_Heu_7. Use shared resources when possible.
		Eff_Heu_8. Determine appropriate scheduling policy.
		Eff_Heu_9. Use synchronization protocols.
	Managing multiple resources	Eff_Heu_10. Balance the computing load.
		Eff_Heu_11. Locality.

- *Integrity:* System absence of improper system alterations;
- *Maintainability:* Ability to undergo modifications and repairs [17].

The definition of an error is the part of the state of the system that may lead to its subsequent failure or deviation from correct behavior. It is important to note that many errors do not reach the system's external state and cause a failure. A fault is active when it causes an error, otherwise it is dormant [17].

Safety concerns are the identification and management of hazardous conditions with catastrophic consequences. Hazards may be considered as sets of system conditions that, together with environmental conditions, will lead to a loss event [18].

The system hazards are caused by faults related to system design, material, workmanship, or operating procedures.

Safety heuristics are organized into four groups: heuristics for hazard avoidance, heuristics for hazard reduction, heuristics for hazard control, and heuristics for mitigation the effects.

6.5.1 Heuristics for Hazard Avoidance

SF_Heu_1: Concentrate on dysfunctional system behavior. Most hazard analysis techniques use the physical architecture of the system rather than its functional architecture. For software-intensive complex systems, Young and Leveson propose an integrated approach based on systems theory where the goal is to ensure the system critical functions and the services the system provides are maintained in the face of disruptions [19]. They propose to identify, in the functional architecture of the system, those unsafe control actions that either lead to a hazard, don't prevent some hazard, are too early or too late of sequence, continue too long, or stop too soon [19].

SF_Heu_2: Minimize the number of components and interactions. This heuristic is related to system structuring heuristics particularly with SA_Heu_10. Here, the heuristic aims to simplify system design through minimizing the number of components and their interactions. The principles of minimum communications and proper partitioning are critical to fault isolation.

SF_Heu_3: Avoid undeterministic behavior. This heuristic aims to enforce a specific sequence of events or actions by controlling all access to shared system resources. This heuristic is related to the efficiency heuristics described above and related to synchronization protocols used for accessing the shared system resources. This heuristic can be used when correct sequencing of events must be ensured, especially when failures are safety concerns. It can be implemented by either hardware or software.

6.5.2 Heuristics for Hazard Reduction

SF_Heu_4: Enforce time requirements. This heuristic aims to enforce execution time onto system components. Time-out is a particularly common mechanism for detecting omission or timing failures that are critical safety concerns. Time-out can

be modeled in the system activity diagrams. Time-out is cheap and easy to implement through either hardware or software. An implicit knowledge of the permissible execution times that the system relevant parts will take is required. Protection actions, such as recovery, are also required upon the missed deadline detection [20].

SF_Heu_5: Sanity check. Wu proposes this heuristic (also known as reasonableness check) aims to enforce the validity or integrity of the output of a specific system component. This heuristic can be applied when the reliability of specific operations or outputs of a component can be anticipated. It is often used to detect value failures of a system component. It has a wide applicability in software intensive systems at different levels ranging from hardware, code to subsystem, and system levels [20].

SF_Heu_6: Redundancy. Redundancy involves multiple part copies but may do it with different design. This heuristic reduces the occurrence of system failures by using multiple copies of similar system parts. One of the redundant components should achieve the functionality allocated regardless the operational state of the other similar parts.

SF_Heu_7: Recovery. Backward recovery heuristic involves returning in time to a previous known state upon the detection of failures of a component and then retrying execution. It simply attempts to simulate the reversal of time, assuming that the earlier known state will not recreate the failure.

6.5.3 Heuristics for Hazard Control

SF_Heu_8: Use partitions. This heuristic aims to contain of a software failure to a particular time and space partition. Separation may be hardware-enforced separation among system execution contexts. In this case, a partition is the basic entity that is typically managed by a hypervisor. It may be considered as a container that consists of isolated processor and memory resources with policies on device access. A hypervisor is traditionally implemented as a software layer but it can also be implemented as code embedded in a system's firmware. A partition as used, for example for the Integrated Modular Avionics (IMA) architectures, is a lighter-weight concept than a virtual machine and could be used outside of the context of virtual machines to provide a highly isolated execution environment.

SF_Heu_9: Select the most appropriate output. Selecting the most appropriate output among the outputs of a set of system parts masks the effect of the failed part(s). The heuristic is applied by a voter that can then be implemented by either hardware or software. This heuristic can be used when a known failure is undetectable, there is nothing to act upon the detection, or there is a highly availability requirement.

SF_Heu_10: Promote system degraded states when possible. This heuristic aims to maintain a partial or degraded system functionality by removing noncritical parts from service. This heuristic is applied when part failures cannot be mitigated

through the recovery or containment heuristics actions. It may not require the combination with the detection function, as failed parts can be removed autonomously.

6.5.4 Heuristics for Mitigation of the Effects

SF_Heu_11:_Implement alerting capabilities. This heuristic recommends implementing system alerting capabilities to notify the operators/users/maintainers of a hazardous condition through graphical/vocal alarm information. This heuristic is applied when human intervention is immediately required to minimize the impact of a hazard.

SF_Heu_12:_Implement operational data recording functions. This heuristic prevents the repeat of an incident or near miss/ accident or loss in the future through on-line recording the operational data allowing the analysis of the possible causes of the hazardous condition.

Safety heuristics are summarized in Table 6.4.

6.6 Resilience Heuristics

Currently, resilience is a popular term with diverse meanings in such fields as psychology, ecology, and materials science. In the systems engineering domain, resilience goes beyond safety in that safety is concerned with avoiding or mitigating harm while resilience deals with standing and recovering from disruptions.

Threats are the source for disruptions and may include human operation and maintenance errors, design flaws, natural disasters, and intentional attacks.

This section proposes resilience heuristics to be applied in the ISE&PPOOA systems architecting process. Therefore, heuristics for physical systems proposed

Table 6.4 Safety Heuristics

Quality	*Category*	*Heuristic*
Safety	Hazard avoidance	SF_Heu_1. Concentrate on dysfunctional system behavior.
		SF_Heu_2. Minimize the number of components and interactions.
		SF_Heu_3. Avoid undeterministic behavior.
	Hazard reduction	SF_Heu_4. Enforce time requirements.
		SF_Heu_5. Sanity check.
		SF_Heu_6. Redundancy.
		SF_Heu_7. Recovery.
	Hazard control	SF_Heu_8. Use partitions.
		SF_Heu_9. Select the most appropriate output.
		SF_Heu_10. Promote system degraded states when possible.
	Mitigation of the effects	SF_Heu_11. Implement alerting capabilities.
		SF_Heu_12_Implement operational data recording functions.

in the literature [4] were selected and adapted for their use in the ISE&PPOOA architecting process. Resilience heuristics related to organizational systems are not considered here.

Jackson and Ferris propose to organize resilience heuristics into four groups: those related to surviving a threat, adapting to a threat, degrading gracefully in the face of a threat, and acting as a whole in the face of a threat [4].

6.6.1 Heuristics for Surviving a Threat

RS_Heu_1: Functional redundancy. This heuristic, also called diversity proposes the design of alternative system parts performing a particular function. This heuristic compensates for the weakness of the SF_Heu_6 redundancy where redundancy is understood as physical redundancy, which recommends using multiple copies of the same part. The weakness is that multiple copies of the same part have the same design flaws.

RS_Heu_2: Layered defense. This heuristic implies the application of two or more heuristics to the same vulnerability in the system. The more layers there are, the more resilient the system will be [4].

6.6.2 Heuristics for Adapting to a Threat

RS_Heu_3: Restructuring. If a threat is affecting the system, it has the capability of dynamically changing the physical architecture that is either the physical parts hierarchy or their dependencies.

RS_Heu_4: Reparability. If a threat is affecting the system, it has the capability for a transition to another state with partial or full functionality in a specified context. This heuristic is related to RS_Heu_3 Restructuring in the case of autonomous self-repairable systems, where the system reconfigures itself considering the physical parts that are healthy and the functionalities they implement.

A robotics solution implementing these heuristics to be resilient in the face of uncertainty proposes a control loop that will leverage at run time the knowledge in the model. This way, the control action executed at run time to achieve it is not determined at design, but performed on the fly by the loop using information of the instantaneous state, in addition to the model, and therefore accommodating the disruptions [21].

RS_Heu_5: Human back-up. As proposed by Madni and Jackson, the human operator should be able to back up system-automated tasks when the system is not sensitive to the threat and there is enough time for human operator intervention [22].

RS_Heu_6: Human in the loop. The concept of human-centered automation is supported by a set of principles described in the literature and applied by Billings in the aviation domain. One of the principles described by Billings is that a human

operator must be informed to conduct operations under changing contextual conditions. Here information is not mere data. Information is presented in a way that is meaningful to the human operator needs in a given context [23]. Humans should be in the loop when rapid cognition and creative opinion is needed [22].

RS_Heu_7: Monitor the operator. This heuristic goes beyond the RS_Heu_6. human in the loop heuristic. Since human error is a frequent contributor to accidents, there is a need to monitor human operator behavior. The intention of the human operators must be explicit and communicated to the parts of the system related to the autonomy capabilities.

6.6.3 Heuristics to Degrade Gracefully

RS_Heu_8: Neutral state. The system is designed to prevent further damage from occurring when affected by a threat until a diagnosis can be performed. Neutral state implies some delay taking action when there is an opportunity to perform the correct action.

RS_Heu_9: Drift correction. Evidence for an approaching threat may be used for either avoiding it or diminishing it through corrective action. Illustrative examples of a solution implementing this heuristic is a sense and avoid functionality on board an unmanned aerial vehicle or the collision avoidance functionality of an autonomous robot.

6.6.4 Heuristics to Act as a Whole in the Face of a Threat

RS_Heu_10.:Identify and reduce undesired interactions between system parts. A lack of a holistic systemic view can produce system parts' interactions that result in unplanned effects. Thermal, vibration, and electromagnetic interferences are typical examples of these interactions.

Table 6.5 summarizes the above resilience heuristics.

Table 6.5 Resilience Heuristics

Quality	*Category*	*Heuristic*
Resilience	Surviving a threat	RS_Heu_1. Functional redundancy.
		RS_Heu_2. Layered defense.
	Adapting to a threat	RS_Heu_3. Restructuring.
		RS_Heu_4. Reparability.
		RS_Heu_5. Human backup.
		RS_Heu_6. Human in the loop.
		RS_Heu_7. Monitor the operator.
	Degrade gracefully	RS_Heu_8. Neutral state.
		RS_Heu_9. Drift correction.
	Act as a whole in the face of a threat	SF_Heu_10. Identify and reduce undesired interactions between system parts.

6.7 Software Architecting Heuristics Using the PPOOA Framework

In this section the heuristics dealing with the application of the building elements supported by the PPOOA architecture framework described in Chapter 4 are presented. As described in Chapter 4, PPOOA is an architecture framework supporting the design of the software architecture for a real-time system. The PPOOA vocabulary of building elements is composed of software components and coordination mechanisms described in Chapters 4 and 7.

Here the heuristics or basic principles related to the software architecture design are described. Users of the ISE&PPOOA process should follow these heuristics when applying the software architecting subprocess (described in Section 4.4.2 of Chapter 4) for those software-intensive subsystems that are part of the main system.

PPOOA_Heu_1. Foundations of the PPOOA framework-explicit concurrency. Frequently, object-oriented software architectures represent concurrency implicitly encapsulating processes or control activities in the objects. In the PPOOA architecture framework, explicit concurrency has been chosen. Therefore, concurrency is represented externally to the software objects by means of the CFA described in Chapter 4. CFAs or flows of activities are controlled by software processes or by a controller object. This principle allows taking into account earlier concurrency issues and temporal behavior evaluation. Therefore, it allows evaluating design decisions before other approaches employing implicit concurrency.

The use of several software processes in the same CFA is permitted. Each software process is in charge of a segment of the CFA activities. This allows having several instances of the CFA at the same time.

PPOOA_Heu_2. Software component selection criteria. The basic criteria for the selection of a software component as subsystem building element are the responsibilities to be implemented in the component, the need to maintain an internal state, and the concurrent activities to support.

The responsibilities to be implemented in a software component can be related to computing, storing of data, control, or signaling activities.

The best candidate for performing pure computation is the algorithm component. If the component has to maintain an internal state, the best candidate is the domain component, but if typical data structures have to be stored, the best candidate is the structure-building element.

Process or controller building elements are required for performing control activities related for example to the triggering of an event response, called CFA in the PPOOA framework. The concurrent activity control is responsibility of this type of building elements as well.

Periodic activities are implemented by a periodic process. Activities triggered by sporadic events need aperiodic process for their implementation, and in certain cases, due to complex event handling, they are supported by a controller component.

PPOOA_Heu_3. Physical device management. Those activities related to physical device management are better supported by a controller component. The controller

building element or component is the most complex of all PPOOA framework building elements from a time responsiveness assessment, but it is the most flexible regarding execution flows or supported interfaces.

PPOOA_Heu_4. Components operations restrictions. The type of operations a PPOOA building element can support is a main issue when selecting it. They are summarized below:

- The algorithm component provides computing operations. These operations are executed immediately when called.
- The domain component may provide reading, writing, and computing operations. These operations are executed immediately when called.
- The structure component may provide reading and writing operations. These operations are executed immediately when called.
- In general cyclic or periodic processes only provide operations for their deactivation. Aperiodic process components provide an operation for their activation. Aperiodic processes can provide other operations as well. These operations are executed immediately when called.
- The controller component has no restrictions for its provided operations.

PPOOA_Heu_5: Use of coordination mechanisms to solve communication and synchronization issues. There are three problems that need to be solved in real-time software intensive systems the mutual exclusion problem, producer-consumer problem, and multiple readers-writers problem.

The mutual exclusion problem occurs when software-building elements or components need exclusive access to resources, shared data, or physical devices. The careful use of a semaphore, or mutex, as coordination mechanism can solve the problem.

The producer-consumer problem arises when a software component needs to communicate with another software component to pass data. The PPOOA framework promotes the buffer coordination mechanism to solve this problem.

The multiple readers-writers problem is similar to the mutual exclusion problem but readers do not need to exclude one another. It is a typical situation related to database access.

6.8 Summary

This chapter presented a collection of heuristics to be applied in the ISE&PPOOA process. This collection of heuristics is the means promoted in the methodology for satisfying the nonfunctional requirements, also called quality attributes requirements. In many case these nonfunctional requirements are not allocated as the functional ones, so the heuristics specify how these requirements can be implemented through design decisions to meet them.

The characteristics of the proposed heuristics are as follows:

Table 6.6 PPOOA Framework Heuristics

View	*Category*	*Heuristic*
Software architecture	Principle	PPOOA_Heu_1. Foundations of the PPOOA framework-explicit concurrency.
	Building elements usage	PPOOA_Heu_2. Software component selection criteria.
		PPOOA_Heu_3. Physical device management.
		PPOOA_Heu_4. PPOOA Components operations restrictions.
		PPOOA_Heu_5. Use of coordination mechanisms to solve communication and synchronization issues.

- A heuristic is a bridge between the quality attribute requirement and the physical architecture.
- A heuristic is based on the knowledge and experience from previous engineering projects in domains, such as maintainability, efficiency, safety, and resilience.
- Heuristics are neither absolute nor independent. The application of a heuristic may also require additional heuristics to be applied
- The application of some heuristics may generate conflicts to be solved.

The application of heuristics may be considered as the art of the system architecting process when human decisions and trade-offs are critical, so their automation by artificial intelligence tools is still an open issue [24].

6.9 Questions and Exercises

1. Identify two dependent heuristics.
2. Describe a solution implementing the functional redundancy heuristic.
3. What do we need to use resilience heuristics?
4. Describe a solution implementing the human in the loop heuristic.
5. Which PPOOA building element is recommended to implement periodic activities?
6. Which PPOOA building element is recommended to handle a physical device?

References

[1] ISO/IEC 25010, "Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) -– System and Software Quality Models," CH-1211 Geneva 20, International Standards Organization, 2011.

[2] Firesmith, D., "Engineering Safety Requirements, Safety Constraints, and Safety-Critical Requirements," *Journal of Object Technology*, Vol. 3, No. 3, March–April 2004, pp. 27–42.

[3] United States Government, The White House, National Security Strategy, Washington, DC: 2010.

[4] Jackson, S., and T. L. J. Ferris, "Resilience Principles for Engineered Systems," *Systems Engineering*, Vol. 16, 2013, pp. 152–164.

[5] Bahill, T. A., and R. Botta, "Fundamental Principles of Good System Design," *Engineering Management Journal*, Vol. 20, No. 4, December 2008, pp. 9–17.

[6] Maier, M. W., and E. Rechtin, *The Art of Systems Architecting*, Second Edition., Boca Raton, FL: CRC Press, 2000.

[7] Wheatcraft, L., "Interface Requirements vs IRDs vs ICDs," November 15, 2013. http://reqexperts.com/blog/2013/11/interface-requirements-vs-irds-vs-icds/.

[8] Wasson, C. S., *System Analysis. Design, and Development. Concepts, Principles and Practices*, Hoboken, NJ: John Wiley & Sons, 2006.

[9] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice,* Second Edition, Reading, MA: Addison Wesley Longman, 2003.

[10] Suh, N. P., *Axiomatic Design. Advances and Applications*, New York: Oxford University Press, 2001.

[11] Ebeling, C., *An Introduction to Reliability and Maintainability Engineering,* Second Edition, Waveland Press, 2010.

[12] Bachmann, F., L. Bass, and M. H. Klein, *Deriving Architectural Tactics: A Step toward Methodical Architectural Design*, CMU/SEI-2003-TR-004, Software Engineering Institute, Carnegie Mellon University, 2003, http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6593.

[13] Bachmann, F., L. Bass, and R. Nord, *Modifiability Tactics*, CMU/SEI-2007-TR-002, Software Engineering Institute, Carnegie Mellon University, 2007, http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8299.

[14] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, 1972, pp. 1053–1058.

[15] Fernandez, J. L., *A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis*, CMU/SEI-93-TR-034, Software Engineering Institute, Carnegie Mellon University, Dececember 1993, https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=12011.

[16] Smith, C. U., and L. L. G. Williams, *Performance Solutions. A Practical Guide to Creating Responsive, Scalable Software*, Indianapolis, IN: Pearson Education, 2002.

[17] Avizienis, A., et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, 2004, pp. 11–33.

[18] Leveson, N. G., *Engineering a Safer World. Systems Thinking Applied to Safety*, Cambridge, MA: The MIT Press, 2011.

[19] Young, W., and N. G. Leveson, "An Integrated Approach to Safety and Security Based on Systems Theory," *Communications of the ACM*, Vol. 57, No. 2, 2014, pp. 31–35.

[20] Wu, W., *Architectural Reasoning for Safety-Critical Software Applications*, D.Phil. thesis, YCST-2007–17, Department of Computer Science, University of York, United Kingdom: 2007.

[21] Hernández, C., J. L. Fernandez-Sánchez, G. Sánchez-Escribano, J. Bermejo-Alonso, and R. Sanz, "Model-Based Metacontrol for Self-adaptation," In H. L. et al., editor, Intelligent Robotics and Applications (ICIRA 2015), vol. 9244 of Lecture Notes in Artificial Intelligence, pp. 643–654, Springer, 2015.

[22] Madni, A., and S. Jackson, "Towards a Conceptual Framework for Resilience Engineering," *IEEE Systems Journal*, Vol. 3, No. 2, 2009, pp. 181–191.

[23] Billings, C. E., Human-Centered Aviation Automation: Principles and Guidelines, NASA Technical Memorandum 110381, Ames Research Center, Moffett Field, CA, 1996.

[24] Fernandez, J. L., and J. Carracedo, "An Autonomous Assistant for Architecting Software," Proc. 20th International Conference on Software and Systems Engineering and their Applications, ICSSEA 2007, Paris, December 4–6, 2007.

CHAPTER 7

Physical Architecture

This chapter deals with the creation of the physical architecture of the system by transformation of the functional architecture, which lies at the core of ISE&PPOOA. First, we discuss the building blocks used in the physical architecture. Next, we explain the allocation of the functions identified in the functional architecture to the physical building blocks to achieve modularity. The key role that modularity and heuristics play in this iterative process is explained. The resulting physical architecture is then explained focusing on the logical and physical connectors obtained to link the building blocks. Finally, the domain model is presented as the bridge in ISE&PPOOA from the system physical architecture to the software architecture, and we discuss of the role of the software components in the architecture.

7.1 Physical Architecture in Systems Engineering

7.1.1 Main Concepts Related to the Physical Architectural Model

Some of the main concepts related to the modeling of a physical architecture were presented in the conceptual model of the ISE&PPOOA method described in Chapter 4; other concepts more specific to the physical architecture and borrowed from the literature are presented here.

There are different models to represent the hierarchical structure of modern, complex engineered systems, but the concepts used are easily translatable from one model to another. For example, Kossciakoff [1] defines two intermediate levels of components and subcomponents in addition to the common levels of subsystems and parts, at the top and the bottom of the system's structure respectively.

The ISE&PPOOA conceptual model presented in Chapter 4 defines the following main elements to represent the hierarchical structure of a system's physical architecture:

System: A combination of interacting parts organized to achieve one or more stated purposes.

Part: A building element of the system. Parts may be composite parts containing other parts. A part depends on other parts. A subsystem is typically considered a major composite part of the system that performs a closely related set of functions. Since ISE&PPOOA addresses the high-level conceptual design of the system, parts here correspond to the main physical building blocks of the system. Other authors, such as Kossiakoff [1], refer to them as components, which are the physical

embodiments of the functional elements consisting of hardware and software. In SysML, a *block* is the basic unit of structure and represents definition of types of building elements, whereas a SysML *part* property represents an instance of one of those types [2].

Physical interface: Physical interfaces are the descriptions of the physical dependencies between system parts. The physical interface between two or more parts of the system is represented by a port and a connector. A connector represents the interaction between two parts, which may consist of the exchange of information (either data or signals), matter or energy. This exchange is modeled in SysML by item flows assigned to a connector. In SysML, ports allow for a robust and flexible definition of interfaces [3]. A port represents a distinct interaction point at the boundary of a part/block, decoupling that interaction from the internal implementation inside the part that the port connects to.

7.1.2 Functional, Physical, and Quality Trees in ISE&PPOOA

The ISE&PPOOA process develops the design of an engineered system, through three simultaneous breakdowns of the problem, resulting in corresponding trees structures:

1. Functional;
2. Quality attributes;
3. Physical.

At each level of the hierarchy, requirements are transformed into a functional architecture and quality models for the nonfunctional requirements (NFRs). Functional allocation (see arrow in Figure 7.1) is used to obtain the main building blocks of the physical architecture at that level, mapping the functional and physical architectures. However, the mapping is not direct between the three hierarchies, since for the NFRs the mapping to the physical architecture is through design heuristics. The application of design heuristics results in a refined physical architecture at that level and requirements for the next level elements (see bottom-right part of Figure 7.1). This breakdown proceeds iteratively to the lower level (double-headed thin arrow in Figure 7.1) until the specification of the basic components to be built or purchased for the system are created.

7.1.3 Other Architecture Models

While ISE&PPOOA maintains an explicit separation of the functional and the physical architecture models of the system and develops them iteratively, this is not the case in other popular MBSE methodologies.

For example, the object-process model OPM [4] mentioned in Chapter 3 is a conceptual modeling language and methodology that can be used to formally specify the function, structure, and behavior of systems. Compared to SysML, OPM claims to offer great simplicity while preserving representational power. Structure is represented via objects and structural relations among them, such as

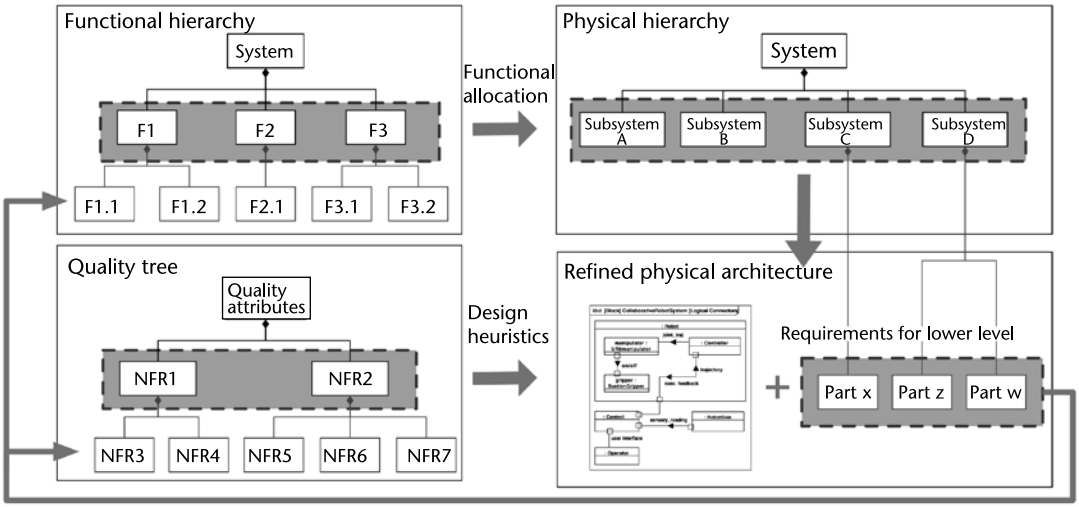


Figure 7.1 The iterative evolution of the three trees, functional, quality, and physical, during the ISE process.

aggregation-participation (whole-part relation) and generalization-specialization ("is-a" relation). The aggregation relation has an associated composition/decomposition mechanism, called in-zooming/out-zooming in OPM, which allows modeling of the hierarchical decomposition. When applied to the objects of an OPM model, it results in the physical hierarchy, although OPM does not explicitly addresses the concept of functional or physical architecture. Physical interfaces do not have an explicit representation in OPM, but the structural links allow defining of the connectivity between objects (parts), and structural tags allow to further refine the connection with semantics (e.g., content of that connection).

The OOSEM [3] method includes explicitly the development of a logical architecture that addresses similar concerns to the ISE&PPOOA functional architecture. However, this logical architecture is developed prior to and as an abstract view of the physical architecture, intended to support the synthesis of candidate physical architectures to satisfy the system requirements, and not in parallel to the physcial in OOSEM architecture, as in ISE&PPOOA. Both logical and physical architectures are defined using SysML block definition diagrams, internal block definition diagrams, and behavioral diagrams, by refining blocks with the stereotype node, which represents an aggregation (or set) of logical/physical components at a particular location. The logical components at each node are allocated to physical components at each node to constitute the node physical architecture. This logical-to-physical allocation can be shaped by design constraints identified during the system requirements analysis (e.g., reuse of COTS), leveraging architectural patterns, or performing trade-off analysis to select the preferred physical architecture based on criteria that optimizes the technical performance measures related to nonfunctional requirements.

7.2 Allocation and Modularity

In ISE&PPOOA, the physical architecture is obtained iteratively from the functional architecture, and in a separate hierarchy. As already discussed, this explicit separation of function (functional hierarchy) and implementation (physical architecture) maximizes the reusability of design solutions. Therefore, once we have the functional architecture of the system, the physical architecture is developed in the three-stage process described in Chapter 4: first we obtain the modular architecture by allocating the functions to the building elements of the solution, and then, we refine the architecture by using heuristics to address the NFRs, and finally the refined architecture is represented.

The first step to create the physical architecture is to identify the building elements (parts). The system design may be constrained to reuse legacy elements or predefined commercial off-the-shelf (COTS) elements. For example, most robotic applications in industry use a commercial robot manipulator, for which a large variety of models are available with different payload, reachability, or safety properties (see Section 9.3 for an example of how the physical architecture is constrained by the reuse of COTS). For this activity, heuristics of modularity/structuring (see Chapter 6) are applied to obtain the modular architecture.

The main criteria for allocation is modularity. The modularity principle stated by heuristics SA_Heu_6 to SA_Heu_10 (Chapter 6) says that the embodiment of functions in the modules must be cohesive and coupled. Cohesive because functions that are related to each other should be closely allocated (see Section 5.3 about cohesion), and coupled because functions with high rates of exchange should not be allocated to separate modules in order to minimize communications across the different modules. The N^2 charts of the functional interfaces are an excellent tool to analyze the coupling (N^2 charts were introduced in Section 5.5). As discussed by Bustnay and Bed-Asher [5], in the N^2 chart interesting issues can be observed, such as functions critically connected to several others, simple flows, where interfaces flow down to the same direction, or control loops. Functions can be reordered in the N^2 chart diagonal based on the coupling criteria, to cluster them and allocate them to the same or close modules. For example, when two functions have a high coupling, it is recommended that they are allocated in the same module. For example, Table 7.1 shows the functional interfaces for a steam generation process (the complete example can be found in Chapter 10), in which the subfunctions have been ordered in the diagonal of the N^2 chart following the modularity criteria for allocation. Consequently, Figure 7.2 shows that the related functions a4, a5, and a6 are allocated to parts in the same boiler subsystem of the modular architecture.

Another useful representation for clustering is the DSM [6], which is equivalent to an N^2. Sharman uses the IR convention in DSM: the inputs to an element are depicted in its row, and its outputs in its column. Algorithms exist to support in this endeavor of clustering for complex system using both formalisms N^2 [5] and the DSM [6].

7.2.1 Representation of the Modular Architecture

In ISE&PPOA the modular architecture is represented using a block definition diagram, internal block diagrams, and behavior diagrams. A SysML block definition

Table 7.1 N^2 Chart of the Functional Interfaces for a Steam Generation Process*

		Work			Flue gas	
F1: Expand steam	Expanded fluid					Work
	F2:Condense steam	Condensed fluid				Heat
		F3:Pump fluid	Compressed fluid			
			F4:Heat fluid	Saturated liquid		Flue gas
			Flue gas	*F5:Evaporate liquid*	Saturated steam	
Superheated steam				Flue gas	*F6:Heat steam*	

*See complete example in Chapter 10.

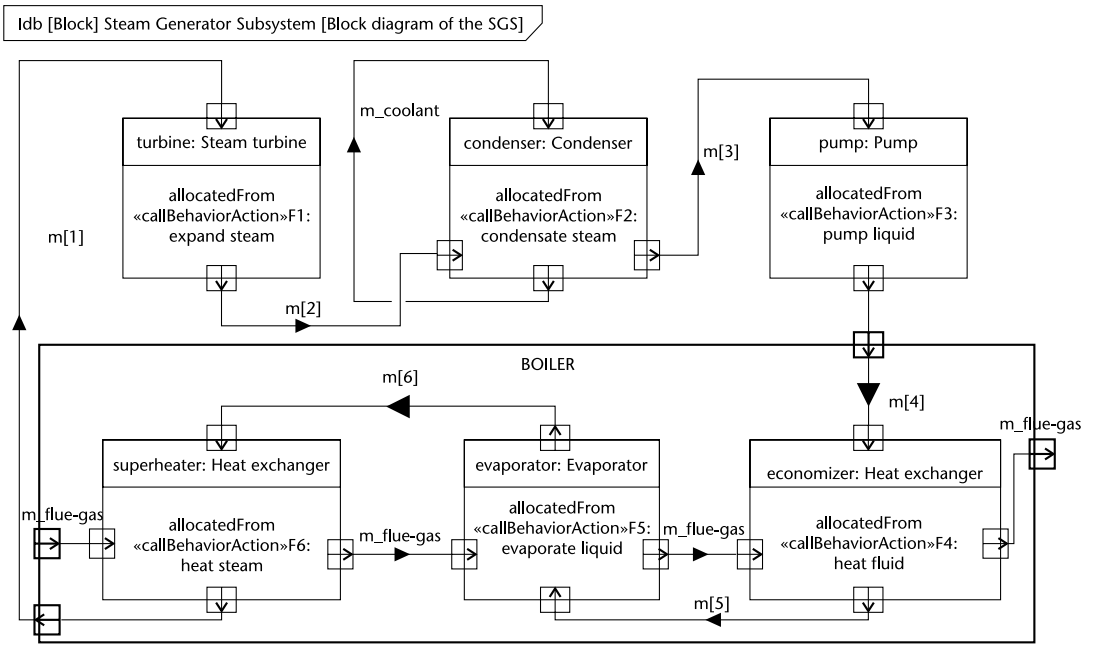


Figure 7.2 SysML internal block definition diagram to represent the modular architecture of the steam generator process (see complete example in Chapter 10).

diagram (BDD) can be used to represent the system's structure [2]. In SysML, the composition association is used to represent the physical architecture breakdown into subsystems and parts. Therefore, the ISE&PPOOA modular architecture is represented by a BDD using composite associations.

The BDD representation of the modular architecture only captures the breakdown of the system into its constituent's parts. This diagram is complemented in ISE&PPOOA with

- SysML internal block diagrams (IBDs), which capture the connections between the internal constituent parts of each module (block). See Figure 7.2 for an example.
- Activity and state diagrams for behavioral description as needed.

Examples of the representation of the physical architecture of different systems, including all the diagrams mentioned, can be found in Chapters 8 (unmanned aerial vehicle), 9 (collaborative robot), and 10 (steam generation in a power plant).

7.2.2 Allocation

Allocations are relationships that cut across the different aspects of your system design: requirements, functions, and physical elements (see [2,3]). In the context of MBSE and SysML, functional allocation is also known as behavioral allocation [2], and refers to allocating a behavioral element (i.e., an activity or an action, interaction, or state machine) to a structural element (a building block in the ISE&PPOOA modular architecture). The key allocation process in ISE&PPOOA is that of functional allocation, where functions in the functional architecture are allocated to the building elements that implement them in the physical architecture.

There are multiple options to represent functional allocation, including graphical and tabular representations, all supported in SysML (see [2,3] for an exhaustive description of the alternative representation of allocations). In ISE&PPOOA allocation is represented by UML/SysML partitions (swimlanes) in the activity diagrams [7], where the activities are allocated to the performing building blocks. Additionally, functional allocation can be represented as a matrix in which the rows are the functions and the columns are parts of the system. This is especially useful as a support representation to facilitate the allocation process. Note that the rows and the columns should refer to consistent levels of their respective functional and physical hierarchies.

This representation provides a useful tool to apply the allocation criteria of the architecting heuristic SA_Heu_9 in Chapter 6, "So each low-level function should be allocated to one physical component." In the model of the allocation through an allocation matrix, this translates into the following criteria: (as far as possible) the rows of the matrix should reflect the appropriate functional level so that the row/function is allocated to only one column/building block. If that is not the case, then that row should be replaced by all the rows corresponding to the subfunctions, which we should try to allocate to individual building blocks. For each row for which the unique allocation is not possible, we have to repeat the decomposition process.

Whereas the allocation of functions results in allocation relationships between functions in the functional hierarchy and corresponding parts in the physical hierarchy, the allocation of NFRs results in the application of design heuristics to elements of the architecture, as discussed in the following section.

7.3 Design Heuristics for Refining the Architecture

Nonfunctional requirements are taking into account in the architecting process by using design heuristics. The application of these heuristics results in the refining of the physical architecture through specific design decisions. In Chapter 6, we discussed heuristics related to maintainability, safety, efficiency, and resilience. The influence of the architecture in the assurance of system safety has been recognized for mission-critical applications. The safety-critical standard IEC 61508 Part 3 states that, "From a safety viewpoint, the software architecture is where the basic safety strategy is developed in the software [8]." Therefore, as an example, in this section we will focus on safety heuristics to explain how they are applied to refine the architecture of the system.

A safety model can support the integration of design heuristics (also referred as tactics in the literature). A very common safety model in software organizes the heuristics based on when they address the failure [9,10]:

- Failure avoidance;
- Failure detection;
- Failure containment.

In software architecting, a tactic (heuristic) is a design building block that embodies a specific development technique (e.g., redundancy) to achieve some facet of a quality requirement [9]. Heuristics have been composed into safety-related architectural patterns [9] that can be easily applied in different applications.

According to Wu [11], "Architectural tactics can thus be seen as the foundational building blocks of architectural patterns." Alexander et al. [12] were the first to describe a pattern (in the architecture of buildings) as a three-part rule expressing a relation between a certain context, a problem, and a solution. This idea has a successful history in object-oriented software, where a design pattern is a reusable solution to a recurring problem of general nature in a specific context of use. Patterns and have also been used in analysis, reflect conceptual structures in the domain, to help obtain domain models [14].

Heuristics offers thus a finer-grained approach to design than patterns, as architectural patterns are composed of architectural heuristics [11].

7.3.1 Control Monitor Pattern

Wu et al. [9] discusses two of the possible patterns resulting from the combination of available heuristics for a control/monitor scenario. The safety requirement in such scenario is that the system must monitor any potentially hazardous conditions in order to execute proper protection mechanisms. Heuristics "condition monitoring", "redundancy", "comparison," and "interlock" are combined into two alternative architectural patterns to address safety by allocating its responsibility to the monitor part, or both the monitor and the control part.

7.3.2 Triple Modular Redundancy Pattern

Another example also discussed by Wu et al. [9] is the triple modular redundancy (TMR) pattern. This is a classic pattern to prevent single-point failures when producing an output, by including three redundant modules to produce the output and a voting element to determine the system output based on the individual outputs (e.g., majority or average). Therefore, it can be immediately noted that this pattern includes the heuristics redundancy and voting. But it can also be assumed to involve the diversity heuristic, since the failure of two (or three) different implementations of a module is less likely. Different combinations of these three heuristics can lead to a variety of the TMR patterns associated with different desired safety properties (see [9] for the detailed discussion).

7.4 Software Architecting with the PPOOA Framework

For software-intensive subsystems, the PPOOA process [15,16] is used to obtain the physical architecture, as explained in Section 4.4. PPOOA bridges the systems engineering modeling process ISE, and its resulting functional architecture, with the software development process through the responsibility-driven analysis using CRC cards. The guiding principle of responsibility-driven modeling is that the central question to formulate when partitioning a software subsystem is what responsibility each part has toward the subsystem.

PPOOA is more than a process, it is an architecture framework [17,18] oriented to design the software of real-time systems. This is also known as architectural style [19], defining the vocabulary of components and connectors that can be used together with a set of constraints on how they can be combined. PPOOA constraints or rules can be regarded as a very general set of heuristics or design patterns that together with the vocabulary constitute a pattern language [20] for the architecting of real-time software subsystems.

PPOOA uses the two most extended viewpoints used in software architecture: structural and behavioral. According to the ISO/IEC/IEEE 42010 standard [21], the structural viewpoint is concerned with the computational elements that compose the software subsystem, how are they organized, what are their interfaces, and how the interconnect to each other.

The behavioral viewpoint has an even longer history and is concerned with the dynamic actions of and within the software subsystem, what kind of actions are there, how do they relate (ordering, synchronization), and how the components interact through them.

PPOOA uses UML[1] class diagrams extended with PPOOA stereotypes to address the structural viewpoint, and UML/SysML activity diagrams to address the behavioral viewpoint.

1. PPOOA was developed before the UML standard publication. Therefore, it did not use UML notation. As UML popularity increased, the author realized the importance of using UML notation. So, partially funded by the European Union CARTS IST project, a UML profile for real-time systems based on PPOOA was developed, and an architecting process named PPOOA_AP. PPOOA and PPOOA_AP were validated in autonomous robots and ground space systems developed by the industrial partners of CARTS (1999-2001).

Whereas Chapter 4 discussed PPOOA as a process in the architecting of software subsystems, in this section we will focus on the main elements of PPOOA as an architectural framework, and how they can be used in the architecting of a system.

7.4.1 Domain Model

The creation of the domain model of the software subsystem is the crucial step in PPOOA. The concept of a domain model comes from object-oriented programming, where in the analysis phase there is an identification of the concepts and their attributes and associations that are considered noteworthy [22]. The domain model represents real-world concepts, not software objects, which are expressed later as components in PPOOA (or classes in the design models when following the traditional object-oriented approach).

A domain model yields a more precise specification of software requirements than the project team has in the results from earlier system requirements. It is described using more formalism than textual descriptions, for example UML class diagrams, and can be used to reason about the internal workings of the software subsystem (see Figure 7.3 for example).

To obtain the domain model classes, one approach promoted by PPOOA is to consider categories of concepts:

- *Tangible things or devices the system must interact with*. For example, in the case of a robotic application for pick and place of products, we could have product, stacks of products, and so on (see Chapter 9 for the complete example).
- *Places*. The relevant locations for our application (if suitable). Continuing with our robotic example, these can be the locations for the robot to pick the different products, and the location where to deliver them.

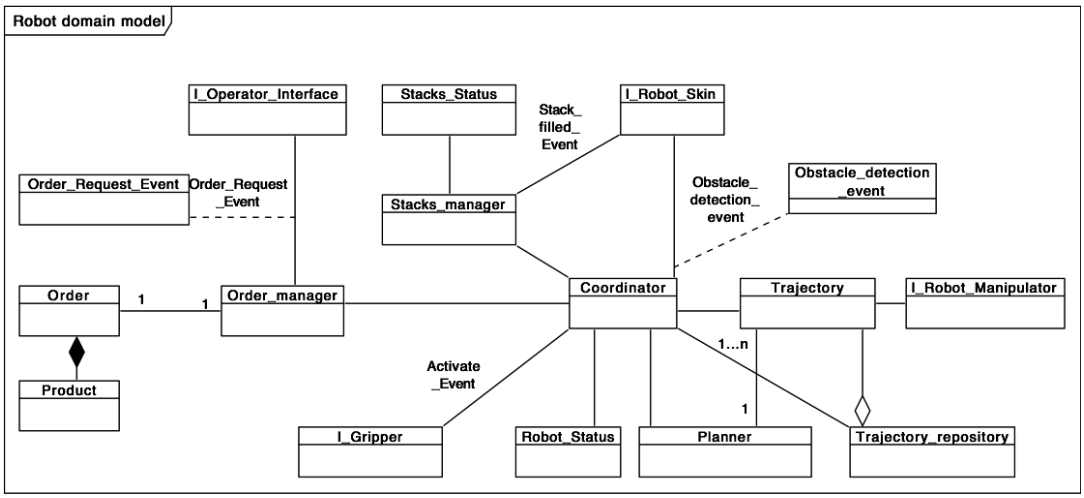


Figure 7.3 Example of the domain model for a collaborative robot application (more details are given in Chapter 9).

- *Events*. An event is an action or occurrence that originates externally to the software subsystem, usually from the environment, and is captured by some sensing subsystem and then passed on to the software subsystem to handle. For example, the main events in our robotic application could include a new order received or an obstacle detected.

However, the previous approach is only a support method to identify additional domain classes. The integration between the systems engineering modeling process ISE and the PPOOA software architecture process is achieved by responsibility-driven software analysis. This approach bridges the functional hierarchy and the domain model through responsibilities, which correspond to functions from the functional hierarchy to be performed by software.

It proceeds as follows:

1. Identify in the functional hierarchy which functions are allocated to software;
2. With those functions, identify substantives in their description to obtain the classes in the domain model.

Substantives (things) become classes in the domain model and verbs (transformations) become responsibilities of those classes.

CRC cards are used to document the responsibilities of each domain model class; (i.e., the mapping to the functional architecture), and the list of classes it collaborates with to achieve those responsibilities; (i.e., the connections between elements in the software architecture).

Larman distinguishes two main classes of responsibilities: doing and knowing [22]. Following his responsibility-driven design, the domain model can inspire responsibilities related to "knowing" because of the attributes and associations it represents. However, in that approach, responsibilities are directly derived from the use cases of the application, whereas in ISE&PPOOA responsibilities are extracted from the functional architecture.

7.4.2 Software Components and the PPOOA Vocabulary

PPOOA provides a vocabulary of elements to build software systems. During the PPOOA process, software components from the PPOOA vocabulary are identified from the domain model classes. Chapter 6 presented software architecting heuristics that can be used (PPOOA_Heu-2, PPOOA_Heu-3, and PPOOA_Heu-4).

7.4.2.1 Components

In PPOOA, a component is a conceptual computation entity that performs some responsibility and may provide and require interfaces to other components. Generally, it may be decomposed into small granularity parts (i.e., subcomponents).

7.4.2.2 Interfaces

Component operations are grouped into interfaces that are meaningful to the architecture solution. In object-oriented programming, interfaces define behaviors of an object as method or operation signatures. This means that in PPOOA, the interface of a component defines its services that other components can use without needing to know their implemention within the component. A component interface is more important from an architectural perspective than the way the interface is realized or implemented. It should be possible to replace one component by another with an equivalent interface without affecting the architecture.

In PPOOA, component interfaces are described in textual and tabular format. An interface description language (IDL) can be used to describe the interfaces in a language/implementation–independent way, but is not strictly needed at this stage.

Next, we summarize the different elements in the PPOOA vocabulary (a complete, detailed description can be found in [23,26]).

- *Algorithmic component.* An element that performs calculations or transforms data from one type to another is but separated from its structural abstraction. Data classification components, Unix filters or data processing algorithms are typical examples. The algorithm component in PPOOA is similar to the utility defined in the UML metamodel, but the algorithmic component may have instances.
- *Domain component.* This element responds directly to the modeled problem. It does not depend on any hardware or user interface. An instance of a domain component corresponds to a software object in traditional object-oriented design.
- *Structure.* A structure is a component that denotes an object or class of objects characterized as an abstract state machine or an abstract data type. Examples are stack, queue, list, and ring. This is a primitive component and cannot be decomposed into other components.
- *Process.* An element that implements an activity or group of activities that can be executed at the same time as other processes. Its execution can be scheduled. The PPOOA architecture framework supports two different types of software processes: cyclic/periodic and aperiodic. The cyclic process is used to implement an activity or group of activities that execute periodically according to an execution period (attribute). The cyclic process communicates with other processes by means of coordination mechanisms, which are described later.

  The aperiodic process provides an activating operation in order to execute its executing flow. This operation should not block the caller. Other operations can be provided, but they should be executed immediately when called. They are called unconstrained operations in the HRT-HOOD terminology [24].

  As they are both aggregate components, the cyclic and acyclic processes can include domain components, structures, cyclic processes, and aperiodic processes.

- *Controller object*. The controller object is responsible for initiating and directly managing a group of activities that can be repetitive, alternative, or parallel. These activities can be executed depending on a set of events or conditions. When an event occurs, the system schedules associated event handlers. It manages other components rather than supplying services that depend on the domain.

7.4.3 Coordination Mechanisms

In addition to the components, PPOOA also models their interactions, distinguishing two types: synchronous and asynchronous. Synchronous interactions are represented as usage relationships. Asynchronous interactions are implemented in the software architecture by the use of coordination mechanisms. The final step in the PPOOA process is to identify the more suitable coordination mechanism for each asynchronous interaction between two components in the software subsystem.

A coordination mechanism provides the capabilities to synchronize or communicate components of the software architecture. Synchronization is the blocking of a software process until some specified condition is met. Communication is the transfer of information between the components of the software architecture [16]. The coordination mechanisms are represented in the PPOOA architecture diagram by UML stereotypes (see Figure 7.4).

