

The MK/OMG PRESS

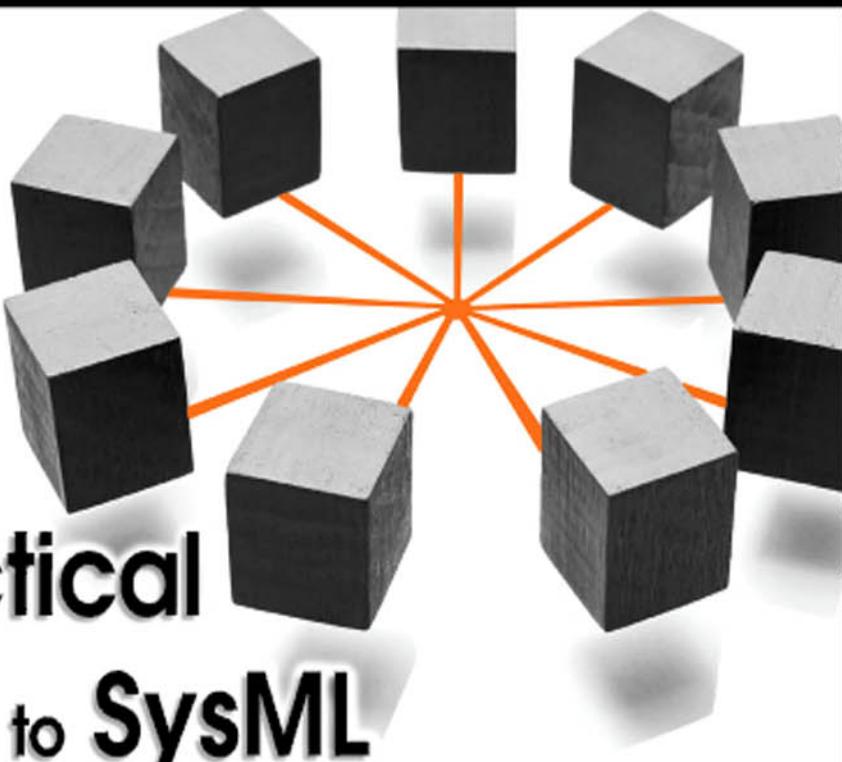
The MK/OMG PRESS



The MK/OMG PRESS

A Practical Guide to SysML

The Systems Modeling Language



The MK/OMG PRESS
Sanford Friedenthal
The MK/OMG PRESS
Alan Moore
The MK/OMG PRESS
Rick Steiner

The MK/OMG PRESS

A Practical Guide to SysML



Morgan Kaufmann OMG Press

Morgan Kaufmann Publishers and the Object Management Group™ (OMG) have joined forces to publish a line of books addressing business and technical topics related to OMG's large suite of software standards.

OMG is an international, open membership, not-for-profit computer industry consortium that was founded in 1989. The OMG creates standards for software used in government and corporate environments to enable interoperability and to forge common development environments that encourage the adoption and evolution of new technology. OMG members and its board of directors consist of representatives from a majority of the organizations that shape enterprise and Internet computing today.

OMG's modeling standards, including the Unified Modeling Language™ (UML®) and Model Driven Architecture® (MDA), enable powerful visual design, execution and maintenance of software, and other processes—for example, IT Systems Modeling and Business Process Management. The middleware standards and profiles of the Object Management Group are based on the Common Object Request Broker Architecture® (CORBA) and support a wide variety of industries.

More information about OMG can be found at <http://www.omg.org/>.

Related Morgan Kaufmann OMG Press Titles

UML 2 Certification Guide: Fundamental and Intermediate Exams

Tim Weilkiens and Bernd Oestereich

Real-Life MDA: Solving Business Problems with Model Driven Architecture

Michael Guttman and John Parodi

Systems Engineering with SysML/UML: Modeling, Analysis, Design

Tim Weilkiens

A Practical Guide to SysML: The Systems Modeling Language

Sanford Friedenthal, Alan Moore, and Rick Steiner

Building the Agile Enterprise: With SOA, BPM and MBM

Fred Cummins

Business Modeling: A Practical Guide to Realizing Business Value

Dave Bridgeland and Ron Zahavi

Architecture Driven Modernization: A Series of Industry Case Studies

Bill Ulrich

A Practical Guide to SysML

The Systems Modeling
Language

**Sanford Friedenthal
Alan Moore
Rick Steiner**



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier



Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400, Burlington, MA 01803

This book is printed on acid-free paper.

Copyright © 2008 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, scanning, or otherwise, without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com. You may also complete your request on-line via the Elsevier homepage (<http://elsevier.com>), by selecting "Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data

Friedenthal, Sanford.

A practical guide to SysML : The Systems Modeling Language / Sanford Friedenthal, Alan Moore, Rick Steiner.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-12-374379-4 (alk. paper)

1. Systems engineering. 2. SysML I. Moore, Alan. II. Steiner, Rick. III. Title.

TA168.F745 2008

620.001'171—dc22

2008024882

For information on all Morgan Kaufmann publications, visit
our Web site at www.mkp.com or www.books.elsevier.com.

Printed in the United States

08 09 10 11 12 10 9 8 7 6 5 4 3 2 1

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER BOOK AID International

Sabre Foundation

Contents

Preface	xi
About the Authors	xv

PART I Introduction 1

CHAPTER 1 Systems Engineering Overview.....	3
1.1 Motivation for Systems Engineering	3
1.2 The Systems Engineering Process.....	4
1.3 Typical Application of the Systems Engineering Process.....	5
1.4 Multidisciplinary Systems Engineering Team.....	9
1.5 Codifying Systems Engineering Practice through Standards	11
1.6 Summary	13
1.7 Questions.....	14

CHAPTER 2 Model-Based Systems Engineering.....	15
2.1 Contrasting the Document-Based and Model-Based Approach	15
2.2 Modeling Principles	21
2.3 Summary	27
2.4 Questions	27

CHAPTER 3 SysML Language Overview	29
3.1 SysML Purpose and Key Features	29
3.2 SysML Diagram Overview	29
3.3 Using SysML in Support of MBSE	31
3.4 A Simple Example Using SysML for an Automobile Design.....	32
3.5 Summary	60
3.6 Questions	60

PART II Language Description 61

CHAPTER 4 SysML Language Architecture	63
4.1 The OMG SysML Language Specification	63
4.2 The Architecture of the SysML Language.....	65
4.3 SysML Diagrams	69
4.4 The Surveillance System Case Study.....	76
4.5 Chapter Organization for Part II	77
4.6 Questions.....	78

CHAPTER 5 Organizing the Model with Packages.....79

5.1	Overview.....	79
5.2	The Package Diagram	80
5.3	Defining Packages Using a Package Diagram	80
5.4	Organizing a Package Hierarchy.....	82
5.5	Showing Packageable Elements on a Package Diagram	85
5.6	Packages as Namespaces.....	85
5.7	Importing Model Elements into Packages.....	87
5.8	Showing Dependencies between Packageable Elements.....	89
5.9	Specifying Views and Viewpoints.....	91
5.10	Summary	92
5.11	Questions	93

CHAPTER 6 Modeling Structure with Blocks95

6.1	Overview.....	95
6.2	Modeling Blocks on a Block Definition Diagram	97
6.3	Modeling the Structure and Characteristics of Blocks Using Properties.....	99
6.4	Modeling Interfaces Using Ports and Flows	120
6.5	Modeling Block Behavior	128
6.6	Modeling Classification Hierarchies Using Generalization.....	134
6.7	Summary	144
6.8	Questions	145

CHAPTER 7 Modeling Constraints with Parametrics.....149

7.1	Overview.....	149
7.2	Using Constraint Expressions to Represent System Constraints	151
7.3	Encapsulating Constraints in Constraint Blocks to Enable Reuse	152
7.4	Using Composition to Build Complex Constraint Blocks	154
7.5	Using a Parametric Diagram to Bind Parameters of Constraint Blocks	155
7.6	Constraining Value Properties of a Block	159
7.7	Capturing Values in Block Configurations	159
7.8	Constraining Time-Dependent Properties to Facilitate Time-Based Analysis	161
7.9	Using Constraint Blocks to Constrain Item Flows.....	163
7.10	Describing an Analysis Context.....	163
7.11	Modeling Evaluation of Alternatives and Trade Studies	166
7.12	Summary	168
7.13	Questions	169

CHAPTER 8 Modeling Flow-Based Behavior with Activities171

8.1	Overview.....	171
8.2	The Activity Diagram	172
8.3	Actions—The Foundation of Activities	174

8.4	The Basics of Modeling Activities.....	176
8.5	Using Object Flows to Describe the Flow of Items between Actions.....	179
8.6	Using Control Flows to Specify the Order of Action Execution.....	187
8.7	Handling Signals and Other Events	191
8.8	Advanced Activity Modeling.....	193
8.9	Relating Activities to Blocks and Other Behaviors	200
8.10	Modeling Activity Hierarchies Using Block Definition Diagrams	206
8.11	Enhanced Functional Flow Block Diagram	208
8.12	Executing Activities.....	208
8.13	Summary	211
8.14	Questions	212
 CHAPTER 9 Modeling Message-Based Behavior with Interactions		 215
9.1	Overview.....	215
9.2	The Sequence Diagram	216
9.3	The Context for Interactions.....	216
9.4	Using Lifelines to Represent Participants in an Interaction	218
9.5	Exchanging Messages between Lifelines.....	220
9.6	Representing Time on a Sequence Diagram.....	225
9.7	Describing Complex Scenarios Using Combined Fragments.....	229
9.8	Using Interaction References to Structure Complex Interactions	234
9.9	Decomposing Lifelines to Represent Internal Behavior.....	235
9.10	Summary	238
9.11	Questions	239
 CHAPTER 10 Modeling Event-Based Behavior with State Machines		 241
10.1	Overview.....	241
10.2	State Machine Diagram.....	242
10.3	Specifying States in a State Machine	243
10.4	Transitioning between States	245
10.5	State Machines and Operation Calls.....	252
10.6	State Hierarchies	254
10.7	Contrasting Discrete versus Continuous States.....	263
10.8	Summary	264
10.9	Questions	266
 CHAPTER 11 Modeling Functionality with Use Cases.....		 269
11.1	Overview.....	269
11.2	Use Case Diagram.....	269
11.3	Using Actors to Represent the Users of a System.....	270
11.4	Using Use Cases to Describe System Functionality.....	271

11.5	Elaborating Use Cases with Behaviors	276
11.6	Summary.....	281
11.7	Questions.....	281
CHAPTER 12	Modeling Text-Based Requirements and Their Relationship to Design	283
12.1	Overview	283
12.2	Requirement Diagram.....	285
12.3	Representing a Text Requirement in the Model	285
12.4	Types of Requirements Relationships.....	287
12.5	Representing Cross-Cutting Relationships in SysML Diagrams.....	289
12.6	Depicting Rationale for Requirements Relationships.....	291
12.7	Depicting Requirements and Their Relationships in Tables.....	292
12.8	Modeling Requirement Hierarchies in Packages	294
12.9	Modeling a Requirements Containment Hierarchy	294
12.10	Modeling Requirement Derivation	296
12.11	Asserting That a Requirement Is Satisfied	298
12.12	Verifying That a Requirement Is Satisfied	298
12.13	Reducing Requirements Ambiguity Using the Refine Relationship	300
12.14	Using the General-Purpose Trace Relationship.....	303
12.15	Summary.....	304
12.16	Questions.....	305
CHAPTER 13	Modeling Cross-Cutting Relationships with Allocations	307
13.1	Overview	307
13.2	Allocation Relationship.....	308
13.3	Allocation Notation.....	308
13.4	Types of Allocation	311
13.5	Planning for Reuse: Specifying Definition and Usage in Allocation	314
13.6	Allocating Behavior to Structure Using Functional Allocation.....	317
13.7	Connecting Functional Flow with Structural Flow Using Functional Flow Allocation	323
13.8	Modeling Allocation between Independent Structural Hierarchies	327
13.9	Modeling Structural Flow Allocation	329
13.10	Evaluating Allocation across a User Model.....	331
13.11	Taking Allocation to the Next Step	332
13.12	Summary.....	333
13.13	Questions.....	334
CHAPTER 14	Customizing SysML for Specific Domains	335
14.1	Overview	335
14.2	Defining Model Libraries to Provide Reusable Constructs	339

14.3	Defining Stereotypes to Extend Existing SysML Concepts	341
14.4	Extending the SysML Language Using Profiles.....	346
14.5	Applying Profiles to User Models in Order to Use Stereotypes.....	347
14.6	Applying Stereotypes when Building a Model.....	348
14.7	Summary	354
14.8	Questions	356

PART III Modeling Examples 357

CHAPTER 15 Water Distiller Example Using Functional Analysis.....359

15.1	Stating the Problem.....	359
15.2	Defining the Model-Based Systems Engineering Approach.....	361
15.3	Organizing the Model	362
15.4	Establishing Requirements	364
15.5	Modeling Behavior	367
15.6	Modeling Structure	376
15.7	Analyzing Performance	382
15.8	Modifying the Original Design.....	386
15.9	Summary	396
15.10	Questions	396

CHAPTER 16 Residential Security System Example Using the Object-Oriented Systems Engineering Method.....397

16.1	Method Overview	397
16.2	Residential Security Example Overview and Project Setup.....	402
16.3	Applying the Method to Specify and Design the System	408
16.4	Summary	485
16.5	Questions	486

PART IV Transitioning to Model-Based Systems Engineering 487

CHAPTER 17 Integrating SysML into a Systems Development Environment

489

17.1	Understanding the System Model's Role in a Systems Development Environment.....	489
17.2	Integrating the Systems Modeling Tool with Other Tools	492
17.3	Data Exchange Mechanisms in an Integrated Systems Development Environment.....	500
17.4	Selecting a System Modeling Tool	504
17.5	Summary	507
17.6	Questions	507

CHAPTER 18 Deploying SysML into an Organization	509
18.1 Improvement Process	509
18.2 Summary	514
18.3 Questions	515
APPENDIX SysML Reference Guide	517
A.1 Overview.....	517
A.2 Notational Conventions	517
A.3 Package Diagram	519
A.4 Block Definition Diagram.....	521
A.5 Internal Block Diagram	525
A.6 Parametric Diagram.....	526
A.7 Activity Diagram.....	527
A.8 Sequence Diagram.....	531
A.9 State Machine Diagram.....	534
A.10 Use Case Diagram.....	537
A.11 Requirement Diagram	538
A.12 Allocation	541
A.13 Stereotypes.....	542
References.....	543
Index	545

Preface

Systems engineering is a multidisciplinary approach for developing solutions to complex problems. The increase in system complexity is demanding more rigorous and formalized systems engineering practices. In response to this demand, along with advancements in computer technology, the practice of systems engineering is undergoing a fundamental transition from a document-based approach to a model-based approach. In the model-based approach, the emphasis shifts from producing and controlling documentation to producing and controlling a coherent model of the system. Model-based systems engineering (MBSE) can help to manage complexity, while at the same time improve design quality and cycle time, improve communications among a diverse development team, and facilitate knowledge capture and design evolution.

A standardized and robust modeling language is considered a critical enabler for MBSE. The Systems Modeling Language (OMG SysML™) is a general-purpose modeling language that supports the specification, design, analysis, and verification of systems. These systems may include hardware, software, data, personnel, procedures, and facilities. SysML is a graphical modeling language with a semantic foundation for representing requirements, behavior, structure, and properties of the system and its components. The modeling language is intended to model systems from a broad range of industry domains such as aerospace, automotive, health care, and so on.

SysML is an extension of the Unified Modeling Language (UML), version 2, which has become the de facto standard software modeling language. Requirements were issued by the Object Management Group (OMG) in March 2003 to extend UML to support systems modeling. UML 2 was selected as the basis for SysML because it is a robust language that addresses many of the systems engineering needs, while at the same time, the systems engineering community is able to leverage the broad base of experience and tool vendors that support UML. This approach also facilitates the integration of systems and software modeling, which is becoming increasingly important for today's software-intensive systems.

The development of the language specification was a collaborative effort between members of the OMG, the International Council on Systems Engineering (INCOSE), and the AP233 Working Group of the International Standards Organization (ISO). Following three years of development, the OMG SysML specification was adopted by the OMG in May 2006 and the formal version 1.0 language specification was released in September 2007. Several vendors have now implemented SysML in their tools. It is expected that OMG SysML will continue to evolve through further revisions to the specification based on feedback from end users, tool vendors, and research activities. Information on SysML is available on the official OMG SysML Web site at <http://www.omg.sysml.org>.

This book provides the foundation for understanding and applying SysML to model systems as part of a model-based systems engineering approach. The book is organized into four parts including the Introduction, Language Description, Modeling Examples, and Transitioning to Model-Based Systems Engineering.

Part I, Introduction, contains an overview of systems engineering, a summary of key MBSE concepts, followed by an overview of SysML. The systems engineering overview and MBSE concepts in Chapters 1 and 2 set the context for SysML, and the language overview in Chapter 3 illustrates how the language is applied to a simple example.

Part II, Language Description, provides the detailed description of the language. Chapter 4 provides an overview of the language architecture, and Chapters 5 through 14 describe key concepts related to model organization, blocks, parameters, activities, interactions, states, use cases, requirements, allocations, and profiles. The ordering of the chapters and the concepts are not based on the ordering of activities in the systems engineering process, but are based on the dependencies between the language concepts. Each chapter builds the readers' understanding of the concepts by introducing language constructs: their meaning, notation, and examples of how they are used. The example used to demonstrate the language throughout Part II is a security surveillance system. This example should be understandable to most readers and has sufficient complexity to demonstrate the language concepts.

Part III, Modeling Examples, includes two examples to illustrate how SysML can support different model-based methods. The first example in Chapter 15 applies to the design of a water distiller system. It uses a simplified version of a classic functional analysis and allocation method that is applied to the design of a system that primarily controls physical processes. The second example in Chapter 16 applies to the design of a residential security system. It uses a comprehensive object-oriented systems engineering method (OOSEM) and emphasizes how the language is used to address a wide variety of systems engineering concerns, including black-box versus white-box design, logical versus physical design, and system distribution. While these two methods are considered representative of how systems engineering can be applied, SysML is intended to support a variety of other systems engineering methods.

Part IV, Transitioning to Model-Based Systems Engineering, addresses how to transition SysML into an organization. Chapter 17 is about how to integrate SysML into a systems development environment. It describes the type of data that are exchanged between a SysML tool and other classes of tools, and some of the types of data exchange mechanisms that can be used. The chapter also includes a discussion on the criteria for selecting a SysML tool. Chapter 18 in this part, and the last chapter of the book, describes how to deploy SysML into an organization. SysML is introduced into the organization along with model-based methods, tools, and training as part of a carefully planned and implemented improvement process.

Questions are included at the end of each chapter to test readers' understanding of the material. The answers to the questions can be found on the following Web site at <http://www.elsevierdirect.com/companions/9780123743794>.

The Appendix contains the SysML notation tables. These tables provide a reference guide for SysML notation along with a cross reference to the applicable sections in Part II of the book.

This book is a “practical guide” targeted to a broad spectrum of industry practitioners and students. It can serve as an introduction and reference for practitioners, as well as an introductory text for courses in systems modeling and its application to model-based systems engineering. In addition, because SysML reuses many UML concepts, software engineers familiar with UML can use this information as a basis for understanding systems engineering concepts. This will also help to bridge gaps in understanding between team members who have diverse expertise, such as is often the case with integrated systems and software engineering teams. Finally, many systems engineering concepts come to light when using an expressive language, and as such, this book can be used to help teach systems engineering concepts.