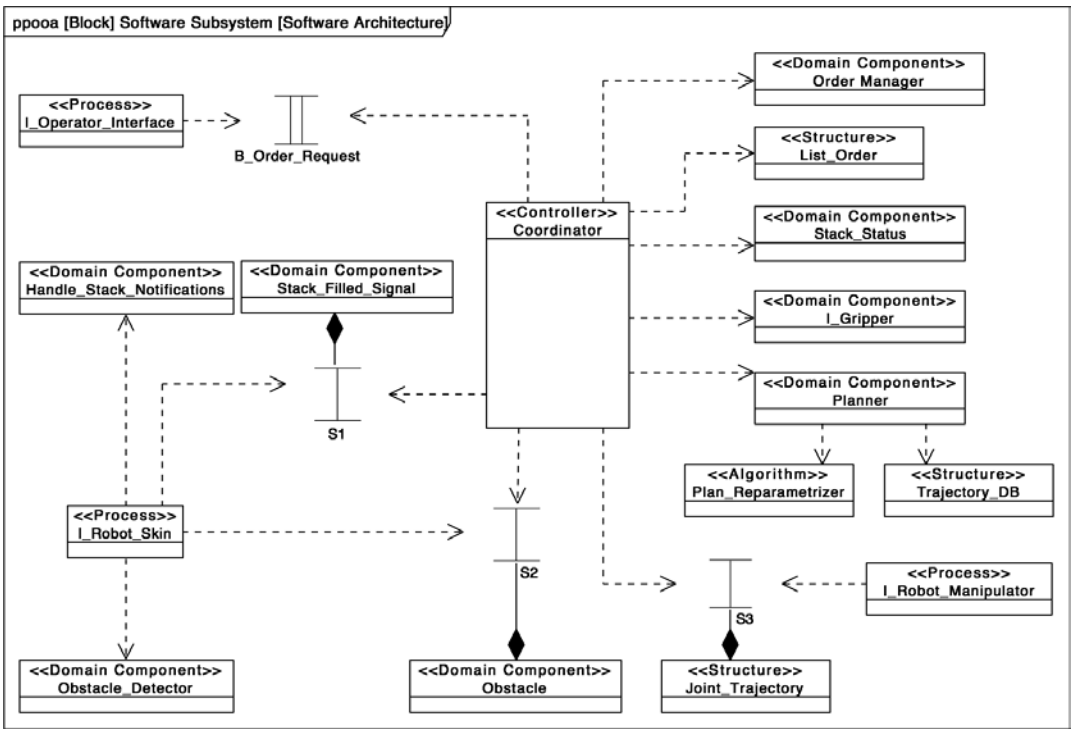


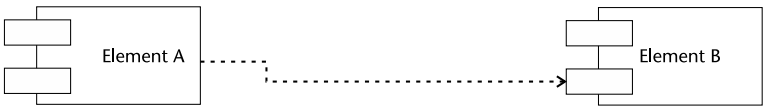
Figure 7.4 Components and coordination mechanisms in the PPOOA architecture diagram of a collaborative robotic application. PPOOA components have stereotypes indicating the type of element, whereas the coordination mechanisms are represented using the shapes for their stereotypes: B_oder_request is a buffer, and S1, S2, and S3 are semaphores protecting the elements they are part of.

The PPOOA vocabulary defines the following set of coordination mechanisms that are implemented by real-time operating systems [25]:

- *Bounded buffer.* The bounded buffer, also called buffer or queue of messages, is a temporary holding area where data producers and consumers make calls to send and get data from.
- *General semaphore.* The general semaphore is a nonnegative integer value used as a synchronization mechanism. It is used to synchronize processes or control the access to a critical region.
- *Mailbox.* The mailbox is a mechanism that is used to pass messages, which can possibly have a variable length, between processes in an asynchronous mode. The process that sends the message has the option of continuing or waiting until the message is received. Only one process can receive a message.
- *Rendezvous.* The rendezvous is a synchronous and unbuffered coordination mechanism that allows two processes to communicate bidirectionaly.
- *Transporter.* A transporter can be considered an active data mover. The transporter makes calls to get and send messages from and to the coordinating partners (producer and consumer). A transporter is strictly a caller that gets a message from one producer and passes the message on to one consumer or another intermediary via a call.

In addition to the previous vocabulary of building blocks, PPOOA defines a set of rules and guidelines [26]. The PPOOA guidelines cover the architectured style, components, coordination, mechanisms, casual flow of activities, and scheduling. The PPOOA rules include rules for the connection of the different elements in the architecture diagram, including composition rules, usage rules (between components for synchronous interactions, between components and coordination mechanisms for asynchronous interaction, and between coordination mechanisms), and inheritance rules. For example, Table 7.2 summarizes the usage rules between components and coordination mechanisms, and Table 7.3 the composition rules between the elements in the PPOOA vocabulary.

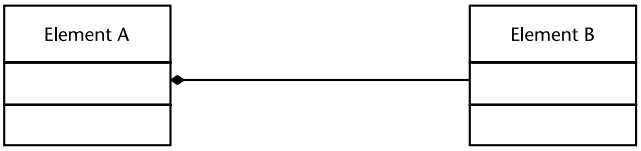
Table 7.2 Usage Rules Between Components* and Coordination Mechanisms



Element A / *Element B*	*Algorithmic Component*	*Domain Component*	*Structure*	*Process*	*Controller Object*
Buffer	No	No	No	Yes	Yes
Transporter	No	No	No	No	No
Semaphore	No	No	No	Yes	Yes
Rendezvouz	No	No	No	Yes	Yes
Mailbox	No	No	No	Yes	Yes

* Note that PPOOA vocabulary also includes a subsystem element that for the sake of simplicity we are not including in the discussion here.

Table 7.3 Composition Rules Between the PPOOA Vocabulary Elements



Element A / *Element B*	*Algorithmic Component*	*Domain Component*	*Structure*	*Process*	*Controller Object*	*Coordination Mechanism*
Algorithmic Component	Yes	Yes	No	Yes	Yes	No
Domain Component	Yes	Yes	No	Yes	Yes	No
Structure	Yes	Yes	Yes	Yes	Yes	No
Process	No	No	No	Yes	Yes	No
Controller Object	No	No	No	No	Yes	No
Coordination Mechanism	No	Limited†	Limited†	Yes*	Yes	No

*Typically, is the case of a process that contains a transporter.
†Either a domain component or a structure may include a semaphore as a mechanism for guaranteeing mutual exclusion. Other coordination mechanisms are not permitted for composition.

Components and coordination mechanism, combined according to the rules discussed, model the structural view of the architecture, which is represented by a PPOOA architecture diagram. This diagram is used instead of the classical UML component diagram, or the SysML internal block definition diagram, to describe the software architecture, and maintains some similarities with the UML class and collaboration/communication diagrams. The PPOOA architecture diagram focuses on software components and coordination mechanisms representation and the dependence relationships between them. Composition relations between components are represented as well.

7.4.4 Software Behavior and Causal Flow of Activities

Behavioral view is necessary to complement the structural one. Describing the static structure of a system can reveal what it contains and how its elements are related, but it does not explain how these elements cooperate to provide the software subsystem functionality.

To model the software subsystem behavior, PPOOA uses the concept of CFA, also called *time threads* by other authors [27]. A CFA is a cause-effect chain of activities that cuts across different building elements of the software architecture. Activities are the smallest division of a response. This chain progresses through time and executes the activities provided by the software architecture components until it gets to an ending point.

Each event response in the software subsystem is captured by a CFA and modeled as an UML/SysML activity diagram in the same way that they were used in the systems engineering subprocess of ISE&PPOOA. Several active CFAs can exist at the same time and may even interact with each other. CFAs can also stop at certain waiting points, where coordination mechanisms or objects with concurrent access

are located. CFAs can split into parallel flow of activities, and flows of activities can also merge together.

CFA activities are allocated to the components and coordination mechanisms of the system architecture using the UML concept of partition (swimlanes). This is a key contribution of the PPOOA architectural framework to a critical issue in the engineering process, since it permits the assessment of the different allocation alternatives. Examples of CFAs for a collaborative robot can be found in Chapter 9.

7.5 Summary

This chapter presented how to develop the physical architecture of a system using the ISE&PPOOA methodology.

The physical architecture representation has three main constituents. The physical hierarchy is represented using a SysML block definition diagram. This diagram is complemented with internal block definition diagrams representing how the different parts are interconnected physically and logically in the subsystems at each level of the hierarchy, and with textual descriptions of the system blocks and their interfaces. Finally, the behavior viewpoint is captured by activity and state diagrams as needed. Allocation of functions may be represented in tabular form or at the system blocks using SysML notation, but more importantly using partitions (swimlanes) in the activity diagrams.

For software subsystems, the PPOOA framework can be used to obtain the software architecture.

7.6 Questions and Exercises

1. Define the physical hierarchy of the washing machine. Its functional hierarchy is an exercise in Chapter 5.
2. Define the physical hierarchy of the electric parking brake. Its functional hierarchy is an exercise in Chapter 5.
3. Model the IBD of the above electric parking brake.

References

[1] Kossiakoff, A., W. Sweet, S. Seymour, and S. Biemer, *Systems Engineering Principles and Practice,* Hoboken, NJ: John Wiley & Sons, 2011.

[2] Delligatti, L., *SysML Distilled: A Brief Guide to the Systems Modeling Language*, Upper Saddle River, NJ: Addison-Wesley, 2014.

[3] Friedenthal, S., A. Moore, and R. Steiner, *A Practical Guide to SysML*, Third Edition, Waltham, MA: Morgan Kaufmann, 2015.

[4] Dori, D., *Model-Based Systems Engineering with OPM and SysML*, New York: Springer, 2016.

[5] Bustnay, T. and J. Z. Ben-Asher, "How Many Systems Are There? Using the N^2 Method for Systems Partitioning.," *Systems Engineering*, Vol. 8, No. 2, 2005, pp. 109–118.

[6] Sharman, D. M., and A. A. Yassine, "Characterizing Complex Product Architectures," *Systems Engineering*, Vol. 7, No. 1, 2003, pp. 35–60.

[7] Fernandez-Sanchez, J. L., and E. E. Betegon, "Supporting Functional Allocation in Component-Based Architectures," *Proc. 18th International Conference on Software & Systems Engineering and their Applications*, Paris, France, December 2005.

[8] IEC 615038, "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems," International Electrotechnical Commission, 1998.

[9] Wu, W., and T. Kelly, "Safety Tactics for Software Architecture Design.," *Proc. of the 28th Annual International Computer Software and Applications Conference*, Vol.1, September 2004, pp. 368–375.

[10] Gawand, H., R. S. Mundada, and P. Swaminathan, "Design Patterns to Implement Safety and Fault Tolerance.," *International Journal of Computer Applications*, 2011.

[11] Wu, W., *Architectural Reasoning for Safety- Critical Software Applications*, PhD thesis, The University of York Department of Computer Science, 2007.

[12] Alexander, C., S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*, Oxford, UK: Oxford University Press, 1977.

[13] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Upper Saddle River, NJ: Addison-Wesley, 1995.

[14] Fowler, M., *Analysis Patterns: Reusable Object Models,* Upper Saddle River, NJ: Addison-Wesley, 1997.

[15] Fernandez, J. L., "An Architectural Style for Object Oriented Real-Time Systems," *Proc. Fifth International Conference on Software Reuse,* June 1998, pp. 280–289.

[16] Fernandez-Sánchez, J. L., "A Vocabulary of Building Elements for Real-Time Systems Architectures.," In *Business Component-Based Software Engineering*, F. Barbier (ed.), The Springer International Series in Engineering and Computer Science. Springer, US: 2002, pp. 209–225.

[17] Systems and Software Engineering, Architecture description ISO/IEC/IEEE 42010 http://www.iso-architecture.org/ieee-1471/afs/.

[18] Survey of Architecture Frameworks http://www.iso- architecture.org/ieee-1471/afs/frameworks-table.html.

[19] Shaw, M., and D. Garlan, *Software Architecture. An Emerging Discipline*, Upper Saddle River, NJ: Prentice Hall, 1996.

[20] Brugali, D., and K. Sycara, "Frameworks and Pattern Languages," *ACM Computing Surveys*, March 2000.

[21] ISO/IEC/IEEE 42010:2011, *Systems and Software Engineering—Architectural Description,* The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) in collaboration with the Institute of Electrical and Electronic Engineers (IEEE), 2011

[22] Larman, C., and P. Kruchten, *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*, Upper Saddle River, NJ: Prentice Hall, 2002.

[23] Fernandez, J. L., and A. Monzon, "Extending UML for Real-Time Component-Based Architectures," *Proc. 14th International Conference on Software & Systems Engineering and Their Applications*, Paris, France, December 2001.

[24] Burns, A., and A. Wellings, *HRT-HOOD: A Structure Design Method for Hard Real-Time Ada Systems,* Amsterdam: Elsevier Science B.V., 1995.

[25] Fernandez, J. L., "A Taxonomy of Coordination Mechanisms Used by Real-Time Processes," *Ada Lett.,* Vol. 17, No. 2, March 1997, pp. 29–54.

[26] PPOOA, *Processes Pipelines in Object Oriented Architectures* http://www.ppooa.com.es/

[27] Buhr, R. J., "Pictures that Play: Design Notations for Real-Time and Distributed Systems," *Software Practice and Experience*, Vol. 23, No. 5, August 1993.

CHAPTER 8

Example of Application: Unmanned Aerial Vehicle-Electrical Subsystem

with Juan Lopez and Alberto Nieto

This chapter illustrates how the ISE&PPOOA methodology has been used to engineer the Seeker Unmanned Aircraft System (UAS) by Aurea Avionics, a lightweight UAS designed to support and cover ISR missions. This example describes the electrical subsystem design, which is not software-intensive, and in this case, shows some characteristics compared to other subsystems:

- It is highly dependent on the eventual selection of onboard components and equipment;
- Some of the functions involved seem simple, such as distribute power or signals, and pose a challenge to be properly defined in early stages;
- A team of engineers applies ISE&PPOOA with strong project constraints regarding time of development, so a good-enough philosophy is maintained throughout the process.

First,we include a short overview of the aircraft and its operation to illustrate a typical mission, and according to ISE&PPOOA, needs and subsequent capabilities are stated in Section 8.1 "Example Overview, Needs, and Capabilities."

Using these capabilities, functional architecture is developed and system requirements, both functional and nonfunctional as well as environmental constraints, have been identified. Hence, functional hierarchy, functional flows, N^2 charts, functional and nonfunctional requirements, environmental constraints, and the data dictionary can be found in Section 8.2.

In Section 8.3 "Physical Architecture and Heuristics," physical architecture diagrams, functional flow allocations to physical components and description of the resulting physical architecture obtained with suitable heuristics are summarized.

Finally, in Section 8.4 a brief recap of the main conclusions of this chapter is included.

8.1 Example Overview, Needs, and Capabilities

The Seeker UAS is an unmanned aircraft designed and manufactured by Aurea Avionics, a Spanish tech company that specializes in UAV/UAS suited for defense and security applications, in which the company is committed to bringing the next generation of cutting-edge autonomous systems and solutions. The Seeker UAS is aimed for quick deploy and very low logistic footprint to support ISR missions and security tasks, such as perimeter security reinforcement and search and rescue, among others. Typical scenarios comprise military theaters or surveillance of locations where private security companies can't make use of traditional surveillance methods: this is the case with nuclear power stations that are partially surrounded by lakes where fixed cams are not practical, or facilities in areas with a complicated orography.

The main characteristics of the Seeker UAS are shown in Table 8.1.

A typical operation might be overseen or preprogrammed for a fully autonomous execution through the ground control station. It provides both daytime and nighttime ISR capabilities, with a communications (COMMS) range up to 15 km. The Seeker UAS is equipped with a payload including gimbaled cameras that deliver real-time color and infrared imagery to the ground segment.

The complete system includes the aircraft, a ground COMMS segment (ground data terminal (GDT)), a laptop to run the C2 software that serves as a graphic user interface (GUI) for the user (ground control station (GCS)), and optionally, a remote handheld control (RHC). The laptop is connected to the GDT that manages COMMS (commands, telemetry, and video downlink).

The GCS allows a completely automatic flight and assists the operator during all the mission phases, including preflight, route definition, and the usual operation maneuvers. In addition, it monitors the aircraft variables and health status (such as position, attitude, velocity, and battery voltage), and shows relevant information for the operator (distance to the target, wind speed, georeferenced video, etc.).

The RHC allows to fly with fixed velocity in situations when is not necessary to introduce waypoints; the aircraft is controlled using simple commands like up-down or right-left. This control allows tilt control of the cam and can send some useful commands in automatic mode such as land, home, and abort.

Table 8.1 Characteristics of the Seeker UAS

Endurance	90 minutes
Weight	3.5 kg
Span	2.0m
Length	1.2m
Takeoff	Hand-launched
Landing	Belly landing
Link range	15 km LOS
Cruise speed	60 km/h
Operating altitude	100m–400m

Figure 8.1 Seeker UAS by Aurea Avionics.

Figure 8.2 Seeker UAS' remote handheld control,

8.1.1 Operational Scenarios and Use Cases

The first step of ISE&PPOOA process (Chapter 4) is to identify operational scenarios. First, the entire context of the system is outlined. Figure 8.3 shows the context of the system and its relationship to the actors involved in the operation [1,2].

The Seeker UAS is operated by the pilot who will use the GDT to send commands and receive information from the aircraft to analyze its behavior. The aircraft uses its onboard cameras to localize targets (people lost in the mountain, shipwrecks, etc.). The aircraft must complete its mission a variety of weather conditions, including sun and wind, and must be able to land in small areas with trees. These conditions are summarized as environmental conditions. In addition, maintenance should be able to be performed by a technician.

Refer to Figure 3.2, which shows a use case diagram for the Seeker UAS. The use cases help to determine the main interactions in order to understand the operational scenarios and the stakeholders' needs.

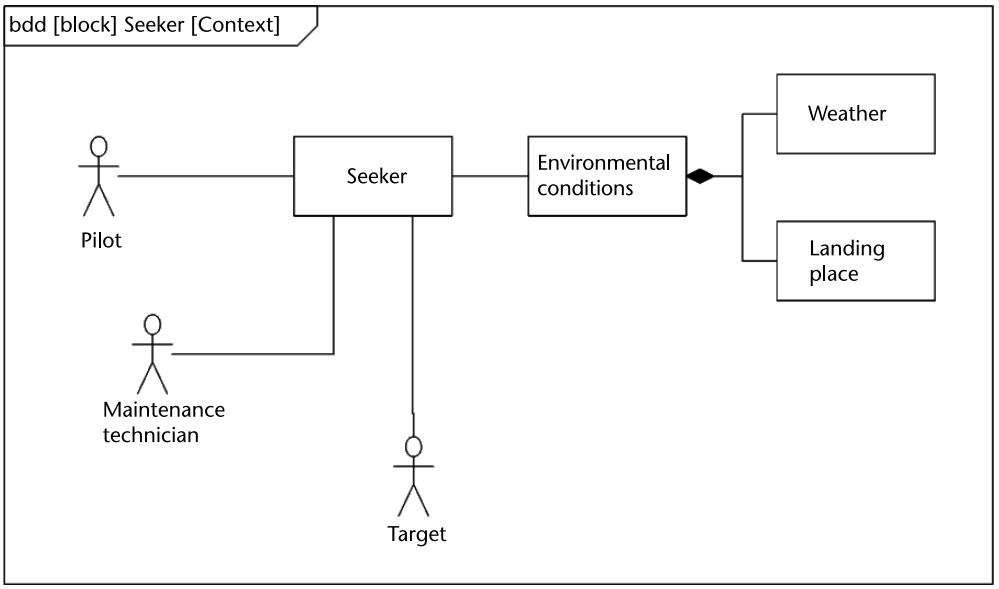


Figure 8.3 Seeker UAS context diagram using an internal block diagram.

Figure 3.2 (see Chapter 3) shows a simplified use case diagram of the Seeker UAS that is composed of the autonomous aircraft, the ground segment (including C2 software and COMMS terminal, GCS+GDT), and the RHC.

UC1. Configure aircraft. This use case is related to the state of the aircraft verification process including setting parameters and variables, sensor verifications, structural integrity, and so on. This use case is performed using a checklist that will be fulfilled before each mission.

UC2. Manage flight plan. This use case includes the process of flight plan definition, modification, checking, and upload to the aircraft. During this use case the pilot will introduce the waypoints by clicking in the map and will define the lost communications point, ground control position, and so on. This use case includes the possibility to change the flight plan during the mission.

UC3. Command aircraft. The pilot needs to send commands to the aircraft. These commands are take off, orbit (perform an orbit in a predefined point), landing, go to home point, and perform the active flight plan.

UC4. Manage landing plan. The pilot sends the landing plan to the aircraft: initial and final of the runway, direction, gliding path, and approach velocity.

UC5. Manage payload. This use case describes the interaction of the pilot with the payload. The payload of the Aurea Seeker is a gyro-stabilized dual camera that must be able to point a target, zoom the image, and switch between the cameras.

UC6. Estimate wind. The pilot needs to obtain an estimation of the direction and the intensity of the wind in the target surroundings. This information is relevant to perform rescue operations.

UC7. Video capture. This use case describes the capacity to capture video from a target and send it to the ground control station.

UC8. Calculate target position: The pilot needs to know the target position. These geographic coordinates are provided in several coordinate systems and units. The pilot selects the units and coordinate system.

UC9. Aircraft monitoring. The aircraft provides information related to aircraft attitude, position, velocity, battery voltage, alarms, and so forth. The pilot uses this information to command the aircraft and to identify malfunctions, estimate the remaining flight time, and so forth.

UC10. Perform maintenance. The system is revised following the manufacturer maintenance manual.

For the sake of brevity only brief descriptions of the use cases have been included here and some have been grouped, but a more detailed explanation and description of the use cases is necessary in order to completely define the needs of the system users.

8.1.2 System Capabilities

The second step of the ISE&PPOOA process consists of specifying system capabilities based on its mission. See Table 8.2 summarizing these capabilities.

Table 8.2 System Capabilities

Id	*Capability*	*Description*
C1	Easy aircraft configuration	The ground control software shall be configured in order to set the parameters and variables of the aircraft.
C2	Guide to system verification	Before the flight, it's necessary to verify the behavior of the sensors, the structural integrity, communications, and so on. The system software must provide a guide to the operator to perform the verification process.
C3	Flight planning management	It is necessary to define and to upload the flight plan prior to the flight and to change it during the flight.
C4	Autonomous flight	The aircraft shall fly in an autonomous mode following the flight plan.
C5	Send and receive communications	The aircraft shall send telemetry and receive commands.
C6	Command payload	The payload (gimbal) shall be pointed and perform zoom.
C7	Send video	The aircraft shall send real-time video continuously.
C8	Support wind estimation	The aircraft shall estimate wind and send the estimation to the ground station.
C9	Long endurance	The endurance of the aircraft shall be higher than an hour.
C10	Range	The range of the aircraft shall be higher than 10 km.
C11	Easy assembly	The aircraft shall be assembled for flight in less than 5 minutes.
C12	Easy to transport	The whole system shall be transported in a box or in two medium-size bags.
C13	Safe during GPS signal outages	The aircraft shall return to base in case of GPS signal outage.
C14	Low noise footprint	The aircraft shall fly in a silence mode.

8.2 Functional Architecture and System Requirements

The deliverables related to the functional architecture are those required by the application of the ISE&PPOOA process step 3:

- BDD diagrams for the functional hierarchy;
- Activity diagrams for the functional flows;
- Functional interfaces description (N^2 chart).

8.2.1 Functional Architecture

A top-level identification of the aircraft main functions is shown in Figure 8.4 adapted from Jackson [3].

F1. Provide communications. Provide communications among subsystems and with the ground control station.

F2. Plan, generate, and control aircraft movement. Plan and analyze aircraft path and generate aircraft movement (thrust, speed, direction, altitude, etc.)

F3. Monitor aircraft conditions for flight. Determine position, attitude, relative position to runway, and velocity.

F4. Distribute communications. Distribute physical signals among subsystems.

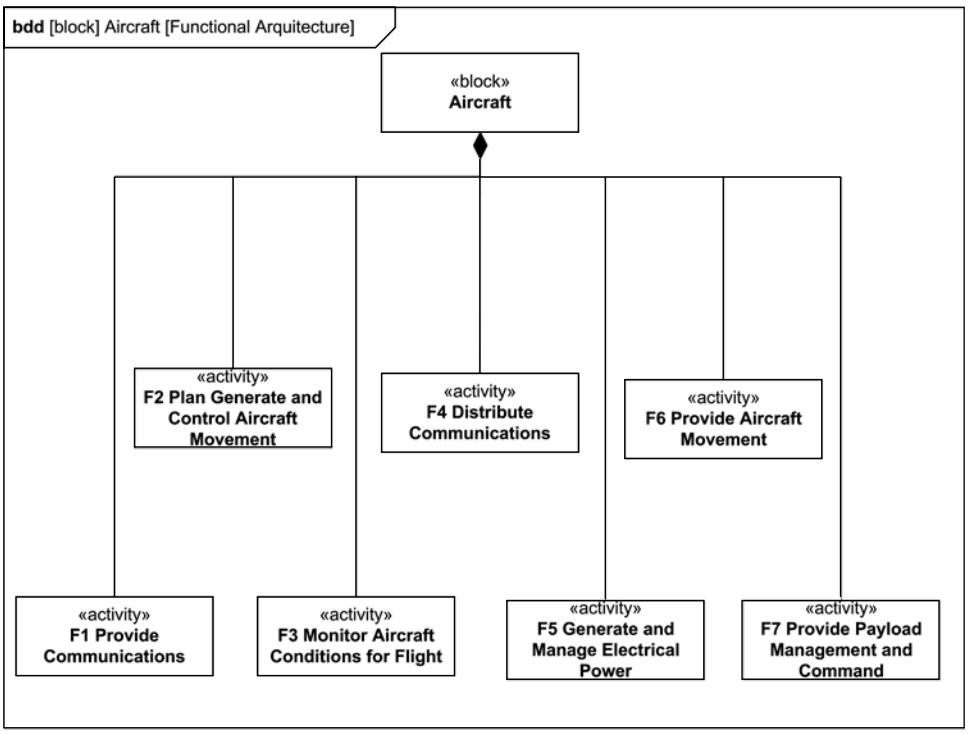


Figure 8.4 Aircraft top level functions represented in a block definition diagram.

F5. Generate and manage electrical power. Provide electrical power and distribute it to the aircraft subsystems.

F6. Provide aircraft movement. Using the calculated movement in F2, provide movement through actuators.

F7. Provide payload management and command. Allow send commands to the payload and receive status information from the payload.

The functions that are related to the electrical subsystem are distribute communications, generate and manage electrical power, and provide payload management and command [4,5]. A more detailed functional hierarchy of these functions is shown hereafter (see Figures 8.5, 8.6, and 8.7).

A textual description in tabular form for the functions is included as the ISE&PPOOA methodology recommends (see Tables 8.3–8.15). For sake of brevity only the first functional decomposition level is shown here.

Finally, in this ISE&PPOOA step related to the delivery of the functional architecture, the functional flows must be modeled as well. Figures 8.8, 8.9, and 8.10 show the functional flows of corresponding to F4, F5, and F7.

Every single subfunction is executed independently as shown in Figure 8.8, since the autopilot must send demanded throttle to the engine and demanded position to actuators simultaneously to control the aircraft. In parallel, sensor measurements are sent to be processed by the autopilot, video captured by the payload is transmitted, telemetry data is transmitted to monitor aircraft's state, and lights must be able to be switched on or off.

The generate and manage electrical power function (see Figure 8.9) includes several steps. The first step provides power, the second step converts the main source's voltage to different voltage levels required, and in the third step power to different subsystems is provided.

Pointing the camera, toggle among the three onboard cameras and zoom can be performed simultaneously as it is shown in Figure 8.10.

The N^2 chart of "F4 distribute communications function is shown in Table 8.16.

The N^2 chart (Table 8.16) shows that all "F4 Distribute Communications" subfunctions are uncoupled. This situation is produced by the fact that function F4 Distribute Communications group subfunctions performing the communications between other subsystems or components of the aircraft.

Table 8.17 shows the N^2 chart *of* "F5 Generate and Manage Electrical Power," representing the functional interfaces between F5 subfunctions and external to F5.

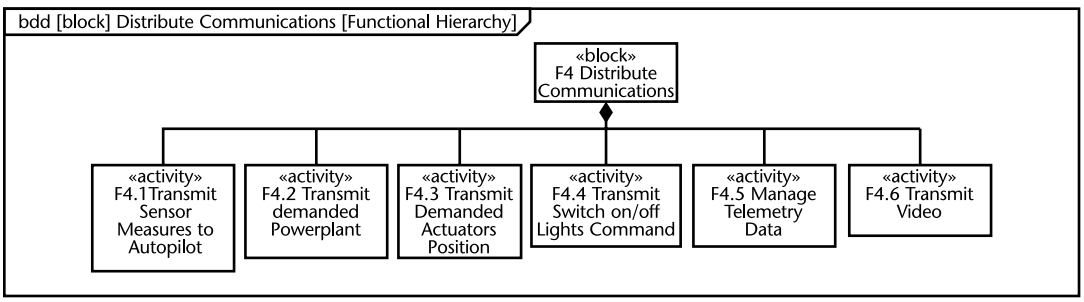


Figure 8.5 "Distribute communications" functional hierarchy in a block definition diagram.

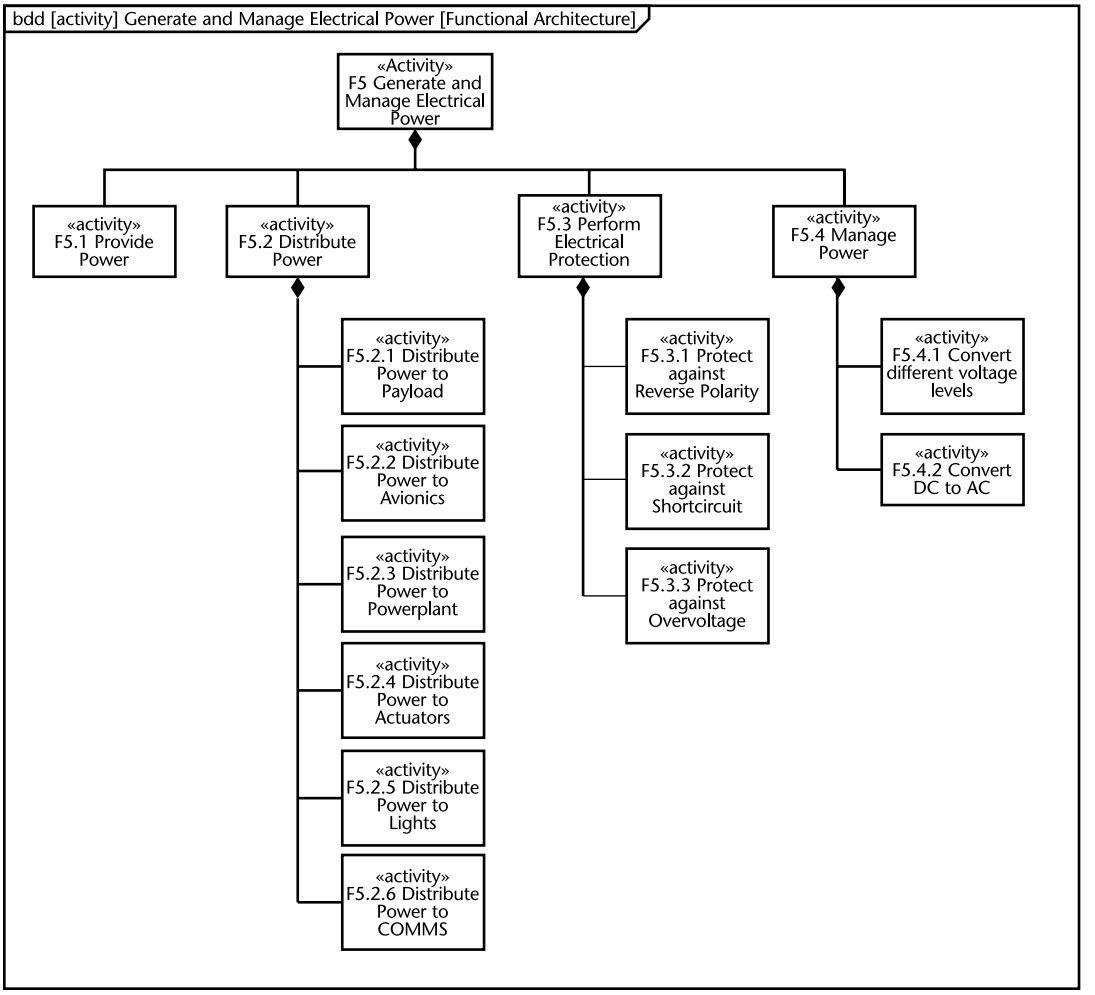


Figure 8.6 "Generate and Manage Electrical Power" functional hierarchy represented on a block definition diagram.

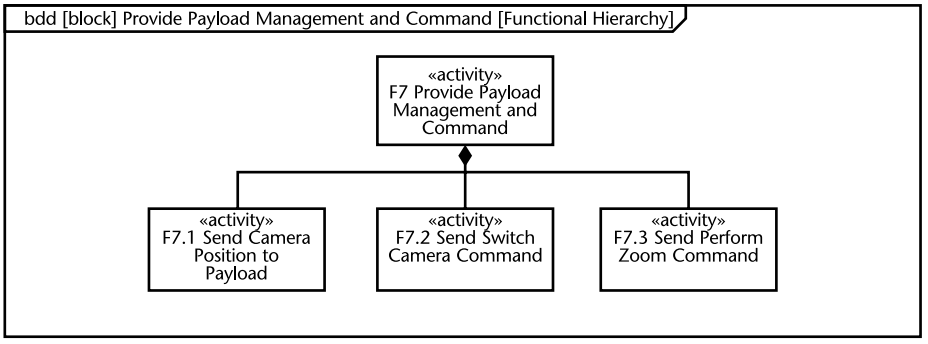


Figure 8.7 Provide payload management and command functional hierarchy.

Table 8.3 Transmit Sensor Measures Function

Function	F4.1 Transmit sensor measures
Description	Transmit measures obtained by the function read sensors
Inputs	Sensor measurements
Outputs	Sensor measurements to autopilot
Parent function	F4
Children functions	TBD

Table 8.4 Transmit Demanded Powerplant* Function

Function	F4.2 Transmit demanded power plant
Description	Transmit the throttle position to the engine
Inputs	Demanded power plant
Outputs	Demanded power plant to engine
Parent function	F4
Children functions	TBD

*Power plant refers to an electric engine regime.

Table 8.5 Transmit Demanded Actuator Position Function

Function	F4.3 Transmit demanded actuators position
Description	Send the signal corresponding to the demanded actuator position to the actuators
Inputs	Demanded actuator position
Outputs	Demanded actuators position to actuator
Parent function	F4
Children functions	TBD

Table 8.6 Transmit Switch On/Off Lights Command Function

Function	F4.4 Transmit switch on/off lights command
Description	Transmit the signal corresponding to switch on or off the navigation lights
Inputs	Signal corresponding to switch on or off the navigation lights
Outputs	Signal corresponding to switch on or off the navigation lights to lights
Parent function	F4
Children functions	TBD

8.2.2 System Requirements

The functional requirements are captured using the templates described in Appendix B. The requirements associated to the functions identified above are as follows:

Function: Transmit Sensor Measures
FR_4.1 The function transmit sensor measures to autopilot shall deliver sensor measurements in order to calculate the control vector.

Function: Transmit Demanded Power Plant

Table 8.7 Manage Telemetry Data Function

Function	F4.5 Manage telemetry data
Description	Send the data to the ground station via radio and send data or commands to the autopilot received by the radio
Inputs	Telemetry data from autopilot
Outputs	Telemetry data to ground station. Command to the autopilot
Parent function	F4
Children functions	TBD

Table 8.8 Transmit Video Function

Function	F4.6 Transmit video
Description	Transmit video signal to process
Inputs	Video signal from camera
Outputs	Video signal to video link
Parent function	F4
Children functions	TBD

Table 8.9 Provide Power Function

Function	F5.1 Provide power
Description	Provide electrical power to all the aircraft systems
Inputs	Stored power
Outputs	Electrical power
Parent function	F5
Children functions	TBD

Table 8.10 Distribute Power Function

Function	F5.2 Distribute power
Description	Power distribution to the aircraft subsystems
Inputs	Conditioned power
Outputs	Electrical power to systems
Parent function	F5.
Children functions	Distribute power to payload, distribute power to avionics, distribute power to power plant, distribute power to actuators, distribute power to lights, and distribute power to COMMS

Table 8.11 Perform Electrical Protection Function

Function	F5.3 Perform electrical protection
Description	Provide electrical protection
Inputs	Electrical power
Outputs	Protected power
Parent function	F5
Children functions	Protect against reverse polarity, protect against short-circuit, and protect against pvervoltage

Table 8.12 Manage Power Function

Function	F5.4 Manage power
Description	This function performs power management
Inputs	Protected power
Outputs	Conditioned power
Parent function	F5
Children functions	Convert to different voltage levels and convert DC to AC

Table 8.13 Send Camera Position to Payload Function

Function	F7.1 Send camera position to payload
Description	Send the desired camera position to point to the target
Inputs	Desired camera position
Outputs	Desired camera position to camera gimbal
Parent function	F7
Children functions	TBD

Table 8.14 Send Switch Camera Command Function

Function	F7.2 Send switch camera command
Description	Send the switch camera command selected by the operator. This function is used to switch between the onboard cameras
Inputs	Switch camera command
Outputs	Switch camera to camera
Parent function	F7
Children functions	TBD

Table 8.15 Send Perform Zoom Command Function

Function	F7.3 Send perform zoom command
Description	Send the desired zoom level selected by the operator. This function is used to apply zoom to the camera
Inputs	Zoom level
Outputs	Zoom level to camera
Parent function	F7
Children functions	TBD

FR_4.2 The function transmit demanded power plant shall deliver the demanded power plant to generate thrust.

Function: Transmit Demanded Actuators Position

FR_4.3 The function transmit demanded actuators position shall deliver the demanded actuator position in order to provide aircraft control.

Function: Transmit Switch On/Off Lights Command

FR_4.4 The function transmit switch on/off lights command shall deliver the switch on/off signal to provide visibility to other traffics.

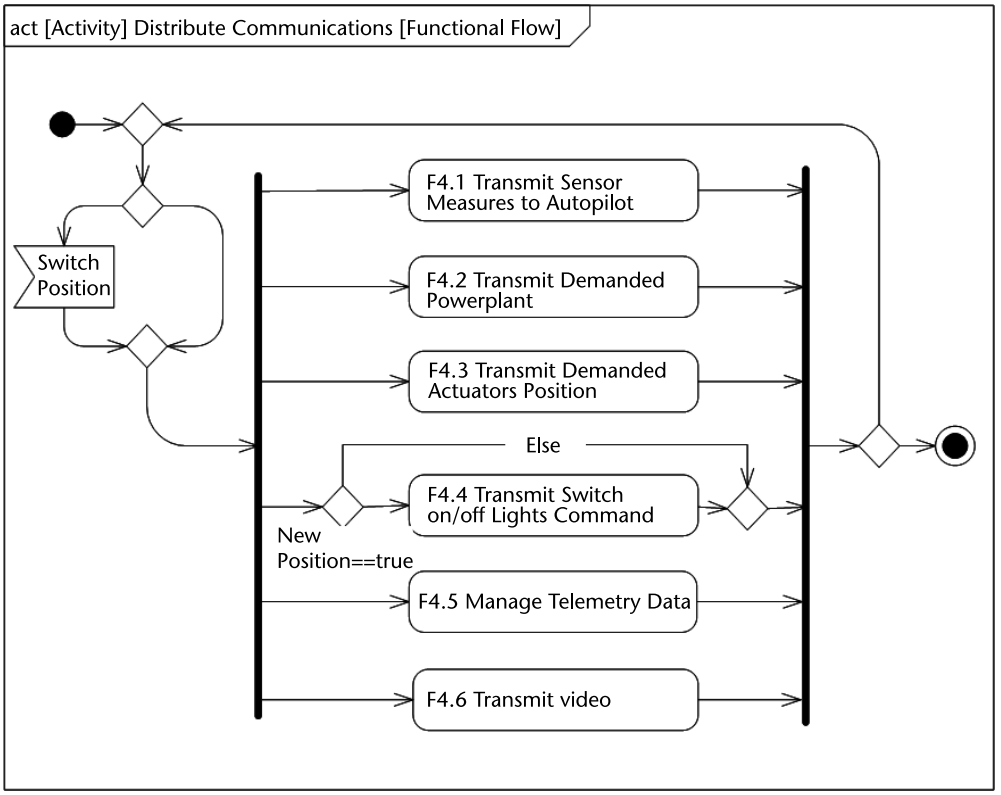


Figure 8.8 Distribute communications functional flow.

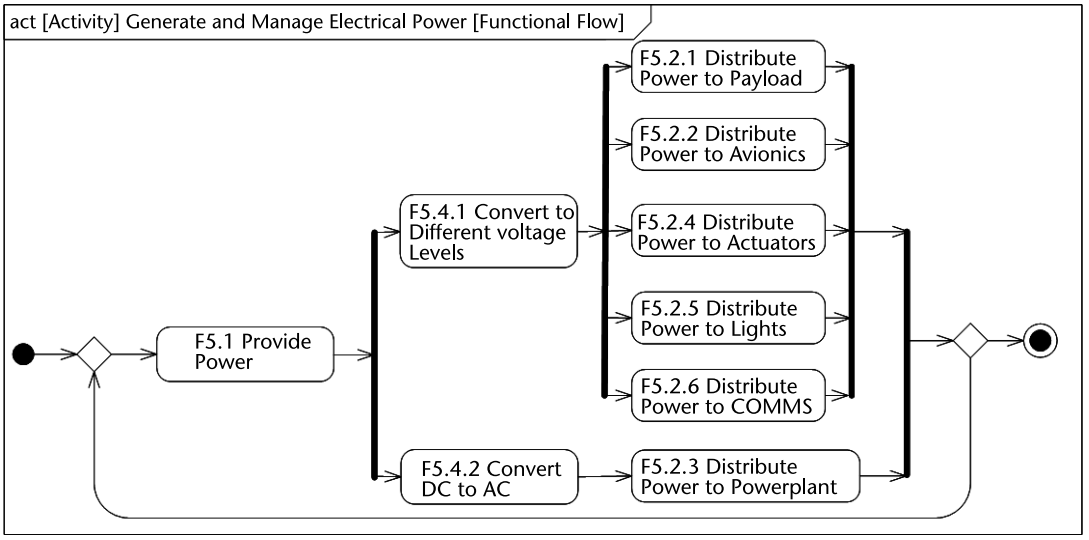


Figure 8.9 Generate and manage electrical power functional flow.

Function: Manage Telemetry Data

FR_4.5a The function manage telemetry data shall transmit state information to the ground station.

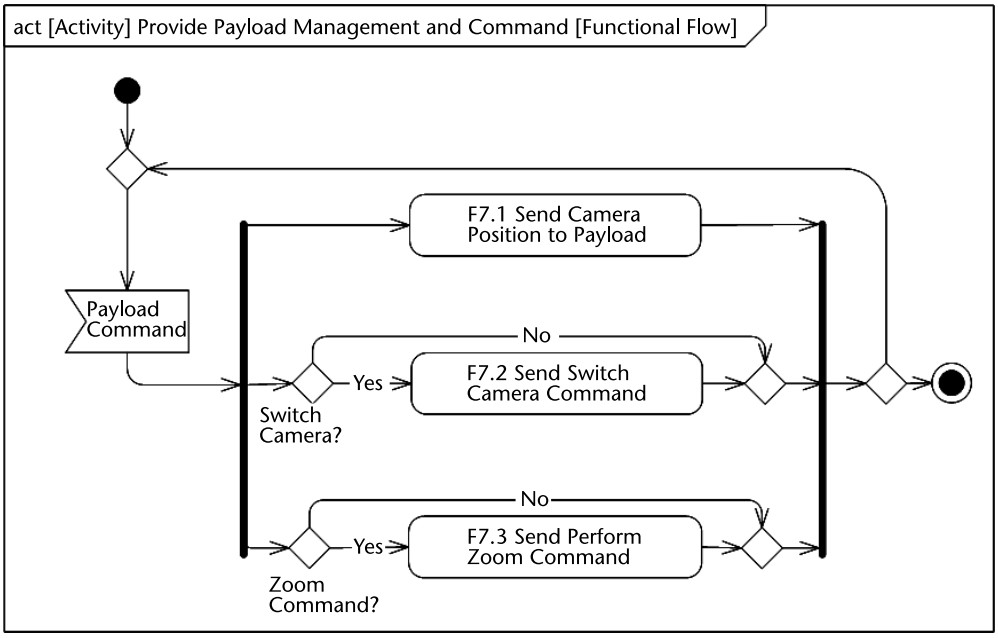


Figure 8.10 Provide payload management and command functional flow.

FR_4.5b The function manage telemetry data shall transmit commands from the ground station.

Function: Transmit Video

FR_4.6 The function transmit video shall deliver the video signal to be processed.

Function: Provide Power

FR_5.1 The function provide power shall provide electrical power to the aircraft subsystems autonomously.

Function: Distribute Power

FR_5.2 The function distribute power shall distribute electrical power to all the electrical equipment.

Rationale: The electrical equipment will depend on the final electrical subsystem solution.

Function: Perform Electrical Protection

FR_5.3a The function perform electrical protection shall provide protection against reverse polarity.

FR_5.3b The function perform electrical protection shall provide protection against short-circuits.

FR_5.3c The function perform electrical protection shall provide protection against overvoltage.

Function: Manage Power

FR_5.4a The function manage power shall provide electrical power to different DC levels .

Rationale: DC levels will depend on the final electrical subsystems selected. The DC levels shall be determined during the detailed design.

FR_5.4b The function manage power shall provide electrical AC power.

Table 8.16 N^2 Chart of the F4 Distribute Communications Functional Interfaces

Measured environment data	Demanded power plant	Demanded actuators position	Signal corresponding to switch on or off the navigation lights	Data for telemetry	Video signal	
F4.1 Send sensor measures to autopilot						Data to autopilot
	F4.2 Send demanded power plant					Demanded power plant to the engine
		F4.3 Send demanded actuators position				Demanded actuators position to actuator
			F4.4 Send switch on/off lights command			Signal corresponding to switch on or off the navigation lights to lights
				F4.5 Send/receive data for telemetry		Data for telemetry to radio link
					F4.6 Send video	Video signal to video link

Table 8.17 N^2 Chart of the F5 Generate and Manage Electrical Power Functional Interfaces

Stored power from battery												
F5.1. Provide power							Electrical power	Electrical power	Electrical power			
	F5.2.1. Distribute power to payload											Power to payload
		F5.2.2. Distribute power to avionics										Power to avionics
			F5.2.3. Distribute power to power plant									Power to power plant
				F5.2.4. Distribute power to actuators								Power to actuators
					F5.2.5. Distribute power to lights							Power to lights
						F5.2.6. Distribute power to COMMS						Power to COMMS
							F5.3.1. Protect against reverse polarity			Reverse polarity protected electrical power	Reverse polarity protected electrical power	
								F5.3.2. Protect against short-circuit		Short-circuit protected electrical power	Short-circuit protected electrical power	
									F5.3.3. Protect against overvoltage	Overvoltage protected electrical power	Overvoltage protected electrical power	
	Conditioned DC power	Conditioned DC power		Conditioned DC power	Conditioned DC power	Conditioned DC power				*F5.4.1. Convert to DC voltage levels*		
			Conditioned AC power								*F5.4.2. Convert DC to AC*	

Function: Send Camera Position to Payload
FR_7.1 The function send the desired camera position to payload shall deliver the target location.
Function: Send Switch Camera Command
FR_7.2 The function send switch camera command shall deliver the command to switch between electro, optical, or thermal images.
Function: Send Perform Zoom Command
FR_7.3 The function send zoom command shall deliver the command to zoom cameras.

At this point performance, environmental constraints, interface, and nonfunctional requirements are identified. An illustrative sample is shown below.

Performance Requirements (Intentionally Incomplete)
PR_FR.5.1: The provide power function shall provide enough power to perform flights 90 minutes long.

Environmental Constraints (Intentionally Incomplete)
EC_1 The operating temperature range of the system shall be between -15 to +60 Celsius degrees.
EC_2. The components of the aircraft shall resist light rain.
EC_3. The engine shall resist operation in dust environment.