A first-time reader should pay close attention to the introductory chapters, may choose to do a cursory reading of Part II, and then review the simplified distiller example in Part III. A more advanced reader may choose to read the introductory chapters, do a more comprehensive review of Part II, and then review the residential security example in Part III. Part IV is of general interest to those interested in trying to introduce SysML and MBSE into their organization or project.

Acknowledgments

The authors wish to acknowledge the many individuals and their supporting organizations who participated in the development of SysML and provided valuable insights throughout the language development process. The individuals are too numerous to mention here but are listed in the OMG SysML specification. The authors wish to especially thank the reviewers of this book for their valuable feedback; they include Conrad Bock, Roger Burkhardt, Jeff Estefan, Doug Ferguson, Dr. Kathy Laskey, Dr. Leon McGinnis, Dr. Øystein Haugen, Dr. Chris Paredis, Dr. Russell Peak, and Bran Selic.

SysML is implemented in many different tools. For this book, we selected certain tools for representing the examples but are not endorsing them over other tools. We do wish, however, to acknowledge some vendors for the use of their tools, including Enterprise Architect by Sparx Systems, No Magic by Magic Draw, and the Microsoft Visio SysML template provided by Pavel Hruby.

The authors would also like to acknowledge the patience, steadfast support, and devotion of their wives throughout the development of this book: Linda Friedenthal, Emma Moore, and Sharon Steiner.

This page intentionally left blank

About the Authors

Sanford Friedenthal is a Principal System Engineer for Lockheed Martin Corporation. His experience includes the system life cycle from conceptual design, through development and production on a broad range of systems. He has also been a systems engineering department manager responsible for ensuring that systems engineering is implemented on programs. He has been a lead developer of advanced systems engineering processes and methods, including the Lockheed Martin Integrated Engineering Process and the Object-Oriented Systems Engineering Method (OOSEM). Sandy also was a leader of the industry team that developed SysML from its inception through its adoption by the OMG.

Mr. Friedenthal is well known within the systems engineering community for his role in leading the SysML effort and for his expertise in model-based systems engineering methods. He has been recognized as an INCOSE Fellow for these contributions. He has given many presentations on these topics to a wide range of professional and academic audiences both within and outside the US.

Alan Moore is an Architecture Modeling Specialist at The MathWorks and has extensive experience in the development of real-time and object-oriented methodologies and their application in a variety of problem domains. Previously at ARTiSAN Software Tools, he was responsible for the development and evolution of Real-time Perspective, ARTiSAN's process for real-time systems development. Alan has been a user and developer of modeling tools throughout his career, from early structured programming tools to UML-based modeling environments.

Mr. Moore is an active member of the Object Management Group and chaired both the finalization and revision task forces for the UML Profile for Schedulability and Performance and Time, and was a co-chair of the OMG's Real-time Analysis and Design Working Group. Alan also served as the language architect for the SysML Development Team.

Rick Steiner is an Engineering Fellow at Raytheon. He has focused on pragmatic application of systems engineering modeling techniques since 1993 and has participated in the International Council On Systems Engineering (INCOSE) Model Driven System Design Working Group since its inception.

He has been an internal advocate, consultant, and instructor of model-driven systems development within Raytheon. Rick has served as chief engineer, architect, or lead system modeler for several large-scale electronics programs, incorporating the practical application of the Object-Oriented Systems Engineering Method (OOSEM), and generation of Department of Defense Architecture Framework (DoDAF) artifacts from complex system models.

Mr. Steiner was a key contributor to the original requirements for SysML and also the development of the SysML specification. His main contribution to this specification was in the area of allocations, sample problems, and updates to requirements. He has provided frequent tutorials and presentations on SysML and model-driven system development at INCOSE symposia and meetings, NDIA conferences, and internal to Raytheon.

PART

Introduction

I

This page intentionally left blank

Systems Engineering Overview

1

The Object Management Group's OMG SysML™ [1] is a general-purpose graphical modeling language for representing systems that may include combinations of hardware, software, data, people, facilities, and natural objects. SysML supports the practice of model-based systems engineering (MBSE) that is used to develop system solutions in response to complex and often technologically challenging problems.

This chapter introduces the systems engineering approach independent of modeling concepts to set the context for how SysML is used. It describes the motivation for systems engineering, introduces the systems engineering process, and then describes a simplified automobile design example to highlight how complexity is addressed by the process. This chapter also summarizes the role of standards, such as SysML, to help codify the practice of systems engineering.

The next two chapters in Part I introduce model-based systems engineering and provide an overview of SysML. The language overview includes a simplified SysML model of the automobile design example introduced in this chapter.

1.1 Motivation for Systems Engineering

Whether it is an advanced military aircraft, a hybrid vehicle, a cell phone, or a distributed information system, these systems are expected to perform at levels undreamed of a generation ago. Competitive pressures demand that the systems leverage technological advances to provide continuously increasing capability at reduced costs and within shorter delivery cycles. The increased capability drives requirements for increased functionality, interoperability, performance, reliability, and smaller size.

The interconnectivity among systems also places increased demands on systems. Systems can no longer be treated as stand-alone, but behave as part of a larger whole that includes other systems as well as humans. Systems are expected to support many different uses as part of an interconnected system of systems (SoS). These uses drive evolving requirements that may not have been anticipated when the system was originally developed. An analogy can be made by looking at how the interconnectivity provided by email impacts the requirements on

day-to-day activities. Clearly, email can result in unanticipated requirements that affect who we communicate with, how often, and how we respond. The same is true for interconnected systems.

The practices to develop these systems must support these increasing demands. Systems engineering is an approach that has been dominant in the aerospace and defense industry to provide system solutions to technologically challenging and mission-critical problems. The solutions often include hardware, software, data, people, and facilities. Systems engineering practices have continued to evolve to address the added complexity of the interconnected SoS challenges, which are no longer limited to aerospace and defense systems. As a result, the systems engineering approach has been gaining broader recognition and acceptance across other industries such as automotive, telecommunications, and medical equipment, to name a few.

1.2 The Systems Engineering Process

Systems engineering is a multidisciplinary approach to develop balanced system solutions in response to diverse stakeholder needs. Systems engineering includes the application of both management and technical processes to achieve this balance and mitigate risks that can impact the success of the project. The management process is applied to ensure that development cost, schedule, and technical performance objectives are met. Typical management activities include planning the technical effort, monitoring technical performance, managing risk, and controlling the system technical baseline. The technical processes are applied to specify, design, and verify the system to be built. The practice of systems engineering is not static, but continues to evolve to deal with increasing demands.

A simplified view of the systems engineering technical processes is shown in Figure 1.1. The *System Specification and Design* process is used to specify the system and component requirements to meet stakeholder needs. The components are then designed, implemented, and tested to ensure that they conform with their requirements. The *System Integration and Test* process includes activities to integrate the components into the system and verify that the system requirements are

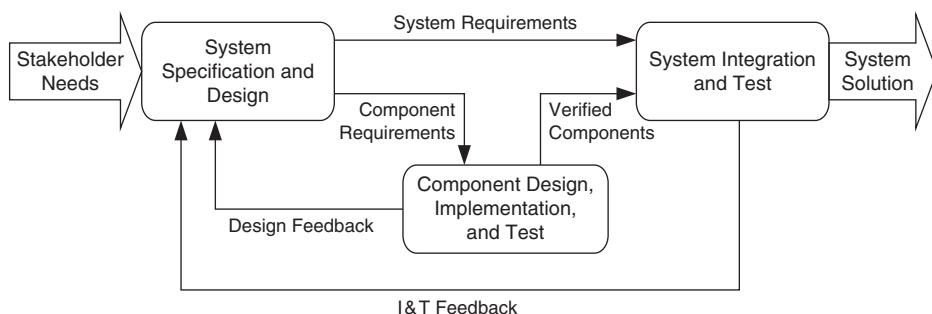


FIGURE 1.1

Simplified systems engineering technical processes.

satisfied. These processes are applied iteratively throughout the development of the system, with ongoing feedback between the different processes. In more complex applications, there are multiple levels of system decomposition beginning at an enterprise or SoS level. In those cases, variants of this process are applied recursively to each intermediate level of the design down to the level at which the components are purchased or built.

The *System Specification and Design* process includes the following activities to provide a balanced system solution that satisfies the diverse stakeholders' needs:

- *Elicit and analyze stakeholder needs* to understand the problem to be solved, the goals the system is intended to support, and the effectiveness measures needed to evaluate how well the system supports the goals
- *Specify system* functionality, interfaces, physical and performance characteristics, and other quality characteristics required of the system to support the goals and effectiveness measures
- *Synthesize alternative system solutions* by partitioning the system design into components that can satisfy the system requirements
- *Perform trade-off analysis* to evaluate and select a preferred solution that satisfies system requirements and provides optimum balance to achieve the overall effectiveness measures
- *Maintain traceability* from the system goals to the system and component requirements and verification results to ensure that requirements and stakeholder needs are addressed

1.3 Typical Application of the Systems Engineering Process

The *System Specification and Design* process can be illustrated by applying this process to an automobile design. A multidisciplinary systems engineering team is responsible for executing this process. The participants and roles of a typical systems engineering team are discussed in Section 1.4.

The team must first identify the stakeholders and analyze their needs. Stakeholders include the purchaser of the car and the users of the car. In this example, the user includes the driver and the passengers. Each of their needs must be addressed. Stakeholder needs further depend on the particular market segment, such as a family car versus a sports car versus a utility vehicle. For this example, we assume the automobile is targeted toward a typical mid-career individual who uses the car for his or her daily transportation needs.

In addition, a key tenet of systems engineering is to address the needs of other stakeholders who may be impacted throughout the system life cycle, so additional stakeholders include the manufacturers that produce the automobile and those who maintain the automobile. Each of their concerns must be addressed to ensure a balanced life-cycle solution. Less obvious stakeholders are governments that express their needs via laws and regulations. Clearly, each stakeholder's concern is not of equal importance, and therefore stakeholder concerns must be properly

6 CHAPTER 1 Systems Engineering Overview

weighted. Analysis is performed to understand the needs of each stakeholder and define effectiveness measures with target values. Target values are used to bound the solution space, to evaluate the alternatives, and to discriminate the solution from competitor solutions. In this example, the effectiveness measures may relate to aesthetics, performance, fuel economy, safety, reliability, repair time, and production cost.

The system requirements are specified to address stakeholder needs and associated effectiveness measures. This begins with a definition of the system boundary so that clear interfaces can be established between the system and external systems and users as shown in Figure 1.2. In this example, the driver and passengers are external users who interact with the automobile. The gas pump and maintenance equipment are examples of external systems that the vehicle interacts with. In addition, the vehicle interacts with the physical environment such as the road. All of these external systems, users, and the physical environment must be specified to clearly demarcate the system boundary and its associated interfaces.

The functional requirements for the automobile are specified by analyzing what the system must do to support its overall goals. This vehicle must perform functions related to accelerating, braking, and steering, and many additional functions to address driver and passenger needs. The functional analysis identifies the inputs and outputs for each function. As shown in the example in Figure 1.3, the functional requirement to accelerate the automobile requires an acceleration input from the driver and produces outputs that correspond to the automobile forces and the speedometer reading for the driver. The functional requirements analysis also includes specifying the sequence and ordering of the functions.

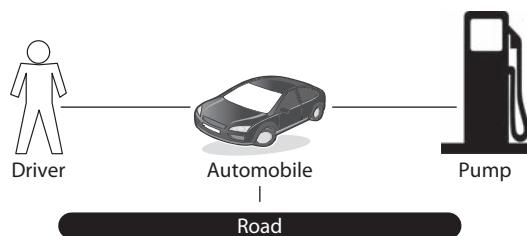


FIGURE 1.2

Defining the system boundary.

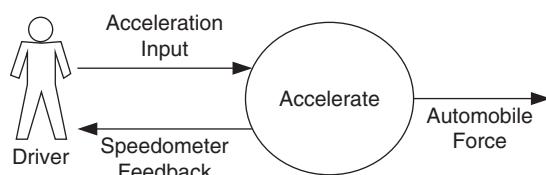
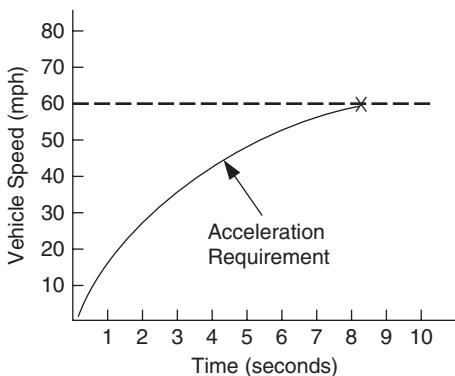


FIGURE 1.3

Specifying the functional requirements.

**FIGURE 1.4**

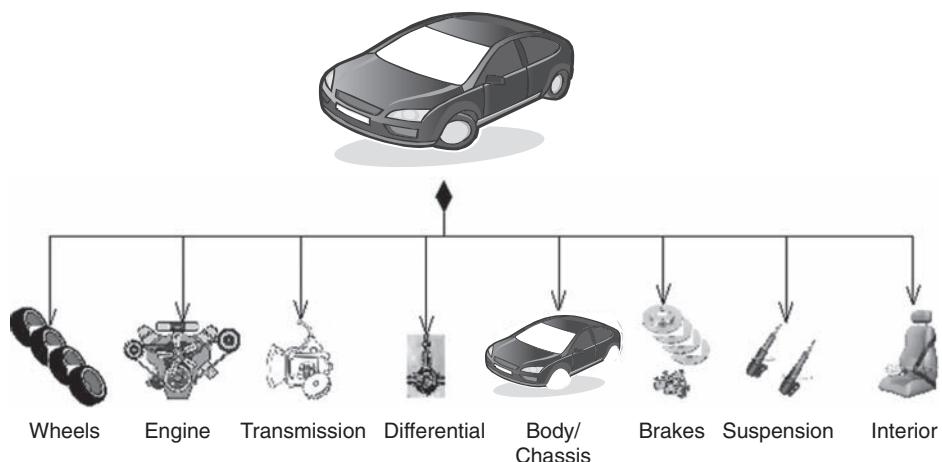
Automobile performance requirements.

Functional requirements must also be evaluated to determine the required level of performance. As indicated in Figure 1.4, the automobile is required to accelerate from 0 to 60 miles per hour (mph) in less than 8 seconds under specified conditions. Similar performance requirements can be specified for stopping distance from 60 to 0 mph and for steering requirements such as the turning radius.

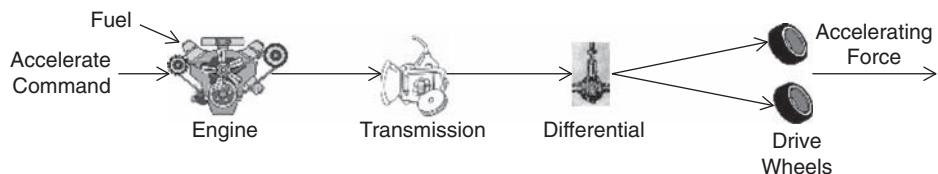
Additional requirements are specified to address the concerns of each stakeholder. Example requirements include specifying riding comfort, fuel efficiency, reliability, maintainability, safety, and emissions. Physical characteristics, such as maximum vehicle weight, may be derived from the performance requirements, or maximum vehicle length may be dictated by other concerns such as standard parking space dimensions. The system requirements must be clearly traceable to stakeholder needs and validated to ensure that the requirements address their needs. The early and ongoing involvement of representative stakeholders in this process is a critical success factor to the overall development effort.

The system design involves identifying system components and specifying requirements for the components needed to satisfy system-level requirements. This may involve first developing a logical system design independent of the technology used, and then a physical system design that reflects specific technology selections. In the example shown in Figure 1.5, the system's physical components include the *Engine*, *Transmission*, *Differential*, *Chassis*, *Body*, *Brakes*, and so on. This system includes a single level of decomposition from the system to component level. However, as indicated earlier, more complex systems may include multiple levels of system decomposition.

Design constraints are often imposed on the solution. A common kind of constraint is to reuse a particular component. For example, there might be a requirement to reuse the engine from the inventory of existing engines. This constraint implies that no additional engine development is to be performed. Although design constraints are typically imposed to save time and money, sometimes analysis reveals that relaxing the constraint would be less expensive and faster. For example, if the engine is reused, expensive filtering equipment might be needed to satisfy newly imposed pollution regulations. On the other hand, the cost of a

**FIGURE 1.5**

Automobile system decomposes into its components.

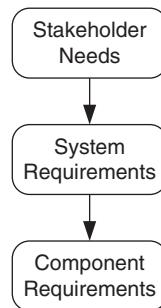
**FIGURE 1.6**

Interaction among components to achieve the functional and performance requirements.

redesign to incorporate newer technology might be easily recovered over the automobile's life cycle. Systems engineers should examine the rationale behind design constraints and inform stakeholders whether the analysis validates the assumptions behind the constraints.

The components are specified such that if their requirements are satisfied, the system requirements are also satisfied. The power subsystem shown in Figure 1.6 includes the *Engine*, *Transmission*, and *Differential* components, and must provide the power to accelerate the automobile. Similarly, the steering subsystem must control the direction of the vehicle, and the braking subsystem must decelerate the vehicle.

The system performance and physical requirements are allocated to the component requirements. This involves engineering analysis to determine how the system performance requirements, such as the vehicle acceleration, must be allocated to the component performance requirements such as engine horsepower, coefficient of drag of the body, and the weight of each component. Similar analysis must be performed to allocate the other system performance requirements related to fuel economy, fuel emissions, reliability, and cost. The requirements for riding comfort may require multiple analysis that address human factors considerations

**FIGURE 1.7**

Stakeholder needs flow down to system and component requirements.

related to road vibration, acoustic noise propagation to the vehicle's interior space-volume analysis, and placement of displays and controls, to name a few.

The system design alternatives are evaluated to determine the preferred system solution that achieves a balanced design that addresses multiple competing requirements. In this example, the acceleration and fuel economy requirement may result in evaluating alternative engine design configurations, such as a 4-cylinder versus a 6-cylinder engine. The alternative designs are then evaluated based on criteria that are traceable to the system requirements and effectiveness measures. The preferred solution is validated with the stakeholders to ensure that it addresses their needs.

The component requirements are input to the *Component Design, Implementation, and Test* process from Figure 1.1. The component developers provide feedback to the systems engineering team to ensure that component requirements can be satisfied by their designs, or they may request that the component requirements be reallocated. This is an iterative process throughout development that is often required to achieve a balanced design solution. The system and component requirements are also provided to the *System Integration and Test Team* to develop the necessary test procedures to verify that the system satisfies its requirements.

As indicated in Figure 1.7, requirements traceability is maintained between the *Stakeholder Needs*, the *System Requirements*, and the *Component Requirements* to ensure design integrity. For this example, the system and component requirements, such as vehicle acceleration, vehicle weight, and engine horsepower, can be traced to the stakeholder needs associated with performance and fuel economy.

A systematic process to develop a balanced system solution that satisfies diverse stakeholder needs becomes essential as system complexity increases. An effective application of systems engineering requires maintaining a broad system perspective that focuses on the overall system goals and the needs of each stakeholder, while at the same time maintaining attention to detail and rigor that will ensure the integrity of the system design. The expressiveness and level of precision of SysML is intended to enable this process.

1.4 Multidisciplinary Systems Engineering Team

To represent the broad set of stakeholder perspectives, systems engineering requires participation from many engineering and nonengineering disciplines. The

participants must have an understanding of the end-user domain, such as the drivers of the car, and the domains that span the system life cycle such as manufacturing and maintenance. The participants must also have knowledge of the system's technical domains such as the power and steering subsystems. The participants must also include understanding of the specialty engineering domains, such as reliability, safety, and human factors, to support the system design trade-offs. In addition, they must have sufficient participation from the component developers and testers to ensure the specifications are implementable and verifiable.