Interface Requirements (Intentionally Incomplete)
IF_1. The connectors shall ensure they are not connected in a wrong way.
Rationale: The exact connectors design will depend on the final components selected. They will be determined when the detailed design will be finished.

Nonfunctional Requirements: Maintainability (Intentionally Incomplete)
NFR_Maint_005. The operator shall mount the aircraft without tools.
NFR_Maint_006. Replaceable parts of the aircraft shall be substituted in less than 5 minutes.
Rationale: Replaceable parts include those that are liable to be broken or damaged during the operation in cases of a harsh landing or due to harsh environmental conditions. Those parts typically include a motor's blade and payload, for instance. Therefore, once the final selection of components is completed, a tag "replaceable" will be included when appropriate and will help list these components and keep their traceability.
NFR_Maint_007. Embedded software shall be updated without disassemble the aircraft.
NFR_Maint_008. Aircraft shall provide a port to check the status of the aircraft.
Nonfunctional Requirements: Safety (Intentionally Incomplete)
NFR_Saf_009. The aircraft shall return to home in case of communications lost.
NFR_Saf_010. The aircraft shall have the mechanical strength and where appropriate the stability to withstand the stresses to which is subjected during the launch without breakage or deformation interfering with its safe flight.
NFR_Saf_012. The aircraft shall have the mechanical strength and tenacity to withstand any impact that may occur at landing.

Figure 8.11 shows the Seeker UAS battery that performs the provide power function.

Figure 8.12 illustrates how IF_1 has been implemented: unmatching connectors are used to interface different elements of the aircraft, thus preventing misconnections.

Data Dictionary (Intentionally Incomplete)

The following requirements define data interchanges among systems' elements, following the notation for data dictionaries recommended in Appendix B.
DD_R_050. Waypoint message definition:
Waypoint = latitude + longitude + altitude + airspeed
DD_R_051. Camera position message definition:
Camera Position = azimuth + elevation
DD_R_052. Payload command message definition:
Payload Command = Camera Position + Camera Index + Zoom Level

8.3 Physical Architecture and Heuristics Applied

Following the steps of the ISE&PPOOA process and once the functional architecture has been defined, the next step (ISE&PPOOA step 4) is to develop a suitable physical architecture. This architecture grows and develops along with the func-

Figure 8.11 Seeker UAS battery.

Figure 8.12 Seeker UAS connectors.

tional architecture, and so the process must be understood as an iterative process rather than a linear forward stepping one.

First and prior to describing the process applied to electrical subsystem, it is relevant to show the physical architecture of the whole aircraft in order to understand the context of the electrical subsystem studied in this chapter. Figure 8.13 shows the high-level subsystems of the aircraft.

The subsystems identified perform all the functions that the aircraft needs in order to perform its mission. A brief subsystem description is shown hereafter:

- Avionics subsystem comprises flight computer, autopilot, COMMS equipment, navigation system, and sensors;
- Mechanical subsystem comprises actuators, hinges, and so on that transform electrical signals in movement;
- Propulsion subsystem comprises engine, propeller, and all elements related;
- Mission subsystem comprises elements related to mission definition, management, and payload;
- Electrical subsystem is formed by the elements that store, transform, condition, and monitor electrical power and signals.

8.3.1 Heuristics Applied

As a fundamental part of the ISE&PPOOA process it is necessary to apply heuristics to build the modular and refined physical architecture in order to make design decisions to satisfy the quality attributes response. Modularity, maintainability, and safety are the main concerns to select the heuristics applied. which are summarized as follows:

- *SA_Heu_6*. Group and allocate functions that are strongly related to each other, separate functions that are unrelated.
- *SA_Heu_8*. Choose a configuration with minimal communications between the subsystems.
- *Man_Heu_8*. Isolate the expected change. Applied here to nonsoftware building elements as well.

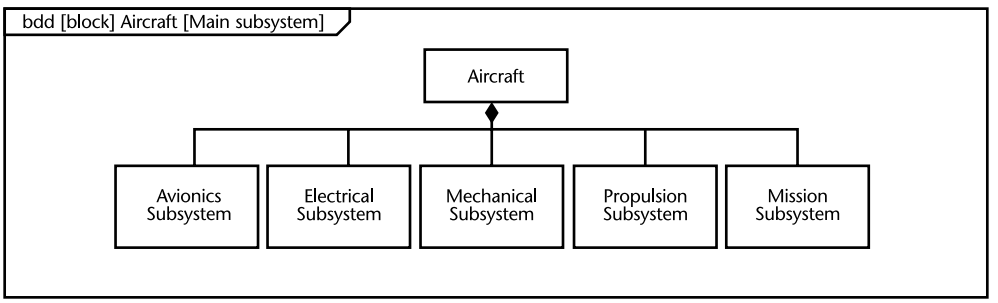


Figure 8.13 Aircraft main subsystems.

- *Man_Heu_9*. Raise the abstraction level. Applied here to nonsoftware building elements as well.
- *Man_Heu_13*. Maintain existing interfaces. Applied here to nonsoftware building elements as well.
- *SF_Heu_2*. Minimize the number of components and interactions.
- *SF_Heu_4*. Enforce time requirements.

The electrical subsystem is a very particular one given that it interconnects all the components that for any reason need to be electrically powered or involve data exchange. A good functional and physical architecture of the whole aircraft will lead to a better and simpler electrical subsystem. There are many examples of the application of heuristics to the whole aircraft design, which are explained below.

SA_Heu_6. As shown in Table 8.16, there are fewer relations between functions that will allow us to separate as independent functions.

SA_Heu_8. During the design, minimal communications between subsystems are pursued. This leads to a simpler and more efficient electrical subsystem.

Man_Heu_8 is used extensively during the design phase. For example, there are some components, such as radios and COMMS equipment, which might change to fulfill final user requirements regarding RF regulations or mission requirements (frequency, transmitted power, etc.). Therefore, it is important to isolate the radio component to simplify the substitution of this element and maintain interfaces (MAN_Heu_13).

Man_Heu_9 has been applied in the block Protection Element in Figure 8.14. The abstraction level has been raised in order to avoid considering protection characteristics of the different components.

SF_Heu_4 has been used to verify lost communications between subsystems.

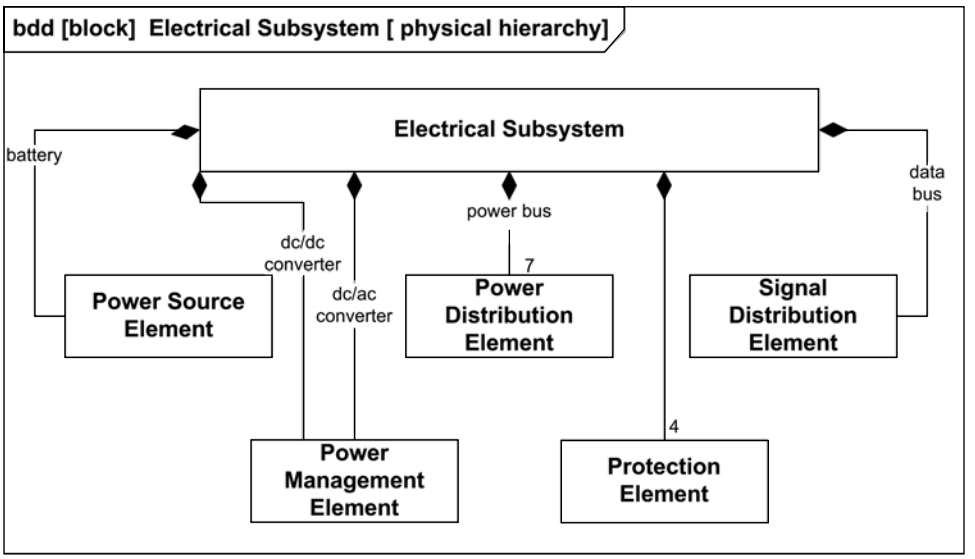


Figure 8.14 Electrical subsystem physical architecture.

8.3.2 Physical Architecture

The Electrical Subsystem physical hierarchy to be developed is obtained by functional allocation using the modularity criteria defined in Chapters 6 and 7, and applying the selected heuristics.

A battery stores energy and delivers it in the form of electrical power, different power supply rails are provided by suitable DC-to-DC power stages, and power for the engine is provided by a DC-to-AC power stage,and protection elements are included as well.

Power buses and signal buses distribute, respectively, power and data where necessary; in this context, a bus, either of power or data, corresponds to an abstract entity that consists of a physical part (wires and connectors), and a logical part (data format and protocol). In contrast to classical aeronautics, military, and commercial standards for avionics data buses, such as the well-known ARINC 429, ARINC 629, and MIL-STD-1553B/STANAG 3838, are not suitable for this application. Even though they are widely spread and show high robustness and modularity levels, they do not comply with overall weight and size constraints for the aircraft (refer to Section 8.1) and avionics for lightweight UAVs are not often compliant with these standards. To avoid being too constraining regarding electrical specifications for the onboard elements, the selection of adequate power and data buses is linked to the eventual choice of these elements. However, it is advisable to include them as power or signal distribution components in this early design stage so functional and nonfunctional requirements may be allocated and assigned to them.

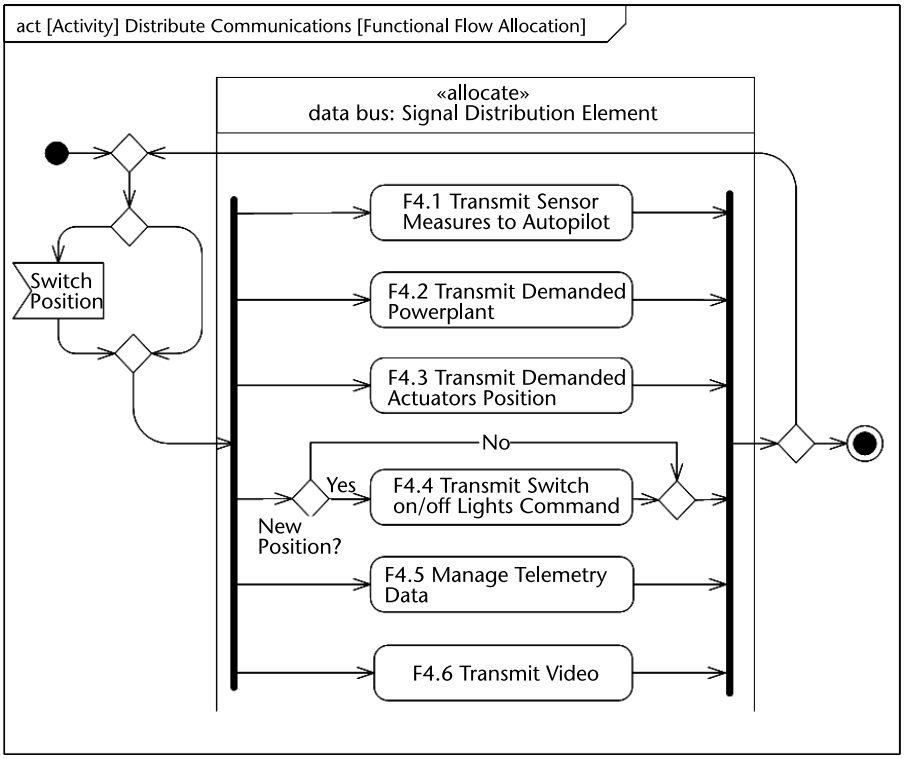


Figure 8.15 Allocated functional flow of distribute communications.

In the following step, functions identified previously are allocated to physical elements (step 4.1 of ISE&PPOOA) as shown hereafter.

Figures 8.15, 8.16, and 8.17 reflect the model that results from this allocation step and are the result of an iterative process that is not shown in this chapter for the sake of brevity.

Once the heuristics have been applied and the allocation has been completed, the refined architecture is done. The internal block description diagram in Figure 8.18, along with Tables 8.18–8.25, serve this purpose.

Tables 8.18–8.25 provide physical parts description.

8.4 Summary

The ISE&PPOOA process has been followed in this chapter to engineer a light-weight UAS. As in many processes in engineering, this should be not understood as a linear, one-way, step-by-step procedure, like a cooking recipe to be followed, but rather as a how-to-think approach developed through many iterative stages, whose results are gathered along the chapter as models and textual descriptions. This is particularly true within the development of functional and physical architecture since both grow in a parallel and iterative way, feeding each other with information as they are refined. This is somewhat similar to "the twin peaks of requirements and architecture" model [6], but with functional and physical architectures refined in a parallel, iterative way.

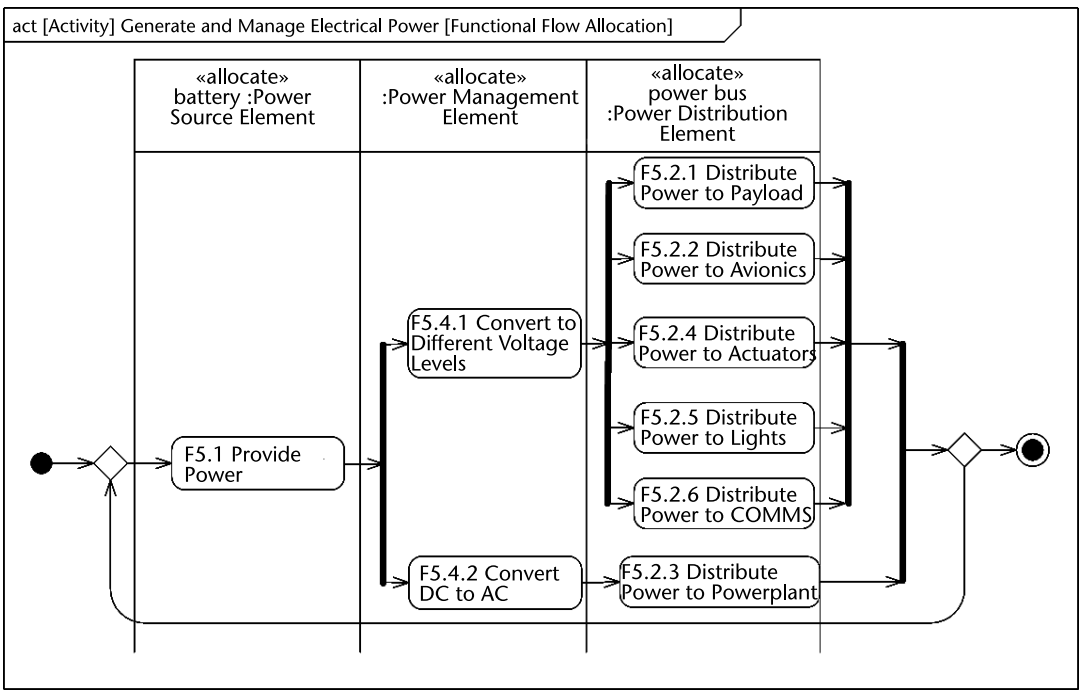


Figure 8.16 Allocated functional flow of generate and manage electrical power.

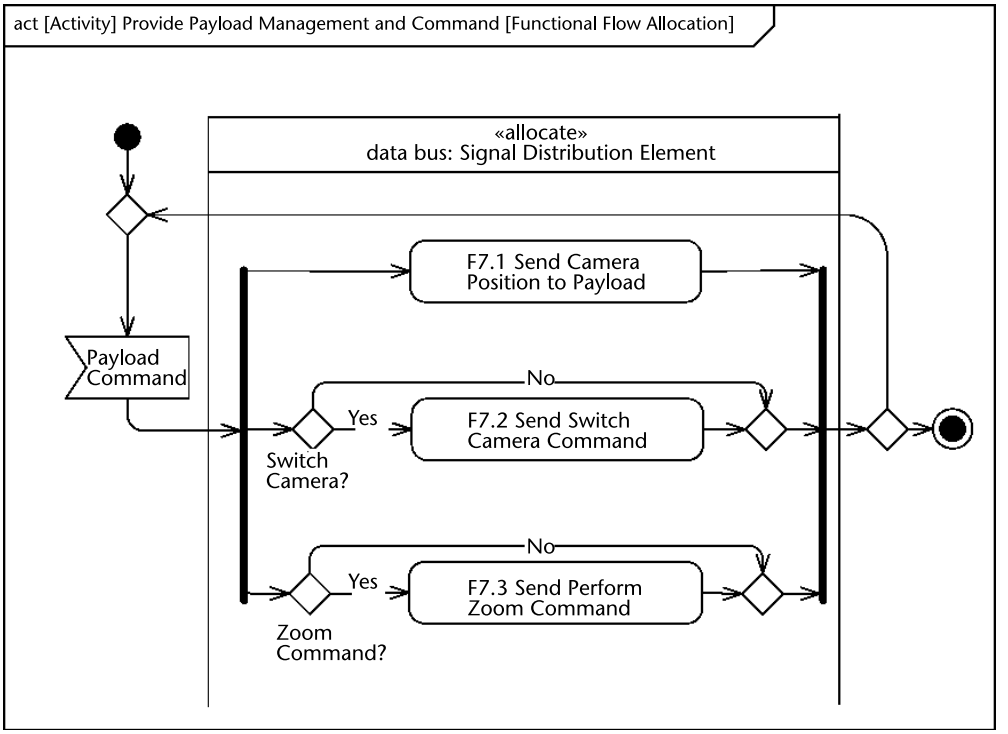


Figure 8.17 Allocated functional flow of provide payload management.

This methodology constitutes a flexible approach that leads to design better systems and products, document and organize information, and points out flaws in designs, architectures, and engineering choices.

In conclusion regarding UAV electrical subsystem design, there are some advantages inherent to the ISE&PPOOA methodology that are worth pointing out, and that are deeply bound to three facts:

1. Generate functional requirements from functions;
2. Capabilities drive the functional hierarchy and hence the requirements;
3. Emergent properties are taken into account through nonfunctional requirements (there is indeed a genuine link between NFRs and emergence).

So, ISE&PPOOA use has led to

- An easy and intuitive way to identify and discover requirements and keep the requirement list long enough (but not longer) and meaningful. There aren't arbitrary detached requirements issued by opaque stakeholders or entities that would enlarge the list and would be hard to trace.
- An effective way to keep traceability of requirements and identify flaws on physical architectures, leading to a better design.
- A well-structured, well-defined, easy to handle, and long enough (but not longer) documentation of the preliminary design that can be easily updated in further stages of development.

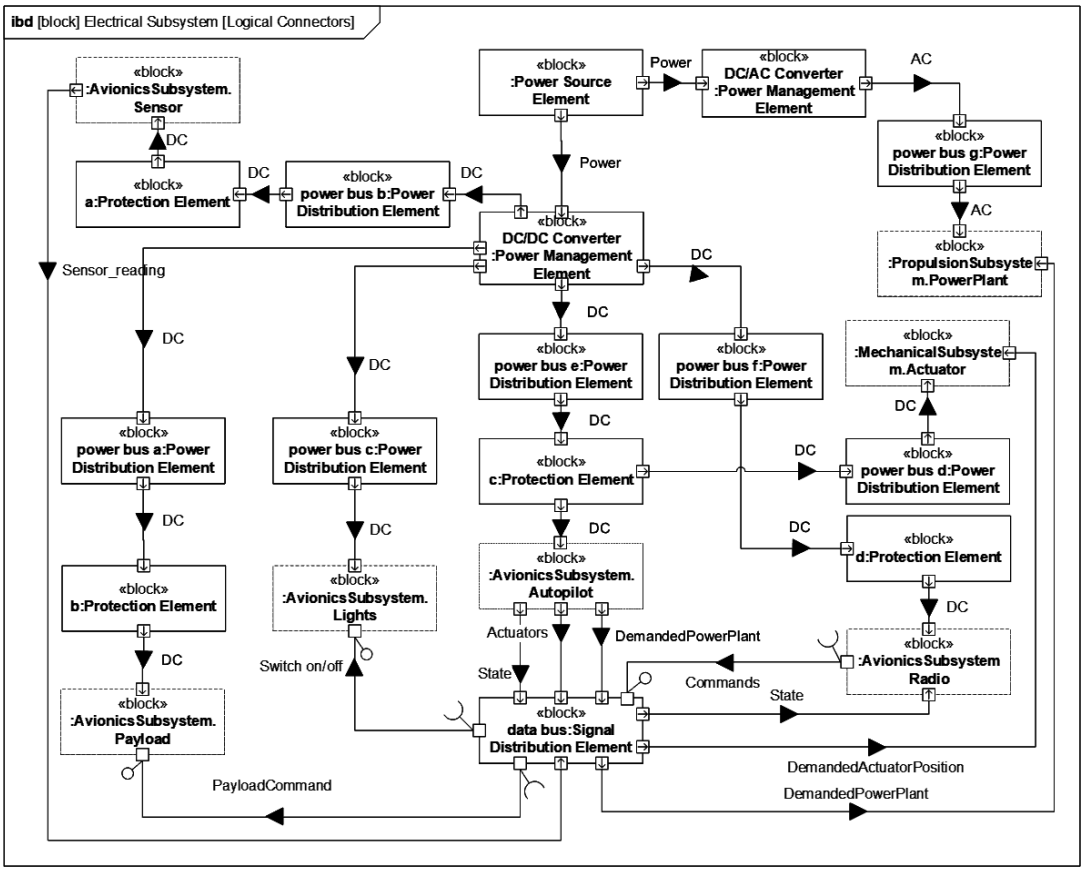


Figure 8.18 Electrical subsystem component relationships.

Table 8.18 Power Source Element Description

: Power Source Element
Inflow: Not applicable
Outflow: power
Allocated functions: • F5.1 Provide power

Table 8.19 Power Management Element Description

DC/DC Converter: Power Management Element
Inflow: power
Outflow: direct current
Allocated functions: • F5.4.1 Convert to different voltage levels

Table 8.20 Power Management Element Description

DC/AC Converter: Power Management Element
Inflow: power
Outflow: alternating current
Allocated functions: • F5.4.2 Convert DC to AC

Table 8.21 Power Distribution Element Description

Power bus a: Power Distribution Element
Inflow: alternating current | direct current
Outflow: alternating current | direct current
Allocated functions: • F5.2.2 Distribute power to avionics

Table 8.22 Power Distribution Element Description

Power bus d: Power Distribution Element
Inflow: alternating current | direct current
Outflow: alternating current | direct current
Allocated functions: • F5.2.4 Distribute power to actuators

Table 8.23 Power Distribution Element Description

Power bus c: Power Distribution Element
Inflow: alternating current | direct current
Outflow: alternating current | direct current
Allocated functions: • F5.2.3 Distribute power to power plant

Table 8.24 Protection Element Description

b: Protection Element
Inflow: direct current
Outflow: direct current
Allocated functions: • F5.3.1 Protect against reverse polarity • F5.3.2 Protect against short circuit • F5.3.3 Protect against overvoltage

- A convenient way to put all the project information in context, suitable to be used as a reference and guide at any level, from developers and engineers to project managers, systems engineers, and managing staff.

Table 8.25 Signal Distribution Element Description

data bus a: Signal Distribution Element
Inflow: state, sensor reading, demanded actuators position, demanded power plant, commands
Outflow: state, demanded actuators position, demanded power plant, payload command, switch on/off
Allocated functions: • F4.1. Transmit sensor measures to autopilot • F4.2. Transmit demanded power plant • F4.3. Transmit demanded actuators position • F4.4. Transmit switch on/off lights command • F4.5. Manage telemetry data • F4.6. Transmit video • F7.1. Send the desired camera position to point to the target • F7.2. Send switch camera command • F7.3. Send perform zoom command

References

[1] Stevens, B. L., F. L. Lewis, and E. N. Johnson, *Aircraft Control and Simulation: Dynamics, Controls Design, and Autonomous Systems,* Hoboken, NJ: John Wiley & Sons, November 2015.

[2] Moir, I., and A. Seabridge, *Design and Development of Aircraft Systems*, Hoboken, NJ: John Wiley & Sons, December 2012.

[3] Jackson, S., *Systems Engineering for Commercial Aircraft*, Second Edition, Surrey, UK: Ashgate Publishing Limited, 2015.

[4] Fletcher, S., et al., "Modeling and Simulation Enabled UAV Electrical Power System Design," *SAE International Journal of Aerospace*, Vol. 4, No. 2, 2011, pp. 1074–1083.

[5] Fletcher, S., et al., "Determination of Protection System Requirements for DC Unmanned Aerial Vehicle Electrical Power Networks for Enhanced Capability and Survivability," *IET Electrical Systems in Transportation*, Vol. 1, No. 4, December 2011, pp. 137–147.

[6] Nuseibeh, B., "Weaving Together Requirements and Architectures," *IEEE Computer*, Vol. 34, No. 3, March 2001, pp. 115–119.

CHAPTER 9

Example of Application: Collaborative Robot

In this chapter, we present the application of ISE&PPOOA to the engineering of a collaborative robot application. Robotic systems are addressing more demanding scenarios; for example, in manufacturing where traditional automation solutions are being replaced by advanced robotic systems. These new robotic systems are very heterogeneous and software-intensive, using novel techniques to meet the need for flexibility and human collaboration. The engineering of these complex systems presents multiple challenges to systems engineers, but the ISE&PPOOA method can help to address them successfully.

This chapter is structured following the steps in the ISE&PPOOA process (see Chapter 4). In Section 9.1, the mission dimension of the robotic application is discussed, applying the steps in the systems engineering subprocess to identify the operational scenarios for the robot and specify the system capabilities and high-level functional requirements. In Section 9.2, the functional architecture is obtained by iteratively identifying the functions needed to realize the capabilities and decomposing them into subfunctions. Section 9.3 elaborates on the allocation of the functions to the building elements of the robotic system and the refinement of the resulting physical architecture by applying some of the heuristics proposed in Chapter 6, with a special focus on safety, a critical quality attribute in collaborative robotic applications. Flexible robot solutions for manufacturing are software-intensive, so Section 9.4 presents how the PPOOA subprocess can be applied to obtain the software architecture for our collaborative robot, which is refined using the PPOOA software-related heuristics.

9.1 Example Overview, Needs, and Capabilities

New technologies, from robotics to artificial intelligence, are becoming available to automate more complex tasks, more flexibly, and in collaboration with humans. The automation industry needs methods and tools to develop these new complex robotic solutions. MBSE offers multiple advantages over traditional methods to

cope with the heterogeneity and complexity of these new software-intensive robotic applications.

In this example, we apply ISE&PPOOA to the engineering of a collaborative robotic application for handling small packages. A collaborative robot manipulator works together with a human operator to process orders that include products of different types in stacked containers. The operator enters the order through the user interface and presses start. The robot collects the products in the order from the containers and places them into the delivery bin. The operator is responsible for supervising the operation, replacing the product containers when empty, or accounting for any product not delivered properly by the robot. The operator can also include a new product type by defining the container position and stacking pattern. The robotic system generates the motions to pick the products automatically. The delivery bin is in a shared space by the robot and the operator. The robot and the operator interact safely in that space thanks to two novel technologies developed in the Factory-in-a-day project [1]. A robot skin allows the manipulator to detect any potential collisions with the operator, and then a dynamic obstacle avoidance controller adapts the robot motions to avoid the collisions.

9.1.1 Identify Operational Scenarios

The first step in ISE&PPOOA is the identification of the operational scenarios for the robotic system in order to model the mission dimension of the system. In this phase, the robotics experts and requirement engineers discuss with the factory-floor managers the intended behavior of the robot. It is important at this stage to involve the future operators that will work with the robot, in order to completely understand the expectations about the robot behavior.

Table 9.1 summarizes the different operational scenarios. Note that all scenarios involve an interaction with the operator. This is very different from traditional automation solutions, in which interactions are limited to deployment and maintenance.

A template form can be used to describe each operational scenario. The operational scenario "pick product" is presented in Table 9.2, following the template. Similar descriptions need to be provided for all the operational scenarios, but for the sake of brevity they are not included here.

9.1.1.1 Operational Needs

An important part in this step is also to identify the needs in all the operational scenarios. As an example, we present below some of these operational needs identified for the "pick product" scenario that are more representative of collaborative robotic applications and relevant for the discussion of ISE&PPOOA. Note that these operational needs capture the demands of the robotic application from the point of view of the customer/mission. At this stage, we do not need to be strictly rigorous, as they will be converted into precise systems requirements later.

Table 9.1 Operational Scenarios

Deployment or Adaptation Phase	
S1 System configuration and calibration	The operator installs the robot arm and the product containers in the frames, and moves the robot arm to the reference pose for each container, so all end-effector poses are calibrated off-line for the robot to be able to reach all the products.
S2 Configure new product*	The operator adds a stack of a new product type, possibly of different dimensions (within a limited variation range), or replaces an existing type with the new one and informs the system through the operator interface. The system is capable of processing orders that include the new product type.
Operation Phase	
S3 System start	The operator powers up all subsystems. The robot arm calibrates its joints.
S4 Manage order	The system receives an order request consisting of several products of one or more product types, and delivers the order in the bin by autonomously handling the products.
S5 Pick product	The robot moves the gripper to the container of the next product in the order, grasps an available product, and retreats from the container holding it.
S6 Deliver product	The robot delivers the product it is holding with the gripper in the delivery bin.
S7 Refill product	When one of the product stacks is empty, the robot notifies the operator and moves to a configuration that allows the operator access to replace the container with one filled with products.
S8 Shut down	The operator shuts down the system through the user interface. As a result, the status of the containers is stored in persistent storage and the robot arm moves to the standby position.
S9 Emergency stop	Upon an anomalous behavior, the emergency stop of the robot arm is activated, stopping any current motion, and this is notified to the operator.

* This is a paradigmatic scenario for the new advanced robotic solutions for manufacturing: the flexibility to quickly adapt the robotic system to handle new products.

Table 9.2 Description of the Operational Scenario "S5 Pick Product"

Operational Scenario	*S5 Pick product*
Preconditions	• System initialized. • There is an order processed and ready for execution or partially executed. • Nonempty containers of products.
Triggering event	The order manager component retrieves the next product in the order request.
Description	The robot arm performs the actions required to pick a product from the stack. The system uses the skin-sensing information to avoid any collisions while moving.
Postconditions	• Robot gripper is holding a product. • The stack has decreased by one unit.

9.1.1.2 Example of Operational Needs from the Scenario "Pick Product"

ON 2_1[1]*:* The system shall be able pick products of cylinder or box shape, with a maximum dimension of 20 cm, offering a top flat surface available for suction

1. It is important to identify the operational needs. An operational need may appear in multiple scenarios. Here ON stands for operational need, the first number corresponding to the scenario and the second number to the operational need.

grasp with a minimum dimension of 0.04m and a maximum weight of 0.2 kg.

ON 2_2: The system shall pick the products from container boxes that have an opening at the top of maximum dimensions 0.6×0.6m, stacked in rows and standing in vertical orientation. The maximum depth of the containers is 0.3m.

ON 2_3: The system shall complete the pick task in less than 2 seconds if there are no potential collisions while performing.

ON 2_4: The system shall move to the stack of the requested type of product without colliding with fixed parts of the work cell.

ON 2_5: The system shall pick the item safely, guaranteeing the safety of the operator sharing its workspace according to ISO/TS 15066:2016 [2].

ON 2_5.1: The area surrounding the delivery bin shall be considered shared workspace with the operator at all times.

ON 2_5.2: The system shall monitor a minimum safe distance of the robot arm to an obstacle while moving within the shared workspace. If violated, the motion will be stopped immediately.

ON 2_5.3: The moving parts of the system shall not present any sharp edges, and the robot arm and the gripper to handle the parts shall be lightweight.

9.1.2 Capabilities and High-Level Functional Requirements

After describing the operational scenarios of our robotic system, the next step is to convert the operational needs into the set of capabilities, functions, and quality attributes that the robotic system needs to implement to address them.

The following are some important considerations that we have applied to identify the capabilities of our robotic system:

- Considering general categories of capabilities can be helpful during the capability analysis for a new application. It helps for the completeness of the analysis, preventing from missing some specific capabilities that are needed and that could otherwise be overseen. These general categories are identified by the domain experts. Figure 9.1 shows the capabilities identified by considering general capabilities of manufacturing robotic systems; for example, high availability is a general capability category important for production systems. Another example is the domain requirement for more flexible production systems that helped us identify a specific capability for the collaborative robot application and quick deployment times. One of the advantages of collaborative robots is that they can be quickly deployed; adapted to a different manufacturing application by simply reprogramming them and adapting their accessories, such as the gripper used to handle parts.

  Considering general capabilities also allows us to identify different concerns in our desired behavior. In our collaborative robot, a critical desired behavior for safety (general category) is that the robot avoids obstacles when moving. This relates to the harmless capability, by preventing the robot from causing any harm to the human operator, to the products, or to the factory equipment. However, the avoid obstacles behavior of the robot also

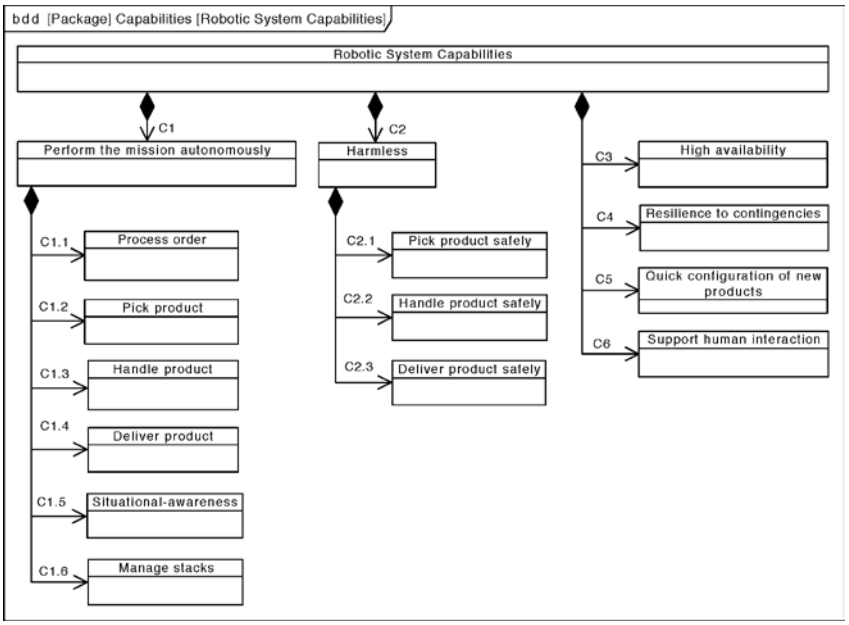


Figure 9.1 SysML block definition diagram that captures the capabilities to be provided by the collaborative robot application.

relates to resilience to contingencies (C4), an important aspect related to high availability (C3).

- We are in the mission dimension, so we do not yet consider the robotic technologies we will use in the solution. The robotic system must be able to pick product (C1.2), handle it by bringing it from the stack to the delivery bin (C1.3), and to deliver it (C1.4), but we are not committing to how the robot will do that (e.g., it might locate the product with a camera to pick it or know the location based on the geometry of the stack and the product dimensions).
- Any capability must be required by at least one operational scenario. To avoid unnecessary capabilities, we create a traceability matrix (see Table 9.3) that identifies which capabilities are needed for each scenario based on the needs of the scenario.
- Identifying the children elements of the capabilities is important to later create the hierarchies for the functional and quality models, and then extract the requirements. For example, C3 high availability will be addressed by including a high-level function to manage errors.

9.1.3 Quality Attributes and System NFRs

To obtain the NFRs, a helpful guide is to first analyze the quality attributes relevant for the system by obtaining a quality attributes tree complementary to the capability tree. ISE&PPOOA recommends a general quality model (see Appendix B) that can be adapted to each particular application. The quality attributes tree for our robotic

Table 9.3 Traceability Matrix of Capabilities (Columns) and Scenarios (Rows)

	C1.1 Process Order	*C1.2 Pick Product*	*C1.3 Handle Product*	*C1.4 Deliver Product*	*C1.5 Situational-Awareness*	*C1.6 Manage Stacks*	*C2.1 Pick Product Safely*	*C2.2 Handle Product Safely*	*C2.3 Deliver Product Safely*	*C3 High Availability*	*C4 Resiliency to Contingencies*	*C6 Quick Configuration of New Products*	*C6 Support Human Interaction*
S1 System Configuration and Caliblibration	—	—	—	—	—	—	—	—	—	X	X	—	X
S2 Config. New Product	—	—	—	—	—	—	—	—	—	X	X	X	X
S3 System Start	—	—	—	—	—	—	—	—	—	X	X	—	X
S4 Manage Order	X	—	—	—	—	X	—	—	—	X	X	—	X
S5 Pick Product	—	X	—	—	X	—	X	—	—	X	X	—	—
S6 Deliver Product	—	—	X	X	X	—		X	X	X	X	—	—
S7 Refill Product	X	—	—	—	—	X	—	—	—	X	X	—	X
S8 Shutdown	—	—	—	—	—	—	—	—	—	X	X	—	X
S9 Emergency Stop	—	—	—	—	—	—	—	—	—	X	X	—	X

system (see Figure 9.2) has been obtained by identifying the quality attributes (QAs) in the general model relevant in our robotic manufacturing application and refining them as suitable. For example, availability in a factory system depends mainly on the reliability of the components and it is related to resiliency subfactors. Safety is a key aspect in collaborative robotic applications. In this example, we will focus on two aspects of asset protection [7]: harm protection, to limit the harm when an incident occurs to acceptable levels (e.g., harm inflicted to the human operator if there is a collision with the robot), and safety incident protection, for measures to prevent such an incident from happening. Another important aspect in modern robotic manufacturing applications is the flexibility to adapt the system to process new types of products, which corresponds to changeability in the ISE&PPOOA generic quality model. It is important to note that quality attributes can be related to multiple concerns. For example, in our robot application the response time for detecting an obstacle is also critical for safety. The actual robotic application involves more detailed quality attributes, but for the sake of this example we are only considering those in Figure 9.2.

Once we obtain the quality attributes tree, the quality models for those attributes will provide us with templates to define associated requirements. For example, let us consider the quality model for efficiency. The most important efficiency aspect in a manufacturing application is the average time to process a product. Therefore, the consideration of the QA response time leads to requirements on the end-to-end time for the functions in the main flow of our system (i.e., the one to process and deliver an order) (see requirement PR_001 in Section 9.2.2). Another example is the QA changeability, which leads to requirements on changeover times (i.e., the average time needed to make changes to the system to handle new products) (see requirment NFR_Maint_001 in section 9.2.2).

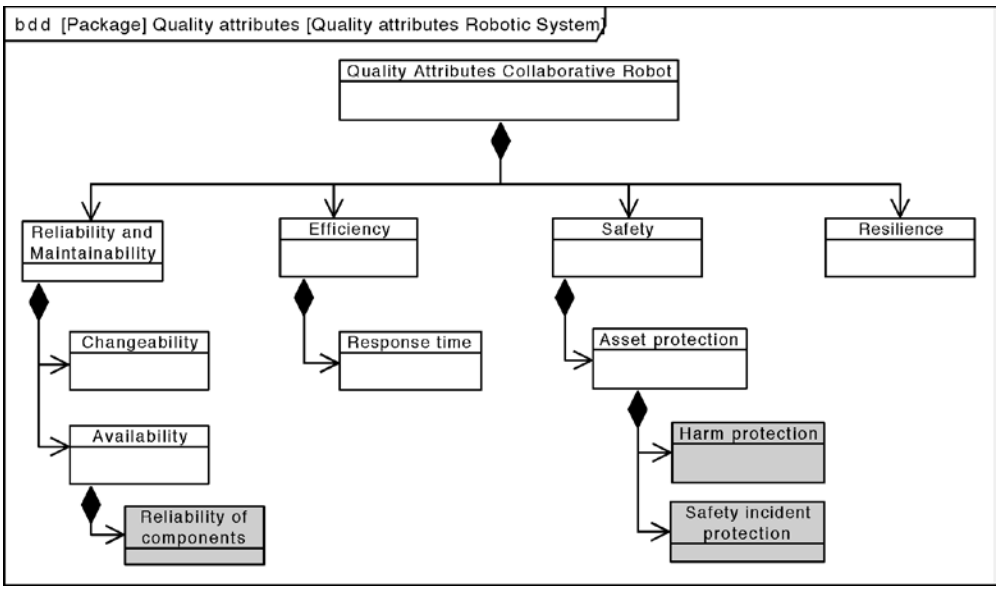


Figure 9.2 Quality attributes of the collaborative robot application considered for this example (intentionally incomplete). The shaded boxes show the QAs added to the general model upon refinement for the robot application.

These nonfunctional requirements derived from the quality attributes will be addressed by applying the corresponding heuristics from Chapter 6 in the architectural design (see Section 9.3).

9.2 Functional Architecture and System Requirements

Once the robotic system has been characterized from a mission viewpoint through the capabilities and quality attributes tree, we have to design the functional architecture that addresses them. This step is the most important one in the ISE&PPOOA process, and a differentiating one with other methods, as explained in Chapter 5.

Figure 9.3 establishes the context for our collaborative robotic application by using a SysML Internal Block Diagram[2], in which the relations between the main elements are defined. As is usual in modern robotic applications in manufacturing, the robotic system consists of an off-the-shelf robot manipulator (the robot), an end-effector designed to handle the products in this application (the suction gripper), complementary sensor devices (the robot skin), and a control subsystem responsible of coordinating the operation and communicating with the operator through a user interface. Note that in this diagram we consider the operator part of the overall system, whereas the product is considered external (dashed box). The rest of the work cell elements (mainly frames and wiring) are omitted from the diagram. The control subsystem is the more demanding element to develop and will be the focus of the example in this chapter.

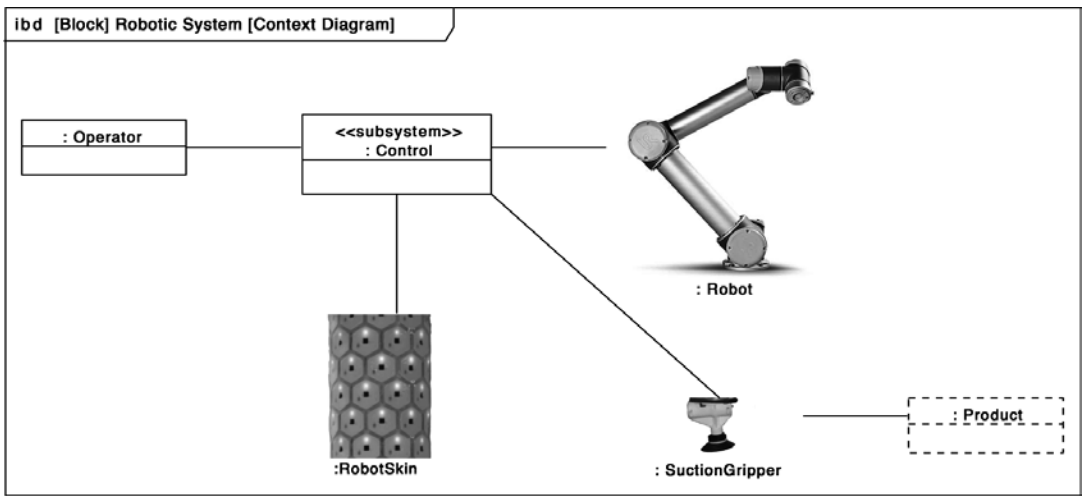


Figure 9.3 Context diagram for a collaborative robotic pick and place application.

2. Note that in Chapter 8 we have used a block definition diagram to define the context. Both are possible, and it is up to the modeler to choose the one that provides a better representation for the context of the particular system.

9.2.1 Functional Architecture

To obtain the functional hierarchy, three approaches were presented in Chapter 5. For our collaborative robotic application, we follow the combined approach as follows:

1. First, we identify the high-level functions from the capabilities (Figure 9.1 and Table 9.4), and the domain knowledge. These high-level functions are described in Table 9.5.
2. Second, we obtain the description of the main responses of the system (i.e., the desired behavior). For this, we take each operational scenario and

Table 9.4 Collaborative Robot Capabilities

Capability	*Description*
C1 Perform the mission autonomously	The robotic system should be able to process and deliver a complete order of products autonomously.
C1.1 Process order	The system should be able to process the order request and decompose it into pick and place operations for the products in the order.
C1.2 Pick product	The robot can pick a product from the stack.
C1.3 Handle product	The robot can move holding any of the products without dropping it.
C1.4 Deliver product	The robot can deliver the products by placing them in the delivery bin.
C1.5 Situational awareness	The robot will be provided with a 3-D model of the work cell and sensors to monitor its surroundings and have runtime information about possible collisions.
C1.6 Manage stacks	The robot keeps an inventory of the stacks to know where to pick a product from, and also notifies the operator when the stack is empty.
C2 Harmless	Harmless is a specific aspect of safety, considering that the robot should not cause any harm. Harmless is a complex capability that requires both the realization of certain functionalities (e.g., C2.1), together with quality attributes (e.g., mobile parts shall not expose sharp edges). For this, the system will comply with ISO/TS 15066:2016 for collaborative robot applications [2].
C2.1 Pick product safely	The robot picks the product while avoiding undesired collisions with any part of the work cell, the products, or the operator.
C2.2 Handle product safely	The robot moves holding a product without losing grip on it or colliding with any obstacles.
C2.3 Deliver product safely	The robot deposits the products in the delivery bin without inflicting any damage to the products, the bin, or the operator.
C3 High availability	The robotic system shall be available 98% of the time (also referred as up time in manufacturing). Increased availability is a complex capability that encompasses different functionalities and quality attributes. This capability will map to reliability requirements for the components of the robotic system.
C4 Resiliency to contingencies	A contingency is an unforeseen and undesired interaction with another actor or its environment. Examples of contingencies our robotic system must be able to manage are small deviations in the stacking of products, or minor irregularities in their shape that could result in picking failures.
C5 Quick configuration of new products	A key aspect in modern robotic systems is their flexibility to quickly configure them to process new types of products. A single operator is able to install a container stack with a new type of products. This involves configuring the control software so that the robot can process orders that include the new product, including performing the new motions required to pick and deliver the product.
C6 Support human interaction	The system will offer an interface for human operators to manage the system.

Table 9.5 Identification of High-Level Functions in the Robotic System

F1 Handle Operator Interface from C6	The system handles communication with the human operator through a suitable device so that the operator can: request an order, monitor the status of the order processing, and cancel it.
F2 Mange Order from C1.1	The system decomposes an order request into the set of pick and place operations needed to deliver in the bin all the products in the order requests.
F3 Coordinate Product Operation from C1.1	The system plans and coordinates the execution of actions needed to pick each product from its corresponding stack and deliver it in the bin.
F4 Manage Stacks of Products from C1.6	The robot keeps an inventory of the stacks to know where to pick a product from.
F5 Move Robot Safely from C2.1, C2.3	The robot moves within the work cell without causing any harmful collision with the environment or the operator.
F6 Pick Product from C1.2, C2.1 and C2.2	The robot picks each of the products requested in the order from the corresponding stacks.
F7 Deliver Product from C1.4, C2.2 and C2.3	The robot delivers all the products in the requested order by dropping them to the delivery bin in a way that none of the products in the bin are damaged.
F8 Manage Errors from C3	Identify any abnormal behavior during the execution of a task and take corrective measures to overcome it. Among the abnormal behavior, we can consider a malformed order request or a malformed trajectory for motion execution. The robot notifies the operator in the case of these errors, and relevant information of the system execution is logged. The system should detect errors to prevent failures (e.g., sanity check for trajectories to prevent a malformed trajectory from being executed). When the failure results in harm it is a safety issue.
F9 Calibrate from C5	The operator can place the new stack of products in the work cell and make the necessary calibration so that the robot can move and pick all the products in the new stack, for example, by introducing the stacking pattern and a reference position, or perform offline teaching of the motions required by the robot to reach and pick all products.

identify the atomic actions it requires from the system. As an example, the atomic actions required for the scenario "S5 pick product" are modeled by the activity diagram in Figure 9.4. When the robot executes the motion to the product stack, it can be interrupted by the signal "sensed obstacle" sent by the parallel flow corresponding to the sensing of dynamic obstacles (on the right of the figure), which runs periodically at high frequency to detect obstacles using the robot skin. Note that the diagram in Figure 9.4 is a draft used to identify and group activities into functions, as annotated in gray. In the activity diagram for the final version of the model, all activities should be labeled to indicate the corresponding functions.