A typical **multidisciplinary systems engineering** team should include representation from each of these perspectives. The extent of participation depends on the complexity of the system and the knowledge of the team members. A systems engineering team on a small project may include a single systems engineer, who has broad knowledge of the domain and can work closely with the hardware and software development team and the test team. On the other hand, the development of a large system may involve a systems engineering team led by a systems engineering manager who plans and controls the systems engineering effort. This project may include tens or hundreds of systems engineers with varying expertise.

A typical multidisciplinary systems engineering team is shown in Figure 1.8. The *Systems Engineering Management Team* is responsible for the management activities related to planning and control of the technical effort. The *Requirements Team* analyzes stakeholder needs, develops the concept of operations, and specifies and validates system requirements. The *Architecture Team* is responsible for developing the system architecture design in terms of system components and their interactions and interconnections. This includes allocating the system requirements to the components that may include hardware and software specifications. The *Systems Analysis Team* is responsible for performing the engineering analysis on different aspects of the system, such as performance and physical characteristics, reliability, maintainability, and cost. The *Integration and Test Team* is responsible for developing test plans and procedures and conducting tests. There are many different organizational structures to accomplish similar roles, and individuals may participate in different roles on different teams.

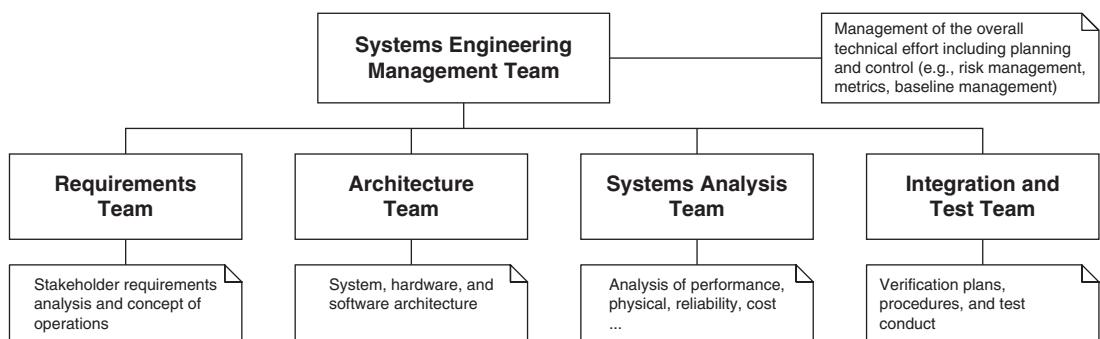


FIGURE 1.8

A typical multidisciplinary systems engineering team needed to represent diverse stakeholder perspectives.

1.5 Codifying Systems Engineering Practice through Standards

As mentioned earlier, systems engineering has been a dominant practice within the aerospace and defense industries to engineer complex, mission-critical systems that leverage advanced technology. These systems include land-, sea-, air-, and space-based platforms; weapon systems; command, control, and communications systems; and logistics systems, to name a few. The emphasis has shifted to treat a system as part of a larger whole, which is sometimes referred to as a **system of systems** (SoS) or an **enterprise**.

The complexity of systems being developed in other industry sectors has dramatically increased due to the competitive demands and technological advances discussed earlier in this chapter. Specifically, many commercial products incorporate the latest processing and networking technology that have significant software content with substantially increased functionality and are more interconnected with increasingly complex interfaces. The products are being used in new ways, such as the integration of cell phones with cameras and the use of global positioning systems in automobiles, that were not previously envisioned. The practice of systems engineering is evolving to other industries to help deal with this complexity. The need to establish standards for systems engineering concepts, terminology, processes, and methods has become increasingly important to advance and institutionalize the practice of systems engineering across industry sectors.

Systems engineering standards have evolved over the last several years. Figure 1.9 shows a partial taxonomy of standards that includes systems engineering process standards, architecture frameworks, modeling standards, and data exchange standards. A comprehensive standards-based approach to systems engineering may implement at least one standard from each layer of this taxonomy.

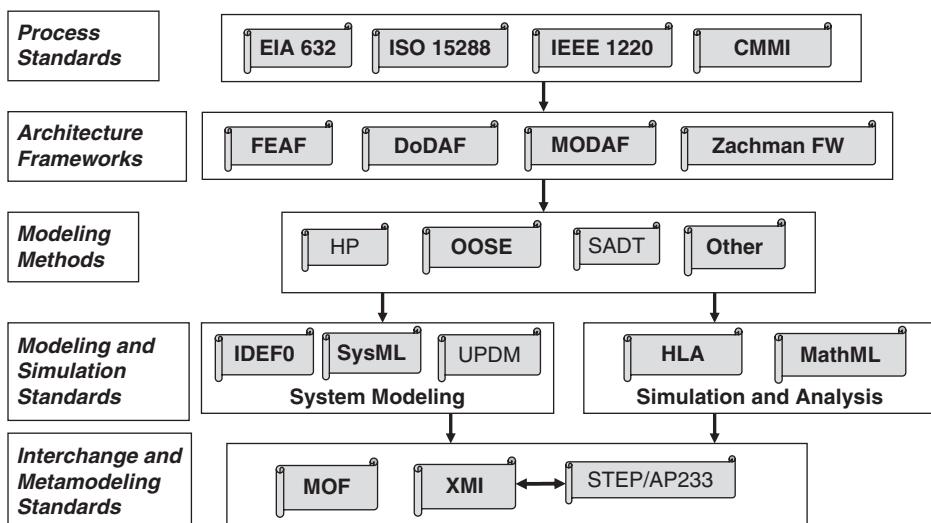


FIGURE 1.9

A partial systems engineering standards taxonomy.

A primary emphasis for systems engineering standards has been on developing process standards that include EIA 632 [2], IEEE 1220 [3], and ISO 15288 [4]. These standards address broad industry needs and reflect the fundamental tenets of systems engineering that provide a foundation for establishing a systems engineering approach.

The systems engineering process standards share much with software engineering practices. Management practices for planning, as an example, are similar whether it is for complex software development or systems development. As a result, there has been significant emphasis in the standards community on aligning the systems and software standards where practical.

The systems engineering process defines what activities are performed, but does not generally give details on how they are performed. A **systems engineering method** describes how the activities are performed in terms of the types of artifacts that are produced and how they are developed. For example, an important systems engineering artifact is the concept of operations. As its name implies, the **concept of operations** defines what the system is intended to do from the stakeholders' perspective. It depicts the interaction of the system with its external systems, but may not show any of the system's internal operations. Different methods may use different techniques and representations for developing a concept of operations. The same is true for many other systems engineering artifacts.

Examples of systems engineering methods are identified in a Survey of Model-Based Systems Engineering Methods [5] and include Harmony [6, 7], the Object-Oriented Systems Engineering Method (OOSEM) [8], the Rational Unified Process for Systems Engineering (RUP SE) [9, 10], the State Analysis method [11], and the Vitech Model-Based Systems Engineering Method [12]. Many organizations have internally developed processes and methods as well. The methods are not official industry standards, but de facto standards may emerge as they prove their value over time. Criteria for selecting a method include its ease of use, its ability to address the range of systems engineering concerns, and the level of tool support. The two example problems in Part III include the use of SysML with a functional analysis and allocation method and a use case driven method (OOSEM). SysML is intended to support many different systems engineering methods.

In addition to systems engineering process standards and methods, several standard frameworks have emerged to support system architecting. An architecture framework includes specific concepts, terminology, artifacts, and taxonomies for describing the architecture of a system. The Zachman framework [13] was introduced in the 1980s to define enterprise architectures; it defines a standard set of stakeholder perspectives and a set of artifacts that address fundamental questions associated with each stakeholder group. The C4ISR framework [14] was introduced in 1996 to provide a framework for architecting information systems for the U.S. Department of Defense (DoD). The Department of Defense Architecture Framework (DoDAF) [15] evolved from the C4ISR framework to support architecting a SoS for the defense industry by defining the architecture's operational, system, and technical views.

The United Kingdom introduced a variant of DoDAF called the Ministry of Defence Architecture Framework (MODAF) [16] that added the strategic

and acquisition view. The IEEE 1471-2000 standard was approved in 2000 as a “Recommended Practice for Architectural Description of Software-Intensive Systems” [17]. This practice provides additional fundamental concepts, such as the concept of view and viewpoint that applies to both software and systems architecting. The Open Group Architecture Framework (TOGAF) [18] was originally approved in the 1990s as a method for developing architectures.

Modeling standards is another class of systems engineering standards that provide a common language for describing systems. Behavioral models and functional flow diagrams have been de facto modeling standards for many years and have been broadly used by the systems engineering community. The Integration Definition for Functional Modeling (IDEF0) [19] was issued by the National Institute of Standards and Technology in 1993. The OMG SysML specification was adopted in 2006 by the Object Management Group as a general-purpose graphical systems modeling language that extends the Unified Modeling Language (UML) and is the subject of this book. Several other extensions of UML have been developed for specific domains. The Unified Profile for DoDAF/MODAF (UPDM) is being developed to describe system of systems and enterprise architectures that are compliant with DoDAF and MODAF requirements. The foundation for the UML-based modeling languages is the OMG Meta Object Facility (MOF) [20], which is a language that is used to specify other modeling languages.

Model and data interchange standards is a critical class of standards that supports model and data exchange among tools. Within the OMG, the XML Metadata Interchange (XMI) specification [21] supports interchange of model data when using a MOF-based language such as UML, SysML, or another UML profile. XMI is summarized in Chapter 17. Another data exchange standard for interchange of systems engineering data is ISO 10303 (AP233) [22], which is also briefly described in Chapter 17.

Additional modeling standards from the Object Management Group relate to the Model Driven Architecture (MDA[®]) [23]. MDA includes a set of concepts that include creating both technology-independent and technology-dependent models. The MDA standards enable transformation between the models and different modeling languages. MDA encompasses OMG standards in both the modeling language and data exchange layers of Figure 1.9.

The development and evolution of these standards are all part of a trend toward a standards-based approach to the practice of systems engineering. Such an approach enables common training, tool interoperability, and reuse of system specification and design artifacts. It is expected that this trend will continue as systems engineering becomes prevalent across a broader range of industries.

1.6 Summary

Systems engineering is a multidisciplinary approach to transform a set of stakeholder needs into a balanced system solution that meets those needs. Systems engineering is a key practice to address complex and often technologically challenging

problems. The process includes activities to establish top-level goals that a system must support, specify system requirements, synthesize alternative system designs, evaluate the alternatives, allocate requirements to the components, integrate the components into the system, and verify that the system requirements are satisfied. It also includes essential planning and control processes needed to manage a technical effort.

Multidisciplinary teams are an essential element of systems engineering to address the diverse stakeholder perspectives and technical domains to achieve a balanced system solution. The practice of systems engineering continues to evolve with an emphasis on dealing with systems as part of a larger whole. Systems engineering practices are becoming codified in various standards, which is essential to advancing and institutionalizing the practice across industry domains.

1.7 Questions

- 1. What are some of the demands that are driving system development?**
- 2. What is the purpose of systems engineering?**
- 3. What are the key activities in the system specification and design process?**
- 4. Who are the typical stakeholders that span a system's life cycle?**
- 5. What are different types of requirements?**
- 6. Why is it important to have a multidisciplinary systems engineering team?**
- 7. What are some of the roles on a typical systems engineering team?**
- 8. What role do standards play in systems engineering?**

Model-Based Systems Engineering

2

Model-based systems engineering (MBSE) applies systems modeling as part of the systems engineering process described in Chapter 1 to support analysis, specification, design, and verification of the system being developed. A primary artifact of MBSE is a coherent model of the system being developed. This approach enhances communications, specification and design precision, design integration, and reuse of system specification and design artifacts.

This chapter summarizes MBSE concepts to provide further context for SysML without emphasizing the specific modeling language, method, or tools. MBSE is contrasted with the more traditional document-based approach to motivate the use of MBSE and highlight its benefits. Principles for effective modeling are also discussed.

2.1 Contrasting the Document-Based and Model-Based Approach

The following sections contrast the document-based approach and the model-based approach to systems engineering.

2.1.1 Document-Based Systems Engineering Approach

Traditionally, large projects have employed a **document-based systems engineering** approach. This approach is characterized by the generation of textual specifications and design documents, in hard-copy or electronic file format, that are then exchanged between customers, users, developers, and testers. System requirements and design information are expressed in these documents and drawings. The systems engineering emphasis is placed on controlling the documentation and ensuring the documents and drawings are valid, complete, and consistent, and that the developed system complies with the documentation.

In the document-based approach, specifications for a particular system, its subsystems, and its hardware and software components are usually depicted in a

hierarchical tree, called a **specification tree**. A systems engineering management plan (SEMP) documents how the systems engineering process is employed on the project, and how the engineering disciplines work together to develop the documentation needed to satisfy the requirements in the specification tree. Systems engineering activities are planned by estimating the time and effort required to generate documentation, and progress is then measured by the state of completion of the documents.

Document-based systems engineering typically relies on a concept of operation document to define how the system is used to support the required mission or objective. Functional analysis is performed to allocate the top-level system functions to the components of the system. Drawing tools are used to capture the system design, such as the functional flow diagrams or schematic block diagrams. These diagrams are stored as separate files and included in the system design documentation. Engineering trade studies and analyses are performed and documented by many different disciplines to evaluate and optimize alternative designs and allocate performance requirements. The analysis may be supported by individual analysis models for performance, reliability, safety, mass properties, and other aspects of the system.

Requirements traceability is established and maintained in the document-based approach by tracing requirements between the specifications at different levels of the specification hierarchy. Requirements management tools are used to parse requirements contained in the specification documents and capture them in a requirements database. The traceability between requirements and design is maintained by identifying the part of the system or subsystem that satisfies the requirement, and/or the verification procedures used to verify the requirement, and then reflecting this in the requirements database.

The document-based approach can be rigorous but has some fundamental limitations. The completeness, consistency, and relationships between requirements, design, engineering analysis, and test information are difficult to assess since this information is spread across several documents. This makes it difficult to understand a particular aspect of the system and to perform the necessary traceability and change impact assessments. This, in turn, leads to poor synchronization between system-level requirements and design and lower-level hardware and software design. It also makes it difficult to maintain or reuse the system requirements and design information for an evolving or variant system design. Also, progress of the systems engineering effort is based on the documentation status that may not adequately reflect the system requirements and design quality. These limitations can result in inefficiencies and potential quality issues that often show up during integration and testing, or worse, after the system is delivered to the customer.

2.1.2 Model-Based Systems Engineering Approach

A model-based approach has been standard practice in electrical and mechanical design and other disciplines for many years. Mechanical engineering transitioned from the drawing board to increasingly more sophisticated two-dimensional (2D) and then three-dimensional (3D) computer-aided design tools beginning in the

1980s. Electrical engineering transitioned from manual circuit design to automated schematic capture and circuit analysis in a similar time frame. Computer-aided software engineering became popular in the 1980s for using graphical models to represent software at abstraction levels above the programming language. The use of modeling for software development is becoming more widely adopted, particularly since the advent of the Unified Modeling Language in the 1990s.

The model-based approach is becoming more prevalent in systems engineering. A mathematical formalism for MBSE was introduced in 1993 [24]. The increasing capability of computer processing, storage, and network technology along with emphasis on systems engineering standards has created an opportunity to significantly advance the state of the practice of MBSE. It is expected that MBSE will become standard practice in a similar way that it has with other engineering disciplines.

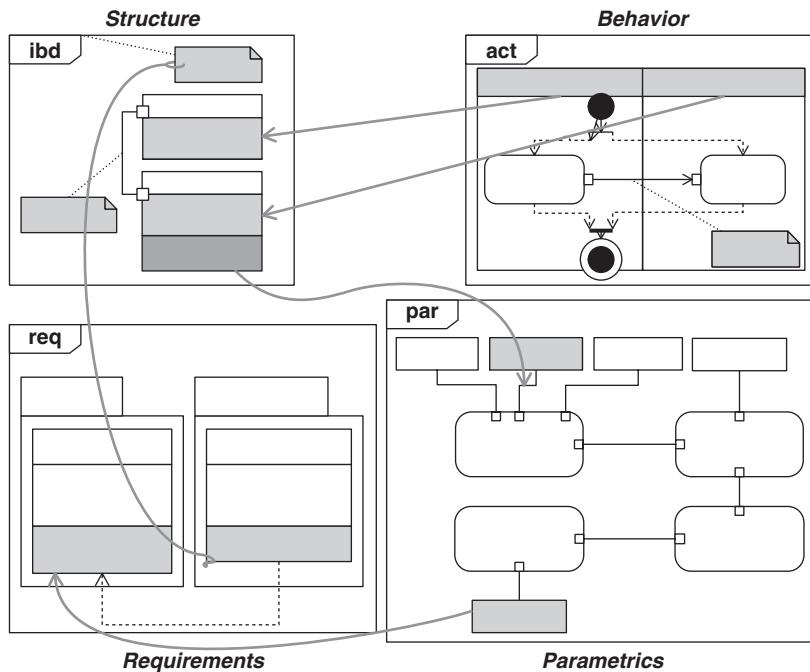
“Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases” [25]. MBSE is intended to facilitate systems engineering activities that have traditionally been performed using the document-based approach and result in enhanced communications, specification and design precision, system design integration, and reuse of system artifacts. The output of the systems engineering activities is a coherent model of the system (i.e., system model), where the emphasis is placed on evolving and refining the model using model-based methods and tools.

The System Model

The **system model** is generally created using a modeling tool and contained in a model repository. The system model includes system specification, design, analysis, and verification information. The model consists of elements that represent requirements, design elements, test cases, design rationale, and their interrelationships. Figure 2.1 shows the system model as an interconnected set of model elements that represent key system aspects as defined in SysML, including its structure, behavior, parametrics, and requirements.

A primary use of the system model is to design a system that satisfies system requirements and allocates the requirements to the system’s components. Figure 2.2 depicts how the system model is used to specify the components of the system. The system model includes component interconnections and interfaces, component interactions and related functions components must perform, and component performance and physical characteristics. The textual requirements for the components may also be captured in the model and traced to system requirements.