3. The next step is to iteratively group the actions in the diagram into activities or subfunctions, trying to map them to the high-level functions and identify their functional interfaces. The cohesion criteria described in Chapter 5 drives this grouping process. One of the simple, practical rules to apply the cohesion criteria is the following:

   If a parent function has children subfunctions, at least one of the children should use the inputs to the parent function, and at least one child subfunction should produce one of the outputs of the parent function.

The N^2 charts described in Chapter 5 are very useful for the grouping process because they offer a compact representation of the functional interfaces. For

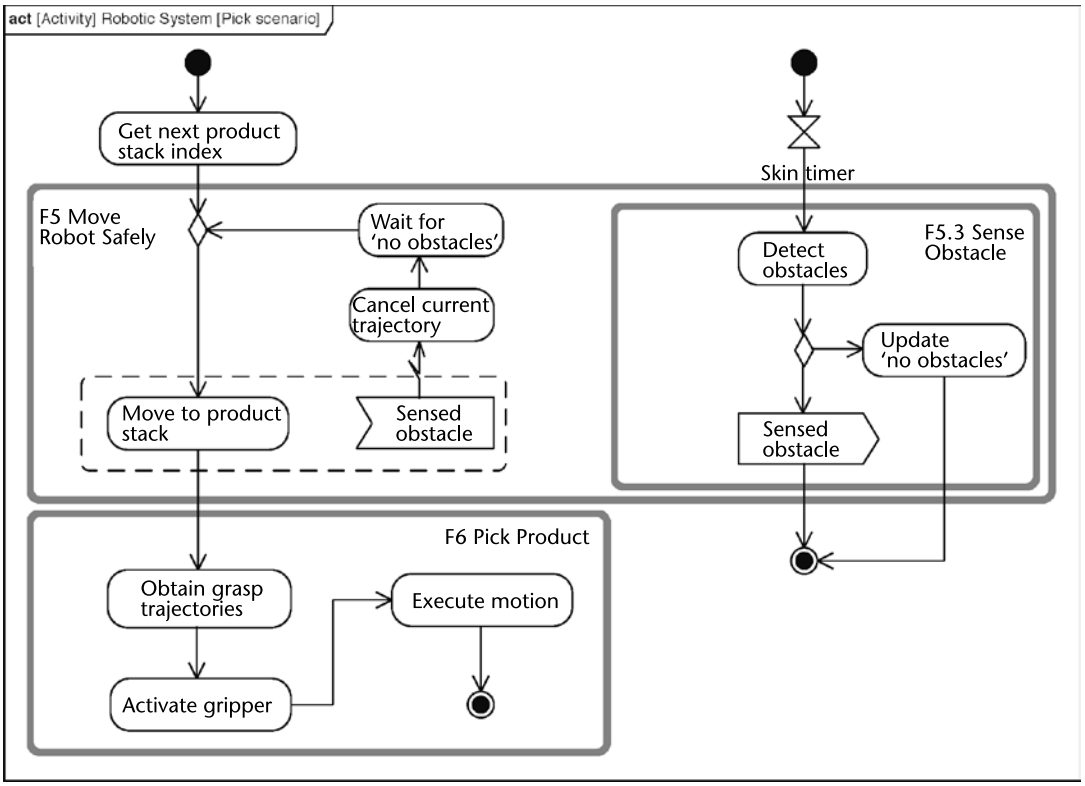


Figure 9.4 Preliminary activity diagram of the robot's desired behavior.

example, Table 9.6 shows an N^2 chart that complements the activity diagram in Figure 9.4 by representing the interfaces and exchanges involved in the functional flow of the scenario "S5 pick product". Note that the N^2 chart of the complete system should have more rows and columns to account for the other scenarios in the operation of the system.

Table 9.6 N^2 Chart of the Functional Interfaces for Scenario S5 Pick Product

Req. available product	Request move to stack Environment 3D model	Request pick	Gripper ON		
F4 Manage stacks of products		Product index			
	F5 Move robot safely	Gripper over product stack			
		Obtain grasp trajectories (F6.1)		Joint trajectory to approach, contact and retreat	
			Activate gripper (F6.3.1.1)	Gripper active	
				Execute trajectory (F6.2)	Product picked

In the N^2 chart it is easy to identify groups of activities with meaningful input and outputs. For example, the activities "obtain grasp trajectories," "activate gripper," and "execute trajectory" result in the meaningful output "product picked," so it is reasonable to group them into the high-level function "F6 pick product." Then, we label its subfunctions accordingly.

An analogous process is done to group the other activities in Figure 9.4 recursively into the high-level function "F5 move robot safely" and its children, such as "F5.3 sense obstacle."

The activity model resulting from this grouping process is shown in the activity diagram for the complete system in Figure 9.5 and associated subdiagrams for the functional flows of the main activities (high-level functions) in Figures 9.6, 9.7, and 9.8. Note that the main functional flow in Figure 9.5 includes most of the activities corresponding to the high-level functions. The function "F3 coordinate product operation" is responsible for coordinating the activation of each funtion, but this is not explicitly represented in activity diagrams (this is later refined in the detailed design, for example, using sequence diagrams). At this stage of the design, in the activity diagrams we have labeled each activity only up to the first sublevel of function that performs them. These activities need to be be further refined later in the lower levels of the functional hierarchy.

Finally, by labeling the consolidated grouping of activities and subactivities in the previous diagrams as functions, we obtain the complete functional hierarchy of the robotic system, represented in the block diagram of Figure 9.9.

Following the ISE&PPOOA process, in addition to the block definition diagram that represents the functional hierarchy, and the activity diagrams that model the functional flow, the functional architecture model is completed by describing

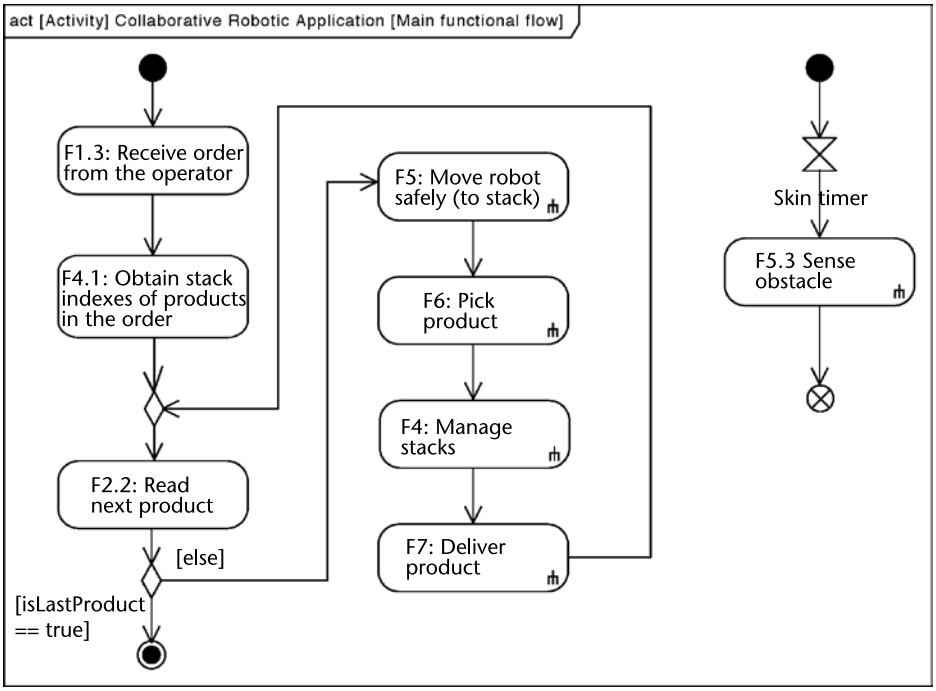


Figure 9.5 Main functional flow of the robotic system to process an order or products.

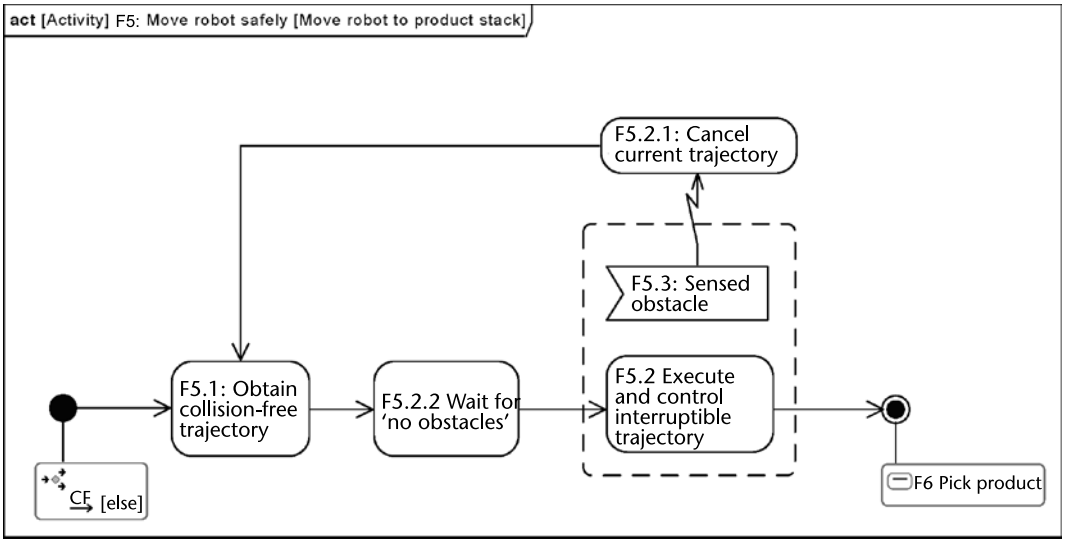


Figure 9.6 Activity diagram for F5 move robot safely.

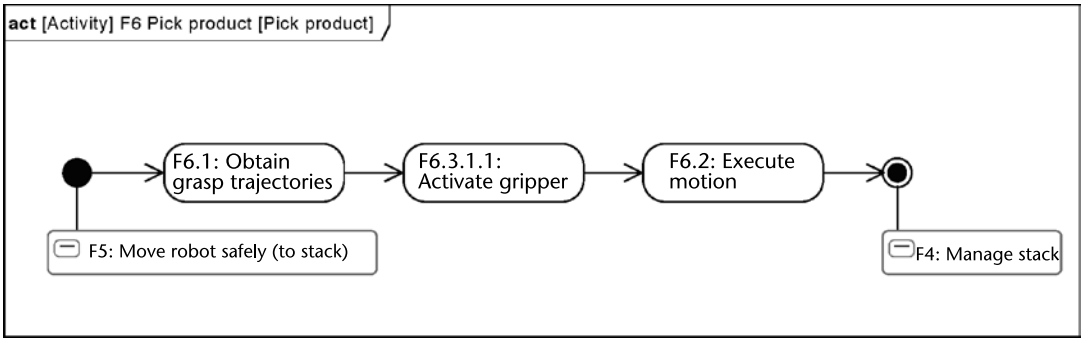


Figure 9.7 Activity diagram of F6 pick product.

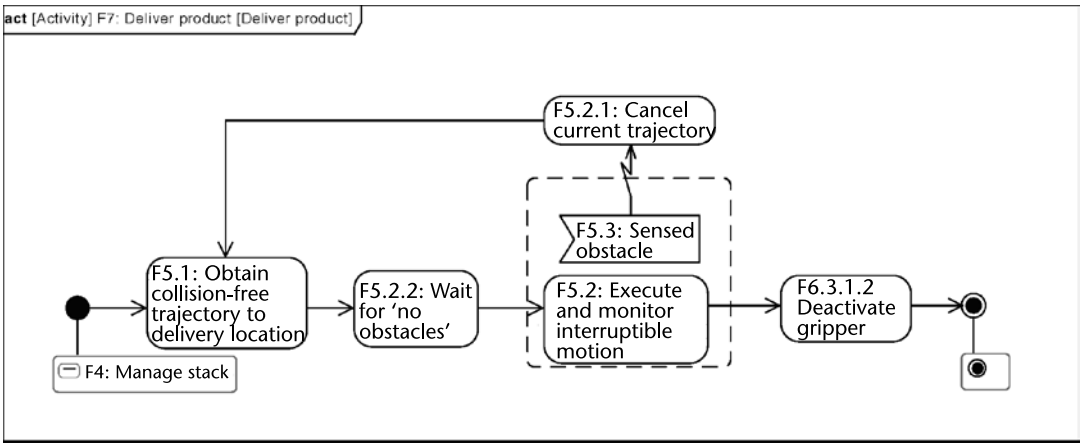


Figure 9.8 Activity diagram of F7 deliver product.

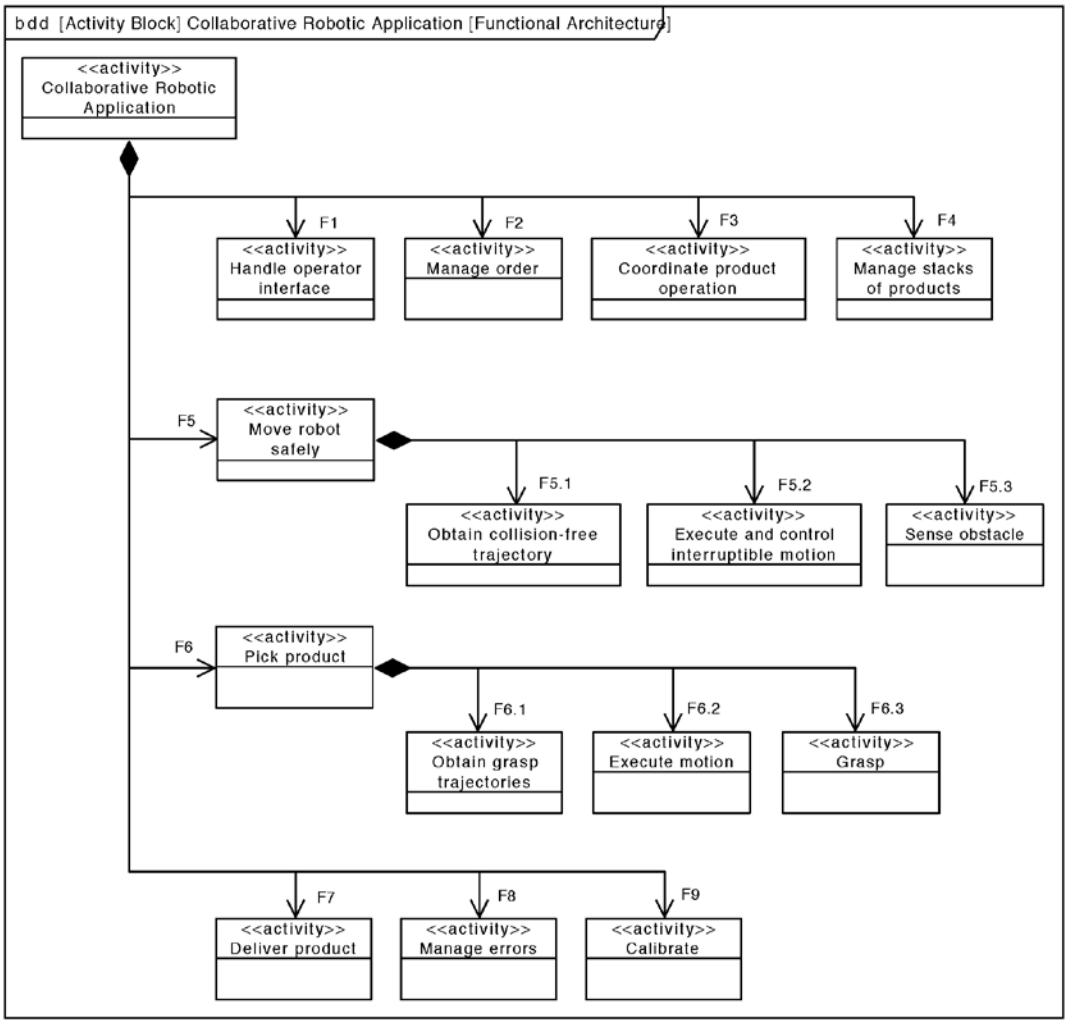


Figure 9.9 Functional hierarchy of the robotic system.

all the functions and their interfaces with the template suggested in Chapter 5. An example is shown in Table 9.7.

It is important to note that the process of obtaining the functional hierarchy is an iterative refinement that progresses in parallel to the refinement of the QA tree through NFRs and the physical architecture, and therefore it can be modified at later stages; for example, when we allocate the different functions to the physical subsystems in the robot (see later in Section 9.3). To ensure the completeness of the design of the robotic application along the ISE&PPOOA process, we have created a traceability matrix that allows us to verify whether all the capabilities identified for the system are addressed by functions and/or QAs (Table 9.8).

9.2.2 System Requirements

Once we have obtained the functional architecture of the robotic system, the next step is to obtain the associated requirements. To capture the requirements, we use

Table 9.7 Function Description for F6 Pick Product

Function	F6 Pick product
Description	Obtains a collision free joint trajectory to move the robot from its current configuration to the given input pose for its end-effector (the gripper).
Inputs	Pick request
	Stack geometry
	Product index
Outputs	Product picked
Parent Function	None
Children Functions	F6.1 Obtain grasp trajectories
	F6.2 Execute motion
	F6.3 Grasp

the templates suggested in Appendix B. Following is a nonexhaustive list of examples of the functional requirements associated to the main functions in the robotic system:

F3. Coordinate Product Operation

[FR_3_1] When < input "gripper over stack" >, the function <F3> shall <generate> <output "gripper ON">
Rationale: The gripper needs to be activated before moving to contact the target product to ensure a fast and successful grasp.

Function F5.1 Obtain Collision-Free Trajectory

[FR_5.1] When <input 'request plan' is received AND input 'index of current item' contains a valid index in the container AND 'environment 3D model' contains the URDF[3] of the robot work cell AND input 'current robot joint configuration' contains the actual values of the robot joints positions> the function <F5.1 Obtain collision-free trajectory> shall <generate output 'joint trajectory'>
Rationale: Function F5.1 provides trajectories to move the robot end-effector to the stack of the target product without colliding with any element in the work cell modeled in the environment 3-D model.

Note that some performance requirements can already be specified for a function if there is a clear mandatory requirement for the functional flow. This is the case of [FR_5.1_001]. However, in other cases, the concrete requirement can only be determined once the function is allocated to the physical element that performs it.

[FR_5.1_001] When <input 'request plan' is received AND input 'index of current item' contains a valid index in the container AND 'environment 3D model' contains the URDF of the robot work cell AND input 'current robot joint configuration' contains the actual values of the robot joints positions> the function <F5.1 Obtain collision-free trajectory> shall <generate output 'joint trajectory'><in less than 10ms>

3. The Unified Robot Description Format (URDF) is an XML format for representing a robot model.

Table 9.8 Traceability Matrix Functions and QAs (Rows) Capabilities

	C1.1 Process Order	*C1.2 Pick Product*	*C1.3 Handle Product*	*C1.4 Deliver Product*	*C1.5 Situational Awareness*	*C1.6 Manage Stacks*	*C2.1 Pick Product Safely*	*C2.2 Handle Product Safely*	*C2.3 Deliver Product Safely*	*C3 High Availability*	*C4 Resiliency to Contin-gencies*	*C5 Configure New Product*	*C6 Support Human Interaction*
F1 Handle Operator Interface	—	—	—	—	—	—	—	—	—	—	—	—	X
F2 Manage Order	X	—	—	—	—	—	—	—	—	—	—	—	—
F3 Coordinate Product Operation	X	—	—	—	—	—	—	—	—	—	—	—	—
F4. Manage Stacks of Products	—	—	—	—	—	X	—	—	—	—	—	—	—
F5.1 Obtain Collision-Free Trajectory	—	X	—	X	—	—	X	X	X	—	—	—	—
F5.2 Execute and Control Interruptible Motion	—	X	—	X	—	—	X	—	—	—	—	—	—
F5.3 Sense Obstacle	—	—	—	—	X	—	X	—	—	—	—	—	—
F6.1 Obtain Grasp Trajectories	—	X	—	—	—	—	X	—	—	—	—	—	—
F6.2 Execute Trajectory	—	X	—	X	—	—	X	—	—	—	—	—	—
F6.3 Grasp	—	X	X	—	—	—	X	X	X	—	—	—	—
F7 Deliver Product	—	—	—	X	—	—	—	—	X	—	—	—	—
F8 Manage Errors	—	—	—	—	—	—	—	—	—	X	—	—	—
F9 Calibrate	—	—	—	—	—	—	—	—	—	—	—	X	—
QA Harm Protection	—	—	—	—	—	—	X	X	X	—	—	—	—
QA Safety Incident Protection	—	—	—	—	X	—	X	X	X	—	—	—	—
QA Response time	X	X	X	X	X	X	X	X	X	—	—	—	—
QA Usability	—	—	—	—	—	—	—	—	—	—	—	—	X
QA Changeability	—	—	—	—	—	—	—	—	—	—	—	X	—
QA Resiliency	—	—	—	—	—	—	—	—	—	—	X	—	—
QA Reliable components	—	—	—	—	—	—	—	—	—	X	—	—	—

Rationale: Derived from PR_001 to complete the overall operation in no more than 2s.

Following we present other requirements derived from [FR_5.1]:

[FR_5.1_002] When <input 'request plan' is received AND input 'index of current item' contains a valid index in the container AND 'environment 3D model' contains the URDF of the robot work cell AND input 'current robot joint configuration' contains the actual values of the robot joints positions> the function <F5.1 Obtain collision-free trajectory> shall <generate output 'joint trajectory'> fulfilling [DD_R_040]
Rationale: The format of provided joint trajectory must comply with the data dictionary.

[FR_5.1_003] When <input 'request plan' is received AND input 'index of current item' contains a valid index in the container AND 'environment 3D model' contains the URDF of the robot work cell AND input 'current robot joint configuration' contains the actual values of the robot joints positions> the function <F5.1 Obtain collision-free trajectory> shall <generate output 'joint trajectory'> <without the need to include values for the 'time' field, see [DD_R_041]>
Rationale: The joint trajectory produced by FR5.1 does not need to include timing information.

[FR_5.1_004] When <input 'request plan' is received AND input 'index of current item' contains a valid index in the container AND 'environment 3D model' contains the URDF of the robot work cell AND input 'current robot joint configuration' contains the actual values of the robot joints positions> the function <F5.1 Obtain collision-free trajectory> shall <generate output 'joint trajectory'>< in which the first trajectory point shall correspond to the current joint values of the robot>
Rationale: The origin of the joint trajectory produced must be the current joint values of the robot so as to be executable.

[FR_5.1_005] When <input 'request plan' is received AND input 'index of current item' contains a valid index in the container AND 'environment 3D model' contains the URDF of the robot work cell AND input 'current robot joint configuration' contains the actual values of the robot joints positions> the function <F5.1 Obtain collision-free trajectory> shall <generate output 'joint trajectory'> fulfilling constraint [C_1]
Rationale: The trajectory will be feasible for the robot that is compliant with its kinematics and joint limits.

Following are examples of performance, constraints, and nonfunctional requirements:

[PR_001] Each <product processed for an order> shall have an end to end time of no more than <2s> from <the product request> to <delivery of the product in the bin>
Example of NFR related to response times.

Constraints

[C_1] (Applies to F5.1) Obtain collision-free trajectory> shall <generate output 'joint trajectory'> that is an executable trajectory by the robot manipulator controller, being compliant with the robot kinematics and joint limits.

[C_2] (Applies to F5.1) Obtain collision-free trajectory> shall <generate output 'joint trajectory'> that avoids collisions with static obstacles in the work cell. Example related to safety, harm protection.

Nonfunctional Requirements

[NFR_Av_001] The system shall normally be available <during all working hours in the factory (16h a day)>, except in the exceptional circumstances of a frequency and duration not to exceed <4h each 28 day> Example of NFR related to availability.

[NFR_Maint_001] Maintainer shall be able to add a new product type, including modifications and testing, with no more than 8 person-hour of effort. Example of NFR related to changeability.

[NFR_Saf_001] Occurrences of any safety incident shall be reported. Example of NFR related to safety, incident reporting.

[NFR_FR3.2_001] "Execute interruptible trajectory" activation shall be allowed only when "F3.3 Sense obstacle" is operational. Example of NFR related to safety, hazard protection.

Data Dictionary

For the data exchanged in our collaborative robotic application, we have followed the data types for the control of robot manipulators defined by Robot Operating System (ROS) [5]. The following requirements define a simplified data dictionary for our robotic example, according the notation recommended in Appendix B.

[DD_R_040]: Contents of a JointTrajectory message JointTrajectory = header + {joint_name}N + {JointTrajectoryPoint}N

N is the number of degrees of freedom of the robot arm selected for the application, in this case the Universal Robot arm UR5 has N = 6 degrees of freedom.

[DD_R_041]: Contents of a JointTrajectoryPoint message JointTrajectoryPoint = {position}N + {velocity}N + (acceleration)N + (effort)N + time

9.3 Physical Architecture and Heuristics Applied

Once we have the functional architecture of the robotic system, the next step in the ISE&PPOOA process is to obtain the physical architecture of the system. This is done following the three-stage process described in Chapter 4: first we obtain the modular architecture by allocating the functions to the building elements of our robotic system, then we refine the architecture by using heuristics to address the NFRs, and finally the refined architecture is represented.

9.3.1 Modular Architecture

The first step to create the physical architecture is to identify the building elements (parts), which in our robotic example also include elements off-the-shelf. These building elements (Figures 9.10 and 9.11) are

- An off-the-shelf industrial robot manipulator, including its controller unit, with the characteristics to be determined by the QAs obtained along the refinement of the system's architecture.
- A suction gripper installed in the robot end-effector to handle the products.
- A sensing robot skin covering the robot manipulator. This device was developed by the Institute of Cognitive Systems at the Technical University of Munich [3], and was deployed in multiples prototypes of collaborative robotic applications during the European project Factory-in-a-day, with the participation of the second author of the book [4]. This robot skin allows detecting dynamic obstacles and can be adapted for different robot manipulators.

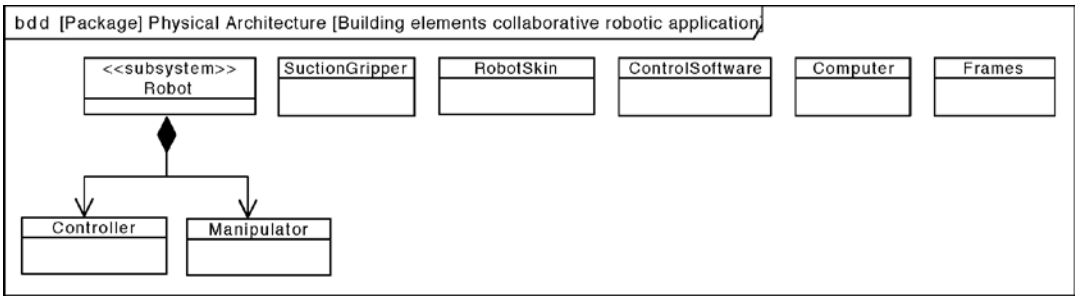


Figure 9.10 Building elements for the collaborative robot application.

Figure 9.11 Photograph of the collaborative robotic application.

- The computer than runs the control software for the application, including a motion planning library that provides routines to dynamically obtain motion plans for robot manipulators [5]. The library is based on the MoveIt! framework [8] and can integrate static 3-D information of the work cell to plan motion trajectories that avoids collisions with the environment, and also dynamic information about obstacles from sensors such as the robot skin.
- The frames structure for the work cell where the robot, computer, and stacks of products are mounted.

Heuristics of modularity/structuring are applied to obtain the modular architecture of the collaborative robot.

Some decisions to structure the modular architecture are given by the reuse of off-the-shelf robotic components that are already designed considering structuring heuristics. The following are some examples:

- *SA_Heu-6 Group functions that are strongly related to each other, separate functions that are unrelated*. Industrial robot manipulators are provided by the manufacturer together with their controller units as an integrated subsystem, providing all the functions required for executing joint trajectories and other functionalities.
- *SA_Heu-7. Promote subsystems implementation independency.*We consider the robot skin as a separate element from the manipulator. Even if physically mounted on it, almost as a part of the manipulator, the function it addresses, together with the independent/separate information and power connections, suggests considering it as a separate subsystem. To make its design reusable with different robot manipulators, the skin was developed manipulator-agnostic, offering a mechanical attachment to adapt to any surface and a generic control interface through a Gigabit Ethernet connection.
- *SA_Heu-8. Choose a configuration with minimal communications between subsystems*. Most manipulators offer the possibility to connect an end-effector, such as a gripper, both physically and logically, by providing the necessary interface to control the end-effector through the robot controller. Applying this heuristic, the suction gripper is designed to connect to the manipulator and be part of the robot subsystem (see Figure 9.12), resulting in the following interface specifications for the components:
  - [I_1]: The gripper will have a mechanical interface compliant with the one of the universal robot UR5 for end effectors, so that it can be bolted and fixed to the tooltip of the robot.
  - [I_2]: The gripper will provide an electrical interface compliant with the electrical interface for end effectors offered by the universal robot UR5.

In addition to the previous heuristics, the user interface required to address "F1 handle operator interface" is included in the control software subsystem, since both the user interface and the rest of the application software run in the same computer, so as not to have too many subsystems.

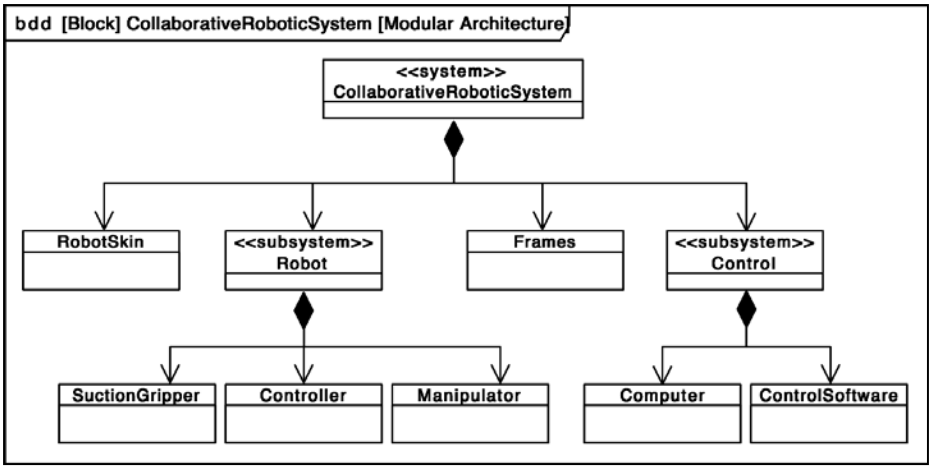


Figure 9.12 Modular architecture of the robotic system.

The result of the previous heuristics and considerations is the high-level modular architecture for the robotic application shown in Figure 9.12.

The structuring of the physical architecture proceeds iteratively with the allocation of the system functions to the different elements and subsystems. An initial allocation is shown in Table 9.9.

The main allocation criterion is to select the functional level to allocate (row) and the physical architecture level (column) so that the function is allocated to a single element. It can be observed in Table 9.9 that this is not the case, as it includes

Table 9.9 Allocation of Functions to the Physical Elements

Function	*RobotSkin*	*Suction-Gripper*	*Robot*	*Control*	*Frames*
F1 Handle Operator Interface	X	—	—	X	—
F2 Manage Order	—	—	—	X	—
F3 Coordinate Product Operation	—	—	—	X	—
F4 Manage Stacks of Products	—	—	—	X	—
F5.1 Obtain Collision-Free Trajectory	—	—	—	X	—
F5.2 Execute and Monitor Interruptible Trajectory	—	—	X	X	—
F5.3.1 Sense Distance to Robot Arm	X	—	—	—	—
F5.3.2 Detect Obstacle	—	—	—	X	—
F6.1 Obtain Grasp Trajectories	—	—	—	X	—
F6.2 Execute Uninterruptible Trajectory	—	—	X	X	—
F6.3 Grasp	—	X	X	X	—
F7 Deliver Product	—	—	—	X	—
F8 Manage Errors	—	—	—	X	—
F9 Calibrate	—	—	—	X	X

subfunctions of F5 and F6 next to high-level functions. This is an indication that further breakdowns are required. This breakdown process also refines the functional architecture in order to accommodate for the allocation to the elements in the physical architecture. The following rule from Chapter 7 can be used for this process of iterative breakdown and allocation:

> If the parent function is allocated to a physical subsystem or element, all children subfunctions must be allocated to the same physical subsystem or element.

As an example, see the breakdown of the function "F6.3 grasp" shown in Figure 9.13. Similar breakdowns need to be performed to other rows and columns of the allocation matrix in Table 9.9, until a one-to-one allocation is achieved.

9.3.2 Application of Heuristics to Refine the Physical Architecture

The next step is to refine the architecture to address the NFRs for our robotic system, applying the design heuristics that address the corresponding QAs. Note that because of the software-intensive nature of this application, more heuristics are applied later in the design of the control software subsystem following PPOOA. Following the requirements breakdown process described in Appendix B, we take the main components identified in the modular architecture, and convert the QAs of their allocated functions into component specifications for them, by making use of the mentioned heuristics.

9.3.2.1 Safety Heuristics

Let us consider the capability "C2 harmless." In industrial collaborative robotic applications, the heuristics addressing the quality attribute requirements related to are given by ISO/TS 15066:2016 [2]. Following those guidelines, design decisions are made resulting in component specifications (constraints) for the manipulator and the gripper (see Figure 9.13):

> [C_3] The manipulator shall be a UR5 universal robot collaborative manipulator. Rationale: Collaborative manipulators, such as the UR5, include in their safety design a torque-triggered safety stop to reduce the harm upon collision.

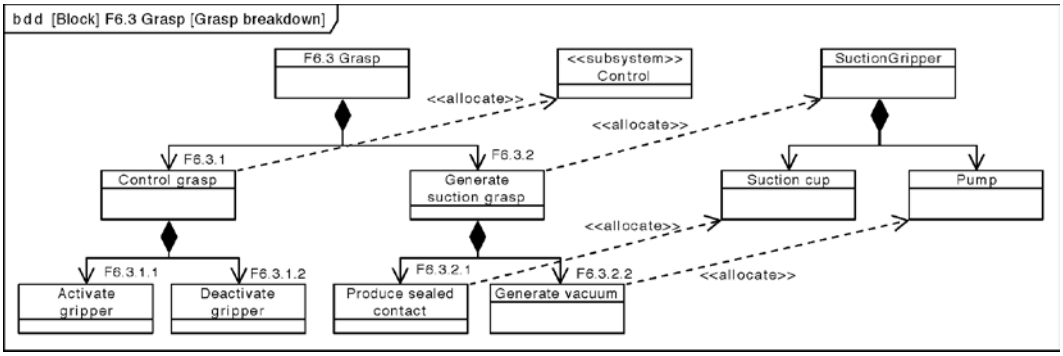


Figure 9.13 Functional breakdown of the function F6.3 grasp.

However, collisions and their associated emergency stop has a negative effect on the performance of the system, reducing the up time (capability C3 High availability) and decreasing the performance of the system [requirement PR_001]. This is the reason for the chosen functionality to detect obstacles (F5.3) and move the robot through the execution of trajectories that can be interrupted (F5.2) when an obstacle is detected.

[P_6][4] The gripper shall be lightweight (less than 0.3 kg)

[P_7] The gripper design shall have rounded edges with a minimum radius of 3 mm.

Rationale: Avoid sharp edges that can harm the human operator.

9.3.2.2 Resilience Heuristics

Heuristics are also applied to address the QA "resiliency to contingencies" identified for the robotic system.

- *RS_Heu_5. Human backup*.This heuristic is used to address the QA intrinsic to the collaborative approach in this robotic application. The collaborative solution including the operator in the solution ensures that a human can intervene in the workspace of the robot and correct minor failures, such us the robot dropping a product.
- *RS_Heu_7. Human in the loop*. In the design of the solution presented here, the human intervention has been considered not only as backup in anomalous operation, but it has actively considered the human in the loop principle. A visual signaling system has been included, using robot motions together with the LEDs of the robot skin, so that the robot notifies the operator the need to refill a product stack when it has been depleted. Figure 9.14 shows the activity diagram corresponding to this flow and the allocation of the different function to the elements of the physical architecture.

As a result of the application of these heuristics, the operator is included as an element of the robotic application (see Figure 9.9), and the function F4.2 replace stack, child of F4 manage stacks, is allocated to her/him, while the other child function F4.1 update stack (inventory) is allocated to the control system (see Figure 9.14).

9.3.3 Representation of the Refined Physical Architecture

After refining the physical architecture of our robotic system, we note that it has not varied much in its structure, but for the important addition of the operator and the component specifications. Figure 9.15 shows the refined physical hierarchy, including as an example the component specifications discussed in the previous section. However, the refinements have a great impact on the behavior of our solution,

4. [P_n] indicates that these requirements correspond to a physical property specified for the component.

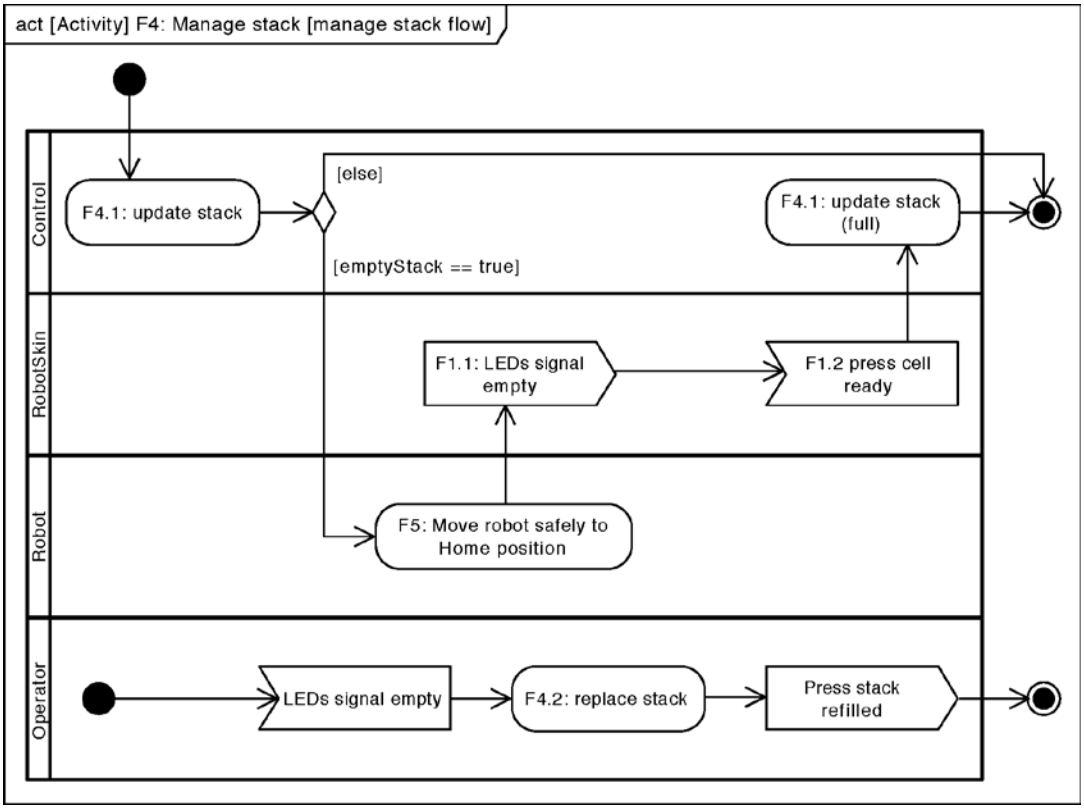


Figure 9.14 Activity diagram modeling F4 manage stacks flow.

modeled through activity diagrams that complement the refined physical architecture deliverable. As an example, Figure 9.16 shows the part of the functional flow in the system corresponding to picking the first product in an order (the actual complete flow will include similar allocations for the "manage stack" and "delivery product" functional subflows).

In addition to the physical hierarchy and the activity diagrams, the final physical architecture deliverable for the robotic system includes internal block diagrams representing the system physical blocks and their logical and physical connectors, accompanied by a textual description of the blocks. Figures 9.17 and 9.18 present a simplified view (for the sake of clarity and brevity) of the physical blocks in the robotic system and their connectors.

9.3.3.1 Physical Parts Description

The previous internal block diagrams are complemented with tables describing each part in the physical architecture. Below we present the description of the robot skin and the control software elements in Figures 9.17 and 9.18 as an example (also see Tables 9.10 and 9.11).

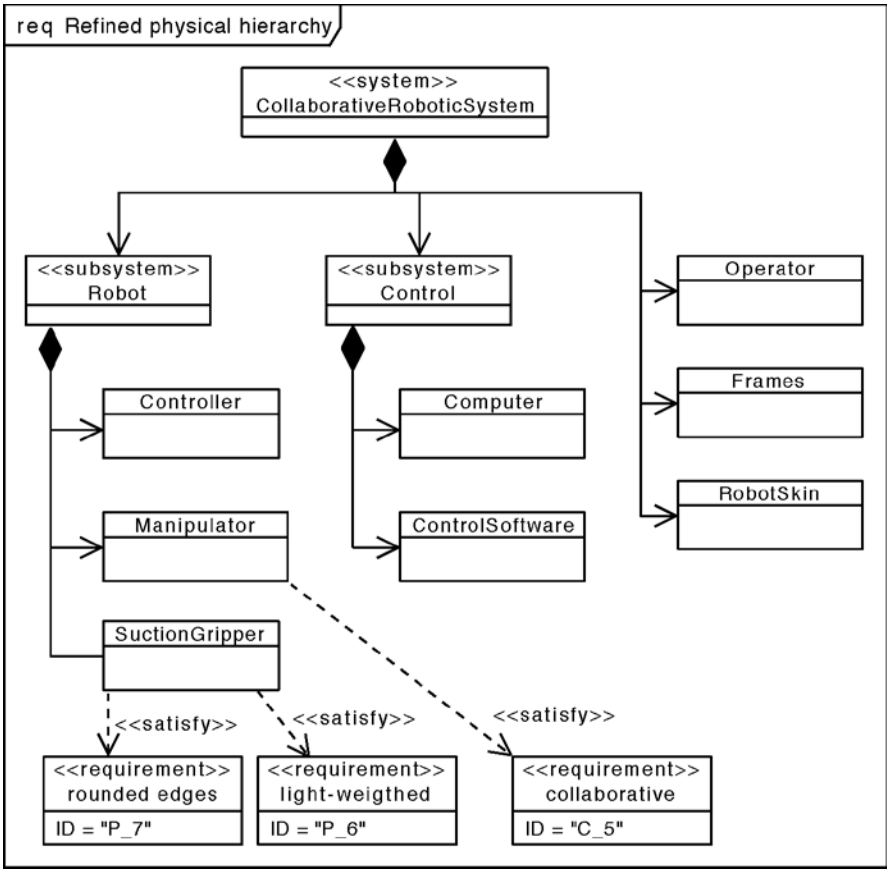


Figure 9.15 Refined physical hierarchy of the collaborative robotic system represented in a requirements diagram.

9.4 Software Architecture

As we have mentioned, the collaborative robotic system is software-intensive, so we will focus now on the systems engineering subprocess to obtain the architecture for the control software subsystem. For this, the PPOOA subprocess discussed in Section 4.4 is used.

The first and most crucial step is to obtain the domain model for the robot control subsystem. Remember that the domain model is a representation of concepts that are important in our control software subsystem and their relations. The domain model involves analysis or domain classes, to which we assign responsibilities.

One approach is to bridge the function hierarchy and domain model through responsibilities:

> Responsibilities are functions from the functional hierarchy.

This approach would proceed as follows:

1. Identify in the functional hierarchy which functions are allocated to software;

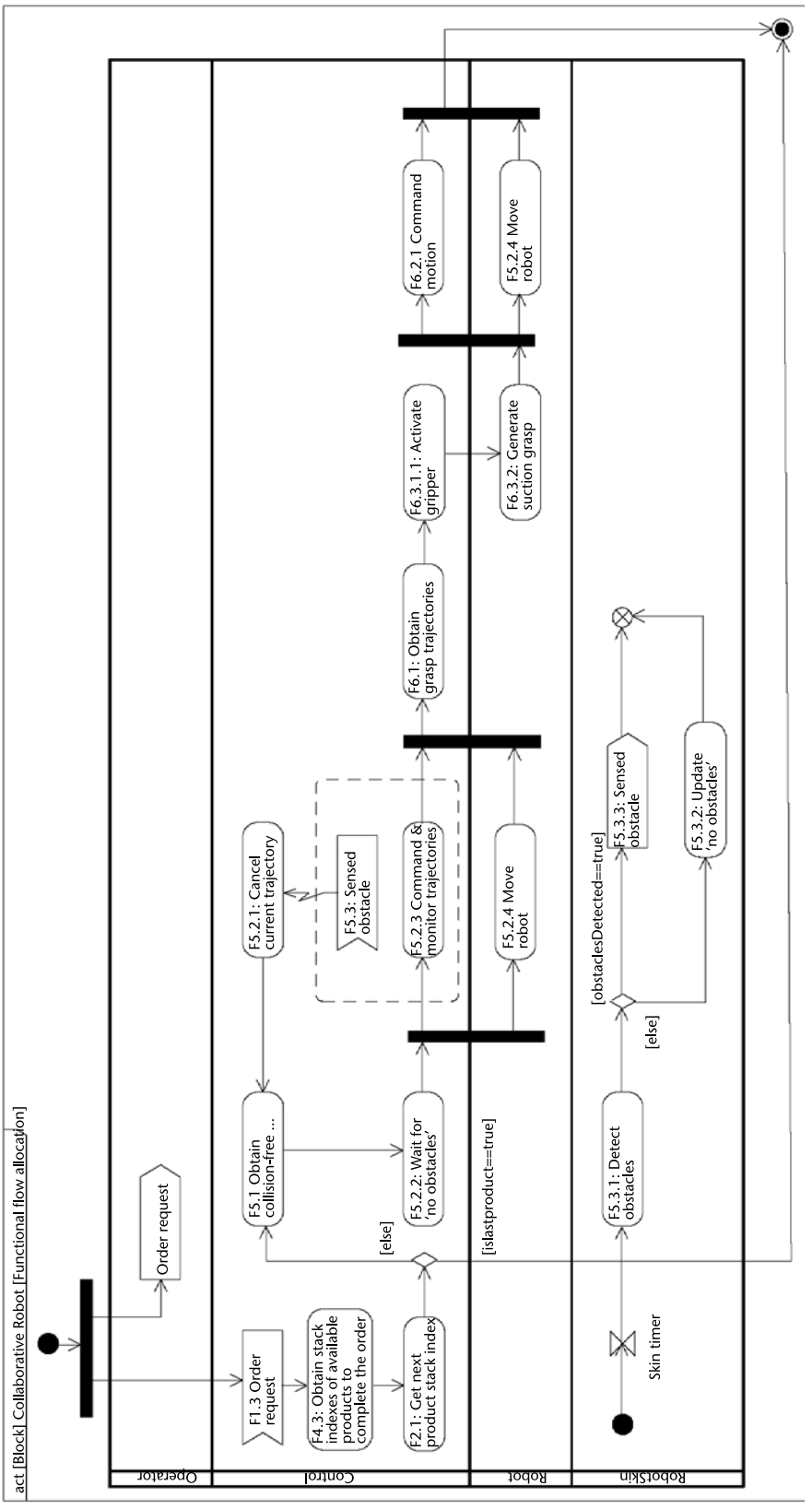


Figure 9.16 Activity diagram modeling the functional flow allocation for picking the first object in an order.