In this regard, the system model is used to specify the component requirements and can be used as an agreement between the system designer and the subsystem and/or component developer. The component developers receive the component requirements in a way that is meaningful to them either through a model data exchange mechanism or by providing documentation that is automatically generated from the model. The component developer can provide information about how the component design complies with its requirements in a similar way. The use of a system model provides a mechanism to specify and integrate

**FIGURE 2.1**

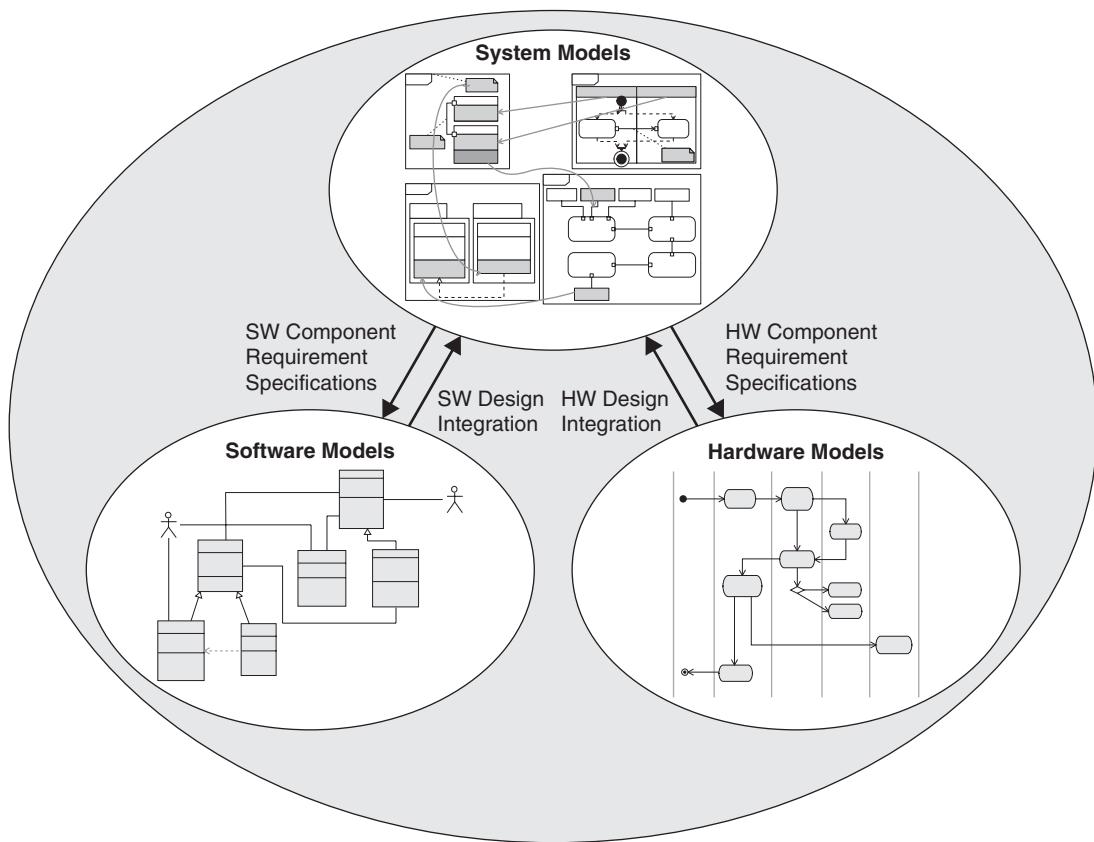
Representative system model example in SysML. (Specific model elements have been deliberately obscured and will be discussed in subsequent chapters.)

subsystems and components into the system and maintain traceability to higher-level requirements.

The system model can also be integrated with engineering analysis and simulation models to perform computation and dynamic execution. If the system model is executed directly, the system modeling environment must be augmented with an execution environment. A brief discussion of executable models is included in Chapter 17.

The Model Repository

The model elements are stored in a **model repository** and depicted on diagrams by graphical symbols. The tool enables the modeler to create, modify, and delete individual model elements and their relationships in the model repository. The modeler uses the symbols on the diagrams to enter the information into the model repository and to view model repository information. The specification, design, analysis, and verification information previously captured in documents is now captured in the model repository. The model can be viewed in diagrams or tables or in reports generated by querying the model repository. The views enable understanding and analysis of different aspects of the same system model. The documents can continue to serve as an effective means for reporting the information,

**FIGURE 2.2**

The system model is used to specify the components of the system.

but in MBSE, the information contained in documentation is generated from the model. In fact, many of the modeling tools have flexible and automated document-generation capability that can significantly reduce the time and cost of building and maintaining the system specification and design documentation.

Model elements corresponding to requirements, design, analysis, and verification information are traceable to one another through their relationships, even if they are represented on different diagrams. For example, an engine component in an automobile system model may have many relationships to other elements in the model. The engine, which is part of the automobile system, is connected to the transmission, satisfies a power requirement, performs a function to convert fuel to mechanical energy, and has a weight property that contributes to the vehicle's weight.

The static semantics of the model impose rules that constrain which relationships can exist. For example, the model should not allow a requirement to contain a system component or an activity to produce inputs instead of outputs. Additional model constraints may be imposed based on the method being employed. An example of a method-imposed constraint may be that all system functions must

be decomposed and allocated to a component of the system. Modeling tools are expected to enforce constraints at the time the model is constructed, or by running a model-checker routine at the modeler's convenience and providing a report of the constraint violations.

The model provides much finer-grain control of the information than is available in a document-based approach, where this information may be spread across many documents and the relationships may not be explicitly defined. The model-based approach promotes rigor in the specification, design, analysis, and verification process. It also significantly enhances the quality and timeliness of traceability and impact assessment over the document-based approach.

Transitioning to MBSE

Models have been used as part of the document-based systems engineering approach for many years, and include functional flow diagrams, behavior diagrams, schematic block diagrams, N2 charts, performance simulations, and reliability models, to name a few. However, the use of models has generally been limited in scope to support specific types of analysis or selected aspects of system design. The individual models have not been integrated into a coherent model of the overall system, and the modeling activities have not been integrated into the systems engineering process. The transition from document-based systems engineering to MBSE is a shift in emphasis from controlling the documentation about the system to controlling the model of the system. MBSE integrates system requirements, design, analysis, and verification models to address multiple aspects of the system in a cohesive manner, rather than a disparate collection of individual models.

MBSE provides an opportunity to address many of the limitations of the document-based approach by providing a more rigorous means for capturing and integrating system requirements, design, analysis, and verification information, and facilitating the maintenance, assessment, and communication of this information across the system's life cycle. Some of the MBSE potential benefits include the following:

- Enhanced communications
 - Shared understanding of the system across the development team and other stakeholders
 - Ability to integrate views of the system from multiple perspectives
- Reduced development risk
 - Ongoing requirements validation and design verification
 - More accurate cost estimates to develop the system
- Improved quality
 - More complete, unambiguous, and verifiable requirements
 - More rigorous traceability between requirements, design, analysis, and testing
 - Enhanced design integrity
- Increased productivity
 - Faster impact analysis of requirements and design changes
 - Reuse of existing models to support design evolution

- Reduced errors and time during integration and testing
- Automated document generation
- Enhanced knowledge transfer
 - Specification and design information captured in a standard format that can be accessed via query and retrieval

MBSE can provide additional rigor in the specification and design process when implemented using appropriate methods and tools. However, this rigor does not come without a price. Clearly, transitioning to MBSE underscores the need for up-front investment in processes, methods, tools, and training. It is expected that during the transition, MBSE is performed in combination with document-based approaches. For example, the upgrade of a large, complex legacy system still relies heavily on the legacy documentation, and only parts of the system may be modeled. Careful tailoring of the approach and scoping of the modeling effort is essential to meet the needs of a particular project. The considerations for transitioning to MBSE are discussed in Chapter 18.

2.2 Modeling Principles

The following sections provide a brief overview of some of the key modeling principles.

2.2.1 Model and MBSE Method Definition

A **model** is a representation of one or more concepts that may be realized in the physical world. It generally describes a domain of interest. A key feature of a model is that it is an abstraction that does not contain all the detail of the modeled entities within the domain of interest. Models are represented in many forms including graphical, mathematical, and logical representations, and physical prototypes. For example, a model of a building may include a blueprint and a scaled prototype physical model. The building blueprint is a specification for one or more buildings that are built. The blueprint is an abstraction that does not contain all the building's detail such as the characteristics of its materials.

A SysML model is analogous to a building blueprint that specifies a system to be implemented. Instead of a geometric representation of the system, the SysML model represents the behavior, structure, properties, constraints, and requirements of the system. SysML has a semantic foundation that specifies the types of model elements and the relationships that can appear in the system model. The model elements that comprise the system model are stored in a model repository and can be represented graphically. A SysML model can also be simulated if it is supported by an execution environment.

A **method** is a set of related activities, techniques, and conventions that implement one or more processes and is generally supported by a set of tools. A **model-based systems engineering method** can be characterized as a method that implements all or part of the systems engineering process, and it produces a system model as one of its primary artifacts.

2.2.2 The Purpose for Modeling a System

The purpose for modeling a system for a particular project must be clearly defined in terms of the expected results of the modeling effort, the stakeholders who use the results, and how the results are intended to be used. The model purpose is used to determine the scope of the modeling effort in terms of model breadth, depth, and fidelity. This scope should be balanced with the available schedule, budget, skill levels, and other resources. Understanding the purpose and scope provides the basis for establishing realistic expectations for the modeling effort. The purposes for modeling a system may emphasize different aspects of the systems engineering process or support other life-cycle uses, including the following:

- Characterize an existing system
- Specify and design a new or modified system
 - Represent a system concept
 - Specify and validate system requirements
 - Synthesize system designs
 - Specify component requirements
 - Maintain requirements traceability
- Evaluate the system
 - Conduct system design trade-offs
 - Analyze system performance requirements or other quality attributes
 - Verify that the system design satisfies its requirements
 - Assess the impact of requirements and design changes
- Train users on how to operate or maintain a system

2.2.3 Establishing Criteria to Meet the Model Purpose

Criteria can be established to assess how well a model can meet its modeling purpose. However, one must first distinguish between a good model and a good design. One can have a good model of a poor design or a poor model of a good design. A good model meets its intended purpose. A good design is based on how well the design satisfies its requirements and the extent to which it incorporates quality design principles. As an example, one could have a good model of a chair that meets its intended purpose by providing an accurate representation of the modeled system. However, the chair's design may be a poor design if it does not have structural integrity. A good model provides visibility to aid the design team in identifying issues and assessing design quality. The selected MBSE method and tools should facilitate a skilled team to develop both a good model and a good design.

The answers to the following questions can be used to assess the goodness of the model and derive quality attributes of it. The quality attributes in turn can be used to establish preferred modeling practices.

Is the model's scope sufficient to meet its purpose?

Assuming the purpose is clearly defined as described earlier, the scope of the model is defined in terms of its breadth, depth, and fidelity. The model scope significantly impacts the level of resources required to support the modeling effort.

Model breadth. The breadth of the model must be sufficient for the purpose by determining which parts of the system need to be modeled. This question is particularly relevant to large systems where one may not need to model the entire system to meet project needs. If new functionality is being added to an existing system, one may choose to focus on modeling only those portions needed to support the new functionality. In an automobile design, for example, if the emphasis is on new requirements for fuel economy and acceleration, the model may focus on elements related to the power train, with less focus on the braking and steering subsystems.

Model depth. The depth of the model must be sufficient for the purpose by determining the level of the system design hierarchy that the model must encompass. For a conceptual design or initial design iteration, the model may only address a fairly high level of the design. In the automobile example, the initial iterations may only model to the engine level, where a future design iteration may model the engine parts if it is subject to further development.

Model fidelity. The fidelity of the model must be sufficient for the purpose by determining the required level of detail for different modeling constructs. For example, a low-fidelity behavioral model may be sufficient to communicate a simple ordering of actions in an activity diagram. Additional detail is required if the behavioral model is intended to be executed to validate the logic. When modeling interfaces, a low-fidelity model may only include the logical interface description, whereas a higher-fidelity model may model the communication protocol. Additional detail is required to model system performance.

Is the model complete relative to its scope?

A necessary condition for the model to be complete is that its breadth, depth, and fidelity must match its defined scope. Other completion criteria may relate to other quality attributes of the model (e.g., whether the naming conventions have been properly applied) and design completion criteria (e.g., whether all design elements are traced to a requirement). The MBSE metrics discussed in Section 2.2.4 can be used to establish additional completion criteria.

Is the model well formed such that model constraints are adhered to?

A well-formed model conforms to its static semantics. For example, the static semantics in SysML do not allow a requirement to contain a system component, although other relationships are allowed between components and requirements such as the satisfy relationship. The modeling tool should enforce the constraints imposed by the static semantics or provide a report of violations.

Is the model consistent?

In SysML, some rules are built into the language to ensure model consistency. For example, compatibility rules can support type checking to determine whether interfaces are compatible or whether units are consistent on different properties. Additional constraints can be imposed by the method used. For example, a method may impose a constraint that logical components can only be allocated to hardware,

software, or operational procedures. These constraints can be expressed in the object constraint language (OCL) [26] and enforced by the modeling tool.

Enforcing constraints assists in maintaining consistency across the model, but it does not prevent inconsistencies. A simple example may be that a modeler inadvertently gives a component two different names that are interpreted by a model checker as different components. The likelihood of inconsistencies increases when multiple people are working on the model. A combination of well-defined model conventions and a disciplined process can limit this from happening.

Is the model understandable?

There are many factors driven by the model-based method and modeling style that can contribute to understandability. A key contributing factor to enhance understandability is the effective use of model abstraction. For example, when describing the functionality of an automobile, one could describe a top-level function as “drive car” or provide a more detailed functional description such as “turn ignition on, put gear into drive, push accelerator pedal,” and so on. An understandable model should include multiple levels of abstraction that represent different levels of detail but relate to one another. As will be described in later chapters, the use of decomposition, specialization, allocations, views, and other modeling approaches in SysML can be used to represent different levels of abstraction.

Another factor that impacts understandability relates to the presentation of information on the diagrams themselves. Often, there is a lot of detail in the model, but only selected information is relevant to communicate a particular design aspect. The information on the diagram can be controlled by using the tool capability to elide (hide) nonessential information and display only the information relevant to the diagram’s purpose. Again, the goal is to avoid information overload for the reviewer of the model.

Other factors that contribute to understandability are the use of modeling conventions and the extent to which the model is self-documenting as described next.

Are modeling conventions documented and used consistently?

Modeling conventions and standards are critical to ensure consistent representation and style across the model. This includes establishing naming conventions for each type of model element, diagram names, and diagram content. Naming conventions may include stylistic aspects of the language, such as when to use uppercase versus lowercase, and when to use spaces in names. The conventions and standards should also account for tool-imposed constraints, such as limitations in the use of alphanumeric and special characters. It is also recommended that a template be established for each diagram type so that consistent style can be applied.

Is the model self-documenting in terms of providing sufficient supporting information?

The use of annotations and descriptions throughout the model can help to provide value-added information if applied consistently. This can include the rationale for design decisions, flagging issues or problem areas for resolution, and providing

additional textual descriptions for model elements. This enables longer-term maintenance of the model and enables it to be more effectively communicated to others.

Does the model integrate with other models?

The system model may need to be integrated with electrical, mechanical, software, test, and engineering analysis models. This capability is determined by the specific method, tool implementation, and modeling languages used. For example, the approach for passing information from the system model using SysML to a software model using UML can be defined for specific methods and tools. In general, this is addressed by establishing an agreed-on expression of the modeling information so that it can be best communicated to the user of the information, such as hardware and software developers, testers, and engineering analysts.

2.2.4 Model-Based Metrics

Measurement data collection, analysis, and reporting can be used as a management technique throughout the development process to assess design quality and progress. This in turn is used to assess status and risk and to support ongoing project planning and control. Model-based metrics can provide useful data that can be derived from the model and can help answer the following questions. This discussion refers to metrics that can be derived from a typical SysML model.

What is the quality of the design?

Metrics can be defined to measure the quality of a model-based system design based on metrics that have been traditionally used in document-centric designs. This includes metrics for assessing requirements satisfaction, critical performance properties, and how well the design is partitioned.

A SysML model can provide explicit relationships that can be used to measure the extent that the requirements are satisfied. The model can provide granularity by identifying model elements that satisfy specific requirements. The requirements traceability can be established from mission-level requirements down to component-level requirements. Other SysML relationships can be used in a similar way to measure which requirements have been verified. These data can be captured directly from the model or indirectly from a requirements management tool that is integrated with the SysML modeling tool.

A SysML model can include critical properties that are monitored throughout the design process. Typical properties may include performance properties, such as latency, physical properties (e.g., weight), and other properties (e.g., reliability and cost). These properties can be monitored using standard technical performance measurement (TPM) techniques. The model can also include relationships among the properties that indicate how they may be impacted as a result of design decisions.

Design partitioning can be measured in terms of the level of cohesion and coupling of the design. Coupling can be measured in terms of the number of interfaces or in terms of more complex measures of dependencies between different model parts. Cohesion metrics are more difficult to define, but measure the extent to

which a component can perform its functions without requiring access to external data. The object-oriented concept of encapsulation reflects this concept.

What is the progress of the design and development effort?

Model-based metrics can be defined to assess design progress by establishing completion criteria for the design. The quality attributes in the previous section referred to whether the model is complete relative to the defined scope of the modeling effort. This is necessary, but not sufficient, to assess design completeness. The requirements satisfaction described to measure design quality can also be used to assess design completeness. Other intermediate metrics may include the number of use case scenarios that have been completed or the percent of logical components that have been allocated to physical components. From a systems engineering perspective, a key measure of system design completeness is the extent to which components have been specified. This metric can be measured in terms of the completeness of the specification of component interfaces, behavior, and properties.

Other metrics for assessing progress include the extent to which components have been verified and integrated into the system, and the extent to which the system has been verified to satisfy its requirements. Test cases and verification status can be captured in the model and used as a basis for this assessment.

What is the estimated effort to complete design and development?

The Constructive Systems Engineering Cost Model (COSYSMO) is used for estimating the cost and effort to perform systems engineering activities. This model includes both sizing and productivity parameters, where the size estimates the magnitude of the effort, and productivity factors are applied to come up with an actual labor estimate to do the work.

When using model-based approaches, sizing parameters can be identified in the model in terms of numbers of different modeling constructs that may include the following:

- #Requirements
- #Use cases
- #Scenarios
- #States
- #System and component interfaces
- #System and component activities or operations
- #System and component properties
- #Components by type (e.g., hardware, software, data, operational procedures)
- #Test cases

The MBSE sizing parameters will need to be integrated into the cost model. Data will need to be collected and validated over time to establish statistically meaningful data. However, early users of MBSE can identify sizing parameters that contribute most significantly to the modeling effort, and use this data for local estimates and to assess productivity improvements over time.

2.2.5 Other Model-Based Metrics

The previous discussion is a sampling of some of the model-based metrics that can be defined. Many other metrics can also be derived from the model, such as the stability of the number of requirements and design changes over time, or potential defect rates. The metrics can also be derived to establish benchmarks from which to measure the MBSE benefits as described in Section 2.1.2, such as the productivity improvements resulting from MBSE over time. Chapter 18 includes a discussion of additional organizational metrics related to deploying MBSE in an organization.

2.3 Summary

The practice of systems engineering is transitioning from a document-based approach to a model-based approach like many of the other engineering disciplines, such as mechanical and electrical engineering, have already done. MBSE offers significant potential benefits to enhance communications, specification and design precision, design integration, and reuse that can improve design quality, productivity, and reduce development risk. The emphasis in MBSE is on producing and controlling a coherent system model, and using this model to specify and design the system. Quality attributes of a model such as model consistency, understandability, and well formedness, and the use of modeling conventions, can be used to assess the goodness of a model and to derive preferred modeling practices. MBSE metrics can be used to assess design quality, progress and risk, and support management of the development effort.

2.4 Questions

1. What are some of the primary distinctions between MBSE and a document-based approach?
2. What are some of the benefits of MBSE over the document-based approach?
3. Where are the model elements of a system model stored?
4. Which aspects of the model can be used to define the scope of the model?
5. What constitutes a good model?
6. What are some of the quality attributes of a good model?
7. What is the difference between a good model and a good design?
8. What are examples of questions that MBSE metrics can help answer?
9. What are possible sizing parameters that could be used to estimate an MBSE effort?

This page intentionally left blank

SysML Language Overview

3

This chapter provides an overview of SysML that includes a simple example showing how the language is applied to the system design of an automobile that was introduced in Chapter 1. The example includes references to chapters in Part II that provide a detailed description of the diagrams and language concepts. Part III includes two more detailed examples of how MBSE methods can be used with SysML to specify and design a system.

3.1 SysML Purpose and Key Features

SysML is a general-purpose graphical modeling language that supports the analysis, specification, design, verification, and validation of complex systems. These systems may include hardware, software, data, personnel, procedures, facilities, and other elements of man-made and natural systems. The language is intended to help specify and architect systems and specify its components that can then be designed using other domain-specific languages such as UML for software design and VHDL for hardware design.

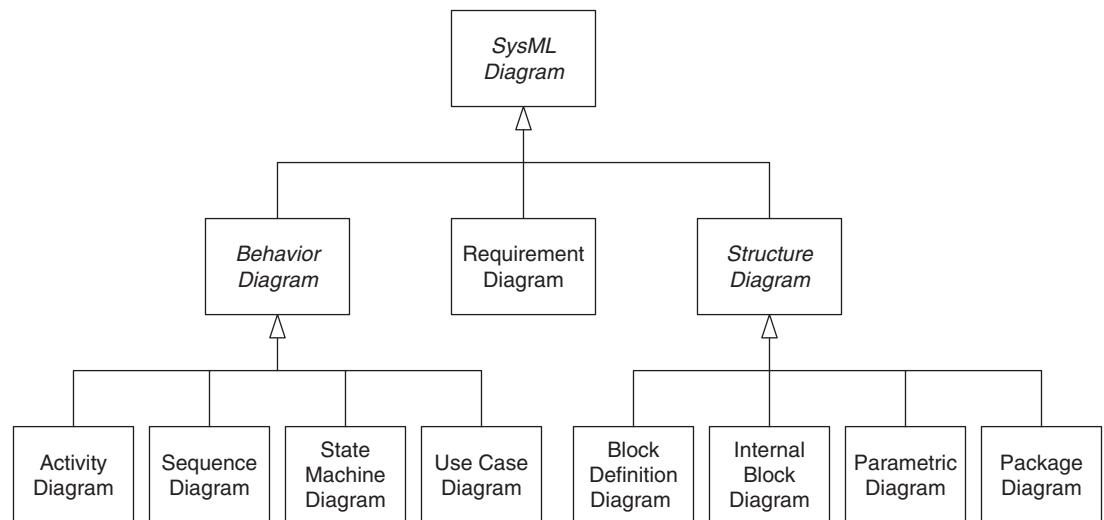
SysML can represent systems, components, and other entities as follows:

- Structural composition, interconnection, and classification
- Function-based, message-based, and state-based behavior
- Constraints on the physical and performance properties
- Allocations between behavior, structure, and constraints (e.g., functions allocated to components)
- Requirements and their relationship to other requirements, design elements, and test cases

3.2 SysML Diagram Overview

SysML includes nine diagrams as shown in the diagram taxonomy in Figure 3.1. Each diagram type is summarized here, along with its relationship to UML diagrams:

- *Requirement diagram* represents text-based requirements and their relationship with other requirements, design elements, and test cases to support requirements traceability (not in UML)

**FIGURE 3.1**

SysML diagram taxonomy.

- *Activity diagram* represents behavior in terms of the ordering of actions based on the availability of inputs, outputs, and control, and how the actions transform the inputs to outputs (modification of UML activity diagram)
- *Sequence diagram* represents behavior in terms of a sequence of messages exchanged between parts (same as UML sequence diagram)
- *State machine diagram* represents behavior of an entity in terms of its transitions between states triggered by events (same as UML state machine diagram)
- *Use case diagram* represents functionality in terms of how a system or other entity is used by external entities (i.e., actors) to accomplish a set of goals (same as UML use case diagram)
- *Block definition diagram* represents structural elements called blocks, and their composition and classification (modification of UML class diagram)
- *Internal block diagram* represents interconnection and interfaces between the parts of a block (modification of UML composite structure diagram)
- *Parametric diagram* represents constraints on property values, such as $F = m*a$, used to support engineering analysis (not in UML)
- *Package diagram* represents the organization of a model in terms of packages that contain model elements (same as UML package diagram)

A diagram graphically represents a particular aspect of the system model as described in Section 2.1.2. The diagram type constrains the type of model elements and associated symbols that can appear on a diagram. For example, an activity

diagram can include diagram elements that represent actions, control flow, and input/output flow, but not diagram elements for connectors and ports. As a result, a diagram represents a subset of the underlying model repository, as described in Chapter 2. Tabular representations are also supported in SysML as a complement to diagram representations to capture model information such as allocation tables.

3.3 Using SysML in Support of MBSE

SysML provides a means to capture the system modeling information as part of an MBSE approach without imposing a specific method on how this is performed. The selected method determines which activities are performed, the ordering of the activities, and which modeling artifacts are created to represent the system. For example, traditional structured analysis methods can be used to decompose the functions and allocate the functions to components. Alternatively, one can apply a use case driven approach that derives functionality based on scenario analysis and associated interactions among parts. The two methods may produce different combinations of diagrams in different ways to represent the system specification and design.

A typical use of the language may include one or more iterations of the following activities to specify and design the system:

- Capture and analyze black box system requirements
 - Capture text-based requirements in a requirements management tool
 - Import requirements into the SysML modeling tool
 - Identify top-level functionality in terms of system use cases
 - Capture the traceability between the use cases and requirements
 - Model the use case scenarios as activity diagrams, sequence diagrams, and/or state machine diagrams
 - Create the system context diagram
 - Identify system test cases to support system verification
- Develop one or more candidate system architectures to satisfy the requirements
 - Decompose the system using the block definition diagram
 - Define the interaction among the parts using activity or sequence diagrams
 - Define the interconnection among the parts using the internal block diagram
- Perform engineering and trade-off analysis to evaluate and select the preferred architecture
 - Capture the constraints on system properties using the parametric diagram to support analysis of performance, reliability, cost, and other critical properties
 - Perform the engineering analysis to determine the budgeted values of the system properties (typically done in separate engineering analysis tools)

- Specify component requirements and their traceability to system requirements
 - Capture the functional, interface, and performance requirements for each component (block) in the architecture
 - Trace component requirements to the system requirements
- Verify that the system design satisfies the requirements by executing system-level test cases

Other systems engineering activities are performed in conjunction with the preceding modeling activities such as configuration management and risk management. Detailed examples of how SysML can be used to support two different MBSE methods are included in the modeling examples in Part III. A simplified example is described next.

3.4 A Simple Example Using SysML for an Automobile Design

The following example was introduced in Chapter 1 without introducing the model-based approach. It is a simplified example that illustrates how SysML can be applied to specify and design a system.

3.4.1 Example Background and Scope

The example includes at least one diagram for each SysML diagram type, but only highlights selected features of the language. It includes multiple references to the chapters in Part II for the detailed language description. The way the diagrams are used to represent the system and the ordering of the diagrams is intended to be representative of applying a typical model-based approach. However, this will vary depending on the specific process and method used.

3.4.2 Problem Summary

The sample problem describes the use of SysML as it applies to the design of an automobile. A marketing analysis that was done indicated the need to increase the automobile's acceleration and fuel efficiency from its current capability. A variant of the process described in Section 3.3 is used to design the system to satisfy the requirements. In this simplified example, selected aspects of the design are considered to support an initial trade-off analysis. The trade-off analysis included evaluation of alternative vehicle configurations that included a 4-cylinder engine and a 6-cylinder engine to determine whether they can satisfy the acceleration and fuel efficiency requirement.

In addition, the proposed vehicle design includes a vehicle controller and associated software to control the fuel-air mixture and maximize fuel efficiency and engine performance. Only a small subset of the design is addressed to

Table 3.1 Diagrams Used in the Automobile Example

Figure	Diagram Kind	Diagram Name
3.2	Requirement diagram	Automobile System Requirements
3.3	Block definition diagram	Automobile Domain
3.4	Use case diagram	Operate Vehicle
3.5	Sequence diagram	Drive Vehicle
3.6	Sequence diagram	Start Vehicle
3.7	Activity diagram	Control Power
3.8	State machine diagram	Drive Vehicle States
3.9	Internal block diagram	Vehicle Context
3.10	Block definition diagram	Vehicle Hierarchy
3.11	Activity diagram	Provide Power
3.12	Internal block diagram	Power Subsystem
3.13	Block definition diagram	Analysis Context
3.14	Parametric diagram	Vehicle Acceleration Analysis
3.15	Timing diagram (not SysML)	Vehicle Performance Timeline
3.16	Activity diagram	Control Engine Performance
3.17	Block definition diagram	Engine Specification
3.18	Requirement diagram	Max Acceleration Requirement Traceability
3.19	Package diagram	Model Organization

highlight the use of the language. The diagrams used in this example are shown in Table 3.1.

SysML diagrams include a **diagram frame**. The **diagram header** in the diagram frame describes the kind of diagram, the diagram name, and some additional information that provides context for the **diagram content**. Detailed information on diagram frames, diagram headers, and other common diagram elements that apply to all SysML diagrams is described in Chapter 4.

The example includes the following user-defined notations, called a stereotype, that are added for this example. Chapter 14 describes how stereotypes are used to further customize the language for domain-specific applications.

```
«hardware»
«software»
«store»
«system of interest»
```

3.4.3 Capturing the *Automobile Specification* in a Requirement Diagram

The **requirement diagram** for the *Automobile System Requirements* is shown in Figure 3.2. The kind of diagram (e.g., req) and the diagram name are shown in the diagram header at the upper left. The diagram depicts the requirements that are typically captured in a text specification. The requirements are shown in a containment hierarchy to depict the hierarchical relationship among them. The *Automobile Specification* is the top-level requirement that contains the lower-level requirements. The line with the crosshairs symbol at the top is the **containment** relationship.

The specification contains requirements for *Passenger and Baggage Load*, *Vehicle Performance*, *Riding Comfort*, *Emissions*, *Fuel Efficiency*, *Production Cost*, *Vehicle Reliability*, and *Occupant Safety*. The *Vehicle Performance* requirement contains requirements for *Maximum Acceleration*, *Top Speed*, *Braking Distance*, and *Turning Radius*. Each requirement includes a unique identification, its text, and can include other user-defined properties, such as verification status and risk, that are typically associated with requirements. The text for the *Maximum Acceleration* requirement is “the vehicle shall accelerate from 0 to 60 mph in less than 8 seconds” and the text for the *Fuel Efficiency* requirement is “the vehicle shall achieve at least 25 miles per gallon under the stated driving conditions.”

The requirements may have been created in the modeling tool, or alternatively, they may have been imported from a requirements management tool or a text document. The requirements can be related to other requirements, design elements, and test cases using **derive**, **satisfy**, **verify**, **refine**, **trace**, and **copy** relationships. These relationships can be used to establish requirements traceability with a high degree of granularity. Some of these relationships are highlighted in Section 3.4.19. Chapter 12 provides a detailed description of how requirements are modeled in SysML. Requirements can be represented using multiple display options to view the requirements, their properties, and their relationships, which includes a tabular representation.

3.4.4 Defining the *Vehicle* and Its External Environment Using a Block Definition Diagram

In system design, it is important to identify what is external to the system that may either directly or indirectly interact with it. The **block definition diagram** for the *Automobile Domain* in Figure 3.3 defines the *Vehicle* and the external systems, users, and other entities that the vehicle may directly or indirectly interact with.

A **block** is a very general modeling concept in SysML that is used to model a wide variety of entities that have structure such as systems, hardware, software, physical objects, and abstract entities. That is, a block can represent any real or abstract entity that can be conceptualized as a structural unit with one or more distinguishing features. The block definition diagram captures the relation between blocks such as a block hierarchy.

The *Automobile Domain* is the top-level block in the block definition diagram in Figure 3.3. It is **composed** of other blocks as indicated by the black diamond

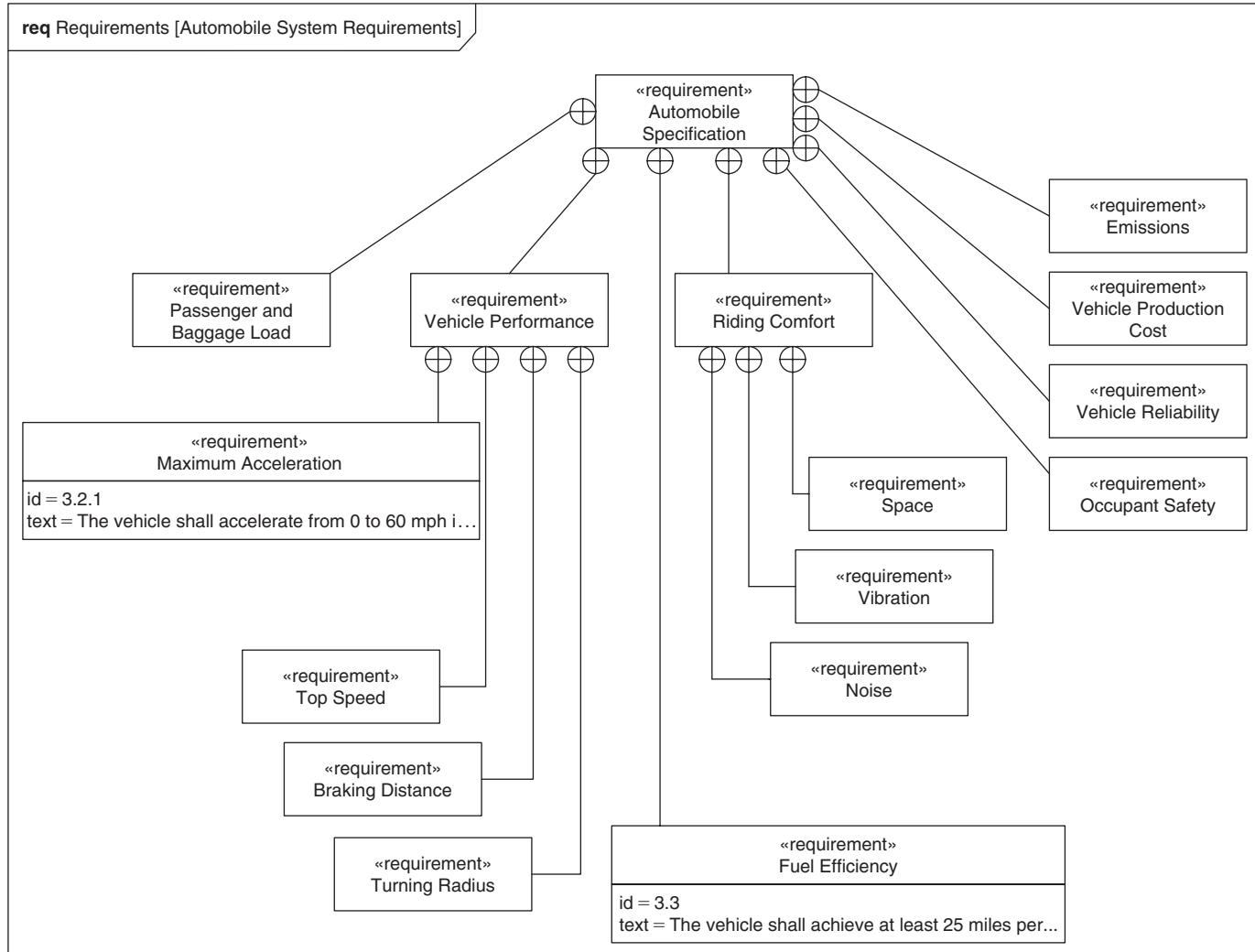
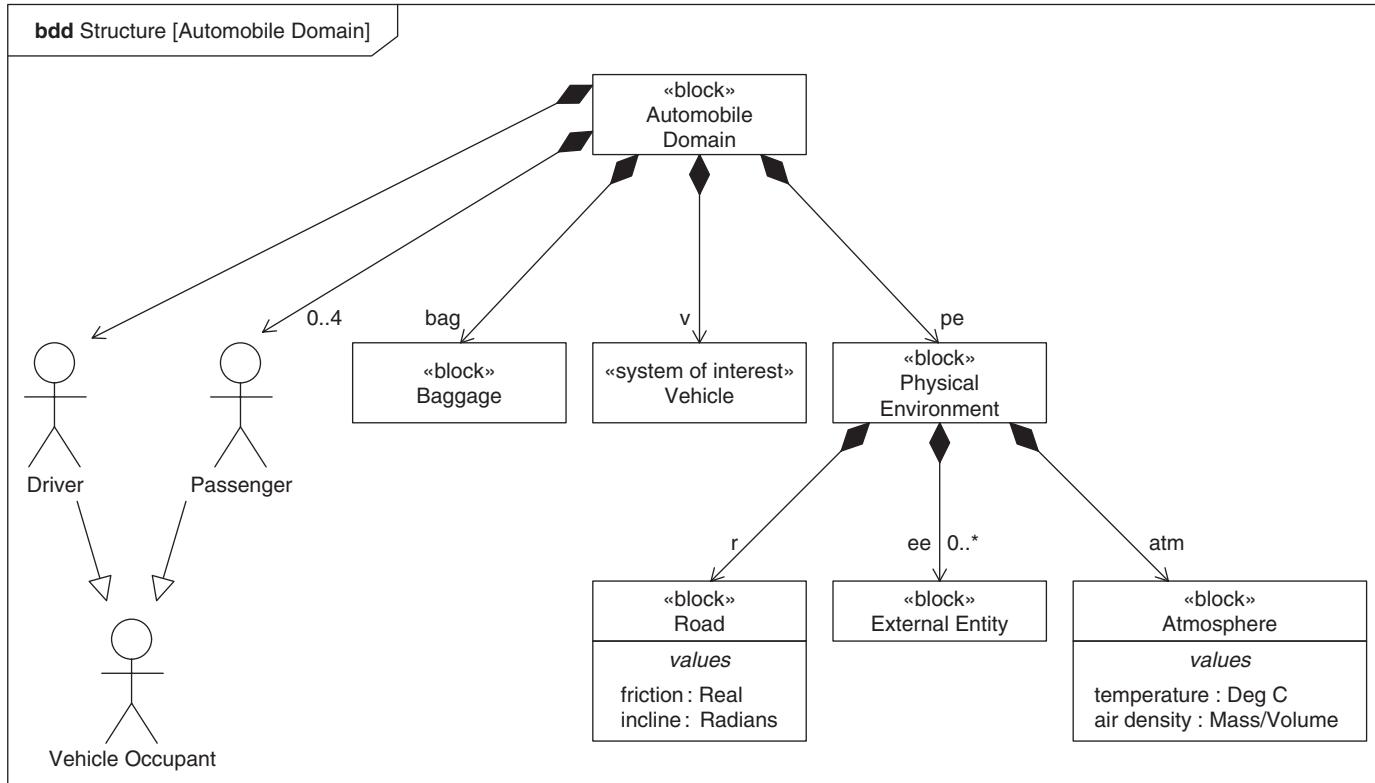


FIGURE 3.2

Requirement diagram showing the system requirements contained in the *Automobile Specification*.

**FIGURE 3.3**

Block definition diagram of the *Automobile Domain* showing the *Vehicle* and its external users and physical environment.

symbol and line with the arrowhead pointing to the blocks that compose it. The differences between the composition hierarchy (i.e., black diamond) and the containment hierarchy (i.e., crosshairs symbol) shown in Figure 3.2 are explained in Part II. The name next to the arrow identifies a particular usage of a block as described later in this section. The *Vehicle* block is referred to as the «system of interest» using the bracket symbol called **guillemet**. The other blocks are external to the vehicle. These include the *Driver*, *Passenger*, *Baggage*, and *Physical Environment*. The *Driver* and *Passenger* are shown using stick-figure symbols. Notice that even though the *Driver*, *Passenger*, and *Baggage* are assumed to be physically inside the *Vehicle*, they are not part of the *Vehicle* structure, and therefore are external to it.

The *Driver* and *Passenger* are **subclasses** of *Vehicle Occupant* as indicated by the hollow triangle symbol. This means that they are kinds of vehicle occupants that inherit common features from *Vehicle Occupant*. In this way, a classification can be created by specializing blocks from more generalized blocks.