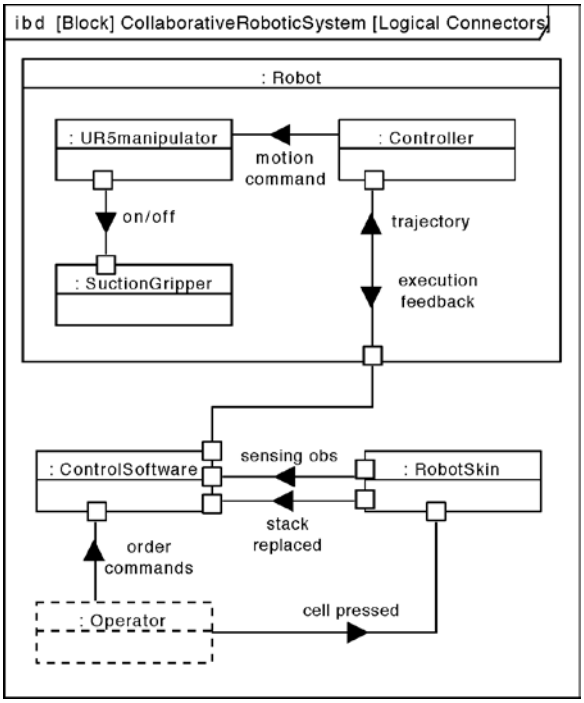


Figure 9.17 Internal block diagram representing the logical connections between the main parts of the robotic system.

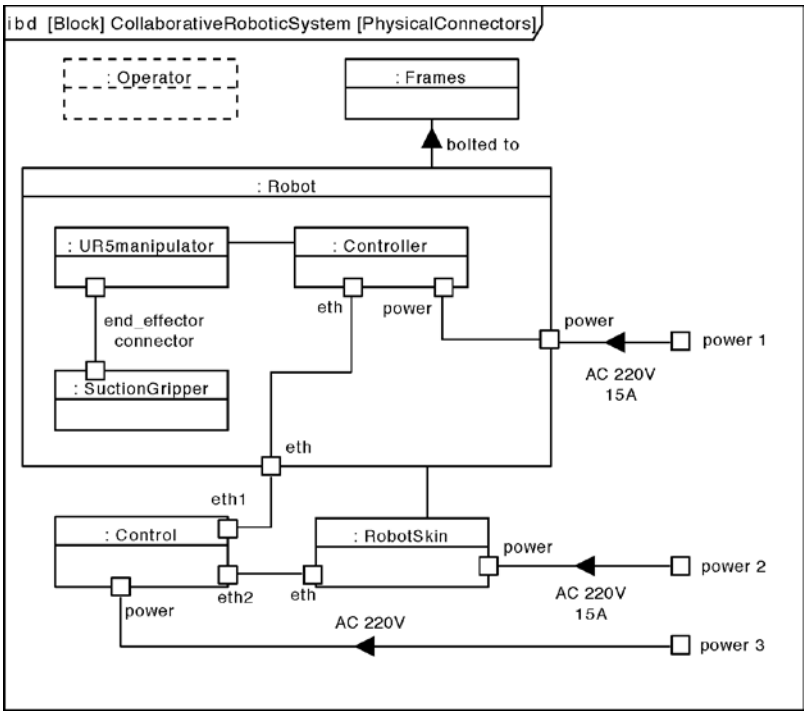


Figure 9.18 Internal block diagram representing the physical connections between the main parts of the robotic system.

Table 9.10 Robot Skin Element Description

: Robot Skin
Inflow: cell pressed (input from the operator to signal product stack refilled)
Outflow: sensing objects, stack replaced
Allocated functions: • F1 Handle operator interface • F5.3.1 Sense obstacle

Table 9.11 Control Software Element Description

: Control Software
Inflow: sensing objects, execution feedback, order commands
Outflow: trajectory
Allocated functions: • F1 Handle operator interface • F2 Manage order • F3 Coordinate product operation • F4 Manage stacks of products • F5.1 Obtain collision-free trajectory • F5.2 Execute and monitor interruptible trajectory • F5.3.2 Detect obstacle • F6.1 Obtain grasp trajectories • F6.2 Execute uninterruptible trajectory • F6.3 Grasp • F7 Deliver product • F8 Manage errors

2. With those functions, identify substantives in their description to obtain the classes in the domain model as an important note:

Work with substantives (things) not with verbs (transformations).

In Table 9.12 we list the functions allocated to software (the control subsystem), and highlight the substantives.

Following this approach, the domain model represented in a block definition diagram in Figure 9.19 has been obtained.

There are two important remarks:

1. An early design decision in the physical architecture was not to include a vision system for the robot to automatically locate the stacks and the products to grasp. Therefore, the control subsystem needs to receive the offline information about the position and geometry of the stacks through the user interface. This information is obtained by the operator during F9 calibrate.
2. Applying the SF_Heu_3 avoid nondeterministic behavior yields the need to avoid online planning of robot motions, which would result in

Table 9.12 Identification of Substantives in the Functions Allocated to the Software of the Robotic System

F1	Handle *operator interface*
F2	Manage *order*
F3	Coordinate *product* operation
F4	Manage *stacks* of products
F5.1	Obtain *collision-free trajectory*
F5.2	Execute and control *interruptible trajectory*
F5.3.2	Detect *obstacle*
F6.1	Obtain *grasp trajectories*
F6.2	Execute *uninterruptible trajectory*
F6.3.1	Control *grasp*
F7	Deliver *product*
F8	Manage *errors*

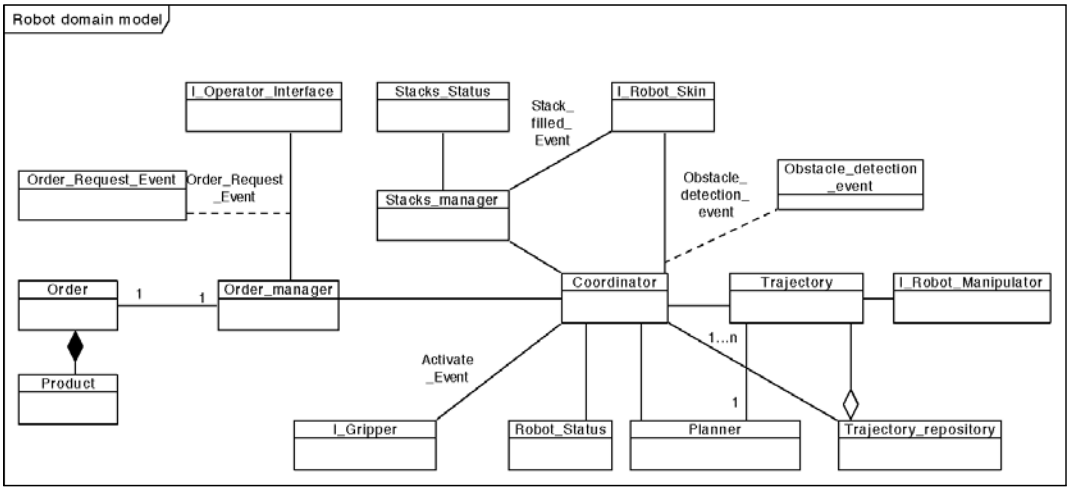


Figure 9.19 Domain model of the control software for the collaborative robot application.

nondeterministic trajectories. For this, a solution based on a database with trajectories obtained offline is used to generate the motions in F5.1 and F6.1.

In PPOOA, CRC cards are used in addition to the block definition diagram to define the responsibilities for each class of the domain model (see Table 4.3 in Chapter 4). Table 9.13 shows examples of the CRC cards for the classes identified in the domain model of the collaborative robot application.

9.4.1 Software Components

Once we have obtained the domain model (see Figure 9.19), we are in a position to start identifying the components that implement each one of the classes in the domain model of the robot. We use the PPOOA architecture diagram in Figure 9.20 to model the software components identified and their interactions: synchronous,

Table 9.13 Examples of CRC Cards for Some of the Classes in the Domain Model for the Robotic Application

Class Name	I_Operator_Interface
Responsibilities	F1 handle operator interface Allows the operator to request the next order. An order consists of a list of types of products and quantity of each type.
Collaboration	Order_Manager, Order_Request_Event
Class Name	Order_Manager
Responsibilities	F2 manage order
Collaboration	Coordinator, I_Operator_Interface, Order_Request_Event, Order
Class Name	Stack_Manager
Responsibilities	F4 manage stack of products
Collaboration	Order, Order_Request_Event, Coordinator
Class Name	Coordinator
Responsibilities	F3 coordinate product operation This component coordinates the operation of the system for the processing of orders through the execution of the pick and place operations, including the control of the motion executions. • F5.1 Obtain collision-free trajectory • F6.2.2 Control interruptible trajectory • F5: Retrieve product • F6.3.1 Control grasp • F6.2.1 Control uninterruptible trajectory
Collaboration	• Order_Manager • Stacks_Manager • I_Robot_Skin • Robot_Status • I_Gripper • Planner
Class Name	I_Robot_Skin
Responsibilities	This class handles the communication with the skin hardware configuration of the skin (thresholds for distance sensors, LED feedback). • F5.3.2 Detect Obstacle • F1.2 press cell “ready”
Collaboration	• Coordinator • Stack Manager
Class Name	I_Robot_Manipulator
Responsibilities	This class handles the communication with the robot manipulator to execute trajectories. • F5.2.2 Execute interruptible trajectory • F6.2.2 Execute interruptible trajectory
Collaboration	• Trajectory

indicated as dashed arrows, or asynchronous, indicated by coordination mechanisms in between the interacting components.

9.4.2 Casual Flows of Activities

The structural view of the software architecture in Figure 9.20 needs to be complemented with a behavioral view. Following PPOOA, the CFAs in the robotic system are modeled with activity diagrams that include the allocation of the activities to the software components, as shown in Figures 9.21, 9.22, and 9.23. In these diagrams, the activities allocated to control in Figure 9.16 are further decomposed and allocated to the different software components. Note that in Figure 9.23 the coordinator component is responsible for coordinating the entire control flow (transitions between the different activities), but the low-level actions corresponding to that coordination have been omitted at this level of modeling for the sake of clarity.

9.4.3 Safety Heuristics

Next we discuss the application of safety heuristics that justify some of the design decisions made for the software architecture, as modeled in the PPOOA architecture diagram (Figure 9.20) and the CFAs (Figures 9.21, 9.22, and 9.23).

SF_Heu_2 Minimize the number of components and interactions.

This heuristic has been applied in the iteration to refine the software components resulting in the robot software PPOOA architecture in Figure 9.20, and concretely the "Coordinator" component. Initially, the state machine governing the sequence of operations to process an order was divided into two state machines, one responsible

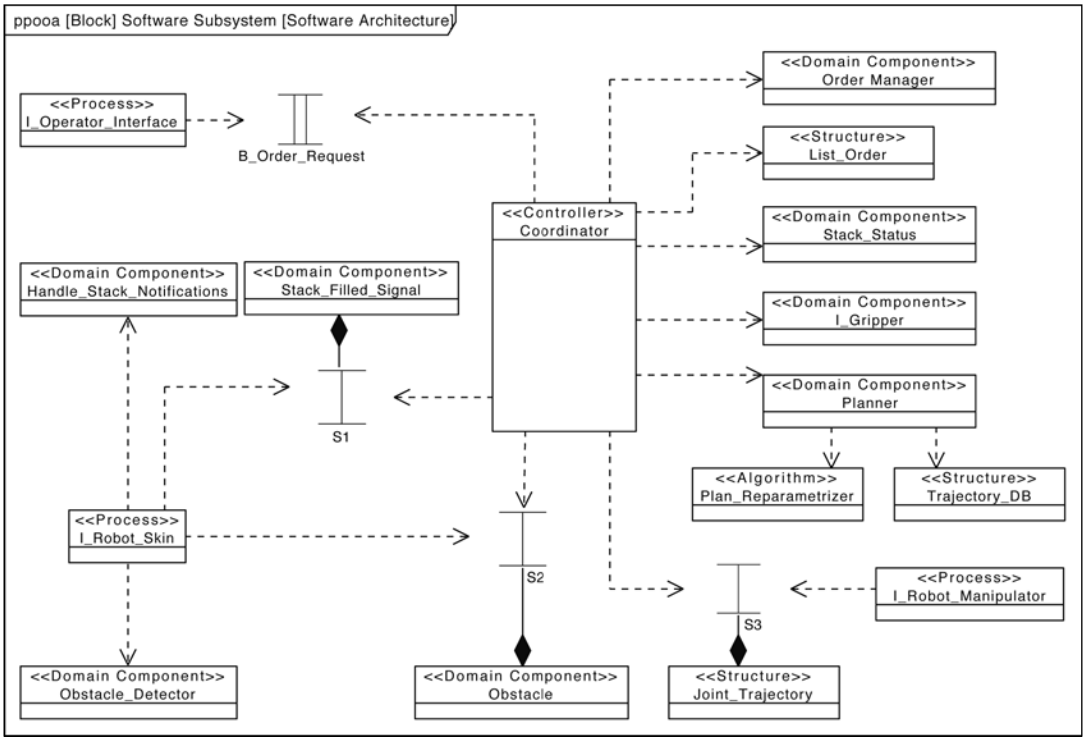


Figure 9.20 Structural view of the software architecture, including the main software components and the coordination mechanisms.

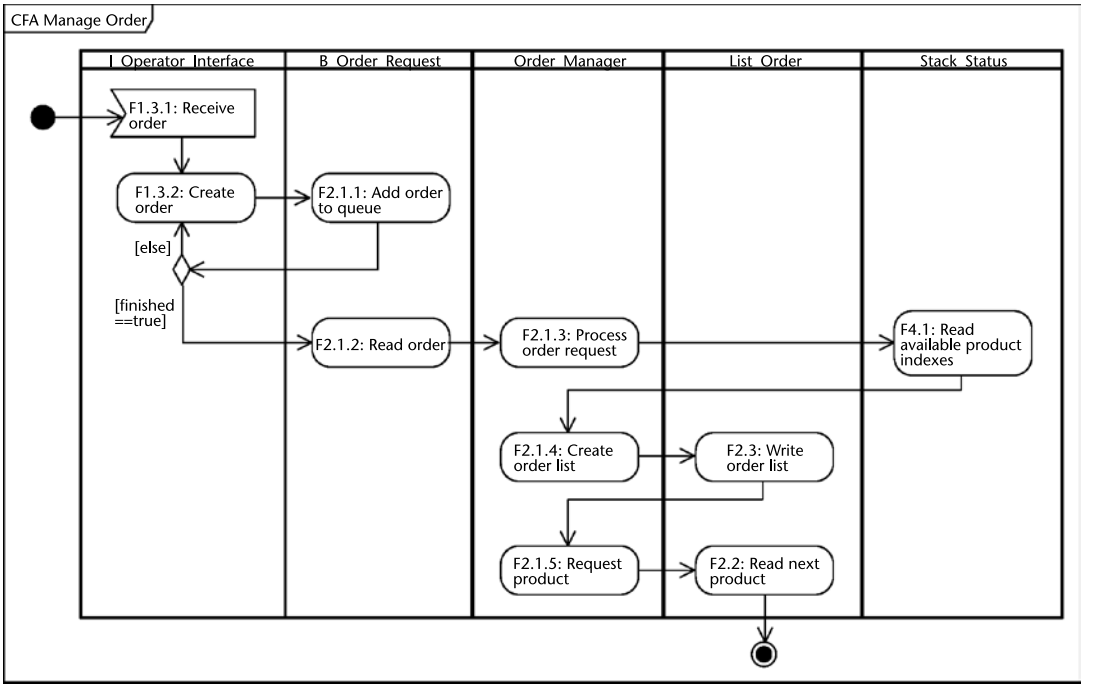


Figure 9.21 Activity diagram of the CFA manage order.

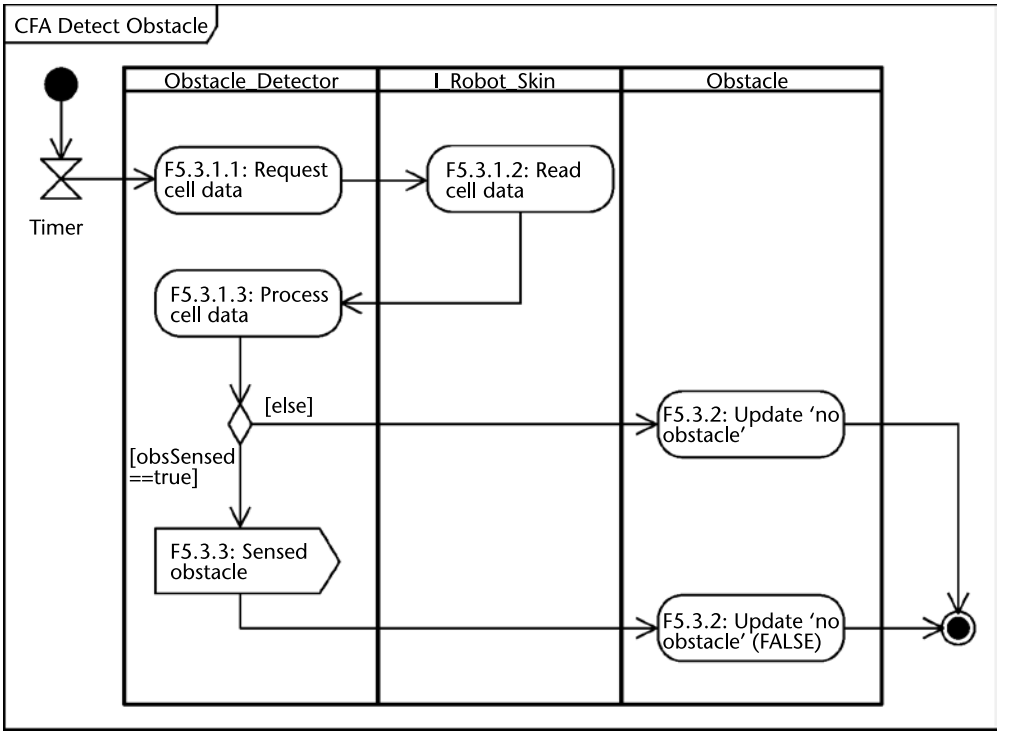


Figure 9.22 Activity diagram of the CFA obstacle detected.

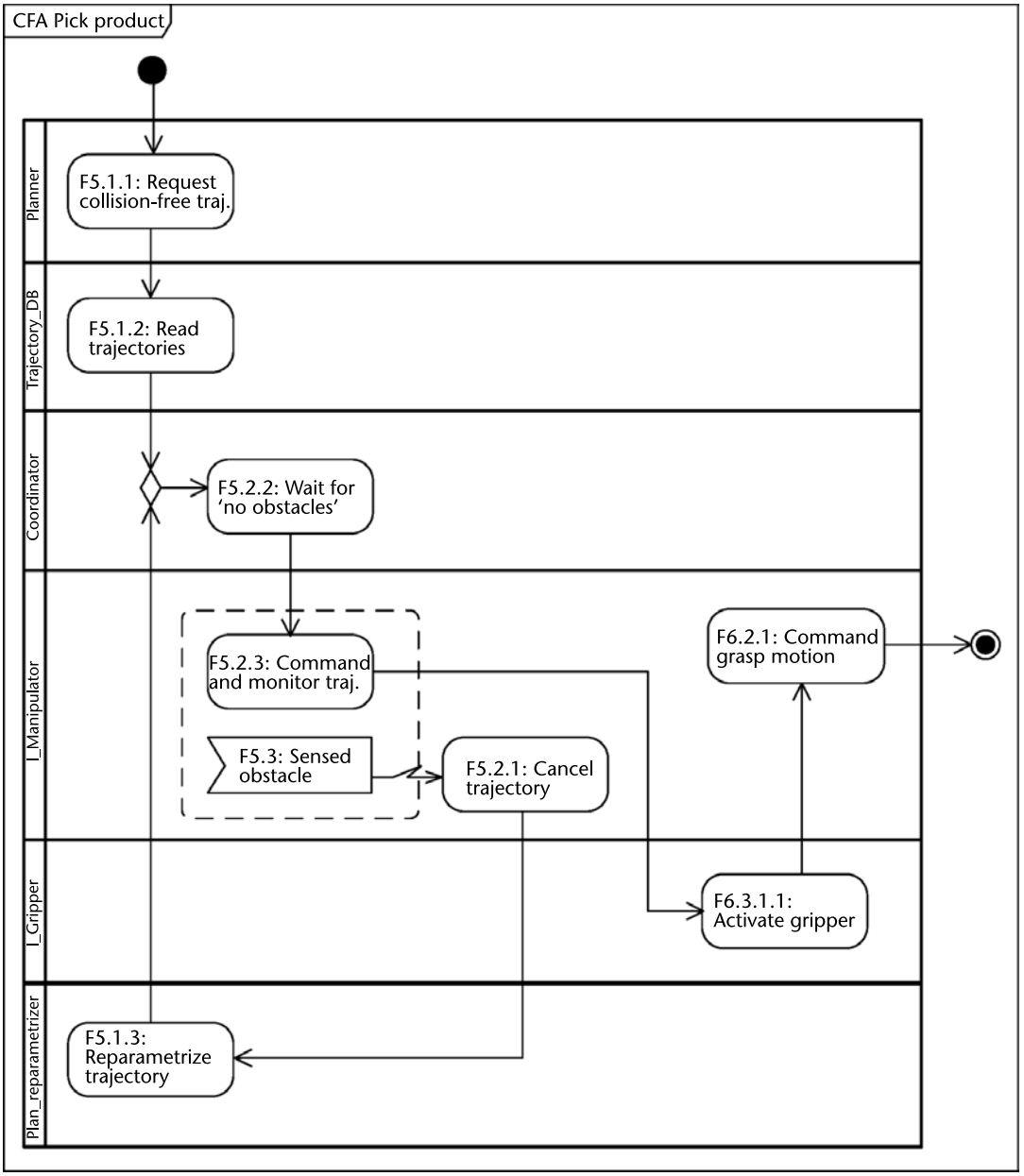


Figure 9.23 Activity diagram for the CFA pick product.

to process the order as a list of products to retrieve, and another one responsible to coordinate the sequence of operations to retrieve each product. During the modeling of the behavior in the CFA to process an order (Figure 9.23), it was clear that both state machines could be merged, eliminating one component.

We could have done otherwise and put a state machine responsible for the functions F5.2 and F6.2, which result in the execution of robot motions, in a separate process component, since that state machine could be reused in a different application in which the handling of the products would be done differently. Sepa-

rating the process controlling the robot motions is a good practice to isolate causes of harmful behavior.

SF_Heu_3 Avoid nondeterministic behavior.

This heuristic has been considered regarding the following design decisions:

- Functional design: Avoid online planning of robot motions that would result in nondeterministic trajectories. For this, a solution based on a database with trajectories obtained offline is used to generate the motions to retrieve products (component Trajectory_DB).
- Interruptible regions only during motion execution in the shared workspace (Figure 9.23), and not during grasping motion.
- Semaphores used for the coordination mechanisms corresponding to the asynchronous interactions involving the protected resources *order, obstacle, trajectory,* and *stack_filled_signal.*

SF_Heu_5 Sanity Check

Heuristic applied to execute the trajectories by checking that the current pose of the robot corresponds to the initial pose of the trajectory (applies to the coordinator component, who verifies this before commanding the execution to I_Manipulator).

SF_Heu_11__Implement Alerting Capabilities

This heuristic was used to extend the software architecture with a domain component that turns the skin LEDs on a red blinking mode to alert the operator of a failure detected, by using the I_Robot_Skin (not shown in the version of the architecture presented so far). This component is commanded by the coordinator.

Additionally, the ROS rosout system[5] shall be used to implement log information for the system developers and maintainers to diagnose and repair the control software subsystem this results in a new system requirement.

SF_Heu_11__Implement Operational Data Recording Functions

An additional requirement could state that the log data referred in the previous requirement will be recorded and stored for at least 2 years using rosbag.[6]

9.5 Summary

In this chapter, we discussed application of the ISE&PPOOA to the design of the functional, physical, and software architectures of a collaborative robotic application. The example showed how the ISE&PPOOA process can support the design of

5. Rosout is the name of the console log reporting mechanism in ROS. Its log messages are human-readable string messages that convey the status of a node [9].
6. Rosbag is a set of tools for recording and playing back ROS messages [10].

complex robotic applications, and especially how its unique use of heuristics helps addressing critical quality attributes in these systems, such as safety.

The chapter followed the step-wise application of ISE&PPOOA presented in Chapter 4. First, the robot operational scenarios and needs were identified and the robot capabilities and most relevant quality attributes specified. The functional architecture was then obtained following the ISE&PPOOA hybrid approach, top-down from the system's capabilities, and bottom-up by grouping the basic actions in the system with the help of an N^2 chart. Modularity heuristics were subsequently applied to obtain the physical architecture through the iterative allocation of the system functions to the robot building blocks, and the refined architecture was represented through SysML block definition diagrams, internal block diagrams, and activity diagrams. Finally, the software architecture of the system was obtained using PPOOA to bridge the software components to the functional architecture and safety and other heuristics from Chapter 6 were applied to refine the solution.

References

[1] Factory-in-a-day European Project, http://www.factory-in-a-day.eu.

[2] ISO/TS, 15066:2016, "Robots and Robotic Devices–Collaborative Robots."

[3] Mittendorfer, P., E. Yoshida, and G. Cheng, "Realizing Whole-Body Tactile Interactions with a Self-Organizing, Multi-Modal Artificial Skin on a Humanoid Robot," *Advanced Robotics*, Vol. 29, No. 1, 2015, pp. 51–67.

[4] Factory-in-a-day: final project video https://youtu.be/DU-y0KH41HI.

[5] Bharatheesha, M., C. Hernandez, M. Wisse, N. Giftsun, and G. Dumonteil, "Dynamic Obstacle Avoidance for Collaborative Robot Applications," in Workshop on IC3 - Industry of the Future: Collaborative, Connected, Cognitive, Novel Approaches Stemming from Factory of the Future and Industry 4.0 initiatives, *IEEE International Conference on Robotics and Automation (ICRA)*, Singapore, May 2017, 2017.

[5] Robot Trajectory Messages in ROS, http://wiki.ros.org/trajectory_msgs.

[6] ISO/IEC 9126-1:2001, Software Engineering–Product Quality–Part 1: Quality Model.

[7] Firesmith, D. G., "Engineering Safety Requirements, Safety Constraints, and Safety–Critical Requirements," *Journal of Object Technology*, Vol. 3, No. 3, March–April 2004, pp. 27–42.

[8] Chitta, S., I. Sucan, and S. Cousins, "MoveIt! [ROS Topics]," *IEEE Robotics Automation Magazine*, Vol. 19, No. 1, March 2012, pp. 18–19.

[9] Rosout–ROS Wiki, http://wiki.ros.org/rosout.

[10] Rrosbag–ROS Wiki, http://wiki.ros.org/rosbag.

CHAPTER 10

Examples of Application: Energy Efficiency for the Steam Generation Process of a Coal Power Plant

with Leticia Moreno

Energy efficiency can be applied at the level of equipment, a process, a system, or a complete industrial facility. This chapter describes the analysis of energy efficiency illustrated with its application to a power plant. The systems approach involves the definition of the context, the identification of the functionality of the system, and its physical architecture, which, combined with the equations of matter and energy balances, allows to evaluate the efficiency of an industrial facility or part of it with the level of appropriate detail, adapting this level of analysis to the process data, equations, graphs, tables, and other correlations available for the particular industrial facility. In the case study the steam generation of a coal power plant is analyzed, choosing one of its 350-MW groups.

10.1 Example Overview

Energy efficiency is now considered as yet another energy resource capable of supplying energy and savings in demand that would reduce electricity generation from other sources such as coal, gas, nuclear, and renewable.

According to the American Council for an Energy Efficient Economy (ACEEE), energy savings through energy efficiency practices by the consumer account for one-third of the cost of other solutions, such as deploying new generation resources.

Conventional energy efficiency solutions have often been adopted at the equipment level of an industrial facility, but this approach does not guarantee that the energy efficiency of such an industrial facility is the sum of the energy efficiency of its parts when these have been optimized independently.

In contrast with the traditional plant engineering approach, a systems approach considers the plant as a whole containing interacting parts, which in turn comprise a system where systems engineering is to be applied [1]. Systems engineering can be applied to a wide range of industrial facilities projects, ranging from a new power

plant construction [2] to energy efficiency of an existing industrial facility or power plant, which is the example presented here.

As well, the best available technique reference documents (BREF) adopted by the European Commission, recommend to optimize the energy efficiency of an industrial facility by means of a systems approach where a set of related parts is chosen for the analysis, a whole from the systemic point of view, which can be a process unit, a subsystem, or a process [3].

Currently, coal is one of the most affordable and largest domestically produced sources of energy in the world. It is used to generate an important amount of our electricity (see Figure 10.1 as an example of a coal power plant). Finding ways to burn it more efficiently is an issue. A diversity of environmental impacts are associated with generating electricity from coal.

In a coal power plant (Figure 10.2) water is transformed to steam, which in turn drives turbine generators to produce electricity since the turbine shaft is connected to the shaft of the generator, where magnets spin within wire coils to produce electricity. Here we summarize how the steam process works. Before the coal is burned, it is pulverized. It is then mixed with hot air and blown into the firebox of the boiler. With the heat released in the combustion process, heat exchange occurs in the boiler. The tubes that go down the boiler are part of the evaporator. Through these tubes, the water goes down in a liquid state (this is the feed water of the water-steam cycle) and with the release of combustion the transformation of water into steam is achieved.

This steam, after being superheated to the necessary pressure and temperature conditions, reaches the turbine where it expands to the condensation pressure. The already condensed steam starts its journey back to the boiler, passing through the pump as shown simplified in Figure 10.2. The heating of the condensate takes place by means of turbine steam extractions. In this way, the water returns to acquire temperature before feeding the boiler. The condensation of the expanded steam in

Figure 10.1 Coal power plant. (Image provided by Banjo © BY-SA 4.0, https://creativecommons.org/licenses/by-sa/4.0.)

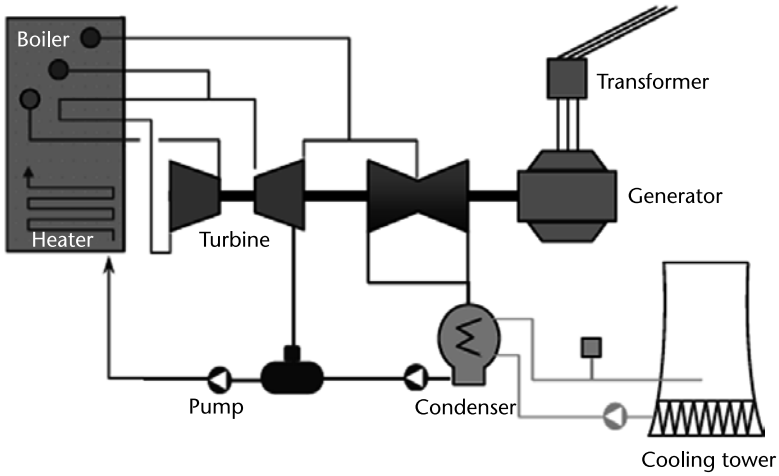


Figure 10.2 Coal power plant schematic.

the turbine is effected by transferring the latent heat of vaporization to the cold sink, usually a large flow of water that comes from a cooling tower, river, or lake.

It is therefore a matter of improving energy efficiency by maintaining the functionality, interfaces, and performance of the system (the power plant) considered in the analysis. The systems approach allows the engineers to determine the context or boundary of the facility where the energy efficiency analysis is to be performed as well as the methods of calculation or simulation that must be chosen, using the laws of thermodynamics and the equations of fluid mechanics, determining numerically the losses of energy, reversibility, efficiency, and other parameters of interest.

There are various examples of calculation and simulation methods published. Some of these methods applied to coal power plants are summarized below.

In some cases, energy efficiency calculation models of coal power plants use the indirect method that requires the calculation of the losses in the boiler, where one of the major problems to be considered is the change of quality and type of coal [4].

Others use simulation models based on more complex computer tools using the principles of coal combustion, heat transfer in boiler elements, matter, and energy balance and thermodynamics of the steam cycle [5].

The use of linear programming algorithms is another alternative to optimize the efficiency of a coal power plant at full capacity, where flows and extractions in turbine stages and fuel consumption are considered as control variables [6].

Another approach uses Modelica to create a software library that contains models of complex thermal power plants with CO_2 capture in static and dynamic modes of operation. The software library is structured according to the main functional groups of the physical processes considered [7].

The above approximations mainly use the enthalpy balance for calculating the efficiency, but in other methods the exergy balances are used, which allows to determine the most efficient process according to the least loss of available work. Using the Aspen Plus computer tool the engineer can do an analysis of a thermal power plant contemplating the second law of thermodynamics and then considering the quality and quantity of energy. The reduction of the irreversible losses of exergy is the best way for the thermal power plant to conserve this energy [8].

Here we use the ISE&PPOOA/energy MBSE approach presented in Chapter 4. ISE&PPOOA/energy combines the models of systems engineering with the balances of matter and energy. This approach allows energy efficiency to be calculated at the appropriate level of abstraction of the coal power plant, which is not well described in other approaches. That is to say, applying the matter and energy balances to the level of decomposition of the plant or facility, where the available equations and correlations allow calculating of the dependent variables.

10.2 Functional Architecture of the Steam Generation Process

The diagrams presented below (Figures 10.3, 10.4, 10.5, and 10.6) illustrate the results of the application of step 2 of the ISE&PPOOA/energy system engineering method described in Chapter 4 to the case study of the steam generation process of an existent coal power plant that is analyzed, choosing one of its 350-MW groups.

Here we identified first the top-level functions of the steam generation process. The top-level function identification follows the recommendations given in Chapter 5 and relies on our previous experience in coal power plant engineering.

The functional hierarchy or functional breakdown structure is the outcome of step 2 of the ISE&PPOOA/energy process described in Chapter 4. Summarizing this hierarchy represents the top-level functions of the system. The functional decomposition of the generate steam process is represented using a SysML block definition diagrams as a hierarchical tree (Figure 10.3).

For the sake of simplicity, the ISE&PPOOA methodology proposes modeling the functional flows as simplified activity diagrams. By this we mean simplified activity diagrams that show the flow of activities/actions but do not represent either tokens or the items (mass or energy) that flow between them. The end to end process of the steam is represented by a SysML activity diagram showing the functional flow dependencies of the steam generation (Figure 10.4). Here the actions are sequential because the item produced by one function is used as an input by the next function to produce its output (as we can see in Table 10.1) that is an N^2 chart representing the functional interfaces.

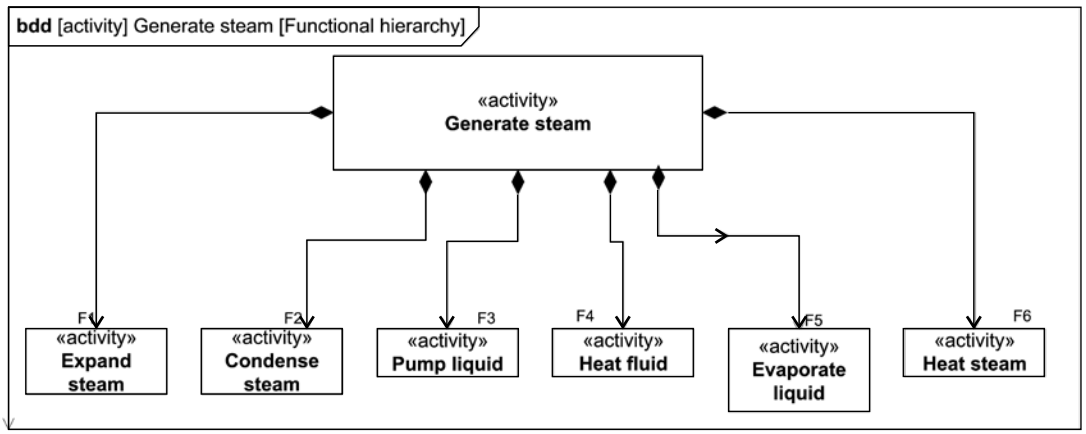


Figure 10.3 Functional hierarchy of generate steam.

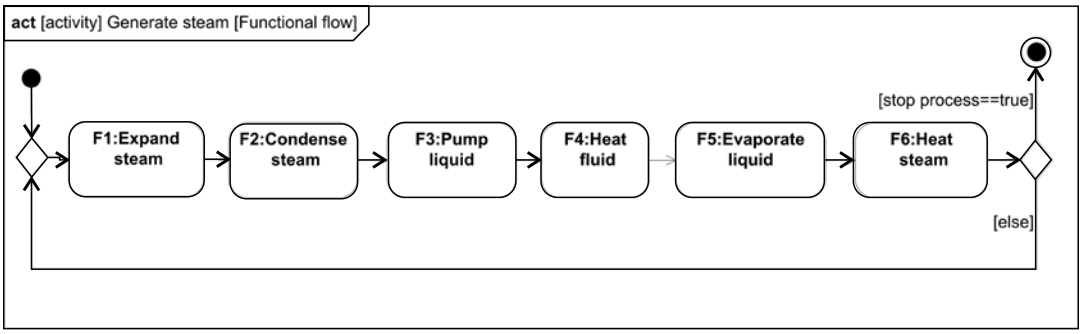


Figure 10.4 Functional flow of generate steam.

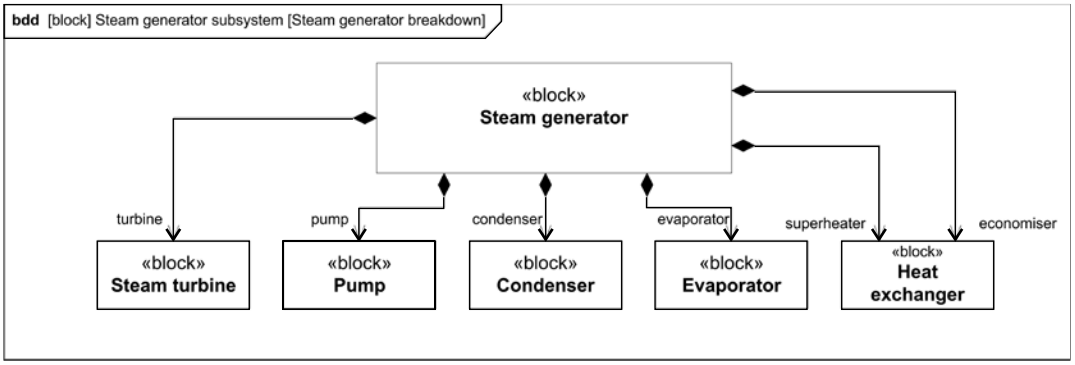


Figure 10.5 Steam generator breakdown.

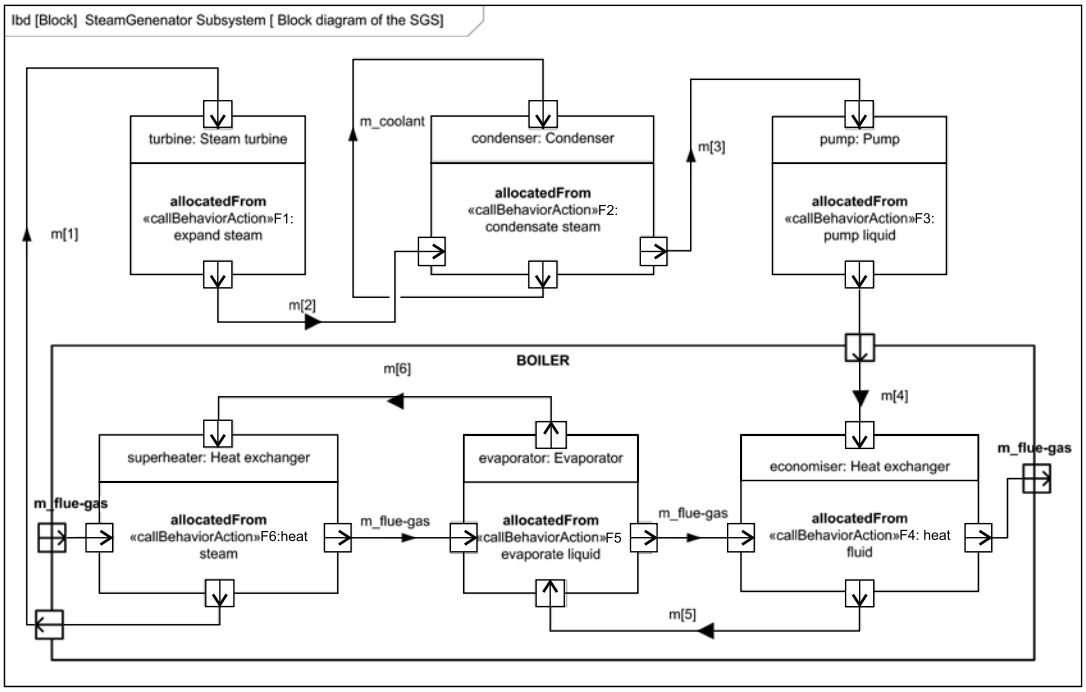


Figure 10.6 Steam generator subsystem internal block diagram.

For the functional interfaces or the item flows the ISE&PPOOA method proposes to use a tabular form, an N^2 chart that provides a compact view of the functional interfaces (items that flow between functions), the external inputs shown in the row above the first entity on the diagonal, and the external outputs shown in the right-hand column (Table 10.1).

10.3 Physical Architecture of the Steam Generation Subsystem

Figure 10.5 represents, by means of a block definition diagram, in SysML notation the constituent parts of the steam generator subsystem. This diagram could have more levels of decomposition if needed, but here because of the purpose of the study, it has not been necessary. The whole—the steam generator—has a composition relation with its parts represented by black diamonds. The parts are represented by blocks that can have more than one role, for example the block heat exchanger can have the role of superheater or economizer. This is an important issue when representing the internal block diagram, shown in Figure 10.6.

Figure 10.6 shows a horizontal view of how the parts of the steam generator are interconnected with each other. This is done using an internal block diagram where flow ports and connectors are represented using SysML notation. Each flow is designated as m [ ], and can be an inflow or outflow to a block. The flow m_fluegas goes through the heat exchangers (superheater and economizer) and the evaporator. Finally, the refrigerant flow m_coolant is shown in the condenser, which will exchange heat with the flow *m* [2], becoming *m* [3] already condensed.

The blocks of Figure 10.6 include the activities or functions represented in Figures 10.3 and 10.4 allocated to the physical parts of the steam generator that perform them. This functional allocation is very useful as information to replace one part or equipment with another with the same functionalities and interfaces, if necessary, for reasons of energy efficiency, as will be seen in the results section.

To clearly understand the operation of the steam generator subsystem,we provide deeper details for each of the aforementioned parts of this subsystem.

Table 10.1 Functional Interfaces

		Work			Flue Gas	
F1:Expand steam	Expanded fluid					Work
	F2:Condense steam	Condensed fluid				Heat
		F3:Pump fluid	Compressed fluid			
			F4:Heat fluid	Saturated liquid		Flue gas
			Flue gas	*F5:Evaporate liquid*	Saturated steam	
Superheated steam				Flue gas	*F6:Heat steam*	

Through the steam turbine of the power plant, energy is extracted from a fluid flow (steam) and turns it into useful work. The work produced by the turbine is used for generating electrical power when combined with a generator (as Figure 10.2 shows). The turbine is a rotary mechanical device (turbomachine) with at least one moving part called a rotor assembly, which is a shaft or drum with blades attached.

The exhaust steam of the turbine contains a lot of energy that cannot escape because it would pollute the surrounding area; however, part of the energy can be saved by condensing the steam. Therefore, a condenser is housed after the turbine. The function of the condenser is to convert the saturated steam at the exhaust of the turbine to water that can be reused for feeding the boiler to again convert it into steam. The major energy loss takes place in the condenser through its cooling water system, but this conversion of the steam into water is needed to improve the efficiency of the cycle, because pumping water (rear process) requires less energy when compared to pumping steam back to the boiler.

Consider the boiler as a closed vessel in which water is converted into steam by the application of heat (coming from the fuel combustion) and which clusters the evaporator and other heat exchangers (including both economizer and super heater). The feed water comes from the return of the circuit and arrives the economizer where the liquid reaches the saturation temperature, and then, in the evaporator, it produces its evaporation at constant temperature and the pressure characteristics. Subsequently, in the superheater, the saturated steam obtained is raised in temperature to the conditions of the turbine.

The evaporator is really an evaporation system that incorporates several evaporators of different types installed in series. All evaporators are fundamentally heat exchangers made of metallic materials to have an adequate heat transfer in order for evaporation to take place.

The other heat exchangers (economizer and superheater) are tubular bundles that are immersed in the exhaust gas stream, exchanging heat with the smoke by convection and radiation. The configuration depends on its location in the boiler and the temperature of the fumes. The economizers are exchangers of the water-gas type, while the superheaters are of the steam-gas type (steam circulates inside and gases outside), so they are subject to greater wear. As for the arrangement, all the pipes have an inlet manifold that concentrates all the flow that will be distributed later, as well as another one to the exit. They are bare tubes (not flapped) to obtain a wide pass section between them.

10.4 Equations and Correlations of the Matter and Energy Balances

The block definition diagrams with constraint blocks, which we will call here constraints diagrams, show the relationships or equations between variables and value properties of interest of the blocks or parts, which in this case make up the steam generation subsystem.

In order to model the behavior of this steam generation subsystem and define these relationships and equations between the different blocks involved in the subsystem, the Rankine cycle has been considered. It is an ideal cycle that represents in a very approximate way the real cycle of a steam power plant and it does not

involve any internal irreversibility (isentropic processes are considered). All the blocks or components associated with the Rankine cycle (the boiler, turbine, condenser, and pump) are steady-flow devices and thus all the processes can be analyzed through the first law of thermodynamics or law of conservation of energy, adapted for thermodynamic systems:

$$(q_{in} - q_{out}) + (w_{in} - w_{out}) = h_e - h_f \tag{10.1}$$

Figure 10.7 represents three physical equipment blocks: economizer, evaporator, and superheater. However, we have no equations constraining these equipment blocks so the energy balance cannot be solved. Therefore, we cluster these three elements into a logical block called boiler to facilitate the identification of the mass and energy balances to be applied.

Boiler logical block has been created following the guidelines of step 5 of the ISE&PPOOA/energy method (see Chapter 4) in order to have zero degrees of freedom and to be able to solve the problem with the balance equations available. So (10.2) is constraining the block called boiler (see Figure 10.8). In this block, water enters as a compressed liquid and leaves as a superheated steam. The boiler is basically a large heat exchanger where the heat originating from combustion gases, is transferred to the water essentially at constant pressure (W = 0).