The *Physical Environment* is composed of the *Road*, *Atmosphere*, and multiple *External Entities*. The *External Entity* can represent any physical object, such as a traffic light or another vehicle, that the *Driver* interacts with. The interaction between the *Driver* and an *External Entity* can impact how the *Driver* interacts with the *Vehicle*, such as when the traffic light changes color. The **multiplicity** symbol $0..*$ represents an undetermined maximum number of external entities. The multiplicity symbol can also represent a single number or a range, such as the multiplicity of $0..4$, for the number of *Passengers*.

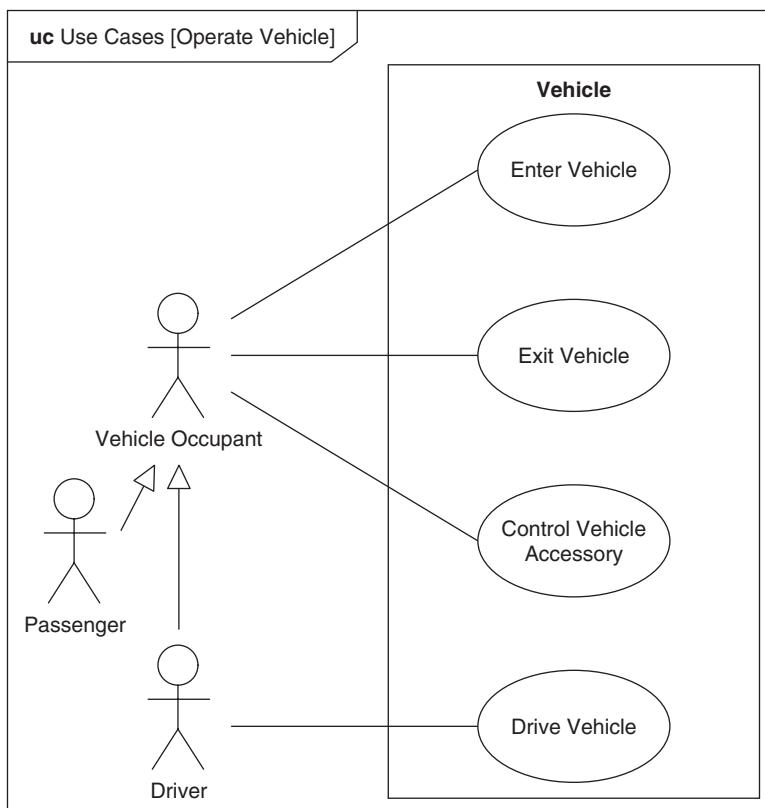
Each block defines a structural unit, such as a system, hardware, software, data element, or other conceptual entity, as described earlier. A block can have a set of **features** that further define it. The features of the block define its **properties** (e.g., weight), its **behavior** in terms of activities **allocated** to the block or **operations** of the block, and its interfaces as defined by its **ports**. Together, these features enable a modeler to specify each block at the level of detail that is appropriate for the application.

The *Road* is a block that has a property called *incline* with units of *Radians* and a property called *friction* that is defined as a real number. Similarly, *Atmosphere* is a block that has two properties for *temperature* and *air density*. These properties are used along with other properties to support analysis of vehicle acceleration and fuel efficiency, which are discussed in Sections 3.4.12 and 3.4.13.

The block definition diagram specifies the blocks and their interrelationships. It is often used in systems modeling to depict multiple levels of the system hierarchy from the top-level domain block (e.g., *Automobile Domain*) down to vehicle components. Chapter 6 provides a detailed description of how blocks are modeled in SysML, including their features and relationships.

3.4.5 Use Case Diagram for *Operate Vehicle*

The **use case diagram** for *Operate Vehicle* in Figure 3.4 depicts the major functionality for operating the vehicle. The **use cases** include *Enter Vehicle*, *Exit Vehicle*, *Control Vehicle Accessory*, and *Drive Vehicle*. The *Vehicle* is the **subject**

**FIGURE 3.4**

Use case diagram describes the major functionality in terms of how the *Vehicle* is used by the actors to *Operate Vehicle*. The actors are defined on the block definition diagram in Figure 3.3.

of the use cases and is represented by the rectangle. The *Vehicle Occupant* is an **actor** who is external to the vehicle and is represented as the stick figure. The subject and actors correspond to the blocks in Figure 3.3. In a use case diagram, the subject (e.g., *Vehicle*) is used by the actors (e.g., *Vehicle Occupant*) to achieve the goals defined by the use cases (e.g., *Drive Vehicle*).

The *Passenger* and *Driver* are both kinds of vehicle occupants as described in the previous section. All vehicle occupants participate in entering and exiting the vehicle and controlling vehicle accessories, but only the driver participates in *Drive Vehicle*. There are several other relationships between use cases that are not shown here. Chapter 11 provides a detailed description of how use cases are modeled in SysML.

Use cases define how the system is used to achieve a user goal. Other use cases can represent how the system is used across its life cycle, such as when

manufacturing, operating, and maintaining the vehicle. The primary emphasis for this example is on the *Drive Vehicle* use case to address the acceleration and fuel efficiency requirements.

The requirements are often related to use cases since use cases represent the high-level functionality or goals for the system. Sometimes, use case textual descriptions are defined to accompany the use case definition. One approach to relate requirements to use cases is to capture the use case descriptions as SysML requirements and relate them to the use case using a refine relationship.

The use cases describe the high-level goals of the system as described previously. The goals are accomplished by the interactions between the actors (e.g., *Driver*) and the subject (e.g., *Vehicle*). These interactions are realized through more detailed descriptions of behavior as described in the next section.

3.4.6 Representing *Drive Vehicle* Behavior with a Sequence Diagram

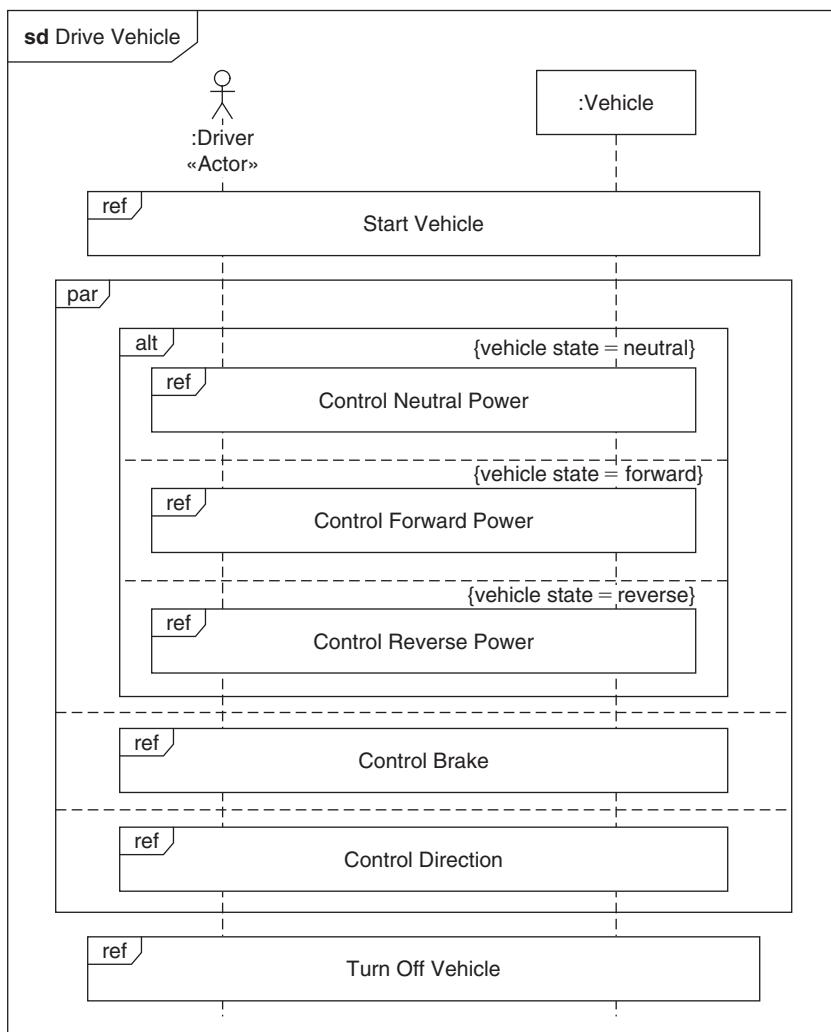
The behavior for the *Drive Vehicle* use case in Figure 3.4 is represented by the **sequence diagram** in Figure 3.5. The sequence diagram specifies the **interaction** between the *Driver* and the *Vehicle* as indicated by the names at the top of the **lifelines**. Time proceeds vertically down the diagram. The first interaction is *Start Vehicle*. This is followed by the *Driver* and *Vehicle* interactions to *Control Power*, *Control Brake*, and *Control Direction*. These three interactions occur in parallel as indicated by **par**. The **alt** on the *Control Power* interaction stands for alternative, and indicates that the *Control Neutral Power*, *Control Forward Power*, or *Control Reverse Power* interaction occurs as a condition of the *vehicle state* shown in brackets. The state machine diagram in Section 3.4.9 specifies the *vehicle state*. The *Turn-off Vehicle* interaction occurs following these interactions.

The **interaction occurrences** in the figure each reference a more detailed interaction as indicated by **ref**. The referenced interaction for *Start Vehicle* is another sequence diagram that is illustrated in Section 3.4.7. The remaining interaction occurrences are references allocated to activity diagrams as described in Section 3.4.8.

3.4.7 Referenced Sequence Diagram to *Start Vehicle*

The *Start Vehicle* sequence diagram in Figure 3.6 is an interaction that is referenced in the sequence diagram in Figure 3.5. As stated previously, time proceeds vertically down the diagram. In this example, the more detailed interaction shows the driver sending a **message** requesting the vehicle to start. The vehicle responds with the **vehicle on reply message** shown as a dashed line. Once the reply has been received, the driver and vehicle can proceed to the next interaction.

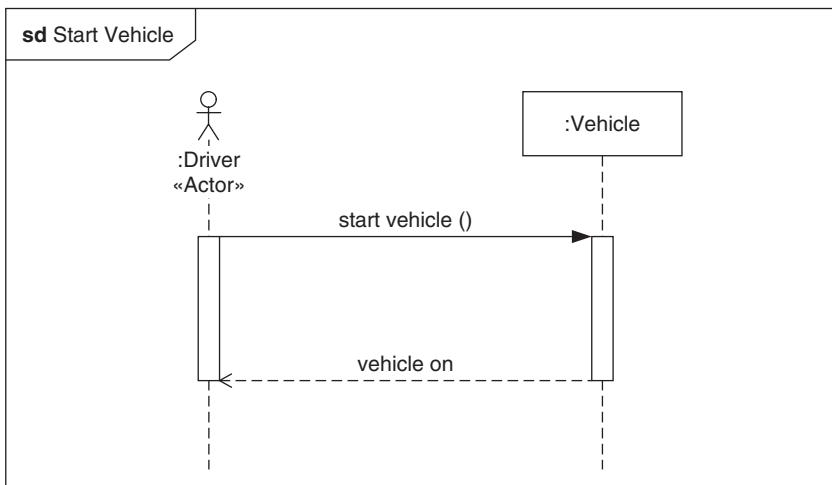
The sequence diagram can include multiple types of messages. In this example, the message is **synchronous** as indicated by the filled arrowhead on the message. The messages can also be **asynchronous** represented by an open arrowhead, where the sender does not wait for a reply. The synchronous messages represent

**FIGURE 3.5**

Drive Vehicle sequence diagram describes the interactions between the *Driver* and the *Vehicle* to realize the *Drive Vehicle* use case in Figure 3.4.

an operation call that specifies a request for service. The arguments of the operation call represent the input data and return.

Sequence diagrams can include multiple message exchanges between multiple lifelines that represent interacting entities. The sequence diagram also provides considerable additional capability to express behavior that includes other message types, timing constraints, additional control logic, and the ability to decompose the behavior of a lifeline into the interaction of its parts. Chapter 9 provides a detailed description of how interactions are modeled with sequence diagrams.

**FIGURE 3.6**

Sequence diagram for the *Start Vehicle* interaction that was referenced in the *Drive Vehicle* sequence diagram, showing the message from *Driver* requesting *Vehicle* to start and the *vehicle on* reply from the *Vehicle*.

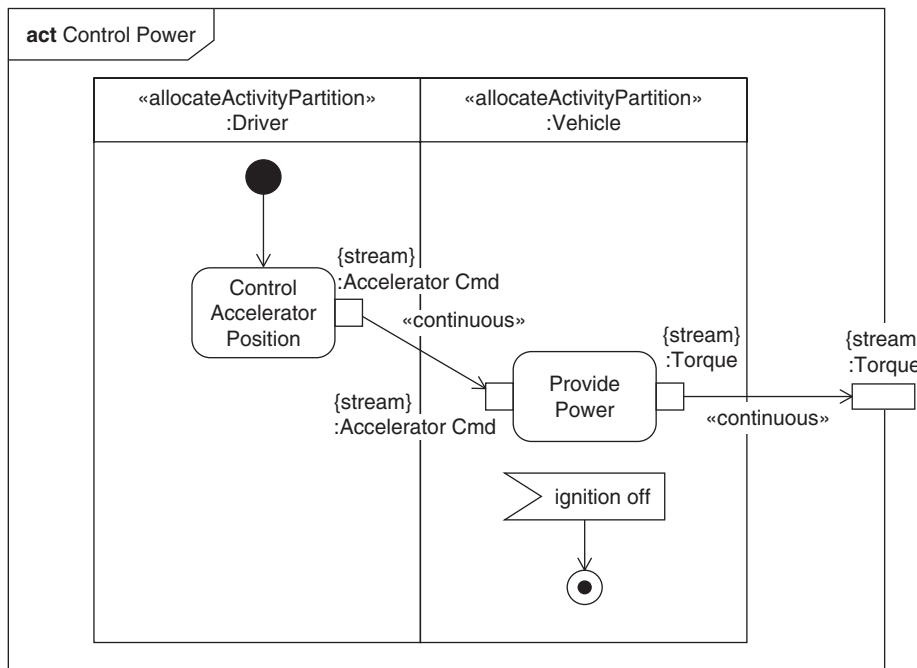
3.4.8 Control Power Activity Diagram

The sequence diagram is effective for communicating discrete types of behavior as indicated with the *Start Vehicle* sequence diagram in Figure 3.6. However, continuous types of behaviors associated with the interactions to *Control Power*, *Control Brake*, and *Control Direction* can sometimes be more effectively represented with activity diagrams.

The *Drive Vehicle* sequence diagram in Figure 3.5 includes the control power interactions that we would like to represent with an activity diagram instead of a sequence diagram. To accomplish this, the *Control Neutral Power*, *Control Forward Power*, and *Control Reverse Power* interactions in Figure 3.5 are **allocated** to a corresponding *Control Power* activity diagram using the SysML allocation relationship (not shown).

The **activity diagram** in Figure 3.7 shows the **actions** required of the *Driver* and the *Vehicle* to *Control Power*. The **activity partitions** (or **swimlanes**) represent the *Driver* and the *Vehicle*. The actions in the activity partitions specify functional requirements that the *Driver* and *Vehicle* must perform.

When the activity is initiated, it starts execution at the **Initial Node** and transitions to the *Control Accelerator Position* action that is performed by the *Driver*. The output of this action is the *Accelerator Cmd*, which is a continuous input to the *Provide Power* action that the *Vehicle* must perform. The output of the *Provide Power* action is the *Torque* generated by the wheels to the road to produce the force that accelerates the *Vehicle*. When the *ignition off* signal is received by the *Vehicle*, the activity terminates at the **Activity Final**. Based

**FIGURE 3.7**

Activity diagram allocated from the *Control Power* interaction that was referenced in the *Drive Vehicle* sequence diagram in Figure 3.5. It shows the continuous *Accelerator Cmd* input from the *Driver* to the *provide power* action that the *Vehicle* must perform.

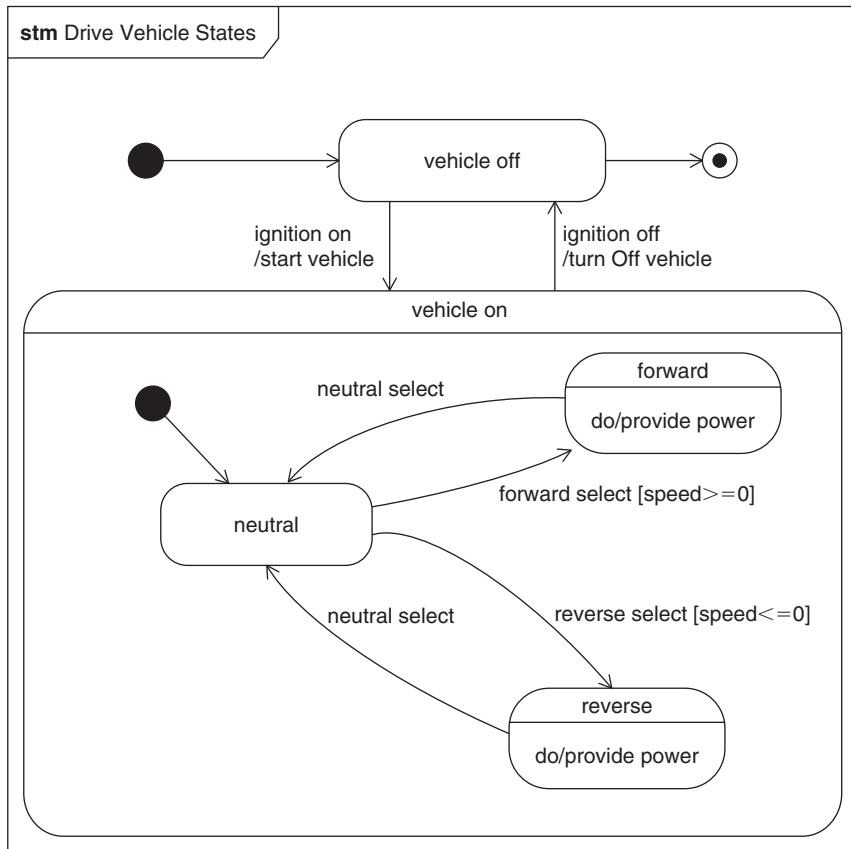
on this scenario, the *Driver* is required to *Control Accelerator Position* and the *Vehicle* is required to *Provide Power*.

Activity diagrams include semantics for precisely specifying the behavior in terms of the flow of control and inputs and outputs. Chapter 8 provides a detailed description of how activities are modeled.

3.4.9 State Machine Diagram for *Drive Vehicle States*

The **state machine diagram** for the *Drive Vehicle States* is shown in Figure 3.8. This diagram shows the states of the *Vehicle* and the **events** that can **trigger** a **transition** between the **states**.

When the *Vehicle* is ready to be driven, it starts in the *vehicle off* state. The *ignition on* event triggers a transition to the *vehicle on* state. The text on the transition indicates that the *Start Vehicle* behavior is executed prior to entering the *vehicle on* state. After entry to the *vehicle on* state, the *Vehicle* immediately transitions to the *neutral* state. A *forward select* event triggers a transition to the *forward* state if the **guard condition** $\{speed \geq 0\}$ is true. While in the *forward* state, the *Vehicle* performs the *Provide Power* behavior that was referred to in the activity diagram in Figure 3.7. The *neutral select* event triggers the transition from

**FIGURE 3.8**

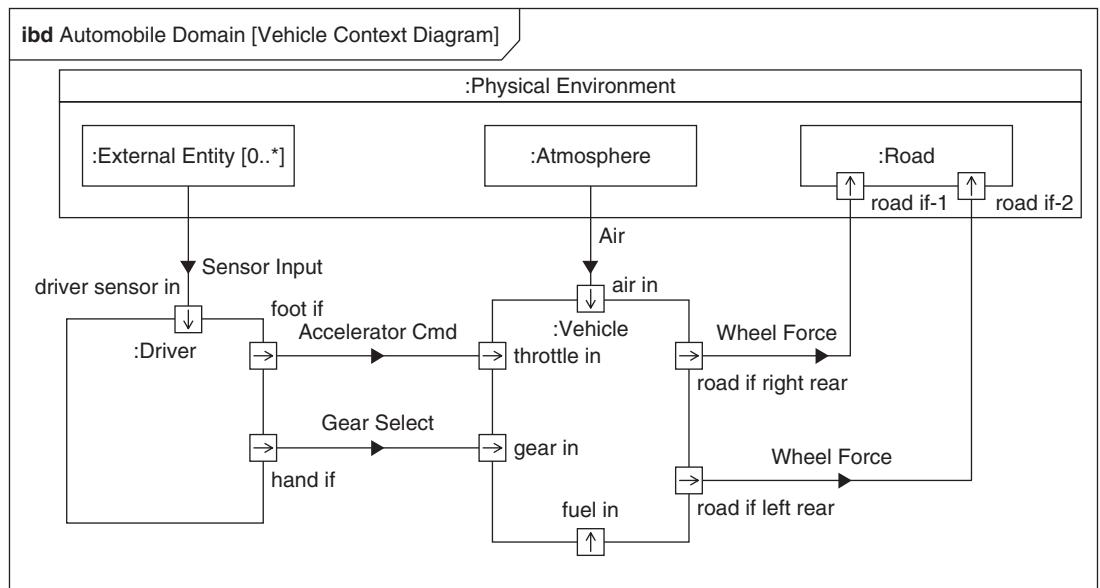
State machine diagram that shows the *Drive Vehicle States* and the transitions between them.

the *forward* state back to the *neutral* state. The state machine diagram shows the additional transitions between the *neutral* and *reverse* states. An *ignition off* event triggers the transition back to the *vehicle off* state. The *Vehicle* can reenter the *vehicle on* state when an *ignition on* occurs.

A state machine can specify the life-cycle behavior of a block in terms of its states and transitions, and are often used with sequence and activity diagrams, as shown in this example. State machines have many other features including **orthogonal regions** and additional transition semantics that are described in Chapter 10.

3.4.10 **Vehicle Context Using an Internal Block Diagram**

The *Vehicle Context Diagram* is shown in Figure 3.9. The diagram shows the interfaces between the *Vehicle*, the *Driver*, and the *Physical Environment* (i.e., *Road*, *Atmosphere*, and *External Entity*) that were defined in the block definition diagram in Figure 3.3. The *Vehicle* has interfaces with the *Driver*, the *Atmosphere*,

**FIGURE 3.9**

Internal block diagram for the *Vehicle Context* shows the *Vehicle* and its external interfaces with the *Driver* and *Physical Environment* that were defined in Figure 3.3.

and the *Road*. The *Driver* has interfaces with the *External Entities* such as a traffic light or another vehicle, via the driver sensor inputs (e.g., seeing, hearing). However, the *Vehicle* does not directly interface with the *External Entities*. The multiplicity on the *External Entity* is consistent with the multiplicity shown in the block definition diagram in Figure 3.3.

This context diagram is an **internal block diagram** that shows how the **parts** of the *Automobile Domain* block from Figure 3.3 are connected. It is called an internal block diagram because it represents the internal structure of a higher-level block, which in this case is the *Automobile Domain* block. The *Vehicle ports* specify interaction points with other parts and are represented as the small squares on the boundary of the parts. **Connectors** define how the parts connect to one another via their ports and are represented as the lines between the ports. Parts can also be connected without ports as indicated by some of the interfaces in the figure when the details of the interface are not of interest to the modeler.

In Figure 3.9, only the external interfaces needed for the *Vehicle* to provide power are shown. For example, the interfaces between the rear tires and the road are shown. It is assumed to be a rear wheel-drive vehicle where power can be distributed differently to the rear wheels depending on tire-to-road traction and other factors. The interface between the front tires and the road is not shown in this diagram, but it would be shown when representing the external interfaces for the steering subsystem where the front tires would play a significant role. It is common modeling practice to only represent the aspects of interest on a particular diagram, even though additional information is included in the model.

The black-filled arrowheads on the connector are called **item flows** that represent the items flowing between parts and may include mass, energy, and/or information. In this example, the *Accelerator Cmd* that was previously defined in the activity diagram in Figure 3.8 flows from the *Driver* port to the *throttle in* port of the *Vehicle*, and the *Gear Select* flows from another *Driver* port to the *gear in* port on the *Vehicle*. The inputs and outputs from the activity diagram are **allocated** to the item flows on the connectors. Allocations are discussed as a general-purpose relationship for mapping one model element to another in Chapter 13.

In SysML, there are two different kinds of ports. The **flow port** specifies the kind of item that can flow in or out of an interaction point, and a **standard port** specifies the services that either are required or provided by the part. The port provides the mechanism to integrate the behavior of the system with its structure.

The internal block diagram enables the modeler to specify both external and internal interfaces of a system or component. An internal block diagram shows how parts are connected, as distinct from a block definition diagram that does not show connectors. Details of how to model internal block diagrams are described in Chapter 6.

3.4.11 *Vehicle Hierarchy Represented on a Block Definition Diagram*

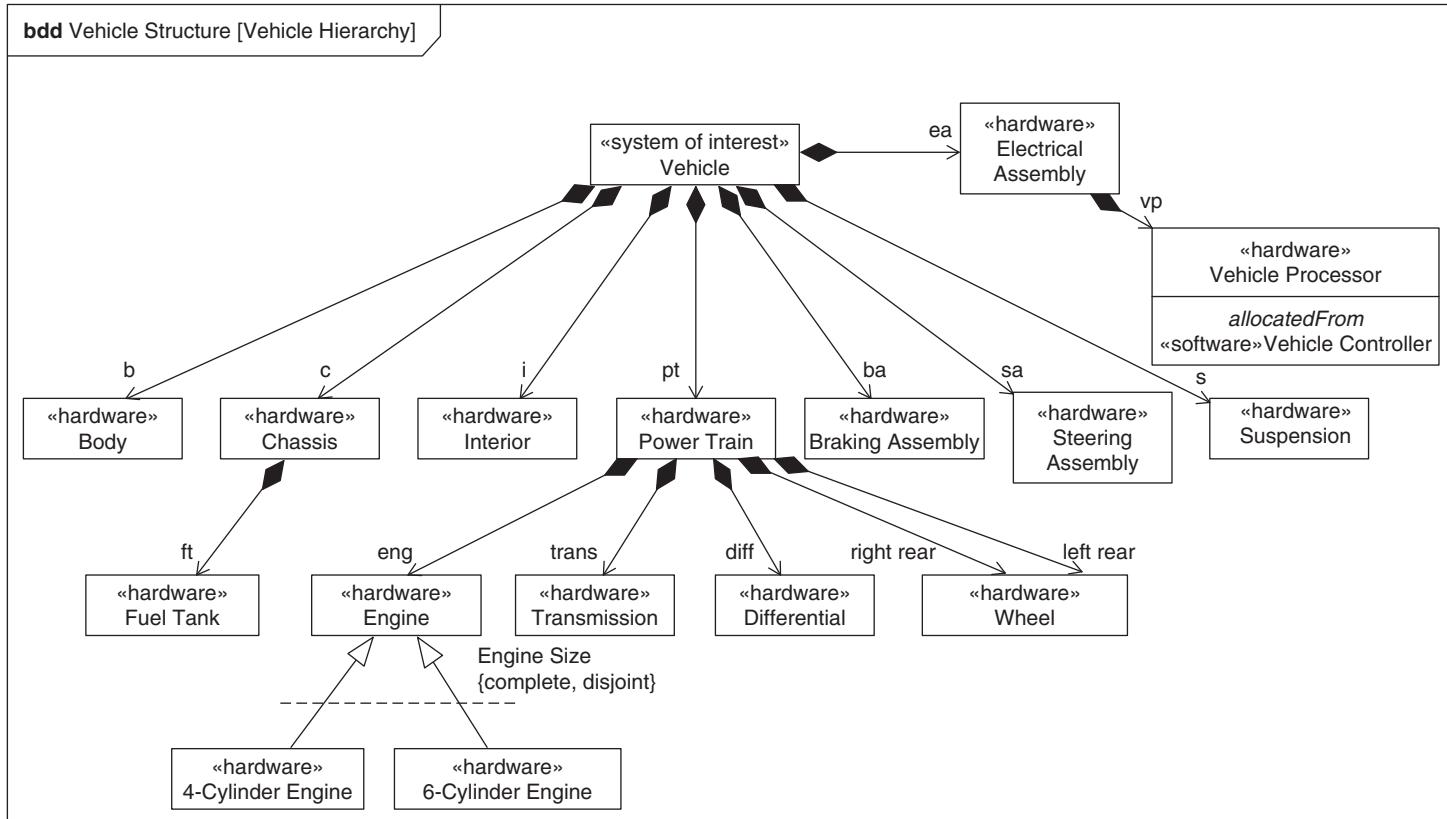
The example to this point has focused on specifying the vehicle in terms of its external interactions and interfaces. The *Vehicle Hierarchy* in Figure 3.10 is a block definition diagram that shows the decomposition of the *Vehicle* into its components. The *Vehicle* is composed of the *Body*, *Chassis*, *Interior*, *Power Train*, and other types of components. Each component type is designated as «hardware».

The *Power Train* is further decomposed into the *Engine*, *Transmission*, *Differential*, and *Wheel*. Note that the *right rear* and *left rear* indicate different usages of a *Wheel* in the context of the *Power Train*. Thus, each rear wheel has a different role and may be subject to different forces, such as is the case when one wheel loses traction. The front wheels are not shown, but could be part of the chassis or part of the steering assembly and would have different roles as well.

The engine may be either 4 or 6 cylinders as indicated by the specialization relationship. The 4- and 6-cylinder vehicle configuration alternatives are being considered to satisfy the acceleration and fuel efficiency requirements. Only one engine type is part of a particular *Vehicle* as indicated by the *{complete, disjoint}* constraint. This implies that the 4- and 6-cylinder engines represent a complete set of subclasses and are mutually exclusive or disjoint.

The *vehicle:Controller* «software» has been allocated to the *Vehicle Processor* as indicated by the allocation compartment. The *Vehicle Processor* is the execution platform for the vehicle control software. The software is being enhanced to control many of the automobile engine and transmission functions to optimize engine performance and fuel efficiency.

The interaction and interconnection between these components is analyzed in a similar way to what was done at the *Vehicle* black box level, and is used to specify the components of the *Vehicle* system as described in the next sections.

**FIGURE 3.10**

Block definition diagram of the *Vehicle Hierarchy* shows the *Vehicle* and its components. The *Power Train* is further decomposed into its components and the *Vehicle Processor* includes the *Controller* software.

3.4.12 Activity Diagram for *Provide Power*

The activity diagram in Figure 3.7 showed that the vehicle must *provide power* in response to the driver accelerator command and generate wheel–tire torque at the road surface. The *Provide Power* activity diagram in Figure 3.11 shows how the vehicle components generate this torque.

The external inputs to the activity include the *Accelerator Cmd* and *Gear Select* from the *Driver*, and *Air* from the *Atmosphere* to support engine combustion. The outputs are the torque from the right and left rear wheels to the road that provides the force to accelerate the *Vehicle*. Some of the other inputs and outputs, such as exhaust from the engine, are not included for simplicity. The activity partitions represent the vehicle components shown in the block definition diagram in Figure 3.10.

The fuel tank stores and dispenses the fuel to the *Engine*. The accelerator command and air and fuel are input to the *generate torque* action. The engine torque is input to the *amplify torque* action performed by the *Transmission*. The amplified torque is input to the *distribute torque* action performed by the *Differential* that distributes torque to the right and left rear wheels to *provide traction* to the road surface to generate the force to accelerate the *Vehicle*.

The actions that are allocated to the *Vehicle* components realize the *provide power* action that the *Vehicle* performs, as shown in Figure 3.7. This approach is used to decompose the system behavior.

A few other items are worth noting in this example. The flows are shown to be continuous for all but the *Gear Select*. The inputs and outputs continuously flow in and out of the actions. *Continuous* means that the delta time between arrival of the inputs or outputs approaches zero. Continuous flows build on the concept of streaming inputs and outputs, which means that inputs are accepted and outputs are produced while the action is executing. Conversely, nonstreaming inputs are only available prior to the start of the action execution, and nonstreaming outputs are produced only at the completion of the action execution. The ability to represent streaming and continuous flows adds a significant capability to classic behavioral modeling associated with functional flow diagrams. The continuous flows are assumed to be streaming.

Many other activity diagram features are explained in Chapter 8; they provide a capability to precisely specify behavior in terms of the flow of control and data, and the ability to reuse and decompose behavior.

3.4.13 Internal Block Diagram for the *Power Subsystem*

The previous activity diagram described how the parts of the system interact to *provide power*. The parts of the system are represented by the activity partitions in the activity diagram. The internal block diagram for the vehicle in Figure 3.12 shows how the parts are interconnected to achieve this functionality and is used to specify the interfaces between the parts. This is a structural view of the system versus the behavioral view that was expressed in the activity diagram.

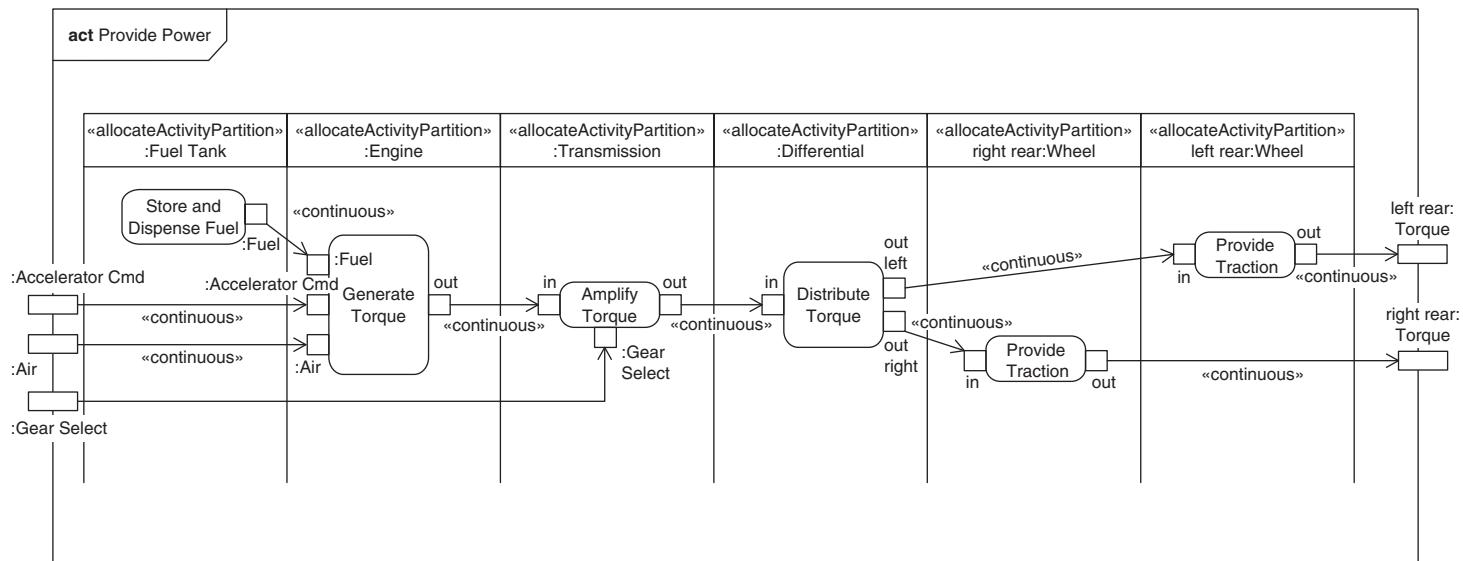


FIGURE 3.11

Activity diagram for *Provide Power* shows how the *Vehicle* components generate the torque to move the vehicle. This activity diagram realizes the *provide power* action in Figure 3.7 with activity partitions that correspond to the components in Figure 3.10.

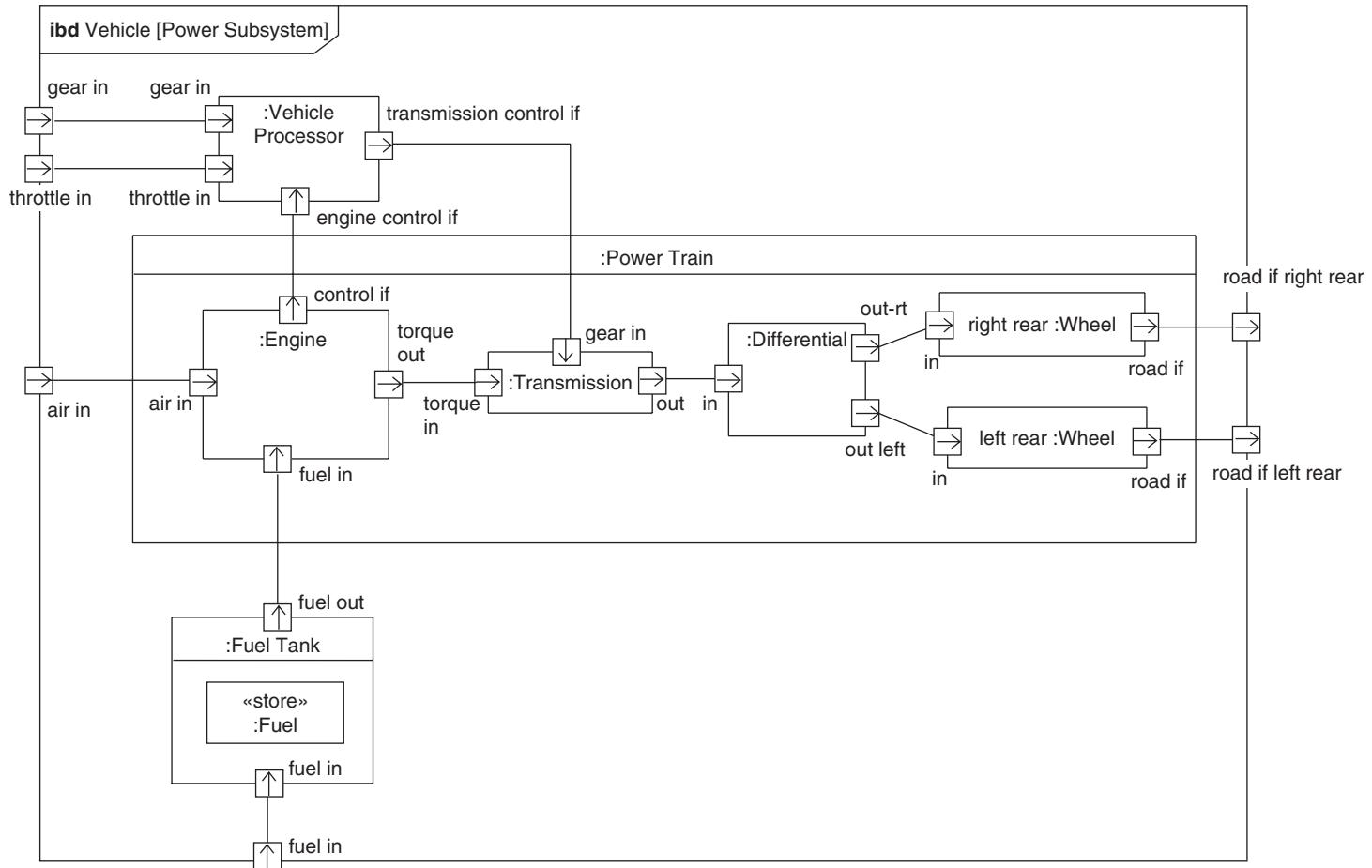


FIGURE 3.12

Internal block diagram for the *Power Subsystem* shows how the parts that *provide power* are interconnected. The parts are represented as the activity partitions in Figure 3.11.