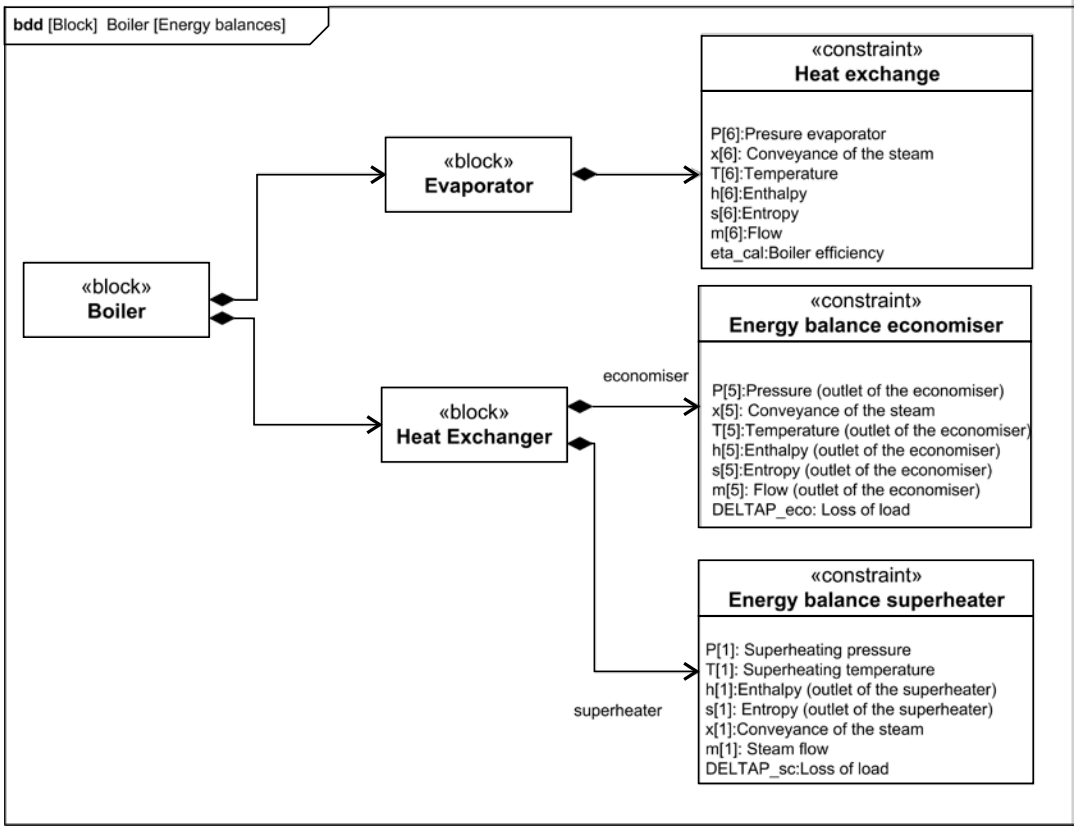


Figure 10.7 Boiler energy balance constraints diagram.

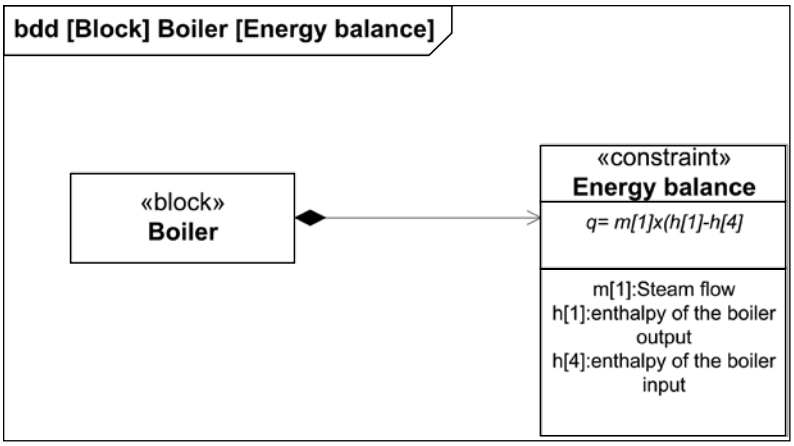


Figure 10.8 Improved boiler energy balance constraints diagram.

$$q = m[1] \times (h[1] - h[4]) \tag{10.2}$$

where q is the heat supplied by the boiler or logic block that contains the economizer, evaporator, and superheater, m [1] is the flow rate that flows through these three blocks (as shown in Figure 10.6), h [1] is the enthalpy of the boiler output that is the enthalpy of the superheated steam at the outlet of the superheater, and h [4] is the input enthalpy to the boiler or enthalpy of the fluid at the economizer inlet.

Water enters the pump as a saturated liquid and is compressed isentropically to the operating pressure of the boiler. The water temperature increases during this isentropic compression process due to a slight decrease in the specific volume of the water.

The equation governing pump behavior is the power (10.3) of the pump (Q = 0):

$$Powp = m[1] \times w_{p4} = m[1] \times (h[4] - h[3]) \tag{10.3}$$

where it appears as new variable h [4], which is the enthalpy of the fluid at the outlet of the pump.

Figure 10.9 is the constraint diagram of the steam turbine. In that case the superheated steam enters the turbine, where is expanded isentropically and produces work by rotating the shaft connected to the electric generator. The temperature and the pressure of the steam drop during this process. In (10.4) it reflects how the steam turbine power can be obtained, taking into account the isentropic expansion (Q = 0):

$$Powtb = m[1] \times w_{tb} = m[1] \times (h[1] - h[2]) \tag{10.4}$$

where h [2], that appears here for the first time, is the enthalpy of the fluid at the exit of the turbine. It is important to note that the enthalpies for this and the previous equations are tabulated and depend on variables that will be parameterized in the execution of the simulation.

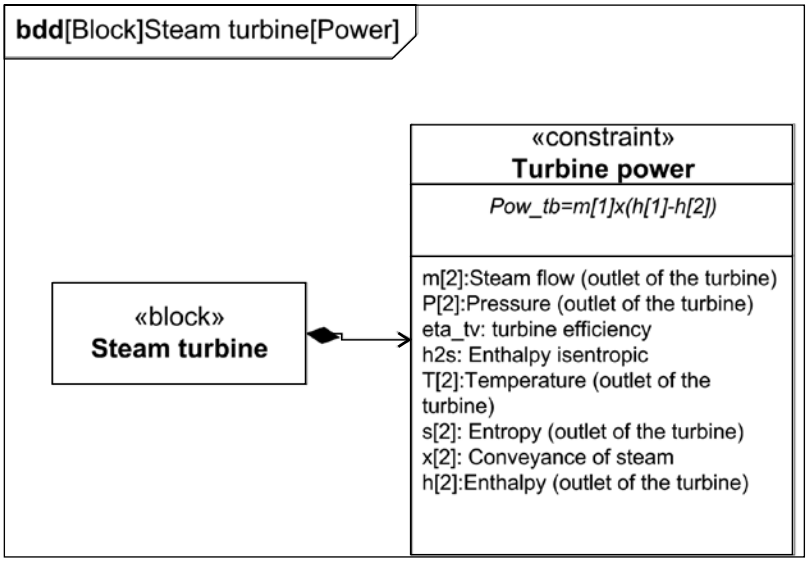


Figure 10.9 Steam turbine power constraints diagram.

Figure 10.10 is the constraint diagram of the condenser. At this block, the steam is a saturated liquid-vapor mixture with a high quality. Steam is condensed at constant pressure in the condenser, which is basically a large heat exchanger, by rejecting heat to a cooling medium from a lake or a river. The steam leaves the condenser as saturated liquid and enters the pump, completing the cycle. The energy balance in this block is reflected in (10.5):

$$m\_steam\ x(h[3] - h[2]) + m\_coolant\ x(hout - hin) = 0 \qquad (10.5)$$

where m_steam is the steam flow rate that reaches the condenser and coincides with *m* [2] shown in Figure 10.6. As new variables appear, hin and hout correspond to the enthalpies of the coolant at the inlet and outlet and whose flow is m_coolant.

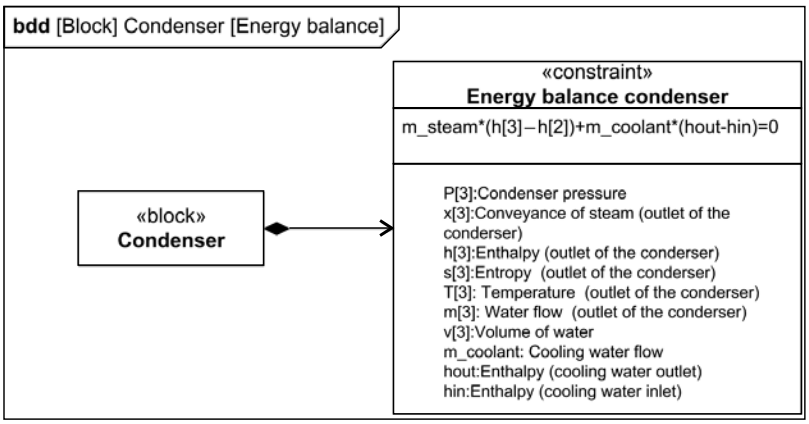


Figure 10.10 Condenser energy balance constraints diagram.

10.5 Results

In order to simulate different situations in the steam cycle, the Engineering Equation Solver (EES) computer tool has been chosen. This tool can solve systems of nonlinear equations as well and provides many useful functions for the solution of thermodynamic equations. In addition it stores thermodynamic properties that facilitate the work and avoid the usage of steam tables or thermodynamic diagrams.

In order to execute the computer tool, some preliminary data of the coal power plant to be analyzed are needed. In this case the public data of a coal power plant of the northwest of Spain have been chosen. Once the results for the given conditions have been obtained and the model has been observed to be coherent and realistic, since it fits quite well with the actual conditions of the plant, it is possible to analyze which parameters would lead to an improvement in the performance of the plant cycle. For this purpose, three parameters have been selected. According to the results obtained, it is decided how these changes could be implemented in practice and if they are of economic interest.

The first parameter that has been selected is the superheat temperature, Tosh or T [1] in Figure 10.7. Thermodynamics proves how the steam cycle efficiency increases when the hot source temperature is reached. The studied cycle allows working with a temperature of superheat equal to 538°C, obtaining a thermal efficiency of 38.81%. If this temperature could be increased, the performance could be improved. However, this is not easy, bear in mind that the temperature limit imposed on steam is defined by the thermal stress conditions of the equipment materials used.

The second parameter to be considered is the superheat pressure, Psh or P [1] in Figure 10.7. It is confirmed that the increase in the superheated steam pressure conditions leads to an increase in cycle efficiency. The cycle studied allows working with a superheating pressure equal to 162 bar obtaining a thermal efficiency of 38.81%. If this pressure could be increased, the performance could be improved. It is necessary to take into account the stress conditions of the materials used. It is known that when the steam circulates through several superheater tube bundles at high speed the pressure drop is high, with values of 10–12 bar being typical for flows of the order of 1,000 Tg/h at 175-bar pressure. It can be concluded that it is not worthwhile to implement this improvement.

Finally, the third parameter to be studied is the condensation pressure, Pcond or P [3] in Figure 10.10. The pressure at which the condenser works in this case is 0.067 bar.

Using the characteristic curves of the condenser shown in Figure 10.11, we analyze which is the most effective condensing pressure for a given load.

The temperature of the cooling water of the water-steam cycle of the selected plant enters the condenser at 18°C. If the load is approximately 350 MW, the condensing pressure can be reduced to approximately 0.058 bar. Introducing this new value it is observed that it would obtain a greater thermal efficiency of the cycle.

It is observed that bringing this improvement to practice can be a good option, since only one would have to act on aspects of the plant operation, and therefore it would not be necessary to improve the design of the equipment or to replace it, an action that is less feasible economically.

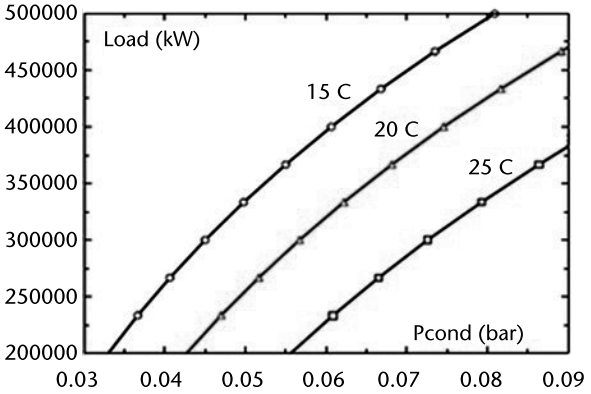


Figure 10.11 Condensing curves.

10.6 Summary

Actions to improve the energy efficiency of an industrial facility may be carried out through operational changes, equipment upgrades, or replacement. It is obvious that a purely technical approach would not suffice when economic and environmental considerations have to be taken into account.

The systems approach, shown in this chapter, involves several activities integrated in a single engineering process called ISE&PPOOA/energy, such as the definition of a context of efficiency analysis, the modeling of the system to be analyzed, and the resolution of the matter and energy balance equations at the appropriate level of abstraction.

This approach allows a study of alternatives that can range from the tuning of operating parameters to replacing equipment for a more efficient but equivalent in functionality and interfaces with other elements of the industrial facility, which is reflected in the internal block definition diagrams obtained by the application of model-based systems engineering.

10.7 Questions and Exercises

1. What does ISE&PPOOA/energy method mean by zero degrees of freedom of a system?
2. What is the difference between a plant unit and a system?
3. What does the SysML constraint block represent?
4. Identify the top-level functions of the pasteurization of milk process.
5. Create a functional flow diagram of the pasteurization of milk process.
6. Create an N^2 chart of the functional interfaces of the pasteurization of milk process.

References

[1] Vanek, F. M., L. D. Albright, and L. T. Angenent, *Energy Systems Engineering: Evaluation and Implementation*, Second Edition, New York: McGraw-Hill, 2012.

[2] Navas, J. et al., "Bridging the Gap between Model-Based Systems Engineering Methodologies and Their Effective Practice: A Case Study on Nuclear Power Plants Systems Engineering," *INCOSE Insight*, Vol. 21 No. 1, March 2018.

[3] European Commission, *Reference Document on Best Available Techniques for Energy Efficiency*, European IPPC Bureau, 2009.

[4] Shi, Y. et al., "On-Line Calculation Model for Thermal Efficiency of Coal-Fired Utility Boiler Based on Heating Value Identification," *Proc. of 2011 International Conference on Modeling, Identification, and Control*, Shanghai, China, 2011, pp. 203–207.

[5] Sanpasetparnich, T., and A. Aroonwilas, "Simulation and Optimization of Coal-Fired Power Plants," *Energy Procedia*, 2009, pp. 3851–3858, doi: 10.1016/j.egypro.2009.02.187.

[6] Tzolakis, G., et al., "Simulation of a Coal-Fired Power Plant Using Mathematical Programming Algorithms in Order to Optimize Its Efficiency," *Applied Thermal Engineering*, Vol. 48, 2012, pp. 256–267, doi: 10.1016/j. applthermaleng.2012.04.51.

[7] Bronnemann, J., et al., "Status of ClaRaCCS: Modelling and Simulation of Coal-Fired Power Plants with CO_2 Capture," *Proc. of the 9th International Modelica Conference*, Munich, Germany, 2012, pp. 609–618, doi: 10.3384/ecp 12076609.

[8] Hou, D., et al., "Exergy Analysis of a Thermal Power Plant Using Modeling Approach," *Clean Technical Environmental Policy*, Vol. 14, 2012, pp. 805–813, doi: 10.1007/S10098-011-0447-0.

CHAPTER 11

Trade-Off Analysis

Analytical trade-off analysis is a complex and interesting issue for systems engineers to explore the solutions space and it is complementary to the heuristics approach proposed by the ISE&PPOOA method.

11.1 Trade-Off and the Architecture Decision Process

Trade-off analysis is part of a more general process dealing with decision making that is part of the main systems engineering tasks described in Chapter 2. The *INCOSE Handbook* [1] defines decision management as a process whose purpose in accordance to ISO/IEC/IEEE 15288 is "to provide a structured analytical framework for objectively identifying, characterizing, and evaluating a set of alternatives for a decision at any point in the life cycle and select the most beneficial course of action" [2].

The ISO definition of the purpose is clearly pointing out that the decision process can be applied in any stage of the system life cycle for either a system project or program. In particular, considering trade studies and systems development, Parnell identifies three main trade spaces to be considered at systems development: conceptual trade space, architectural trade space, and design trade space [3].

There are many and diverse assessment of alternatives evaluation techniques. Here, in Table 11.1, for the sake of brevity, we summarize some of the techniques presented by Kenley, Whitcomb, and Parnell in Chapter 8 of the Parnell book [3].

Here as part of architectural trade studies, we will explain the application of trade-off analysis as an approach to be used to help in the decision making inherent to step 4.3 "refine the architecture" of the ISE&PPOOA process described in Chapter 4. As we mention there, trade studies may be performed to select the preferred physical architecture that optimizes the performance and other criteria selected.

The "refine the architecture" step of the ISE&PPOOA process is used to refine the modular architecture obtained from the functional allocation, selecting the most appropriate design heuristics (see Chapter 6) that are based on meeting the specified nonfunctional requirements for the particular system of interest. Trade-off

Table 11.1 Assessment of Alternative Techniques*

Technique	*Brief Description*
Decision theory	This technique uses value measures derived from objectives and value functions and may account for uncertainty as well (see Sections 11.2 and 11.3).
Pugh method	This technique compares alternative attributes as better (+), same (S), or worse (-). The weakness of this technique is that it ignores the relative importance of performance measures.
Design of experiments (DoE)	This technique uses statistics to develop a model to predict the response for factor-level combinations.
Quality function deployment (QFD)	This technique, also called the house of quality matrix, provides a way to trace development aspects from the customer needs
Analytic hierarchical process (AHP)	This technique captures explicitly stakeholder needs and determines the relative importance of the attributes by pairwise comparison (see Section 11.3).

* Adapted from [3].

analysis is in this context a complementary technique useful for identifying a candidate solution for either a module or subsystem implementing one or several key system functions. The result of the application of heuristics and trade-off analysis is the refined architecture (see Figure 11.1). Although it seems that this process is sequential, it can be applied iteratively as well.

Nonfunctional requirements and particularly those related to performance at component level are the source for defining the trade-off criteria to be applied to the solution component to be selected between candidate components performing the same functionalities using different technologies.

Nonfunctional requirements are also used to identify from the collection of heuristics (see Chapter 6) which are those to be applied in our system of interest.

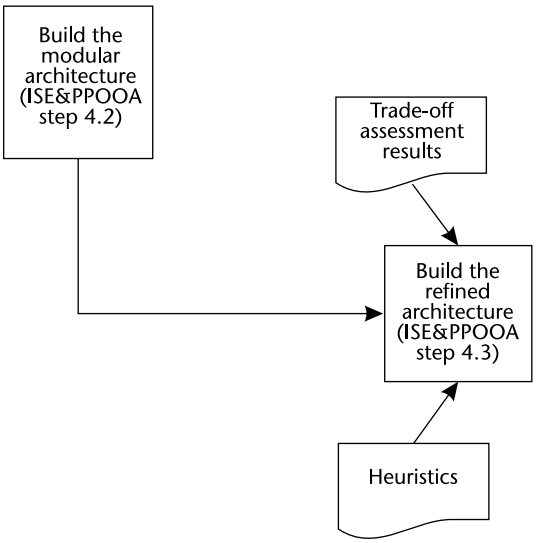


Figure 11.1 Building the refined architecture by means of trade-off analysis and heuristics application.

11.2 Trade-Off Assessment Criteria and Utility Functions

Parnell recommends building a value hierarchy to represent the trade-off assessment criteria. This value hierarchy has at least three levels: the decision purpose, the criteria that define value, and the measures for each criteria to assess potential value [3].

In the case of the system functions allocated to the building elements of the modular architecture, the trade tree represents criteria that may be the performance objectives for each allocated function defining value and the value measure for each criteria to assess its potential value. See Figure 11.2 where we represent a generic trade tree containing functional performance criteria, cost criteria, and other criteria that may be, for example, those related to reliability or maintainability.

Utility or value functions are used to establish a consistent scale for dissimilar criteria. The utility or value function represents the relationship between a measure for each selection criterion (x-axis) and a common scale (y-axis).

For the trade study, it is necessary to represent as part of the system model the selected assessment criteria and the utility or value functions associated to them. Utility functions can be discrete or continuous. As shown in Figure 11.3, they follow three basic shapes: linear, curve, and S-shape curve.

When creating an increasing utility function for a particular criterion, the systems engineer ascertains whether the stakeholders believe its minimum value measure to be accepted, mapping it to the 0 value on the score scale (y-axis). The measure beyond which an alternative provides no additional value is mapped to the highest score scale (y-axis). When working with curve shapes, it is important

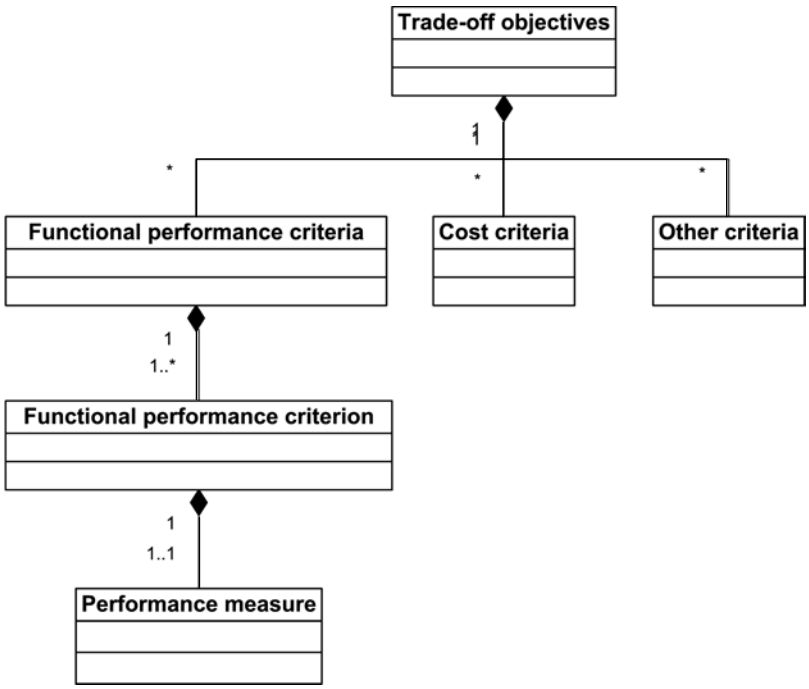


Figure 11.2 Generic trade tree.

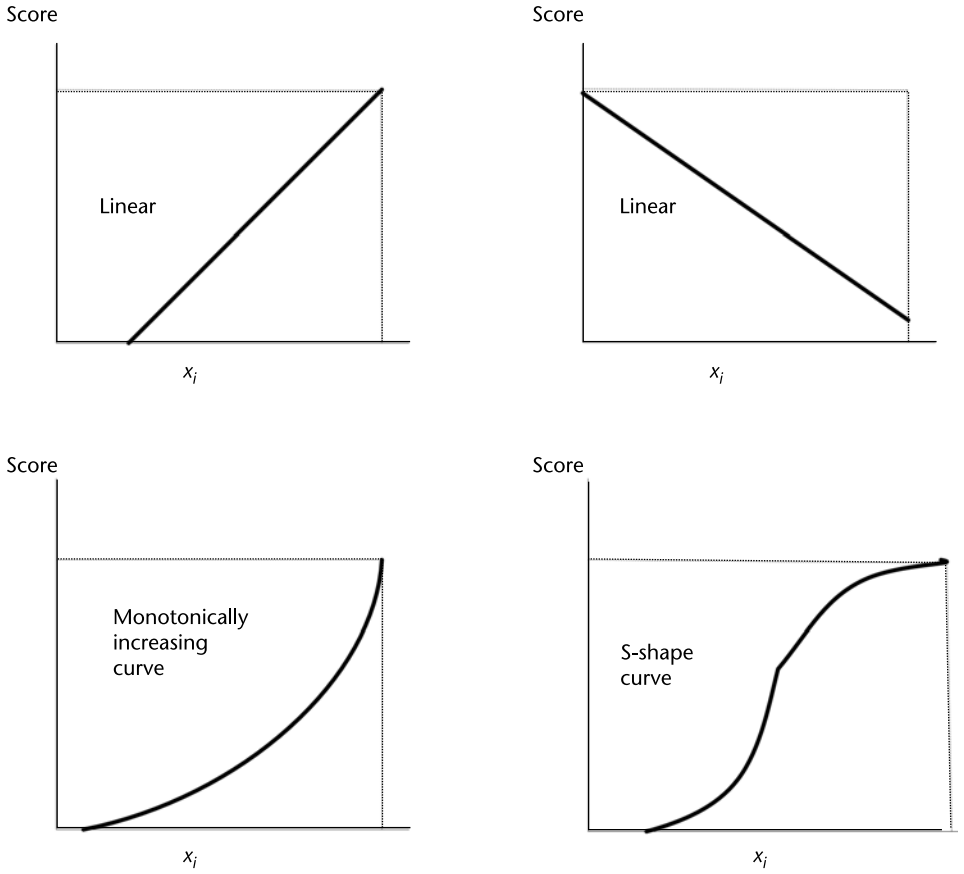


Figure 11.3 Basic shapes for a utility function.

to pick the appropriate inflection points for drawing the curve that may be either convex or concave [3].

11.3 A Trade-Off Subprocess to be Used with the ISE&PPOOA Process

Based on diverse processes for trade-off analysis found in the literature, [4–6], we propose here a trade-off analysis process that can be integrated as part of the ISE&PPOOA architecting process using its outputs and producing inputs to the ISE&PPOOA architecting process presented in Chapter 4. Classical approaches such as NASA and Cross do not use SysML system models, but recent approaches such as IBM [6] use SysML notation and diagrams. The proposed trade-off subprocess of ISE&PPOOA has the steps summarized as follows:

1. *Identify the system modules to be used in the trade-off study*. From the modular architecture obtained in step 4.2 of the ISE&PPOOA process, select which modules (i.e., logical building elements clustering cohesive functionality) may be implemented using alternative technical solutions identified in the next step.

2. *Identify credible alternative technical candidates for implementing the system logical building blocks or modules under consideration.* The list of technical alternatives selected during brainstorming sessions may be reduced considering the requirements to be met. Some alternatives may be eliminated based on cost or technology readiness. The remaining alternatives should be described in the detail needed to be assessed.
3. *Define objectives and criteria to evaluate the technical solutions.* Objectives related to stakeholder needs are transformed into a set of performance, cost, and other criteria to be used for the trade-off study. Building a trade tree (see Figure 11.2) and discussing it with the stakeholders will be helpful.
4. *Assign relative weightings to the criteria.* Using pairwise comparison or the AHP [4] and establish the relative weights to the criteria at the same level so that all weights sum to 1.0.
5. *Generate the utility function for each criterion.* Utility functions giving scores for each criterion to be used can be represented using SysML parametric diagrams with constrain properties blocks.
6. *Assess each alternative.* Estimate the performance of every alternative for a given criterion in terms of its score based on the utility function applied. The assessment executes the utility functions represented by parametric diagrams and using attributes values from each alternative obtained from test data, vendor provided data, engineering practice, or other sources.
7. *Show the trade study results.* Generally a summary table (see Table 11.2) of criteria versus alternatives is presented to summarize the results from the preceding steps.

11.4 Summary

This chapter presented trade-off analysis as a complementary technique to be used in the system architecture as it is modeled using the ISE&PPOOA MBSE methodology. In particular, we recommend using it when evaluating technical alternatives either at the system, subsystem, or component level.

Table 11.2 Trade-Off Study Summary Table

Criteria		Technical Alternatives							
Criterion	Weight	Alt 1		Alt 2		Alt....		Alt *n*	
		Score	Weighted Score	Score	Weighted Score	Score	Weighted Score	Score	Weighted Score
Criterion 1									
Criterion 2									
.....									
Criterion *n*									
Total									

References

[1] Walden, D. D., et al., *Systems Engineering Handbook–A Guide for System Life Cycle Processes and Activities*, INCOSE-TP-2003-02-04, Hoboken, NJ: John Wiley & Sons, 2015.

[2] ISO, *ISO/IEC/IEEE 15288:2015*, *Systems and Software Engineering–System Life Cycle Processes*, Geneva: International Standards Organization, 2015.

[3] Parnell, G. S. (ed.), *Trade-off Analytics: Creating and Exploring the System Tradespace*, Hoboken, NJ: John Wiley & Sons, 2017.

[4] Goldberg, B. E., et al., *System Engineering Toolbox for Design-Oriented Engineers*, Alabama NASA Marshall Space Flight Center, NASA Reference Publication 1358, December 1994.

[5] Cross, N., *Engineering Design Methods: Strategies for Product Design*, Hoboken, NJ: John Wiley & Sons, 2000.

[6] Bleakley, G., A. Lapping, and A. Whitfield, "6.6.2 Determining the Right Solution Using SysML and Model Based Systems Engineering (MBSE) for Trade Studies," *INCOSE International Symposium*, Vol. 21, 2011, pp. 783–795.

CHAPTER 12

Other Topics of Interest and Next Steps

There are other topics of interest for the MBSE practitioners as described in this chapter, and for scope and space reasons they are described briefly. Agile development is growing and has a large market potential for software development. We will show that the ISE&PPOOA methodology can be integrated and used in an agile project. Architecture evaluation and particularly model checking is still a research topic that deserves some explanation and literature references. The last section of this chapter is a recommendation to the readers of the next steps to follow to apply the ISE&PPOOA process to use MBSE in their organizations.

12.1 Agile Development

The main objective of this section is to show how systems engineering and particularly MBSE can be more agile and how the ISE&PPOOA MBSE approach can support agility in product development. Therefore, this section will describe briefly the principles of agility, how agility is scaled to large or complex products, and how the ISE&PPOOA approach can support agility and be agile as well. It is out of scope to explain here the diversity of agile methods used by industry. Next, we provide literature references that expand on agile approaches to system and software development.

12.1.1 Principles and Misconceptions about Agility

It is important to recognize that what we called an evolutionary approach to development (see Chapter 2) is a foundation of the later so-called agile methodologies. Iterative and incremental approaches to development can be traced back to the 1960s. During the 1990s industry invented approaches such as rapid application development (RAD), Scrum, extreme programming (XP), or feature driven development (FDD), most of which were later termed as agile methods.

One of the main milestones in agile software development was the publication of the *Agile Manifesto* [1] coauthored by 17 software developers that met at a resort in Snowbird, Utah in 2001. The principles behind the *Agile Manifesto* are summarized here:

- Early and continuous delivery of value software to satisfy the customer;
- Changing requirements are welcome;
- Deliver working software frequently;
- Business people and developers must work together on a daily basis;
- Build projects around motivated individuals;
- Face-to-face conversation as the most efficient and effective method of conveying information to and within a development team;
- The primary measure of progress is working software;
- The sponsors, developers, and users should be able to maintain a constant pace indefinitely promoting sustainable development;
- Agility is enhanced by continuous attention to technical excellence and good design;
- Simplicity or maximizing the amount of work not done is essential;
- Self-organizing teams are necessary for the emergence of the best architectures, requirements, and designs;
- At regular intervals, the team tunes and adjusts its behavior to become more effective.

In some cases due to misunderstandings or inadequate application of the above principles, some misconceptions about agility are disseminated. Carlson [2] describes and demystifies the most common misconceptions or agile myths that we summarize here:

- *Agile development is undisciplined and not measurable.* Agile is not waterfall but involves a disciplined method and measurements applied to short iterations.
- *Agile development has no project management.* It is a more or less real issue but the fact is that there are other roles with responsibilities similar to the project manager.
- *Agile applies only to software development.* Some industrial experiences in aerospace, automotive, and other domains negate this statement.
- *Agile development has no documentation.* Several industrial agile projects and mainly those related to certifiable products deliver the required documentation. These documents follow the same delivering pattern as all other deliverables: for example, software and hardware items.
- *Requirements are not necessary in agile development.* Features and stories used in agile development are true requirements.
- *Agile methods only work with small teams.* The scaled agile approaches, summarized in the following section, debunk this misconception.
- *Agile methods do not include planning.* It is not true since agile methods include incremental planning of the diverse iterations.
- *Agile development does not scale.* Some scale agile approaches described in a following section debunk this misconception.

12.1.2 Scalability in Agile Approaches

Scalability is one of the main concerns when applying agile methods to either large or complex projects. Current scalable agile frameworks blend agile and lean practices to address industry needs. Here, for illustrative purposes we will describe some scalable agile methods used by industry. The main objective of this section and the next one is to convince the reader of the feasibility of applying agile methods in the projects or programs developing complex products combining hardware, software, and firmware.

For the sake of brevity we choose three illustrative scalable agile approaches to describe here. For each one we describe its scope and its process. There are other comparison studies of scalable agile frameworks to be consulted as well [3].

12.1.2.1 Scrum@Scale

Scrum is one of the most popular agile frameworks used to develop and deliver software products by a single team. Scrum@Scale is designed to achieve scalability by coordinating the diverse teams involved. Scrum@Scale accomplishes this goal by its scale-free architecture.

As defined by Sutherland: "Scrum@Scale is a framework for scaling Scrum. It radically simplifies scaling by using Scrum to scale Scrum. It consists only of Scrum teams coordinated via Scrum of Scrums and MetaScrums" [4].

The products delivered using Scrum@Scale may be hardware, software, complex integrated systems, services, processes, and others.

Scrum concepts such as backlog and sprint [5] are still used:

- *Product backlog:* This backlog is an ordered list of everything needed in the product development and is the single source of requirements for any product changes;
- *Sprint:* One iteration of an agile project, representing a timebox of one month or less during which, an increment in the product is created that can be used and potentially deployed,
- *Sprint backlog:* This backlog contains the product backlog items selected for the sprint plus a plan for delivering the product increment and realizing the sprint goal.

Scrum@Scale proposes two cycles, which are represented in Figure 12.1: the Scrum master cycle and the product owner cycle. The first one deals with the how, and the second one deals with the what. These two cycles synchronize at two points and together produce a framework coordinating the efforts of multiple teams along a single path [4].

12.1.2.2 Disciplined Agile Delivery

As defended by Ambler, disciplined agile delivery (DAD) is an agile process framework that covers the entire solution life cycle from initiation of the project through construction to the point of solution release into production [6].

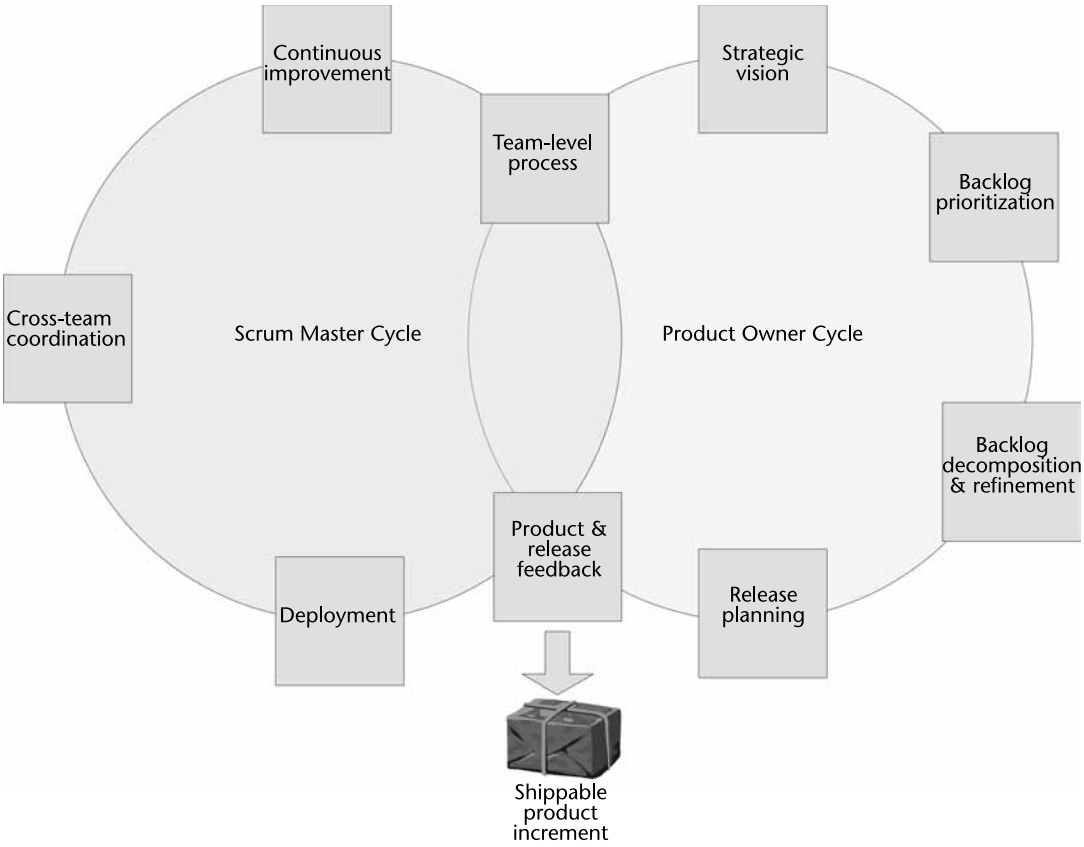


Figure 12.1 Scrum@Scale cycles.

DAD framework adopts practices from other approaches, such as the backlog from Scrum; continuous integration, refactoring, test driven development, and collective ownership from XP; the importance of providing the architecture in early iterations from unified process (UP); and limiting the work in progress and visualizing work from Kanban.

DAD addresses software, hardware, documentation, business process, and organizational structure issues [6].

The life cycle of a DAD project is summarized in Figure 12.2. As described by Ambler this life cycle has three differential aspects:

1. A delivery life cycle extending the construction phase;
2. Explicit phases that are inception, construction, and transition;
3. Context, preproject, and postproject activities are considered as well.

12.1.2.3 Scaled Agile Framework

Scaled agile framework (SAFe) is an agile framework that synchronizes alignment, collaboration, and delivery for large numbers of teams. It supports both software and systems development, from projects under 100 practitioners to the largest software solutions, and complex cyberphysical systems, which are systems that require thousands of people to create and maintain.

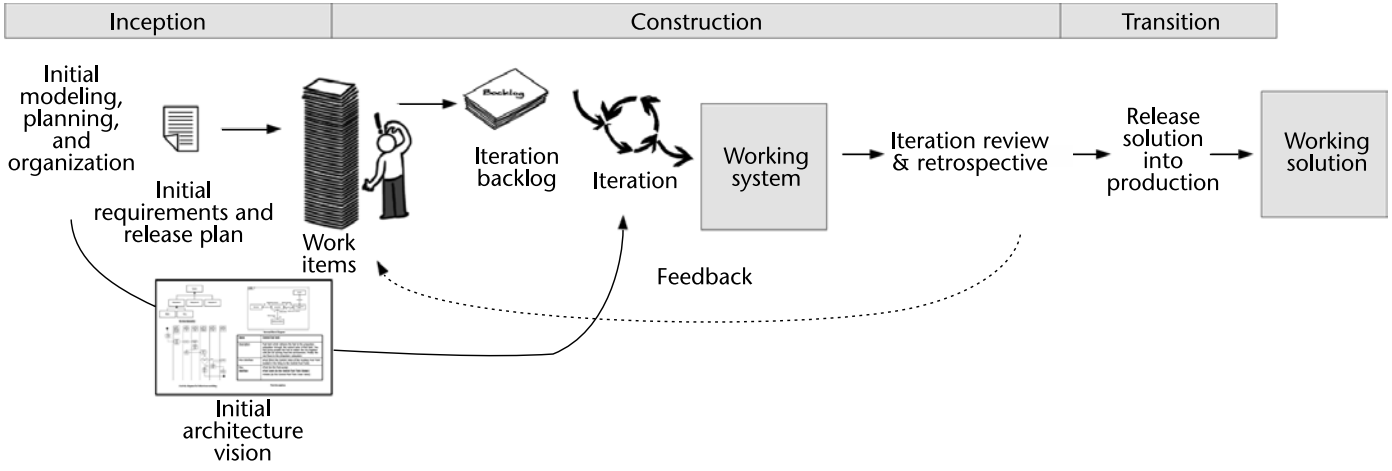


Figure 12.2 Basic DAD life cycle.

SAFe combines three bodies of knowledge: agile development, lean product development, and systems thinking.

SAFe supports four configurations: essential SAFe, large solution SAFe, portfolio SAFe, and full SAFe [7]. Here for the sake of brevity we will describe the essential SAFe. A complete description of the four configurations can be found in the *SAFe Reference Guide,* Version 4.5 [7].

The essential SAFe is the simplest starting point for implementation of the SAFe framework.

The core of SAFe is the program level, which revolves around an organization called the Agile Release Train (ART) (see Figure 12.3). The ART includes all the roles that are necessary to transform ideas from concept through deployment. Each ART delivers valuable and tested system increments every 2 weeks. ARTs build and maintain a continuous delivery pipeline to regularly develop and release small increments of value [7].

Figure 12.3 is a simplified representation of essential SAFe where solutions are released on demand, during or at the end of a program increment (PI), based on the needs of the business. PIs provide fixed timebox increments for planning, execution, and inspecting and adapting [7].

12.1.3 ISE&PPOOA Process and Agility

This section addresses the answers to two important questions: how MBSE and particularly ISE&PPOOA can support agile approaches and how ISE&PPOOA as a MBSE approach can be more agile.

The first question is easy to answer considering how popular scalable agile approaches, such as DAD and SAFe (described and referenced above) consider the

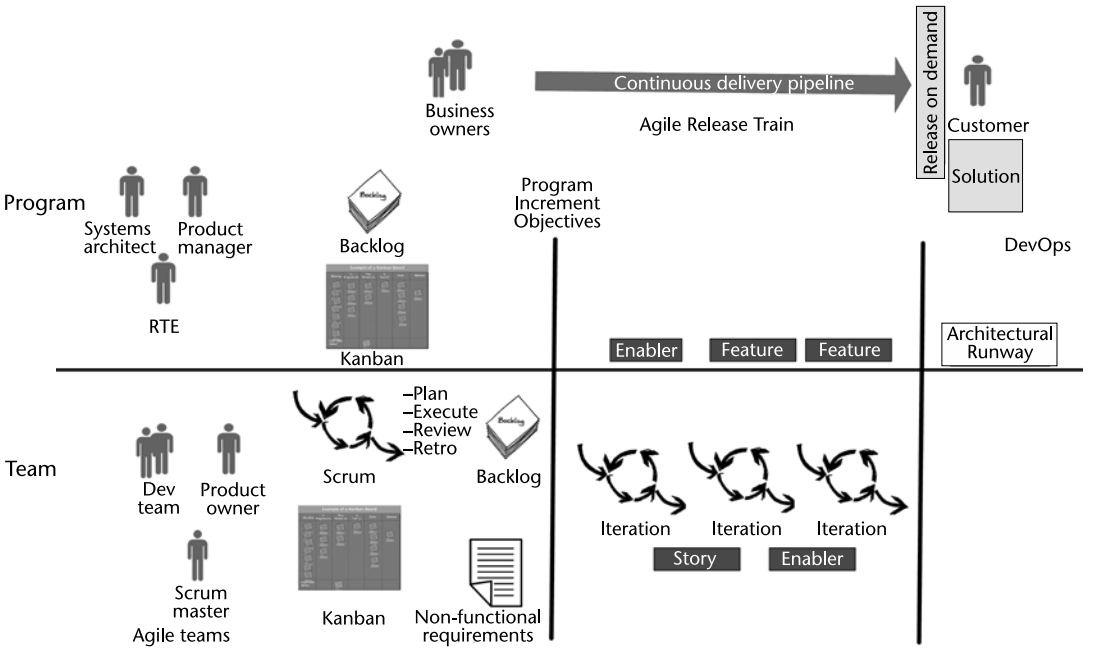


Figure 12.3 Main elements of essential SAFe.

importance of the systems architecture as a previous step to the agile iterations or sprints.

Regarding DAD, the reader can see in Figure 12.2 that the system architecture is an iteration input since DAD focuses on the importance of proving out the architecture in the early iterations, reducing all types of risk early in the life cycle [6].

SAFe Reference Guide, Version 4.5 (see Figure 12.3) is more explicit regarding the importance of the system architect role, MBSE, and models as the way to explore the structure and behavior of system elements, evaluate design alternatives, and validate assumptions as early as possible in the system life cycle. In particular, those products, such as commercial aircraft, cars, trains, and medical appliances with regulatory requirements, are the best candidates to use models to generate the certification documentation. Here the models act as a single source of truth [7].

The second question of how ISE&PPOOA as a MBSE approach can be more agile is answered by developing the ISE&PPOOA models described in Chapters 4, 5, and 7 iteratively and incrementally.

The ISE&PPOOA process may be applied iteratively and to deliver in increments or builds, by incremental we mean here to use the functional architecture of the system of interest and slice it into common groups of system response functions that are implemented in the next increment. That is, each system response is implemented as an increment containing the functions and physical building blocks participating in the concrete response. This approach may facilitate as well the end-to-end testing of each system response. The NFRs applicable to the selected building blocks of the modular architecture are implemented using design heuristics to build the refined architecture of the system increment.

A similar approach is described by B. P. Douglas in the harmony agile MBSE process, where he proposes to do an iteration for each use case or a hybrid approach grouping and handing off together a small number of developed use cases [8].

12.2 Architecture Evaluation and Model Checking

12.2.1 Architecture Evaluation

Verification and validation is applied during the system development life cycle, and one of its main activities is the verification and validation of the system architecture that is performed using a combination of manual and automatic practices that are summarized and referenced to the literature below.

First, before describing the objectives of the architecture evaluation, it is important to realize that what we mean by system architecture is not only the structural and behavioral models produced by the application of the ISE&PPOOA method but the design decisions made as well. Therefore, when we evaluate a system architecture we evaluate the models and the design. As mentioned by Friedenthal, a good model is the model that meets its intended purpose and a goodness of a design is based on how well it satisfies the requirements and the extent to which it incorporates quality design principles [9].

Regarding models, Balci et al. [10] give the following definition for model verification, validation, and testing:

- Model verification deals with building the model correctly; that is, the model is transformed from one into another with sufficient accuracy;
- Model validation deals with building an accurate model; that is, the model within its domain of applicability that behaves with satisfactory accuracy consistent with the modeling objectives;
- Model testing determines whether inaccuracies or errors exist in the model.

Engel [11] proposes three main goals to be taken into account in the evaluation of the system architecture:

- Consistency of the system architecture versus the system requirements and interface requirements;
- Feasibility of the design within the bounds of cost, time, and other project constraints;
- Meeting product existing standards, environmental protection issues, certification, and other external requirements.

The scope of the system architecture evaluation process is as follows:

- Verify that the system architecture contains a full identification of the system to which it applies and its mission;
- All the documents and information sources referenced are identified;
- The system architecture model diagrams contain the system inputs and outputs, the main system responses, and the handling of improper inputs;
- System and subsystem building blocks or parts and their relations are modeled;
- Verify the two-way traceability between each system building block identified in the system architecture and the system requirements allocated to it;
- Verify how quality attribute requirements are implemented in the architecture as design decisions and patterns based on the heuristics applied.

12.2.2 Diverse Practices for Architecture Evaluation

This section identifies the practices—some that are manual and human-intensive, and other that are automated—that can be applied in the system architecture evaluation to meet the evaluation scope described above (see Figure 12.4).

Reviews are formal meetings with the participation of the diverse stakeholders that may be the system architects, developers, maintainers, integrators, testers, performance engineers, security expert project managers, and other relevant parties. The objectives for the review are based on the participant stakeholder concerns and focus on specific aspects of the architecture. The use of checklists and a specific process for the architecture review are very advisable.

The use of metrics is well known in software development. The use of metrics requires the presence of a design or implementation artifact on which to take the measures. The literature about software metrics is extensive [12–16]. The older

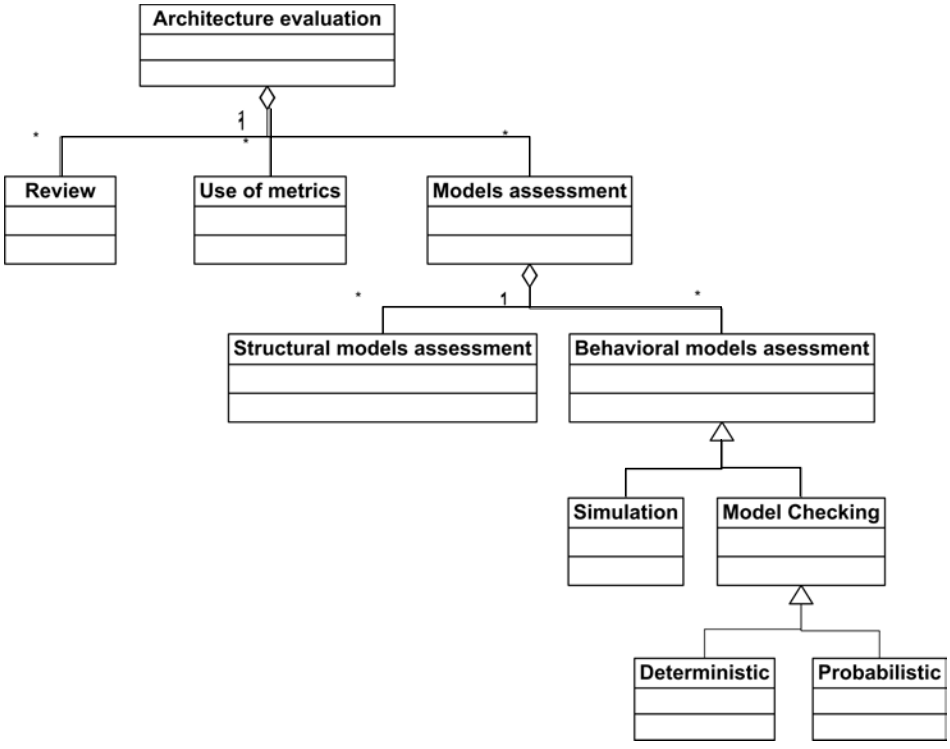


Figure 12.4 A framework of system architecture evaluation practices.

software metrics are used for coding artifacts measurements. When the object-oriented paradigm emerged, new metrics were used to evaluate the quality of object-oriented software architectures. The first author used them in the domains of avionics and air traffic control systems.

Some software engineering metrics can also be used for the systems engineering models. As an example we propose two metrics for afferent and efferent coupling adopted and adapted from object-oriented design (see Table 12.1).

An example of hybrid practice that combines reviews and measurements is the architecture trade-off analysis method (ATAM) [17]. The main goal of ATAM is to evaluate nonfunctional quality attributes of the architecture; for example, those

Table 12.1 Example of Coupling Metrics Used for Instability

Afferent Coupling (Ca)	
Artifact	Module M
Description	Total number of external modules couple to module´s parts due to incoming coupling. That is, the number of modules that have parts depend on module M.
Efferent Coupling (Ce)	
Artifact	Module M
Description	Total number of external modules couple to module´s parts due to outgoing coupling. That is, the number of modules that have parts in module M depend on.
Instability	I = Ce/(Ce + Ca)

related to its efficiency and maintainability properties. Other nonfunctional quality attributes may be evaluated following this process if the required inputs are available. The proposed process for architecture evaluation is presented here as a sequence of activities, but it can be executed in several iterations since different groups of scenarios may be selected to be validated.

- *Activity 1*: Presentation of the evaluation.
- *Activity 2*: Presentation of the system to be evaluated.
- *Activity 3*: Presentation of the architecture.
- *Activity 4*: Generate the quality attribute utility tree. A utility tree is primarily used to understand how the architects perceive and handle quality attribute architectural drivers.
- *Activity 5*: Brainstorm and prioritize scenarios. Here scenarios are used to represent stakeholders' interest and understand quality attribute requirements.
- *Activity 6*: Evaluate scenarios.
- *Activity 7*: Present the results.

The third group of evaluation practices related to models assessment are described next.

12.2.3 Model Assessment

One of the main deliverables related to architecting the system is the model of the system architecture represented by SysML structural and behavioral diagrams, and UML diagrams for the structure and behavior of the software intensive subsystems.

As stated by Friedenthal, a good model is the model that meets its intended purpose [9]. A model must be complete relative to its breadth, which system parts need to be modeled, its depth, level of design hierarchy considered, and its fidelity determining the required level of detail [9].

The structural model assessment is based on the concept of a well-formed model where the model conforms to the building constraints imposed by the standard (e.g., SysML) or the building constraints imposed by the architectural framework used. ISE&PPOOA for example has the building constraints imposed by SysML for the system structure diagrams (BDDs and IBDs), and the building constraints of UML and PPOOA for the class diagrams and the PPOOA architectural diagram of the software architecture. PPOOA building constraints (composition and dependency relations) were summarized in Chapter 7. These building or structural constraints must be enforced by the modeling tools [18].

In the context of behavioral model assessment, two main groups of techniques are considered: simulation and model checking. It is not the purpose of this section to be a survey of all the simulation and model checking techniques offered for systems architecture assessment but to provide some illustrative examples.

Simulation execution of the SysML behavioral diagrams does not suffice for the assessment of behavior because simulation can only reveal the presence of errors but cannot prove their absence. In the case of PPOOA, we developed two tools for simulation of behavioral models represented in UML.

The first one is called PPOOA-Cheddar [19], because it combines the PPOOA method and tool with the Cheddar tool for simulation and evaluation of real-time systems [20]. When the software architect has completed the software architecture models in the PPOOA-Visio tool, he or she can execute the PPOOA-XML add-on. This add-on performs the identification of the different components of the system architecture and the dependencies among them. The architecture is described in an XML file that is used as an input to the Cheddar tool. Cheddar offers a simulation engine that allows the performance engineer to describe and run simulations of the architected system. When the simulation is executed, Cheddar determines the following for each system task and during the simulation time: the number of task preemptions, the number of context switches the blocking times, and the missed deadlines.

The second tool xalled Deadlock Risk Evaluation of Architectural Models (DREAM) was developed to assess deadlock situations in real-time systems architectural models. PPOOA was used to represent platform-independent models of the software architecture to be assessed [21].

Model checking techniques allow tracking the behavior and provide a faithful assessment based on desired specifications of a given property. While the assessment of structural diagrams can be performed directly, behavioral diagrams need to be transformed into semantic or computable models. SysML/UML activity or state diagrams are endowed with formal semantics. Thus model checking operates on the formal semantic describing the meaning of the behavioral model. The assessment consists of exploring the state space and checking whether a given property holds or fails [22].

Extended formalisms, such as probabilistic, timed automata, and Markov chains [23] enable the description of the semantics of behavioral diagrams that specify stochastic, probabilistic, and or time-constrained behavior.

12.3 Next Steps Recommended to the Reader

Here we would like to give to the reader some recommendations for the deployment of the ISE&PPOOA method in his/her organization. By organization, we mean either a group of engineering students, a group of researchers in a research center, or a small-and medium-sized enterprise (SME). Examples of these groups are the contributors to the examples presented in the book. Chapter 8 was based on the development of a SME developing fixed-wing UAVs, Chapter 9 was based on the development of a robotics research center, and Chapter 10 was based on the final degree works of university engineering students. Based on these experiences and previous experiences in larger companies, we can identify obstacles and recommendations for the deployment of a MBSE approach and particularly ISE&PPOOA described in this book:

1. *Identify the needs of the group*. It is important to know how systems engineering is currently being practiced to identify improvement goals. In the case of ISE&PPOOA deployment some typical improvement goals we found are those related to requirements flowdown, functional architec-

ture, nonfunctional requirements, architecture synthesis, or functional and physical interfaces.

Here the main obstacle is an organization feeling that MBSE is not required for their own products.

2. *Plan and execute a pilot and demonstration project.* Apply the ISE&PPOOA method to a pilot project; for the scope of the pilot project may be a subsystem of a larger system as well (see Chapters 8 and 9 examples). A pilot project helps understanding the methodology and creates trust. The pilot project should exercise the ISE&PPOOA method and the MBSE tool. Here the most important issue is to mentor people during the pilot project. SysML notation should be as simple as possible and the MBSE tool selected for the pilot should be easily adopted. It is better to put the main effort in the ISE&PPOOA methodology understanding.

   The main obstacle is that mentoring by experts can be difficult or expensive for some organizations.

3. *Identify how to integrate the ISE&PPOOA process with the technical and management process of the organization.* Here the analysis of the diverse dependencies the organization can face in a product development project is the main issue. These dependencies are, for example, knowledge dependencies, development process dependencies, and product dependencies [24].

   Organizations with low maturity levels in project management and knowledge management are an obstacle for this integration.

4. *Master plan of action.* Based on the existing organization structure, people competences, and tools create a master plan of the activities to implement the ISE&PPOOA process step by step. Always remember that people are the main concern, so people training and empowerment are very important.

   The main obstacle is that some organizations do not understand that MBSE requires high up-front expenditure but there is a relation between this expenditure and the resulting benefits [25].

12.4 Summary

This chapter summarized topics that we consider important for a systems engineer developing complex products, but for book scope and length reasons were not addressed in the core chapters of the book. The explanations and references provided here will allow the reader to expand his/her knowledge in these interesting topics.

The deployment of a MBSE methodology in an organization is always a challenge, and this chapter has provided some recommendations to help the reader.

References

[1] Beck, K., et al., *Agile Manifesto*, Agile Alliance, 2001.

[2] Carlson, D., "Debunking Agile Myths," *CrossTalk, The Journal of Defense Software Engineering*, May–June 2017, pp. 32–37.

[3] Ebert, C., and M. Paasivaara, "Scaling Agile," *IEEE Software*, November–December 2017, pp. 98–103.

[4] Sutherland, J., *The Scrum@Scale Guide, The Definitive Guide to Scrum@Scale: Scaling that Works*, Scrum Inc., 2018.

[5] Schwaber, K., and J. Sutherland, *The Scrum Guide, The Definitive Guide to Scrum: The Rules of the Game*, November 2017.

[6] Ambler, S., and M. Holitza, *Agile for Dummies,* Hoboken, NJ: John Wiley & Sons, 2012.

[7] Leffingwell, D., et al., *SAFe Reference Guide*, Pearson Education/Scaled Agile Inc., 2018.

[8] Douglass, B. P., *Agile Systems Engineering*, Waltham, MA: Morgan Kaufmann, 2016.

[9] Friedenthal, S., A. Moore, and R. Steiner, *A Practical Guide to SysML*, Burlington, MA: Morgan Kaufmann, 2008.

[10] Blaci, O., et al., "Planning for Verification, Validation, and Accreditation of Modeling and Simulation Applications," *Proc. of the 2000 Winter Simulation Conference*, Orlando, FL: December 2000.

[11] Engel, A., *Verification, Validation, and Testing of Engineered Systems*, Hoboken, NJ: John Wiley & Sons, 2010.

[12] Abreu, F. B., "The MOOD Metrics Set," *ECOOP '95 Workshop on Metrics*, 1995.

[13] Basili, V. R., L. C. Briand, and W. L. Melo, "A Validation of Object Orient Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, Vol. 21, 1996, pp. 751–761.

[14] Briand, L. C., J, W. Daly, and J. K. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems,' *IEEE Transactions on Software Engineering*, Vol. 25, January–February 1999, pp. 91–121.

[15] Chidamber, S. R., and C. F. Kemerer, "A Metric Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, June 1994, pp. 467–493.

[16] Churches, N. I., and M. J. Shepperd, "Comments on 'A Metrics Suite for Object-Oriented Design,'" *IEEE Transactions on Software Engineering*, Vol. 21, 1995, pp. 263–5.

[17] Clements, P., R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Indianapolis, IN: Addison Wesley/Pearson Education, 2002.

[18] Fernandez-Sanchez, J. L., and J. C. Martinez-Charro, "Implementing 'A Real Time Architecting Method in a Commercial CASE Tool,'" *Proc. 16th International Conference on Software and Systems Engineering (ICSSEA)*, Paris, France, December 2003.

[19] Fernandez, J. L., and G. Marmol, "KR10 An Effective Collaboration of a Modeling Tool and a Simulation and Evaluation Framework," *Proc. INCOSE International Symposium*, Vol. 18, 2008, pp. 1509–1522, doi:10.1002/j.2334-5837.2008.tb00896.x.

[20] Singhoff, F., J. Legrand, L. Nana, and L. Marcé, "Cheddar: A Flexible Real Time Scheduling Framework," *Proc. of the ACM SIGAda International Conference*, Atlanta, November 2004, pp. 15–18

[21] Monzon, A., J. L. Fernandez, and J. A. de la Puente, "Application of Deadlock Risk Evaluation of Architectural Models," Software: Practice and Experience, Vol. 42, 2012, pp. 1137–1163, doi:10.1002/spe.1118.

[22] Debbadi, M., et al., *Verification and Validation in Systems Engineering: Assessing UML/SysML Design Models*, Berlin: Springer Verlag, 2010.

[23] Balsamo, S., et al., "Model-Based Performance Prediction in Software Development: A Survey," *IEEE Transactions on Software Engineering*, Vol. 30, No. 5, May 2004.

[24] Martinez Leon, C., J. L. Fernandez, and J. A. Cross, "Taxonomy on Product Dependencies for Project Planning," *Proc. of the 2018 IISE Annual Conference*, K. Barker, D. Berry, C. Rainwater, Eds., 2018.

[25] Madni, A. M., and S. Purohit, "Economic Analysis of Model-Based Systems Engineering," *Systems*, Vol. 7, No. 12, 2019.

APPENDIX A

SysML Notation

This appendix provides a summary of the SysML notation used in the ISE&PPOOA methodology.

A.1 Use of SysML in the ISE&PPOOA Methodology

The ISE&PPOOA uses SysML and an extension of it as the modeling language to represent the outcomes in the different steps of the ISE&PPOOA methodology. In the following sections, a description of these modeling elements and diagrams is provided together with an explanation of their use in the methodology. For a detailed account of the SysML modeling language, the reader is referred to [1] and for a handy explanation of how to use SysML [2]. Friedenthal et al. [3] and Weilkiens et al. [4] also provide a detailed discussion of the use of SysML.

Table A.1 summarizes the different SysML diagrams and elements used in the different steps of ISE&PPOOA, with references to corresponding figures throughout the book.

A.1.1 SysML Diagrams in ISE&PPOOA

From the nine types of diagrams in SysML (see Figure 3.2 in Chapter 3), ISE&POOA makes intensive use of three:

1. Block definition diagram;
2. Internal block diagram;
3. Activity diagram.

A SysML diagram contains diagram elements (mostly nodes connected by paths) that represent elements in the SysML model, such as activities, blocks, and associations [1]. Most diagrams in SysML come from UML [7]. Since ISE&PPOOA was initiated before SysML, some of the diagrams (e.g., class diagrams, component diagrams) are based on UML. However, the corresponding SysML diagram (e.g., block definition diagram) can be used instead if preferred.

Table A.1 ISE&PPOOA Outputs and Diagrams and SysML Notation Used

ISE&PPOOA Step and Deliverable	*SysML Diagram*	*Example Figures*
1. Identify operational scenarios	(optional)	
• Context diagram	Block definition diagram or internal block definition diagram	8.3, 9.3
• Use cases		
• Scenario	Use case diagram	3.2
	Activity diagram (optional, in some cases it is captured in textual form)	9.4
2.a. System capabilities with a hierarchical decomposition	Block definition diagram	9.1
3.1,3.2, and 3.3 Functional architecture: functional hierarchy	Block definition diagram	8.5, 8.6, 8.7, 9.9, 10.3
3.3 Functional architecture: functional flows or behavior	Activity diagrams	8.8, 8.9, 8.10, 9.5, 9.6, 9.7, 9.8, 10.4
4.1 Allocation	Block definition diagram	9.13
4.2 Modular architecture–structural view	Block definition diagram	9.10, 9.12, and 10.5
	Internal block definition diagrams	
4.4 Physical architecture–behavioral view	Activity diagrams with allocation activity partitions	8.15, 8.16, 8.17, and 9.14
ISE&PPOOA/energy		
Step 2 Functional and Physical architecture– structural view	Block definition diagrams are used for the functional and physical hierarchies	10.3 and 10.5
Step 2 Functional and Physical architecture– behavioral view	Activity diagrams are used for functional flows	10.4
Step 3 Flows of matter and energy	Internal block definition diagrams with item flows	10.6
Step 4 Write equations, equalities, and correlations	Block definition diagram with constraints	10.7, 10.8, 10.9, and 10.10
PPOOA		
1. Domain model	UML class or SysML block definition diagram	9.19
2.a.1 Structural view of the software architecture	PPOOA architecture diagram (UML profile)	9.20
2.b Software subsystem behavior	UML/SysML activity diagram for each CFA	9.21, 9.22, 9.23
2.c Coordination mechanisms	Coordination mechanisms added as stereotyped elements to the PPOOA architecture diagram	9.20

The most notable difference between UML and SysML diagrams is that SysML diagrams have a frame with a contents area, a heading, and a diagram description [1]. The heading has the following syntax [2]:

diagramKind [modelElementType] modelElementName [diagramName]

Figure A.1 shows the diagram frame for the block definition diagram that captures the functional hierarchy of the collaborative robotic system from Figure 9.9 in Chapter 9.

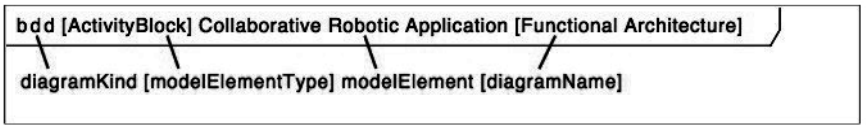


Figure A.1 Diagram frame example.

A.2 SysML for the ISE&PPOOA Structural Perspective

A.2.1 Blocks and Block Definition Diagram

The most used SysML diagram in ISE&PPOOA is the BDD. A SysML *block* corresponds to an ISE&PPOOA *part* (Chapter 4, Figure 4.1) (i.e., a building element of the system). BDDs are used to convey system information about a system's structure, such as decomposition, associations, dependencies, or type classification[1]. In ISE&PPOOA, they are mainly used to convey decomposition (e.g., functional and physical hierarchies). A SysML block defines a type [2] that defines a number of features and properties [5]. That is, SysML blocks represent the types of elements that your system is built of, both physical (e.g., motors, sensors, pumps, electronic circuits) and logical (e.g., sensor driver, graphical user interface application).

Note that there is a special type[2] of SysML block, the SysML activity, which is intensively used in ISE&PPOOA to model functions. This will be discussed in Section A.3.

In SysML, blocks can be related through three kinds of relationship: association, dependency, and generalization.

Associations (lines in a BDD connecting blocks) are a way in SysML to represent the structural relationships within a system, with block properties (as compartments in the block rectangle) being the alternative way [2]. ISE&PPOOA uses mainly

- *Composite associations* between blocks to represent hierarchical decomposition of functions and physical structure of the system. Composite associations are intensively used in ISE&PPOOA for the functional and physical hierarchies. For an example see all the composite associations in Figure A.2.
- *Reference associations* between blocks to indicate that the instances of the blocks are connected and can access each other for some purpose across the connection [2]. ISE&PPOOA stresses more the direct modeling of the connections in internal block diagrams through ports and flows (see Section A.2.2).

Dependency relationships convey the idea that one element in the system depends on another one to deliver its functionality. Dependency associations are denoted with a dashed line with an arrowhead. In ISE&PPOOA dependencies can be used, for example, in the BDD diagram for the modular architecture of the system.

1. The model elements that you display on BDDs serve as types for the other model elements that appear on the other kinds of SysML diagrams [2].
2. SysML uses UML profiling mechanism to define special types. For example, a SysML activity is a special type of SysML block with the stereotype activity.

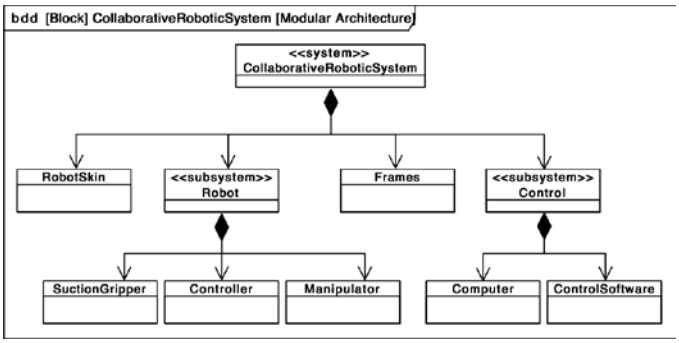


Figure A.2 Example of block definition diagram: the physical architecture of the robotic application seen in Figure 9.11 from Chapter 9.

However, just as with reference associations, ISE&PPOOA stresses the modeling of the connections through ports and flows in internal block definition diagrams for the refined architecture, where dependencies are better refined into required and provided interfaces (see Figure 8.18 for examples). A type of dependency, with the UML stereotype «use» is used in the PPOOA architecture diagram (see Section A.5).

Figure A.2 shows an example of a block definition diagram from Chapter 9: the modular architecture of the collaborative robotic system.

Blocks are displayed as rectangles with a name compartment and an empty properties compartment (e.g., RobotSkin).

The composite associations are represented by solid lines with black diamonds on the composite end. An open arrowhead on the part-end indicates unidirectional access (i.e., the block can access its constituent parts, but the part cannot access its composite), where its absence indicates bidirectional access. At the first two levels of the physical hierarchy diagram, elements are virtual and therefore there is no consequence of using one notation or the other. This is not the case for the lower levels, where blocks correspond to actual system parts and the unidirectional or bidirectional access has an actual impact on the solution designed. Also, note the «system» and «subsystem» stereotypes applied to certain blocks, which provide additional information about the nature of them. Their use is suggested but not mandatory in ISE&PPOOA.

A.2.2 SysML Internal Block Diagram: Parts, Ports, Connectors, and Flows

The internal block diagram (IBD) in SysML is a view of the internal structure of a block. In ISE&PPOOA it is used to capture the physical architecture in terms of parts and connectors between parts. In an IBD, the ISE&PPOOA subparts that constitute a part (SysML block instance) are represented as SysML property parts or simply parts. A SysML part is a typed slot into which an instance will play the role provided by the slot [5]. For a detailed discussion of the difference between SysML block, part, and instance, see [2, 5].

A *connector* between two parts on an IBD conveys that the two parts will have some way to access each other [2]. The connector can be given a name and a type to add more information about the medium that connects the two parts.

Ports specify allowable types of interactions between the parts [1], adding restrictions and specification to the general interaction represented by the connector.

SysML *flow* properties specify the kinds of items that can flow between two parts through a connector, whereas item flows specify what does actually flow in a particular usage context [1]. Item flows are represented with a filled triangle.

Ports can be typed with *interfaces,* which defines a set of operations, and receptions, a behavioral contract for a port. A provided interface is denoted by a lollipop symbol, whereas a required interface uses the socket notation.

The modeling of ports has evolved with the different versions of the SysML language. For simplification, ISE&PPOOA recommends the use of the proxy port for the latest SysML specification (v1.5) that can account for both flows and operations specifications.

Figure 8.18 shows an IBD that models the parts, connectors, and flows of the electrical subsystem in the UAV example from Chapter 8. The electrical subsystem operation includes its own component parts (represented with solid-edge rectangles, such as the *power source element*), whereas reference properties (structures external to the *electric subsystem* block) are denoted with dashed-edge rectangles (e.g., *AvionicsSubsystem.sensor*).

The flows of energy are represented through ports with an arrow that indicate the direction of the energy flow and connectors with a flow item (e.g., the connectors and associated ports in the *power source element*). The information flows are represented through ports types with interfaces and connectors with associated item flows that indicate the information that is being communicated (e.g., the *data bus:Signal Distribution Element* has two ports with required interfaces to send payload command and switch on/off information, and another port offering an interface to receive *commands*).

A.3 SysML for the ISE&PPOOA Behavioral Perspective

The most important SyML behavioral view in ISE&POOA is the activity diagram (ACT). An activity diagram expresses sequences of behaviors and event occurrences over time [2], and emphasizes the inputs, outputs, sequences, and conditions for coordinating those other behaviors [1].

ISE&PPOOA makes intensive use of the activity diagram instead of other behavior diagrams, such as sequence diagrams and state machine diagrams, because they are the best way to model the functional flows and system behavior. Activity diagrams express the order in which actions are performed in a behavior and they can also express allocation: activity partitions allow to represent which structure (part) performs each action (function) [2].

Note that activity and activity diagram are not synonyms [2]. The nodes and edges that an activity diagram typically contains correspond to actual elements in the model. They are contained in the activity model element of which the activity diagram is only a particular view.

Edges connect nodes to form ordered sequences in an activity. There are two types of edges:

- Object flow is an edge through which instances of matter, energy, or data flow through an activity from one node to another when the activity executes;
- Control flow, the most used edge in ISE&PPOOA, conveys the order of execution of the nodes in an activity diagram.

In ISE&PPOOA, the solid line with an open arrowhead for both edges is used since in ISE&PPOOA activity diagram edges usually represent control flows.

A.3.1 Activity Nodes

An *action node* models a basic unit of functionality within the activity, which is a transformation that will occur when the activity gets executed during system operation [2].

A *call behavior action* is a specialized action that invokes another behavior when it becomes enabled [2]. A rake symbol in the lower-right corner of a call behavior action indicates that the behavior getting called is an activity. This is typically the case in ISE&PPOOA activity diagrams, and it indicates that the function corresponding to the call behavior action node is further decomposed as an activity in another AD (see examples in Figure 9.5: F5 Move robot safely to stack, F6 Pick product, F4 Manage stacks, and F7 Deliver product).

In ISE&PPOOA, an action represents the usage of a function (see Chapter 5), a particular execution of a function in the context of the encompassing activity. We capture this by using the function name for the actions in the activity diagram. Figure 9.5 in Chapter 9 shows the different functions involved in the main functional flow of the operation of the robotic system. Note that high-level functions are call behavior actions, indicating that the function behavior is detailed in another activity diagram.

SysML includes special actions to represent distributed and concurrent behavior that are used in ISE&PPOOA:

A *send signal action* asynchronously generates and sends a signal instance to a target when it becomes enabled [2].

An *accept event action* is the partner of the send signal action in asynchronous behaviors; it indicates that the activity must wait for an asynchronous event occurrence before it can continue its execution. Typically, that asynchronous event occurrence is the receipt of a signal instance [2].

A *wait time action* waits for a time event occurrence is called a and is represented (e.g., see the skin timer node in Figure 9.5).

A.3.1.1 Interruptible Activity Region

An interruptible region is designated by a dashed line and an interrupting edge connected to an accept event action that represents the interruption. It indicates that when the interruption is raised, the node within the region in execution terminates,and the control flow follows the interrupting edge instead of the execution flow following the terminated node.

An interruptible region contains activity nodes. When the flow exits an interruptible region via edges designated by the region as interrupting edges (notated with a lightning bolt), all behaviors in the region are terminated [7].

An example is the causal flow of the activity "process order," for the collaborative robot application in Chapter 9, Figure 9.23. If the sensed obstacle event is triggered (interruption) while execute trajectory to product stack is in execution, that activity is terminated and cancel execution is executed next instead of activate gripper.

A.3.2 Control Nodes

Control nodes in ADs are used to guide the flow of execution in the activity.

Initial and *final* nodes mark the starting and final points within an activity [2]. The initial node is denoted with a filled circle. We have to distinguish two types of final nodes. The flow final node marks the end of a single flow of control, whereas an activity final node marks the end of all flows of control (no matter where they are currently in their execution) and the entire activity terminates [2]. The notation for a flow final node is a circle containing an X. The notation for an activity final node is a circle that contains a smaller, filled-in circle.

A decision node denotes alternative flows in an activity. The notation is a hollow diamond (e.g., Figure 9.14 "is empty" node). A merge node indicates the end of alternative flows in an activity. The notation is also a hollow diamond.

A fork node and a join node indicate the start and end of concurrent flows in an activity, respectively. The notation for both is a line segment.

Some typical constructs of the classical EFFBD can be easily mapped to SysML activity diagrams using control nodes [6].

The activity diagram in Figure 9.14 in Chapter 9 provides an example of the use of control nodes. If during operation, when the control subsystem updates the state of the stacks of products, the stack of the last product picked is empty (decision node), the robot moves to the home position, the robot skin notifies the situation through the visual LED signal empty and waits to receive the signal ready from the operator, who is responsible for replacing the empty stack and notifying it to the robot by pressing a cell in the robot skin.

Note that SysML also includes state machine diagrams which, as mentioned in Chapter 4, are recommended in ISE&PPOOA when the system has modes of operation.

A.4 Other SysML Elements and Views in ISE&PPOOA

A.4.1 Allocation

Allocations describe ways to allocate behavior to structural elements and thus are cross-cutting relationships [2]. In SysML, the allocation relationship can provide an effective means for navigating the model by establishing cross relationships and ensuring the various parts of the model are properly integrated [1]. SysML provides a basic capability to support allocation in the broadest sense [1].

SysML supports different ways to represent allocation on diagrams, in compartment, as a relationship and tool vendors usually support more traditional formats, such as tables.

Next, we present how SysML allocations are used in the ISE&PPOOA methodology. In ISE&PPOOA, the most important SysML mediums to represent allocation are the use of block compartments, allocation activity partitions, and matrices. The matrix format is implemented differently in the various modeling tools; SysML does not define a particular format. It is a compact format that is convenient to display a large number of allocations.

An example of the use of allocation compartments can be found in Chapter 10, Figure 10.6, part of which is also reproduced in Figure A.3.

Allocation relationships can also be represented as a dashed line with an open arrowhead and the «allocate» stereotype applied to it (see Figure A.4). The element being allocated appears at the tail end of the line; the element receiving the allocation appears at the arrowhead end of the line [2].

Allocation activity partitions are a mechanism to allocate behaviors to structures. The behavior element is typically an action. The structure element can be either a block or a part. Placing an action in an activity partition conveys that the structural element represented by the activity partition is responsible for performing the contained actions. If the structural element is a part, it means that that part is responsible to perform the contained actions. If it is a block, it means that all instances of that block perform the contained actions. An example of allocation activity partitions is Figure 9.23.

A.4.2 Use Case Diagram

Use case diagrams express information about the services your system provides and the stakeholders who require those services [2]. In ISE&PPOOA, use case diagrams can be used to summarize the system scope and involved stakeholders in the initial step of identifying the operational scenarios (see Appendix B, Section B.4), as exemplified in Figure 3.2. Use case and scenario are not synonyms. Each path of

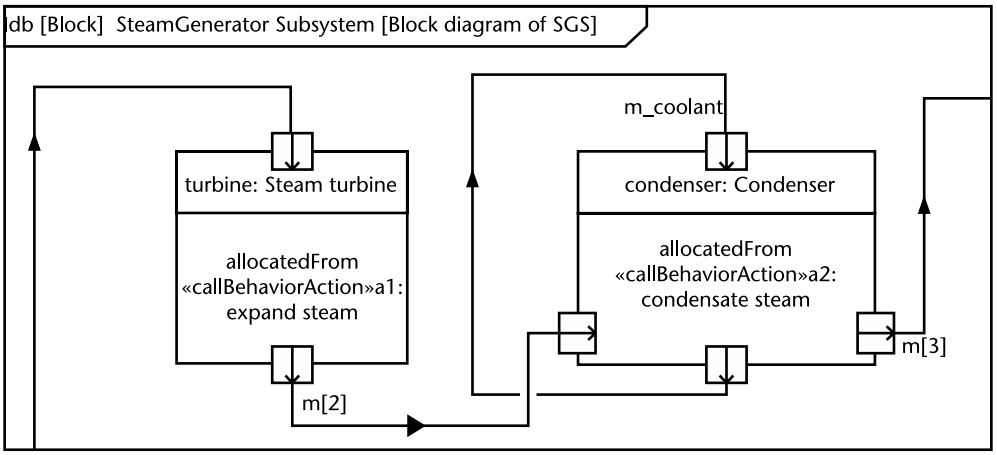


Figure A.3 Allocation compartments in the parts represented in the IBD of the steam generator subsystem seen in Chapter 10, Figure 10.6.

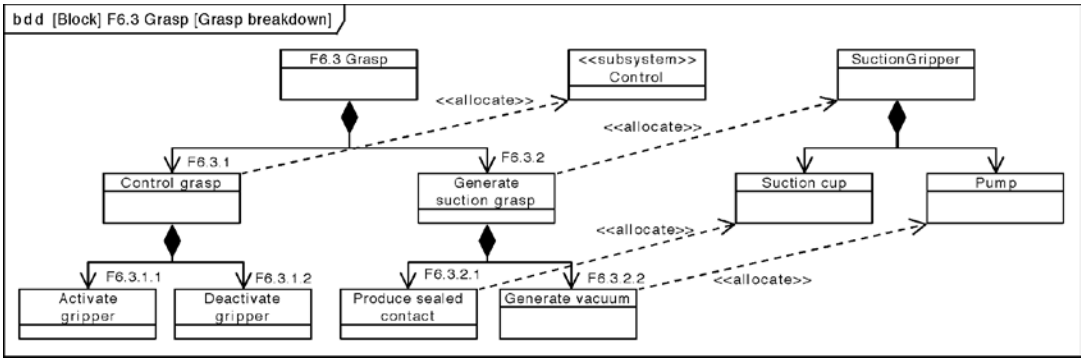


Figure A.4 Allocation relationships capture the allocation of the functions related to grasping to the gripper and control elements in the physical architecture of the robotic application seen in Chapter 9, Figure 9.12.

execution through a use case from beginning to end is a distinct scenario. A use case therefore consists of one or more scenarios.

A.4.3 Constraint Blocks and Parametric Diagrams

SysML Constraint blocks allow to integrate engineering analysis, such as performance and reliability models, with other SysML models, identifying critical performance parameters and their relationships to other parameters, which can be tracked throughout the system life cycle [1].

In ISE&PPOOA/energy, SysML constraint blocks are used to show how the properties associated with the flows of matter and energy in the system are constrained. Block definition diagrams with constraint blocks, which we will call constraint diagrams in ISE&PPOOA, show the relationships or equations between variables and value properties of interest of the blocks or parts.

ISE&PPOOA does not commit to a specific modeling tool to bind the parameters of a constraint to a particular situation for analysis. SysML parametric diagrams can be used for this purpose. Parametric diagrams include usages of constraint blocks to constrain the properties of another block by binding the parameters of the constraint to specific properties of the block, which provide values for the parameters [1].

A.4.4 Requirements

The ISE&PPOOA approach to requirements engineering manages textual requirements and requirements models to facilitate requirements extraction and requirements analysis.

SysML provides a diversity of modeling constructs to represent text-based requirements and relate them to other modeling elements [1]. For example, the requirements diagram can depict the requirements in graphical, tabular, or tree structure format. In ISE&PPOOA, a traditional approach in a tabular format is preferred to specify the requirements in text form. However, it encourages showing the relationships of requirements to other modeling elements on diagrams, as supported by SysML. SysML provides either the trace, refine, or satisfy relationship between the requirement and the model element. The trace requirement

relationship provides a general-purpose relationship between a requirement and another model element, with no semantics. Therefore ISE&PPOOA recommends using the refine and satisfy relationships. Figure 9.15 shows an example of satisfy relationships that relate requirements to elements in the physical architecture of the robotic application from Chapter 9.

Requirements model or models of specification are a more rigorous approach to engineer consistent requirements. ISE&PPOOA uses the following SysML diagrams for requirement modeling: use case diagrams, activity diagrams, and state machine diagrams (see Appendix B).

A.5 Complementing SysML: PPOOA Architecture Diagram

As discussed in Chapters 4 and 7, for the architecting of the software subsystem PPOOA uses UML/SysML activity diagrams to address the behavioral viewpoint, but it incorporates a new diagram for the structural view of the software subsystem: the PPOOA architectural diagram. The PPOOA architectural diagram is an extension of the UML class diagram with PPOOA stereotypes for the PPOOA components and coordination mechanisms [8] discussed in Chapter 7 and it is used in the robot example in Chapter 9 (see Figure 9.20).

References

[1] OMG, *OMG Systems Modeling Language,* Version 1.5, technical report, OMG, May 2017.

[2] Delligatti, L., *SysML Distilled: A Brief Guide to the Systems Modeling Language,* Upper Saddle River, NJ: Addison-Wesley, 2014.

[3] Friedenthal, S., A. Moore, and R. Steiner, *A Practical Guide to SysML*, Third Edition, Waltham, MA: Morgan Kaufmann, 2015.

[4] Weilkiens, T., J. Lamm, S. Roth, and M. Walker, *Model-Based System Architecture*,, Hoboken, NJ: John Wiley & Sons, 2015.

[5] Douglass B., *Agile Systems Engineering*, Upper Saddle River, NJ: Morgan Kaufmann, 2015.

[6] Bock, C., "SysML and UML 2 Support for Activity Modeling," *Systems Engineering*, Vol. 9, No. 2, 2006.

[7] Object Management Group, *OMG Unified Modeling Language (OMG UML),* Version 2.5.1, technical report formal/2017-12-05, Object Management Group, December 2017.

[8] PPOOA, Processes Pipelines in Object Oriented Architectures, http://www.ppooa.com.es

APPENDIX B

Requirements Framework

In the ISE&PPOOA methodology for systems development, requirements play a fundamental role. In particular, the various ISE&PPOOA deliverables, such as functional and physical architectures, are mainly based on the requirements that are specified by the requirements engineering discipline and are largely dependent on the quality characteristics of these particular requirements or the sets of requirements.

Generally, natural language is what is used to express requirements. Natural language may be ambiguous but ambiguity can be avoided using requirements templates or boilerplates and writing clear and precise requirements statements where requirements are quantified.

How requirements are expressed differs through the systems engineering life cycle. As the system is engineered and solutions designed down through the levels of abstraction, we expect requirement statements to become more and more specific.

Modeling of requirements for specification purposes is a new way where diagrams in standard notations; for example, UML or SysML, replace or refine textual requirements in natural language. Section B.4 describes which models are recommended and how they can be used to extract or refine textual requirements.

B.1 Needs, Capabilities, and Requirements

Here we will briefly explain three main concepts already defined in the conceptual model of the ISE&PPOOA methodology presented in Chapter 4, but consider how they are used during system development.

First, we consider the mission dimension where the users have some needs that are identified using mission scenarios describing how the system is used, operated, maintained, and other interactions of interests. Therefore, needs, as defined in Chapter 4, are the answers to the question "What problem are we trying to solve with the new system operating in a particular environment?" Here we can call them operational needs or user needs. The *INCOSE Guide for Writing Requirements* considers needs as the expectations stated in the language of business [1].

The second concept used in the mission dimension is capability. In Chapter 4, a capability was defined as the ability to perform an effect under specified standards

and conditions through combinations of means and ways to perform a set of tasks. For the sake of understandability, we can see an analogy between the capability concept for a system and the competency concept for a person. For achieving his/her complex professional task an employee must have some competencies that are typically demanded in the particular job position; for example, systems thinking is a systems engineer competency. Similarly a system must have the capabilities that are needed to perform its mission; for example, long endurance is a capability in an UAV performing a wildfire prevention mission.

Based on the mission operational context and scenarios, the engineer translates the set of specific needs into a set of system capabilities that should be solution-independent. Each capability is a container of system properties that may be either system quality attributes, physical properties, states, or functions.

In Chapter 4, we define a requirement as the statement of a property that a system or one of its parts shall exhibit.

INCOSE Guide for Writing Requirements defines a requirement as the result of the transformation of one or more needs into a function or a quality attribute of some entity or single thing the requirement refers [1].

ISO/IEC/IEEE 29148 defines a requirement as a statement that translates or expresses a need and its associated constraints and conditions; in other words, a constraint is an externally imposed limitation on system requirements, design, or implementation, or on the process used to develop or modify a system and a condition is a measurable qualitative or quantitative attribute that is stipulated for a requirement [2].

B.2 Requirements Classification

In the conceptual model of the ISE&PPOOA methodology presented in Chapter 4, four types of requirements are considered: functional requirements, state requirements, nonfunctional requirements, and physical properties. This classification is based on how these requirements are taken into consideration when designing the solution represented by the architecture. Generally, functional requirements are allocated to the physical elements that perform the corresponding functionality, but nonfunctional requirements are not allocated in the same way since they represent emergent properties of the system and so allocation is not always possible. We therefore propose the use of heuristics to implement them as described in Chapter 6.

Here we extend the types of requirements shown in Figure 4.1 with the requirements classification presented in Figure B.1 that includes interface requirements and constraints as well.

Behavioral requirements describe the behavior the system or part must exhibit under specific conditions. Behavioral requirements are mainly the functional requirements that are related to the outputs of a function under specific conditions. In other cases, such as reactive systems, behavior is better modeled as states and transitions that may be expressed by textual requirements statements as well.

Frequently functions transform data so data requirements are needed to complement some functional requirements of functions transforming data. It is recommended to use a data dictionary for this purpose. We recommended using the Hatley and Pirbhai notation summarized below for the data items to be defined in

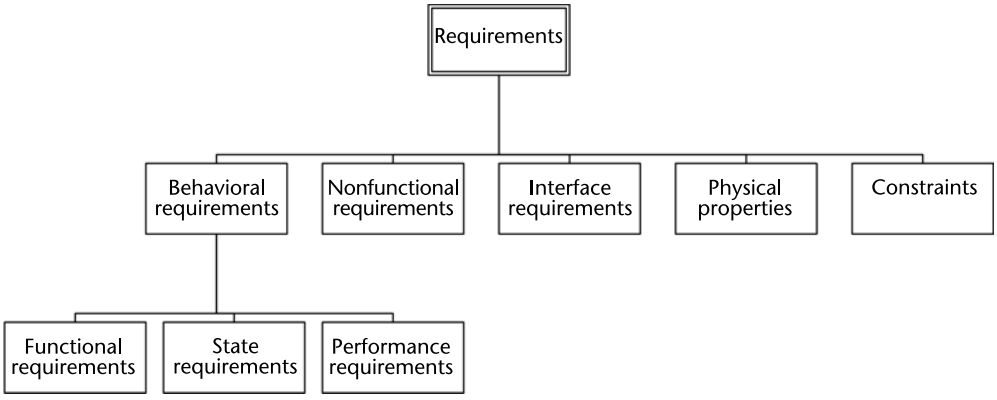


Figure B.1 Requirements classification.

the data dictionary. The data dictionary defines each data item and decomposes it further into items of its own, breaking it down until primitive items that terminate the data decomposition are identified [3]. They propose the following symbols to use in data decomposition:

- =, meaning composed of. The data item named on its left is composed of the data items named on its right.
- +, meaning together with. Collects data items into a group.
- { }, meaning iterations of. The expression enclosed with curly brackets may occur any number of times in a given instance of the data item.
- [ ], meaning select one of. The square brackets contain two or more data items separated by vertical bars.
- ( ), meaning optional. The expression enclosed with parentheses may or may not appear in a given instance of the data item.
- " " meaning literal.

Performance requirements define how the system or part fulfills its behavior. Functional and performance requirements may be combined when specified. Some authors consider performance as part of the group of nonfunctional requirements. We prefer to consider performance as part of behavior because performance as functionality can be allocated.

Nonfunctional requirements known as well as quality attribute requirements are used to specify the criteria that can be used to assess the operation or evolution of a system rather than its specific behaviors. Therefore, nonfunctional requirements are based on quality models representing the quality factor of concern and how these quality factors break down into subfactors and criteria. We will describe the quality model we proposed for specifying nonfunctional requirements below.

Interface requirements are those specifying functional or physical relationships between the system and external elements (external interfaces) or between system elements (internal interfaces). Hooks and Farry classify external interfaces into two broad categories: human interfaces and those that involve everything else [4].

Physical properties requirements are used to specify properties of mass, shape, color, temperature, and so on.

Constraints are a type of requirements that is an externally imposed limitation or restriction on the system. Constraints bound the design solutions space. They may be technical, regulatory, or environmental.

Nonfunctional requirements are specified by using the quality model shown in Figure B.2 adapted from diverse sources, such as ISO 9126 [5], Firesmith for safety [6] and Jackson and Ferris for resilience [7]. It is important to realize that nonfunctional requirements may be applied at different levels either to the complete system, a subsystem, or one of its parts.

The quality attributes or quality factors and the subfactors shown in the quality model of Figure B.2 are described below. The quality factors and subfactors allow us to propose nonfunctional requirements templates or boiler plates as those presented in Section B.5.

Reliability and maintainability are represented together. Reliability may be defined as the extent to which the system or one of its parts performs its functions without failure. Maintainability is the capability of the system or one of its parts to be modified. Modifications may include corrections, improvements, or adaptation of the element to changes in environment and in requirements.

Reliability and maintainability break down into some subfactors described below.

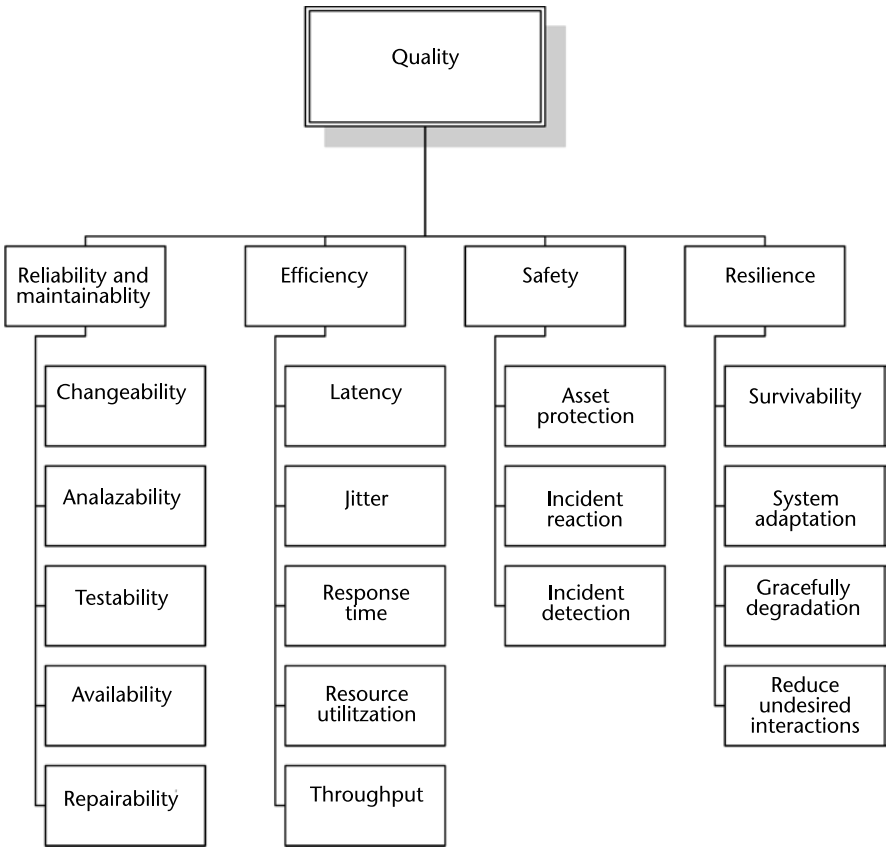


Figure B.2 Proposed quality model.

Changeability is the ability of the system or part to enable a specified modification to be implemented.

Analyzability is the ability of the system or part to be diagnosed for defects or causes of failures.

Testability is the ability of the system or part to be validated.

Availability is defined by a measure of the probability of the system to be in an operable condition at the start of a mission initiated at random times.

Repairability is the ability of a failed or damaged system or part to be restored to acceptable condition within an acceptable period of time. The MTTR defines how long it takes to repair the part on average.

Efficiency as defined by ISO 9126 [5], is the quality factor used here in the quality model. Other sources term it as performance. Efficiency is the ability of the system to provide appropriate performance, relative to the amount of resources used, under stated conditions. Efficiency breaks down into the subfactors described below.

Latency is the time delay between the cause and the effect of some physical change in the environment interacting to the system. However, Gregg defines resource latency, for example, for a storage disk, as the time interval sending the I/O request and receiving the completion interrupt [8].

Jitter is the irregularity of a time-based event response. In a reactive control system jitter implies that the system behaves in a nonperiodic manner and the system performance is degraded with respect to the expected response [9].

Response time is related to the ability of the system to provide appropriate end-to-end times when performing its mission functions under stated conditions.

Resource utilization is the ability of the system to use appropriate amounts and types of resources when the system performs its mission functions under stated conditions.

Throughput is related to the rate at which the system, part, or a particular interface must be able to perform some input or output processing.

Safety is defined by Firesmith as the degree to which accidental harm is prevented, identified, reacted to, and adapted to [6]. Therefore, he identifies four quality subfactors, three of which we adopted here because the fourth one identified by Firesmith is considered here as part of resilience.

Asset protection is the degree to which valuable assets are protected from harm. Protection is based on decreasing the likelihood of hazards and their impact.

Incident detection is the degree to which safety incidents are recognized in a timely manner by the system.

Incident reaction is the degree to which the system reacts to an incident by reporting it and activating its safeguards.

The term resilience has diverse meaning in such fields as psychology, ecology, and engineering. The U.S. government defines resilience as the ability to adapt to changing conditions and prepare for withstand and rapidly recover from disruptions [10]. Jackson and Ferris apply resilience for engineered systems. These resilience engineered systems are able to restructure during the threat encounter, withstanding the situation by retaining partial or full functionality [7]. Based on Jackson and Ferris, the following subfactors are considered in the quality model we proposed here.

Survivability is the ability of the system to deal with the threat disruption that it encounters.

System adaptation is the degree to which the system changes itself to adapt to the threat.

Graceful degradation is the ability of the system to maintain a limited functionality even when a large portion of it has been destroyed or rendered inoperative.

Reduce undesired interactions, also called hidden interactions. These interactions are caused when there is a lack of whole system design or system architecting principles as opposed to emphasizing component design [7]. Hidden interactions are the cause of diverse accidents. Jackson and Ferris describe a Nimrod aircraft accident as reported by RAF in which fuel from a leaking fuel line contacted a heated pipe, resulting in an accident [7]. Undesired interactions are reduced by reducing system complexity.

B.3 Requirements Flowdown in Systems Development

Requirements flowdown is one of the most challenging issues for the requirements engineering practitioners. The first author's experience as consultant, trainer, and educator was that mixing different level requirements in the requirements specification documents is a common problem for industry and students.

A disciplined requirements flowdown process is one that transforms requirements and designs architecture concurrently and interleave their development as shown in Figure B.3. Although this approach develops requirements specifications and architecture concurrently, it separates problem dimension (specification) from solution dimension (architecture) in an iterative process that produces progressively more solution-oriented or detailed requirements.

Figure B.3 summarizes the approach we recommend. Beginning with a set of high-level system requirements, systems engineers transform them into a functional architecture and diverse quality models for the diverse quality attributes and nonfunctional requirements.

Using functional allocation and design heuristics, systems engineers identify the main subsystems and the interfaces between them. They represent the system architecture by using SysML BDD, and IBD diagrams. For each subsystem they define the functional, performance, and nonfunctional requirements as well. Next, each subsystem is developed by the team responsible for it. Based on the subsystem requirements the particular subsystem team transforms these requirements into the functional architecture of the particular subsystem followed by the allocation of the identified functions to the new subsystem components. A refined architecture of each subsystem is produced considering nonfunctional requirements and constraints as well. This subsystem architecture representing the subsystem components, their interfaces, the interfaces to other subsystems, and the external interfaces is modeled by SysML BDD, and IBD diagrams.

For the sake of simplicity, Figure B.3 only shows IBD diagrams. Furthermore, each component has requirements that specify the functional behavior, performance, and nonfunctional requirements required for the particular component to work with other components. At the component level the approach followed de-

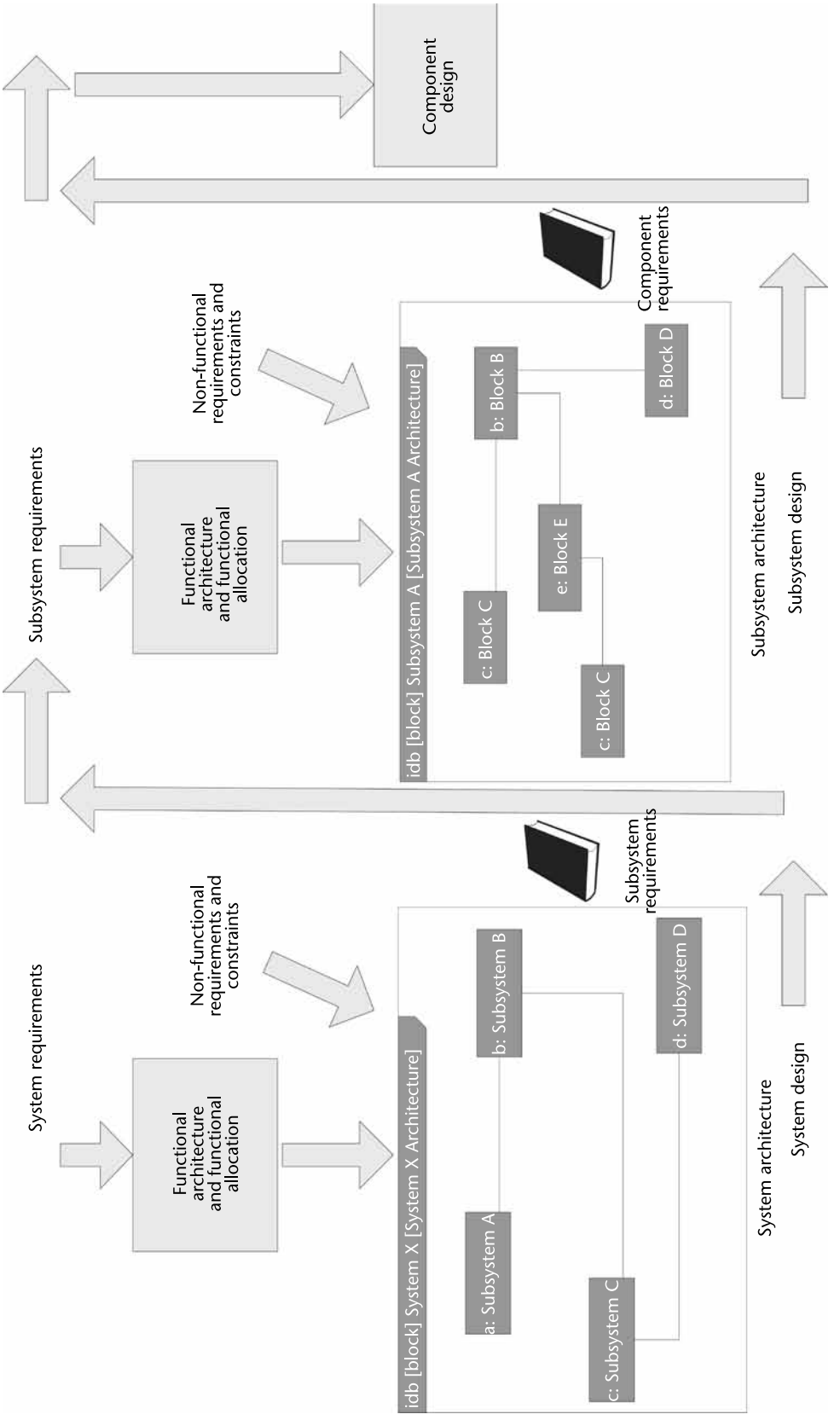


Figure B.3 Disciplined requirements flowdown.

pends of the type of component for example hardware or software component, so specific approaches are used.

For software components, design may be based on the use of the PPOOA component-based development process explained in Chapter 4. Other alternatives for software development are either agile methods (see Chapter 12), or the twin peaks model described by Nuseibeh where software requirements and software architecture are intertwined in an incremental development [11].

In Figure B.3 requirements specifications are represented as documents using a book icon, but models as described in the section below can be used as well. A requirements management tool can be helpful for requirements changes and change impact analysis.

A disciplined approach, such as the one presented here, ensures that all requirements are flowed down properly and provides a number of benefits to the project, including:

- Due to the use of hierarchies, all system-level requirements are properly allocated;
- The use of IBD diagrams allows the identification of possible internal interfaces;
- Due to the use of functional, nonfunctional, and physical hierarchies and allocation, requirements traceability is facilitated;
- Functional and quality attribute hierarchies aid in finding redundant requirements;
- The approach presented aids in ensuring completeness of requirements.

B.4 Models and Requirements

The approach to requirements engineering proposed here manages textual requirements and requirements models to facilitate requirements extraction and requirements analysis. Therefore, requirements model or models of specification are a more rigorous approach to engineer consistent requirements.

Requirements models, as opposed to architectural models or design models, try to represent the problem dimension and not the solution dimension defined by architectural models such as SysML IBDs.

IREB Handbook of Requirements Modeling [12] summarizes the diverse uses of requirements modeling:

- Requirements models replace textual requirements statements;
- Requirements models are used to uncover inconsistencies or omissions in the textual requirements statements;
- Requirements models are used to refine the textual requirements for the sake of understandability.

We can add an additional approach we use in our development projects, which is to use models to extract or derive requirements. That is the case of the

hierarchical functional trees and functional flows that allow us to derive functional requirements.

Diverse sources, for example *IREB Handbook* referenced above or Wiegers and Beatty, propose several requirements models mainly for use in business analysis [13]. Here we propose the requirements models we are using for systems or product development, which are use case diagrams, data flow diagrams, activity diagrams, and state-transition diagrams.

Use case diagrams show the actors external to the system and the use cases with which these actors interact. Therefore, the functionality is represented from the users or actors perspective. The main modeling elements of use case diagrams as provided in UML/SysML notation are actors represented as stick figures, use cases represented as ovals, and associations represented by lines between an actor and a use case (see Figure B.4).

Since use cases represent interactions, they are decomposed into scenarios or sequences of steps that are represented by activity diagrams or commonly by textual tabular forms as the one shown in Table B.1. Preconditions are the system conditions that must be true before the use case start. Postconditions are the system conditions that must be true when the use case ends, no matter which use case scenario, main or alternatives, is executed. Steps are the interactions between actors and system that are necessary to achieve the use case.

By identifying in the use case description table which steps are to be taken by the system, they can be converted into functional needs or functional requirements statements.

DFDs are described in Chapter 5 as well, because they are used in the functional architecture modeling by some methodologies and tools. The DFDs represent the system functional architecture as a network of functions that accept and produce data items. The DFD shows terminators or external entities as squares, the functions at the same level of hierarchy as bubbles, and the data flows represented

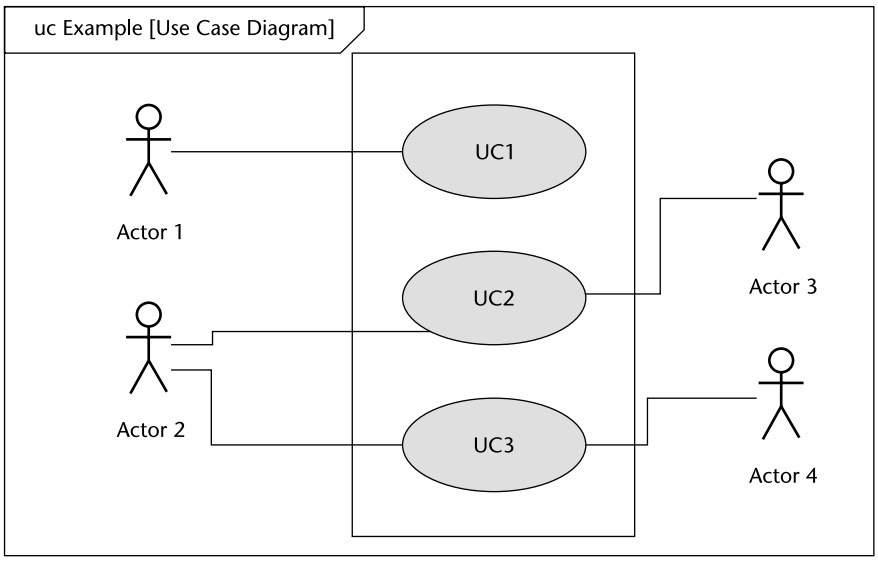


Figure B.4 Use cases diagram.

Table B.1 Use Case Description

Use Case Name	
Preconditions	
Description	Step 1
	Step 2
	Step n
Postconditions	
Alternatives	Step x
	Step y
	Step z

as arrows between them, external to them, and to and from stores. Stores are used to represent an item or set that is operated by a group of functions (see Figure B.5).

When using DFDs to model the functions of a system, we can derive functional requirements statements from the outputs of each function represented using the template presented in the next section to specify each transformation to produce the output. Similarly each complex data item and each data store are specified

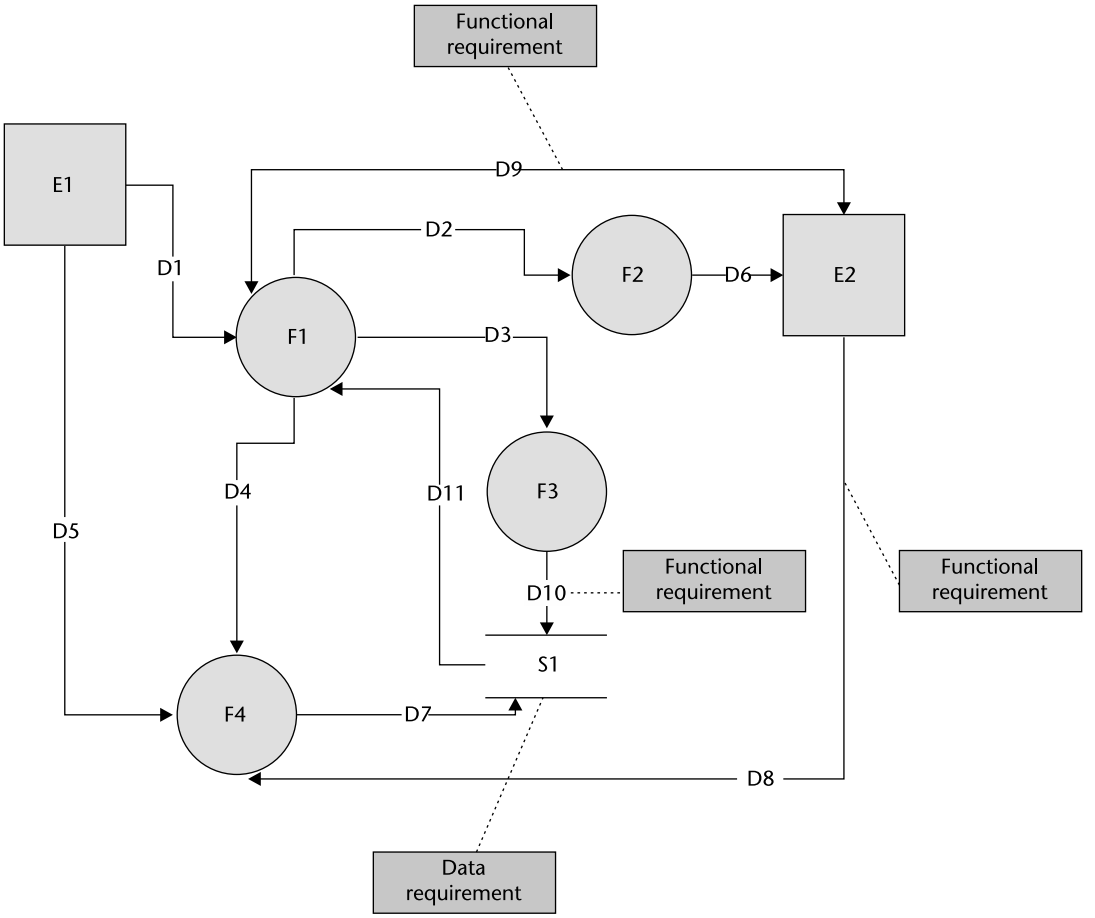


Figure B.5 DFD and functional and data requirements.

using the template for data requirements presented in the next section and using the Hatley and Pirbhay notation shown above.

Activity diagrams are an alternative to represent functional behavior. The SysML activity diagram defines the actions in the activity along with the flow of input/output and control between them. in other words, an activity decomposes into a set of actions that describe how the activity executes and transforms its inputs to outputs. Activity diagrams allow modeling object and data flows as well, as shown in Figure B.6.

When using activity diagrams to model the functional architecture of a system, we can derive functional requirements statements from the outputs of each action represented using the template presented in the next section to specify each transformation to produce the output. Similarly, each complex data item is specified using the template for data requirements presented in the next section and using the Hatley and Pirbhay notation shown above. When we use the N^2 chart to represent the functional interfaces (see Chapter 5) instead of modeling object and data flows in the activity diagram, we can identify the outputs of each function in the N^2 chart and derive functional requirements from them.

State transition diagrams or state machine diagrams represent the system behavior as states instead of functions. A state is described in Chapter 4 as the condition of a system defined by its current condition/configuration and the function provided.

The main modeling elements of the state diagrams as provided in UML/SysML notation are states, transitions, initial state, and final state (see Figure B.7). A state is entered when a transition is passed that leads to this state. A state is abandoned when a transition is passed through that leads away from the state. Each state may have entry and exit behaviors that are performed whenever the state is entered or exited, respectively. In addition, the state may perform a do behavior that executes once the entry behavior has completed. A state is represented by a round-cornered box with its name. Composite states or superstates are those composed out of two or more substates. Composite states are represented by a simple state graphic with a special composite icon as state 3 in Figure B.7. A transition is shown as an arrow

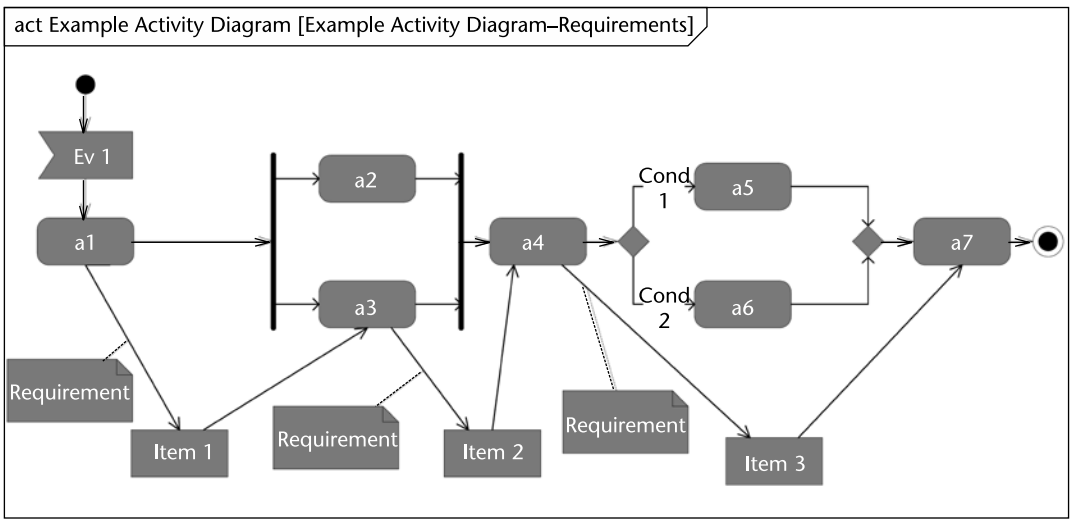


Figure B.6 Activity diagram and functional and data requirements.

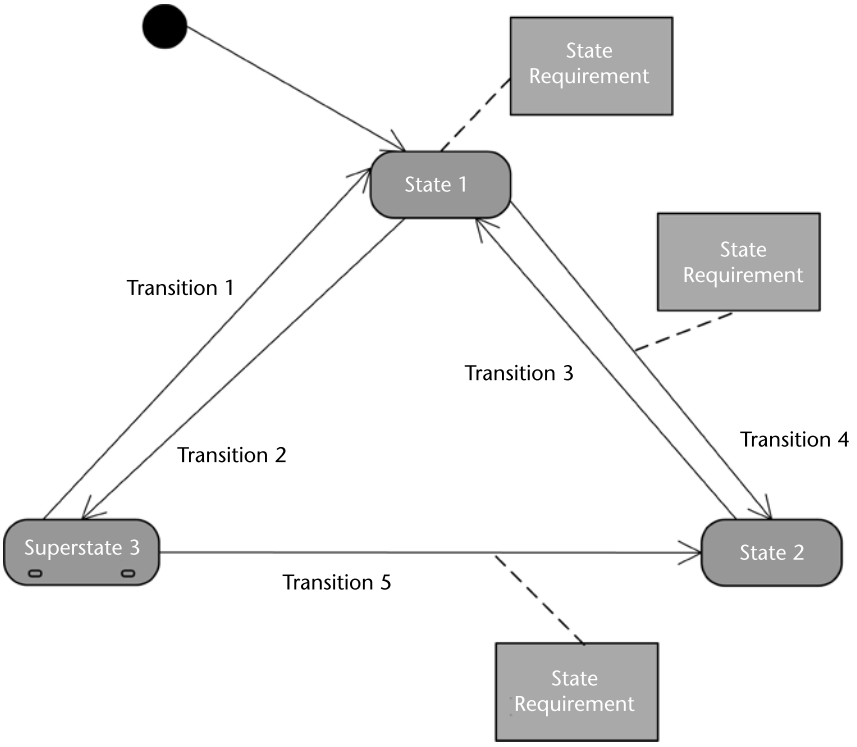


Figure B.7 State diagram and state requirements.

between two states, with the arrow pointing to the target state. Transitions to self, not used here, are shown with both ends of the arrow attached to the same state.

Textual requirements statements may be derived from the state diagram identifying the conditions required for each transition to occur and identifying the actions (functions) occurring as a result of each transition. Data processing performed during a state should be derived as functional requirements.

B.5 Requirements Templates

Generally, natural language complemented with requirements models is what is used to express requirements. Natural language may be ambiguous but ambiguity can be avoided using requirements templates or boilerplates and writing clear and precise requirements statements where requirements are quantified.

Adopting and adapting diverse sources such as the ISO quality model [2], Firesmith [6], Hatley and Pirbhai [4], and Withall [14], we propose the following requirements templates consistent with the quality model shown in Figure B.2. The requirements templates are shown in Table B.2. We recommend the reader to adopt and extend them based on the quality model more appropriate for the system to be developed.

Table B.2 Requirements Templates

Functional requirement	When <condition clause>, the <subject clause> shall <action verb clause><object clause> <constraint of action>
Nonfunctional requirement-efficiency-response time	Each <system response> shall have an end to end time of no more than <Tolerable length of time> from <Start event> to <Timing boundary end> [when using «Indicative hardware set-up»]
Nonfunctional requirement-efficiency-throughput	<Part of system> shall handle <Throughput object type> transactions at a rate of at least <Throughput quantity> per <Unit time period>
Nonfunctional requirement- availability	The system shall normally be available to its users <Availability extent description> [, except in exceptional circumstances of a frequency and duration not to exceed <Tolerated downtime qualifier>]
Nonfunctional requirement-maintainability- changeability	Developers shall add a new system <function or quality attribute requirement>, including modifications and testing , with no more than <personxhour> of effort
Nonfunctional requirement-safety-harm protection	The system shall not injure any human sufficiently to require his/her hospitalization at an average rate greater than <quantity> per <mission duration>
Nonfunctional requirement-safety-hazard protection	<Function> activation shall be allowed only when < associated hazard prevention condition> is operational
Nonfunctional requirement-safety-incident identification	The system shall identify a safety incident due to the combination of <function> activation and <hazard prevention condition> malfunction with a probability of at least <quantity>
Nonfunctional requirement-safety-incident reporting	The system shall report the occurrences of identified safety incidents at least < quantity> of the time
Data requirement	Data = <primitive data item> + <primitive data item>+{<primitive data item >}N+ [<alternative primitive data item> | <alternative primitive data item>] + (<optional primitive data item>)

B.6 Summary

Requirements are the main driver of the systems architecting process. As presented in this Appendix, requirements engineering and systems architecting are two processes that are intertwined to facilitate requirements allocation and requirements flowdown.

Furthermore, requirements commonly represented by textual requirements statements, should be necessary (essential capability, characteristic, constraint, or quality factor), appropriate (amount of detail of the requirement is appropriate to the level), unambiguous (it can be interpreted in only one way), complete (sufficiently describes the necessary capability, characteristic, constraint, or quality factor to meet the entity), and other characteristics described in detail in the *INCOSE Guide for Writing Requirements* [1].

The above requirements characteristics are supported in this Appendix by an adequate requirements flowdown process, complementing requirements with requirements models, and using requirements templates.

References

[1] Ryan, M., et al., "Guide for Writing Requirements," International Council on Systems Engineering (INCOSE), San Diego, California, 2017.

[2] ISO/IEC/IEEE 29148, "Systems and Software Engineering: Life Cycle Processes-Requirements Engineering," ISO, Geneva, Switzerland, IEC, Geneva, Switzerland, and Institute of Electrical and Electronics Engineers, New York, 2011.

[3] Hatley, D .J., and I. A. Pirbhay, *Strategies for Real-Time System Specification,* New York: Dorset House, 1988.

[4] Hooks, I. F., and K. A. Ferry, *Customer Centered Products, Creating Successful Products Through Smart Requirements Management,* New York: AMACON, 2001.

[5] ISO/IEC FDIS 9126-1, "Information Technology: Software Product Quality, Part 1: Quality Model" ISO, Geneva, Switzerland, 2000.

[6] Firesmith, D., "Engineering Safety Requirements, Safety Constraints, and Safety-Critical Requirements," *Journal of Object Technology,* Vol. 3, No. 3, March–April 2004, pp. 27–42.

[7] Jackson, S., and T. Ferris, "Resilience Principles for Engineered Systems," *Systems Engineering*, Vol. 16, No. 2, 2013, pp. 1098–1241.

[8] Gregg, B., "Visualizing System Latency," *Communications ACM,* Vol. 53, No. 7, July 2010, pp. 48–54.

[9] Lluesma, M., et al., "Jitter Evaluation of Real-Time Control Systems," *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.

[10] U.S. Government, *National Security Strategy,* Washington, DC, 2010.

[11] Nuseibeh, B., "Weaving Together Requirements and Architectures," *IEEE Computer*, Vol. 34, No. 3, March 2001, pp. 115–119.

[12] Cziharz T., et al., *Handbook of Requirements Modeling*, IREB Standard. International Requirements Engineering Board (IREB), Karlsruhe, Germany, September 2015.

[13] Wiegers, K., and J. Beatty, *Software Requirements*, Redmond, WA: Microsoft Press, 2013

[14] Withall, S., *Software Requirement Patterns*, Redmond, WA: Microsoft Press, 2007.

About the Authors

Jose L. Fernandez has a Ph.D. in computer science and an engineering degree in aeronautical engineering, both from the Universidad Politecnica de Madrid.

He has over 30 years of experience in industry as a systems engineer, project leader, researcher, department manager, and consultant. He has been involved in projects dealing with software development and maintenance of large systems, specifically real-time systems for air traffic control, power plant supervisory control and data acquisition (SCADA), avionics, and cellular phone applications.

For 20 years he was an associate professor at the E.T.S. Ingenieros Industriales, Universidad Politecnica de Madrid (UPM). His areas of interest are systems engineering, real-time systems, software engineering, CASE tools, and project management.

He is the methodologist of the PPOOA architectural framework for real-time systems and ISE&PPOOA, an integrated systems and software MBSE methodology for complex systems.

He has more than 50 publications in international journals and conference proceedings, mainly in the fields of systems engineering, software development, real-time systems, and project management.

He is a senior member of the IEEE and a member of International Council on Systems Engineering (INCOSE), participating in the software engineering body of knowledge, systems engineering body of knowledge, and requirements engineering working groups of these associations. He is a member of the Project Management Institute (PMI), participating as reviewer of the *PMBoK* sixth edition, 2017, and *Requirements Management, Practice Guide,* 2016.

Carlos Hernandez is an assistant professor at the Delft University of Technology and has extensive experience in the design and integration of advanced robotic applications. In 2016, he led the development of Team Delft's robotic system that won the Amazon Picking Challenge.

He received his engineering degree from the Universidad Politecnica de Madrid in industrial technologies and electronics and automation in 2006, a M.Sc. in automation and robotics in 2008, and his Ph.D. in 2013 from the same university.

Carlos currently coordinates the H2020 project ROSIN (Grant Agreement No. 732287). Previously he was scientific lead in the Factory-in-a-day FP7 project, to develop flexible robotic components to reduce the system integration costs for the robotizing of factories, and participated in the ICEA and HUMANOBS projects

from the European program FP7 and other national research projects in the area of cognitive robotics. His main areas of current research include autonomy, model-based engineering for robot control design, and self-adaptive systems. He has more than 20 publications in international journals and conference proceedings, mainly in the field of robotics.

Index

A

Action Language for Foundational UML (Alf), 26
Activities, 211
Activity diagrams
  activity nodes, 211, 212–13
  collaborative robot application, 147, 149, 160, 162, 168–69
  control nodes, 213
  defined, 211
  ISE&PPOOA use of, 211
  in modeling functional architecture, 227
  in representing functional behavior, 227
  SysML, 62
  UML, 63
Activity nodes, 211, 212–13
Agile Architecture Framework, 30
Agile development
  disciplined agile delivery (DAD), 195–96, 197, 199
  ISE&PPOOA process and, 198–99
  misconceptions about, 194
  principles of, 193–94
  scalability, 195–98
  scaled agile framework (SAFe), 196–97
  Scrum@Scale, 195, 196
Agile Manifesto, 193
Alerting capabilities, 87
Allocations
  collaborative robot application, 157
  criteria for, 98
  defined, 100
  functional, 100
  of NFRs, 100
  relationships, 214, 215
  Seeker UAS application, 130, 131, 132
  SysML, 213–14
Analyzability, 221
Architecture evaluation
  ATAM, 201–2
  defined, 199–200
  diverse practices for, 200–201
  framework of practices, 201
  goals, 200
  process scope, 200
  verification and validation, 199–200
Architecture trade-off analysis method (ATAM), 201–2
Asset protection, 221

B

Block definition diagrams (BDD). *See* SysML block definition diagrams
Bottom-up approach, functional hierarchy, 65
Building elements
  collaborative robot application, 155
  supported by PPOOA architecture framework, 45–46
  vocabulary of, 45

C

Changeability, 221
Collaborative robot application
  activity diagrams, 147, 149, 160, 162, 168–69
  allocation of functions, 157
  building elements, 155
  capabilities and high-level functional requirements, 140–41
  capabilities list, 145
  casual flows of activities, 167
  context diagram, 144
  CRC cards, 166
  defined, 137–38
  domain model, 165
  element descriptions, 164

Collaborative robot application (continued)
example needs and capabilities, 137–44
functional architecture, 145–50
functional flows, 148
functional hierarchy, 150
identification of high-level functions, 147
modular architecture, 155–58
N^2 charts, 146–48
operational needs, 138–40
operational scenarios, 138–40
overview, 137
performance, constraints, and nonfunctional requirements, 153–54
physical architecture, 154–61
physical parts description, 160
quality attributes and system NFRs, 141–44
representation of refined physical architecture, 159–61
resilience heuristics, 159
responsibilities, 161–64
safety heuristics, 158–59, 167–70
software architecture, 161–70
software components, 165
structural view of software architecture, 167
summary, 170–71
SysML block definition diagram, 141
SysML internal block diagrams, 163
system requirements, 150–54
traceability matrix functions and QAs capabilities, 152
traceability matrix of capabilities, 142
Commercial off-the-shelf (COTS) elements, 97, 98
Component interfaces
PPOOA, 105–6
specifying, 48
Components
algorithmic, 46, 105
domain, 46, 105
identifying, 47–48
operations restrictions, 91
PPOOA, 104, 106
selection criteria, 90
usage rules between coordination mechanisms and, 107
Computing load, balancing, 83
Concept stage, system life cycle, 9–10
Concurrent engineering, 13
Constraint blocks, 215
Control monitor pattern, 101
Control nodes, 213
Coordination mechanisms
defined, 106
in PPOOA architecture, 106
selecting, 49
set of, 107
usage rule between components and, 107
Coupling metrics, 201
CRC cards, 46–47, 166

D

Data dictionary, 127, 154, 219
Data flow diagram (DFD), 59, 60, 225–26
Data recording functions, 87
Deadlock Risk Evaluation of Architectural Models (DREAM), 203
Development specification, 13
Development stage, system life cycle, 10
Disciplined agile delivery (DAD), 195–96, 197, 199
Disciplined requirements flowdown, 222, 223
Diverse paths for reading this book, *5*
Domain model
classes, 103–4
collaborative robot application, 165
defined, 103
example, 103
substantives as classes in, 104
Drift correction, 89

E

Efficiency
characteristics, 74
defined, 74, 221
heuristics for arbitrating demand, 82–83
heuristics for managing demand, 81–82
heuristics for multiple resources, 83–84
performance, 81
Energy efficiency application
coal power plant, 174
coal power plant schematic, 175
enthalpy balance, 175
functional architecture, 176–78

linear programming algorithms, 175
matter and energy balances and, 179–82
overview, 173–76
physical architecture, 178–79
results, 183–84
summary, 184
Engineering Equation Solver (EES) computer tool, 183
Enhanced functional flow block diagram (EFFBD), 61
Events
domain model, 104
in SysML, 58
Evolutionary approach, 17–18
Extreme programming (XP), 194

F

Feature-driven development (FDD), 194
Fixing point, 82
Foundational UML (fUML), 26
Functional architecture
collaborative robot application, 145–50
core model representation, 56
deliverables, 41
describing functions and functional interfaces in, 66–67
functional requirements, 67–68
importance of, 55–56
of ISE process, 75
main concepts, 57–58
modeling the functional flows and, 65–66
modeling the functional hierarchy, 63
models, 58–63
N^2 charts for, 67
as representation of the problem space, 56
representing, 42
Seeker UAS application, 116–19
for software intensive systems, 56
steam generation application, 176–78
summary, 68
in systems engineering, 55–56
Functional flow block diagram (FFBD), 59
Functional flows
functional hierarchy consistency and, 64
guidelines for modeling, 66
modeling, 65–66
Seeker UAS application, 122
steam generation application, 177
Functional hierarchy
bottom-up approach, 65
collaborative robot application, 150
as framework, 67
functional flow consistency and, 64
illustrated, 64
level of detail, 63
modeling of, 63–64
Seeker UAS application, 117, 118
steam generation application, 176
tabular format for description, 63
top-down approach, 64–65
Functional interfaces
describing, 66–67
N^2 charts, 66
steam generation application, 178
Functional requirements, 67–68
Functions
describing, 66–67
Seeker UAS application, 119, 120–21
in SysML, 57
as transformations, 57–63

G

Goals and readers, this book, 1–2
Guide to the Systems Engineering Body of Knowledge (SEBoK), 15

H

Harmony, 29
Heuristics
to act as a whole in face of a threat, 89
for adapting to a threat, 88–89
application of, 75
for arbitrating demand, 82–83
collaborative robot application, 158–59
defined, 71
to degrade gracefully, 89
design, for refining architecture, 101–2
efficiency, 80–84
framework, 71–75
for hazard avoidance, 85
for hazard control, 86–87
for hazard reduction, 85–86

Heuristics (continued)
implementations in architecture, 74
for managing demand, 81–82
for mitigation of the effects, 87
for multiple resources, 83–84
PPOOA framework, 90–91, 92
for preventing unintended effects, 80
processing versus frequency, 82
quality model, 71, 72, 73
reliability, 77–80
resilience, 87–89, 159
for restricting visibility of responsibilities, 79–80
safety, 84, 158–59, 167–70
security, 84
Seeker UAS application, 128–29
selection of, 71
software architecting, 90–91
summary, 91–92
for surviving a threat, 88
system architecting, 75–77
Hidden interactions, reducing, 222
Human back-up, 88
Human in the loop, 88–89

I

ICAM Definition (IDEF) techniques, 59
IDEF0 diagram, 59, 60–61
IEEE Std. 1220, 11
Incident protection, 221
Incident reaction, 221
Incremental approach, 17
Instrument the system, 81–82
Integrated Computer Aided Manufacturing (ICAM), 59
Integrated logistics support (ILS), 13
Integration, 14
Interface management, 13
International Council on Systems Engineering (INCOSE), 7–8, 11
IREB Handbook of Requirements Modeling, 224, 225
ISE&PPOOA
activity diagrams, 211
conceptual model for systems engineering with, 36
creativity and, 2
guidelines, xiv
outputs and diagrams, 208
as requirements-driven approach, 1
research and development, xiii
structural perspective, SysML for, 209–11
SysML diagrams in, 207–8
SysML use in, 207–8
Visio add on, xiv
ISE&PPOOA/energy process
defined, 49
plant models, 50
steps for, 51–52
study of alternatives, 49
ISE&PPOOA process
agile development and, 198–99
challenges of, 34–36
concepts, 36–38
dimensions and main steps of, 38–49
extension for energy efficiency concerns, 49–52
functional, physical, and quality trees in, 96
functional and physical interfaces and, 35–36
implementation of nonfunctional requirements and, 34–35
modeling the functional flows, 65–66
software architecting subprocess (PPOOA), 44–49
summary, 53
systems engineering subprocess (ISE), 39–44
three dimensions of, 38–39
trade-off subprocess, 190–91
UML notation, 38
ISE subprocess
architecture refinement, 43
functional architecture, 42, 75
functions allocation, 42–43
functions decomposition, 42
high-level functions identification, 42
illustrated, 40
modular architecture representation, 43
operational scenarios identification, 39–40
physical architecture of, 75–77
quality attributes specification, 41–42
reliability and maintainability heuristics, 78
system capabilities specification, 41

system physical architecture representation, 44
See also ISE&PPOOA process

J

Jitter, 221
JPL State Analysis, 30

L

Latency, 221
Linear programming algorithms, 175
Locality, 83–84

M

Maintainability
characteristics, 73
defined, 73
heuristics, 77–80, 81
ISE process, 78
PPOOA process, 78–79
requirements, 220
Model assessment, 202–3
Model-based systems engineering (MBSE)
benefits of, xiii
defined, 1
industry acceptance, 22
methodologies, 27–28
modeling in, 22–25
modeling languages, 25–27
need for, 21–22
potential benefits of, 22
shortcomings addressed by, 21–22
summary of methodologies, 29
tools, 28–30
Models
for communication, 22–23
example of, 24
in MBSE, 22–25
reasons for use, 22–24
requirements and, 224–28
in robotic pick and place application, 23
for simulation, 23
views and, 24
Modifiability. *See* Maintainability
Modular architecture
collaborative robot application, 155–58
representation of, 43, 98–100
SysML block definition diagram, 99
Modularity principle, 98

N

N^2 charts
benefits of, 36
collaborative robot application, 146–48
defined, 35
for functional architecture, 67
functional interfaces in, 66
illustrated, 35
ISE&PPOOA, 66
Seeker UAS application, 117, 124, 125
use of, 35–36
NASA handbook, 11
Neutral state, 89
Next steps for reader, 203–4
Nonfunctional requirements (NFRs)
allocation of, 100
collaborative robot application, 141–44
implementing, 34–35
quality models for, 96
trade-off analysis and, 188
use of, 219

O

Object-Oriented Systems Engineering Method (OOSEM)
defined, 29
logical architecture development, 97
Object Process Methodology (OPM)
defined, 27
for function, structure, and behavior specification, 96–97
list of MBSE methodologies, 28
summary, 29
Operational scenarios
collaborative robot application, 138–40
identifying, 39–40
Seeker UAS application, 113–14
Operations and maintenance, 15
Organization, this book, 2–5

P

Parallel processing, 82–83

Parametric diagrams, 215
Physical architecture
allocation and modularity, 98–100
application of heuristics, 158–59
collaborative robot application, 154–61
deliverables, 44
design heuristics for refining, 101–2
heuristics, 158–59
of ISE process, 75–77
model, concepts related to, 95–96
representing, 44
resilience heuristics, 159–60
Seeker UAS application, 130–31
software architecting with PPOOA framework, 102–9
steam generation application, 178–79
in systems engineering, 95–97
Physical device management, 90–91
Pipelines of Processes in Object-Oriented Architectures. *See* ISE&PPOOA
PPOOA architecture diagram, 216
PPOOA-Cheddar, 203
PPOOA framework heuristics, 90–91, 92
PPOOA process
algorithmic component, 46, 105
as architecture framework, 45
building elements, 45–46
CFA execute subwork order, 49, 50
components, 104
composition rules between vocabulary elements, 108
controller object, 46, 106
coordination mechanisms, 49, 106–8
CRC cards, 46–47
domain component, 46, 105
domain model, 103–4
illustrated, 45
interfaces, 105–6
maintainability for, 78–79
modeling subsystem functional behavior, 48–49
software behavior and causal flow of activities, 108–9
software component interfaces specification, 48
software components identification, 47–48
structural view, 46
structure, 46, 105
See also ISE&PPOOA process
Preplanned product improvement, 14
Production and development, 15
Production stage, system life cycle, 10
Project heuristic description template, 43

Q

Quality assurance and management, 14
Quality attributes, specifying, 41
Quality models
functions of, 71
for heuristics classification, 73
for nonfunctional requirements (NFRs), 96
proposed, 220
for refining solution architecture, 72

R

Rapid application development (RAD), 194
Recovery, 86
Redundancy, 86, 88
Reliability
characteristics, 73
defined, 72–73
heuristics, 77–80
ISE process, 78
requirements, 220
Reliability, availability, and maintainability (RAM), 14
Reliability engineering, 13
Repairability, 88, 221
Requirements
behavioral, 218
classification, 218–22
constraints, 220
data, 218–19, 226
expression of, 217
flowdown in systems development, 222–24
framework, 217–29
functional, 226, 228
guide for writing, 218
interface, 219
models and, 224–28
needs and capabilities and, 217–18
performance, 219
physical properties, 220

reliability and maintainability, 220
specifications, 224
state, 228
summary, 229
SysML notation, 215–16
textual, 228
See also Nonfunctional requirements (NFRs)
Requirements templates, 228–29
Resilience
defined, 74, 221
engineered systems, 74
heuristics, 88–89, 159
heuristics for adapting to a threat, 88–89
heuristics for surviving a threat, 88
heuristics to act as a whole in face of a threat, 89
heuristics to degrade gracefully, 89
in systems engineering, 87
Resource utilization, 221
Response time, 221
Restructuring, 88
Retirement stage, system life cycle, 10
Risk analysis, 13

S

Safety
characteristics, 74
defined, 74, 221
as emergent property, 84
heuristics, 85–87, 158–59, 167–70
heuristics for hazard avoidance, 85
heuristics for hazard control, 86–87
heuristics for hazard reduction, 85–86
heuristics for mitigation of effects, 87
Sanity check, 86
Scalability in agile approaches
disciplined agile delivery (DAD), 195–96, 197, 199
overview, 195
scaled agile framework (SAFe), 196–97
Scrum@Scale, 195, 196
Scaled agile framework (SAFe), 196–97
Scheduling policy, 83
Scrum, 194
Scrum@Scale, 195, 196
Security heuristics, 84
Seeker UAS application
aircraft dop level functions, 116
aircraft main subsystems, 128
allocated functional flow, 130, 131, 132
battery, 127
characteristics of, 112
connection, 127
context diagram, 114
defined, 112
distribute communications functional flow, 122
distribute communications functional hierarchy, 117
distribute power function, 120
electrical subsystem physical architecture, 129
example needs and capabilities, 112–-15
functional architecture, 116–19
functional flows, 122
generate and manage electrical power functional flow, 122
generate and manage electrical power functional hierarchy, 118
heuristics applied, 128–29
illustrated, 113
manage power function, 121
manage telemetry data function, 120
N^2 charts, 117, 119, 124, 125
operational scenarios, 113–14
overview, 111
payload management and command functional hierarchy, 118
perform electrical protection function, 120
physical architecture, 130–31
physical parts description, 131, 133–35
provide payload management and command functional flow, 122
provide power function, 120
remote handheld control, 113
send camera position to payload function, 121
send perform zoom command function, 121
send switch camera command function, 121
summary, 131
system capabilities, 115
system components, 112
system requirements, 119–27

Seeker UAS application (continued)
transmit demand actuator position function, 119
transmit demand powerplant function, 119
transmit sensor measures function, 119
transmit switch on/off lights command function, 119
transmit video function, 120
use cases, 114–15
Sequential approach, 16
Shared resources, 83
Smart products, 12
Software architecting
heuristics, 90–91
integrating systems engineering and, 33–34
maintainability for, 78–79
with PPOOA framework, 102–9
Solution-oriented quality model, 72
Specialty engineering, 14
State diagrams, 227–28
State transition diagrams, 227
Steam generation application
boiler energy balance constraints diagram, 180
breakdown, 177
condenser energy balance constraints diagram, 182
condensing curves, 184
evaporator, 179
exhaust steam of turbine, 179
functional architecture, 176–78
functional hierarchy, 176
functional interfaces, 178
heat exchangers, 179
improved energy balance constraints diagram, 181
matter and energy balances and, 179–82
overview, 173–76
physical architecture, 178–79
pump behavior equation, 181
results, 183–84
steam turbine power constraints diagram, 182
subsystem internal block diagram, 177
summary, 184
Structured analysis and design technique (SADT), 56
Support stage, system life cycle, 10
Survivability, 222
Synchronization protocols, 83
SysML
activity diagram, 62
allocation, 213–14
behavioral diagrams, 97, 202
category of actions, 61
constraint blocks, 215
defined, 25–26
diagrams in ISE&POOA, 35, 207–8
events in, 58
fUML, 26
functions in, 57
for ISE&PPOOA behavioral perspective, 211
in ISE&PPOOA methodology, 207–8
for ISE&PPOOA structural perspective, 209–11
modular architecture, 98–99
parametric diagrams, 215
PPOOA architecture diagram and, 216
requirements notation, 35
standard extension, 23
taxonomy of diagrams in, 26
text-based requirements, 215–16
use case diagrams, 214–15
Viewpoint and View elements, 26
SysML block definition diagrams
associations, 209
collaborative robot application, 141
defined, 97
dependency relationships, 209–10
as most used, 209
SysML internal block diagrams
collaborative robot application, 144, 163
connectors, 211
defined, 97, 210
flows, 211
parts, 210–11
ports, 211
steam generation application, 177
SysMod, 29
System adaptation, 222
System architecting heuristics, 75–77
System capabilities
Seeker UAS application, 115

specification of, 41
System Definition Language (SDL), 27
System development alternatives, 15–16
System disposal, 15
System requirements
collaborative robot application, 150–54
Seeker UAS application, 119–27
Systems
defined, 7–8
hierarchical structure, 8, 9
interconnected, 8–9
life cycle, 8, 9–10, 13
operation environment, 8
properties of, 8
quality attributes, 8
Systems engineering
definitions of, 11
evolutionary approach, 17–18
functional architecture in, 55–56
incremental approach, 17
with ISE&PPOOA, conceptual model, 36–38
key tasks of, 12–15
milestones of, 12
need for, 11–12
physical architecture in, 95–97
sequential approach, 16
software architecting integration, 33–34
system development alternatives, 15–18
See also Model-based systems engineering (MBSE)
Systems engineering management, 15
Systems Modeling Language. *See* SysML

T

Testability, 221
Test and evaluation, 14
Threats
defined, 87
heuristics for adapting to, 88–89
heuristics for surviving, 88
heuristics to act as a whole in face of, 89
Throughput, 221
Top-down approach, functional hierarchy, 64–65
Trade-off analysis
architectural decision process and, 187–88
architecture method (ATAM), 201–2
assessment criteria and utility functions, 189–90
assessment of alternative techniques, 188
defined, 187
generic trade tree, 189
nonfunctional requirements (NFRs) and, 188
subprocess, 190–91
summary, 191
Training, 15
Triple modular redundancy (TMR) pattern, 102

U

UML
activity diagram, 63
behavioral models, 202
class diagrams, 102
Use case description, 226
Use case diagrams, 214–15, 225
Use cases
interaction representation, 225
Seeker UAS application, 114–15
Utility function, shapes of, 190
Utilization stage, system life cycle, 10

V

Verification and validation
architecture evaluation, 199–200
defined, 14