The internal block diagram represents the *Power Subsystem* that only includes the parts of the *Vehicle* that collaborate to *provide power*. The frame of the diagram represents the *Vehicle* black box. The ports on the diagram frame correspond to the same ports shown on the *Vehicle* in the *Vehicle Context* diagram in Figure 3.9. This enables the external interfaces to be preserved as the internal structure of the *Vehicle* is further specified.

The *Engine*, *Transmission*, *Differential*, *right rear* and *left rear Wheel*, *Vehicle Processor*, and *Fuel Tank* are interconnected via their ports. The *Fuel* is stored in the *Fuel Tank* as indicated by «store». The item flows on the connectors are not shown, but represent the items that flow through the system and are allocated from the inputs and outputs on the *Provide Power* activity diagram in Figure 3.11.

Additional subsystems can be created in a similar way to realize specific functionality such as provide braking and provide steering. A composite view of all of the interconnected parts across all subsystems can also be created in a composite *Vehicle* internal block diagram. An example of this is included in the residential security example in Chapter 16.

It is appropriate to elaborate on the usage concept that was first introduced in Section 3.4.11 when discussing the *right rear* and *left rear* wheels. A part in an internal block diagram represents a particular usage of a block. The block represents the generic definition, whereas the part represents a usage of a block definition in a particular context. Thus, the right rear and left rear are different usages of the *Wheel* block in the context of the *Vehicle*. A usage (or part) is the same as a role. The parts in the diagram are indicated by the colon (:) notation. A part enables the same block to be reused in many different contexts and be uniquely identified by its usage. Each part may have unique behaviors, properties, and constraints that apply to its particular usage.

The concept of definition and usage is applied to many other SysML language constructs as well. One example is that the item flows themselves can have a definition and usage. For example, an item flow entering the fuel tank can be in : Fuel and the item flow exiting the fuel tank can be out : Fuel. Both flows are defined by fuel, but “in” and “out” represent different usages of Fuel in the *Vehicle* context.

As mentioned previously, Chapter 6 provides the detailed language description for both block definition diagrams and internal block diagrams, and includes the concept of role, references, and many other key concepts for modeling blocks and parts.

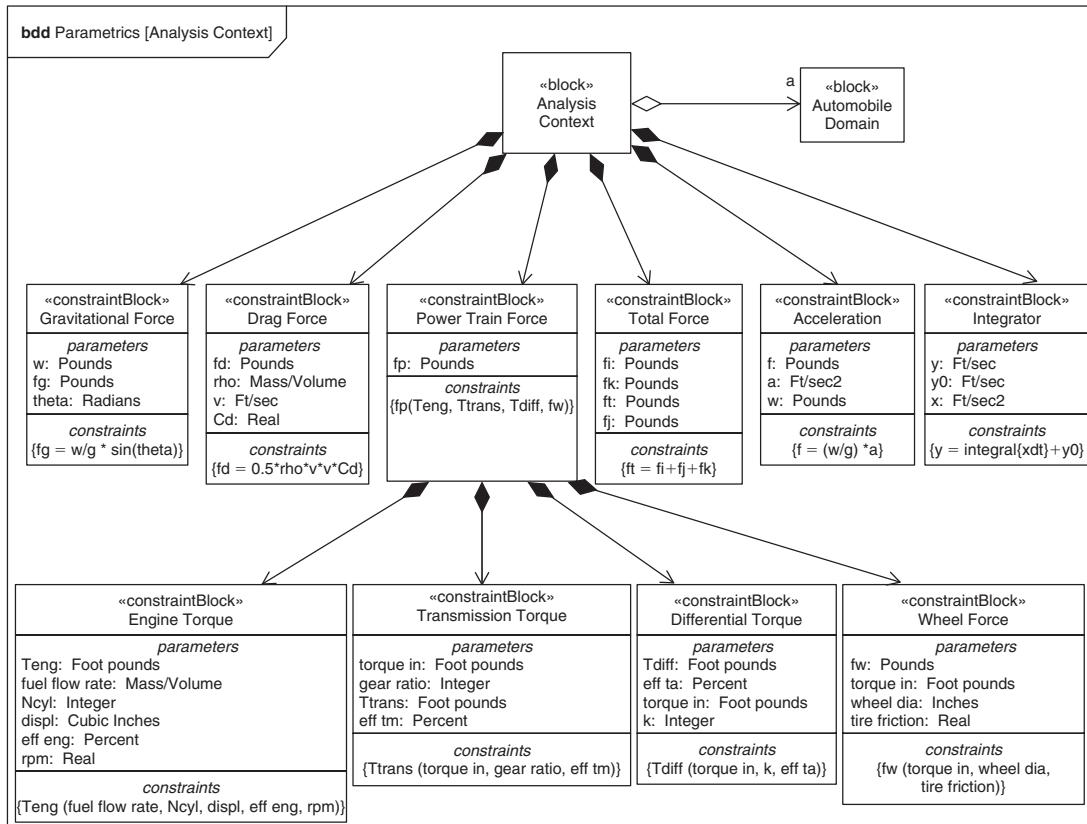
3.4.14 Defining the Equations to Analyze Vehicle Performance

Critical requirements for the design of this automobile are to accelerate from 0 to 60 mph in less than 8 seconds, while achieving a fuel efficiency of greater than 25 miles per gallon. These two requirements impose conflicting requirements on the design space, such that increasing the acceleration capability can result in a design with lower fuel efficiency. Two alternative configurations, including a 4- and 6-cylinder engine, are evaluated to determine which configuration is the preferred solution to meet the acceleration and fuel efficiency requirements.

The 4- and 6-cylinder engine alternatives are shown in the *Vehicle Hierarchy* in Figure 3.10. There are other design impacts that may result from the automobile configurations with different engines, such as the vehicle weight, body shape, and electrical power. This simplified example only considers the impact on the *Power Subsystem*. The vehicle controller is assumed to control the fuel and air mixture, and control when the gear changes the automatic transmission to optimize engine and overall performance.

The block definition diagram in Figure 3.13 introduces a new type of block called a **constraint block**. Instead of defining systems and components, the constraint block defines constraints in terms of equations and their **parameters**.

In this example, the *Analysis Context* block is composed of a series of constraint blocks to analyze the vehicle acceleration to determine whether either the 4- or 6-cylinder vehicle configuration can satisfy its requirement. The constraint blocks define generic equations for *Gravitational Force*, *Drag Force*, *Power Train*

**FIGURE 3.13**

Block definition diagram for the *Analysis Context* that defined the equations for analyzing the vehicle acceleration requirement. The equations and their parameters are specified using constraint blocks. The *Automobile Domain* block from Figure 3.3 is referenced since it is the subject of the analysis.

Force, Total Force, Acceleration, and an Integrator. The *Total Force* equation, as an example, shows that ft is the sum of fi , fj , and fk . Note that the parameters are defined along with their units and/or dimensions in the constraint block.

The *Power Train Force* is further decomposed into other constraint blocks that represent the equations for torque from the *Engine*, *Transmission*, *Differential*, and *Wheels*. The equations are not explicitly defined, but the critical parameters of the equations are. This is important since it may be of value to identify the critical parameters, and to defer definition of the equations until the detailed analysis is performed.

The *Analysis Context* block also references the *Automobile Domain* block that was originally shown in the block definition diagram in Figure 3.3. The intent of this diagram is to identify both the equations for the analysis and the subject of the analysis. Referencing the *Automobile Domain* enables the equations to constrain the properties of the *Vehicle*, its components, and the physical environment. The parameters of the generic equations are bound to the properties of the system and the environment that is being analyzed, as described in the next section.

3.4.15 Analyzing Vehicle Acceleration Using the Parametric Diagram

The previous block definition diagram defined the equations and associated parameters needed to analyze the system. The **parametric diagram** in Figure 3.14 shows how these equations are used to analyze the vehicle acceleration to determine the time for the *Vehicle* to accelerate from 0 to 60 mph.

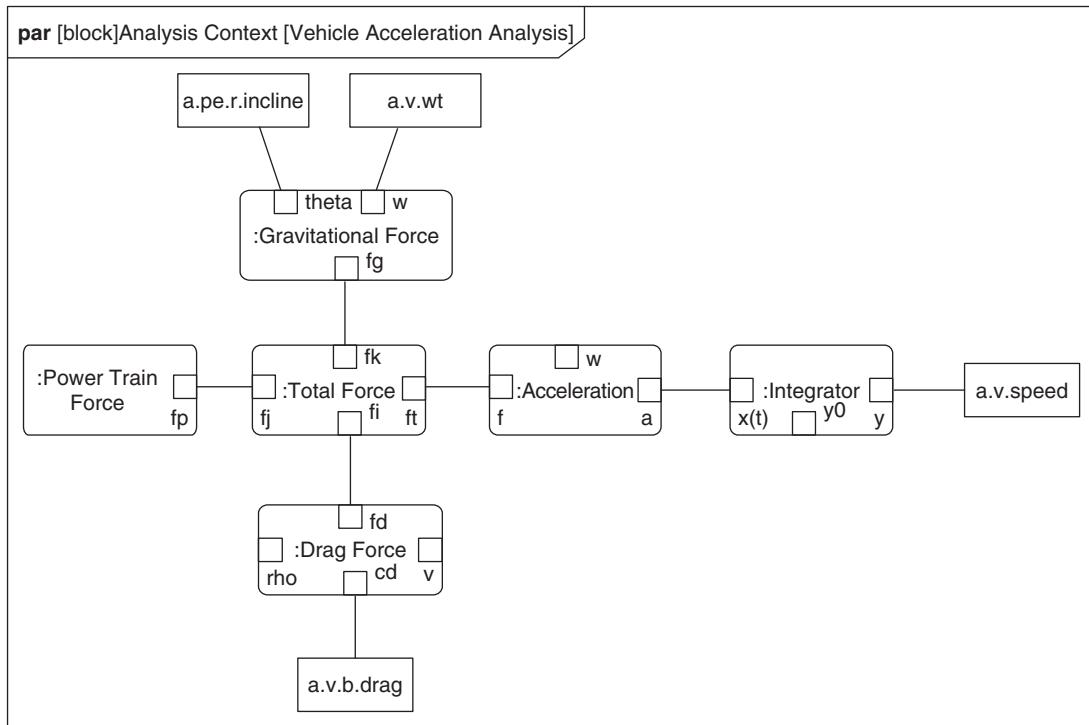
The parametric diagram shows a network of constraints (equations). Each constraint is a usage of a constraint block defined in the block definition diagram in Figure 3.13. The parameters of the equation are shown as small rectangles flush with the inside boundary of the constraint.

A parameter in one equation can be bound to a parameter in another equation by a **binding connector**. An example of this is the parameter ft in the *Total Force* equation that is bound to the parameter f in the *Acceleration* equation. This means that ft in the *Total Force* equation is equal to f in the *Acceleration* equation.

The parameters can also be bound to **properties** of blocks to make the parameter equal to the property. The properties of blocks are shown as the rectangles in the diagram. An example is the binding of the coefficient of drag parameter cd in the *Drag Force* equation to the drag property called *drag*, which is a property of the vehicle *Body*. The dot notation “*a.v.b*” that precedes the drag property specifies that this is a property of the body, which is part of the vehicle that is part of the *Automobile Domain*. Another example is the binding of the road *incline* angle to the angle *theta* in the gravity force equation. This binding enables parameters of generic equations to be set equal to specific properties of the blocks. In this way, generic equations can be used to analyze many different designs.

The parametric diagram and related modeling information can be provided to the appropriate simulation and/or analysis tools to support execution. This engineering analysis is used to perform sensitivity analysis and determine which property values are required to satisfy the acceleration requirement.

Other analysis can be performed to determine the required property values for the system components (e.g., *Body*, *Chassis*, *Engine*, *Transmission*, *Differential*,

**FIGURE 3.14**

Parametric diagram that uses the equations defined in Figure 3.13 to analyze *Vehicle Acceleration*. The parameters of the equations are bound to parameters of other constraints and to properties of the *Vehicle* and its environment.

Brakes, Steering Assembly) to satisfy the overall system requirements. In addition to the acceleration and fuel efficiency requirements, other analyses may address requirements for braking distance, vehicle handling, vibration, noise, safety, reliability, production cost, and so on. The parametrics enable the critical properties of the system to be identified and integrated with analysis models. Details of how to model constraint blocks and their usages in parametric diagrams are described in Chapter 7.

3.4.16 Analysis Results from Analyzing *Vehicle Acceleration*

As mentioned in the previous section, the parametric diagram is expected to be executed in an engineering analysis tool to provide the results of the analysis. This may be a separate specialized analysis tool that is not provided by the SysML modeling tool, such as a simple spreadsheet or a high-fidelity performance simulation depending on the need. The analysis results from the execution then provide values that can be incorporated back into the SysML model.

The analysis results from executing the constraints in the parametric diagram are shown in Figure 3.15. This example uses the **UML timing diagram** to display the results. Although the timing diagram is not currently one of the SysML diagram

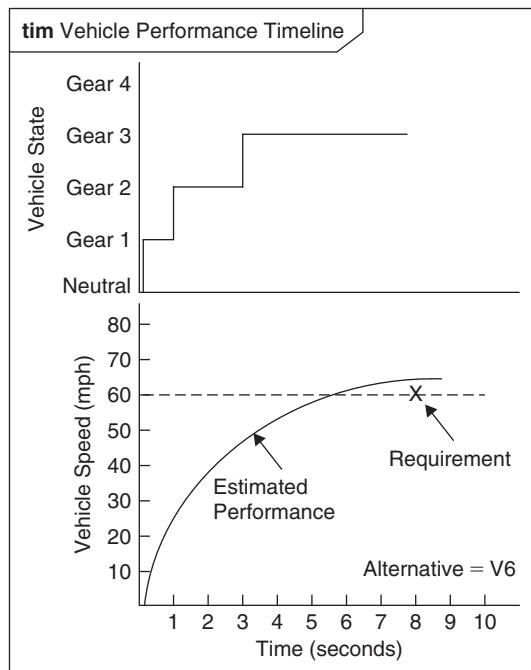


FIGURE 3.15

Analysis results from executing the constraints in the parametric diagram in Figure 3.14 showing the vehicle speed property and *Vehicle* state as a function of time. This is captured in a UML timing diagram.

types, it can be used in conjunction with SysML if it is useful for the analysis, along with other more robust visualization methods. The vehicle speed property is shown as a function of time, and the *Vehicle* state is shown as a function of time. The *Vehicle* states correspond to nested states within the *forward* state in Figure 3.8. Based on the analysis performed, the 6-cylinder (V6) vehicle configuration is able to satisfy its acceleration requirement but a similar analysis showed that the 4-cylinder (V4) vehicle configuration does not satisfy the requirement.

3.4.17 Using the *Vehicle Controller* to Optimize Engine Performance

The analysis results showed that the V6 configuration is needed to satisfy the vehicle acceleration requirement. Additional analysis is needed to assess whether the V6 configuration can satisfy the fuel efficiency requirement for a minimum of 25 miles per gallon under the stated driving conditions as specified in the *Fuel Efficiency* requirement in Figure 3.2.

The activity diagram in Figure 3.16 is a refinement of a portion of the *Provide Power* activity diagram in Figure 3.11. In this figure, the *vehicle controller* software has been added as an activity partition to support the analysis needed to optimize

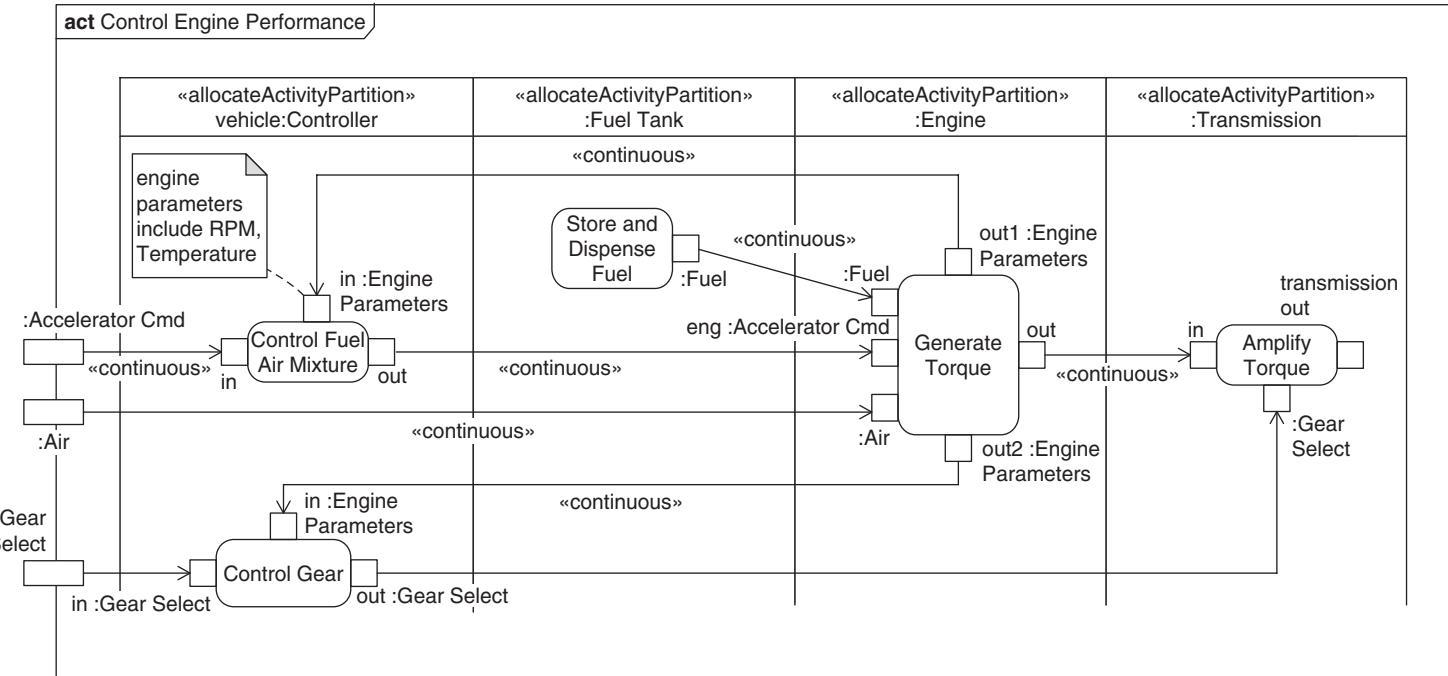


FIGURE 3.16

Activity diagram used to analyze the vehicle controller software interaction with the engine and transmission to optimize fuel efficiency and engine performance. This diagram is a refinement of a portion of the activity diagram in Figure 3.11.

fuel efficiency and engine performance. The *vehicle Controller* includes an action to *control fuel-air mixture* that in turn produces the engine accelerator command. The inputs to this action include the *Accelerator Cmd* from the *Driver* and *Engine Parameter* such as revolutions per minute (RPM) and engine temperature. The *vehicle Controller* also includes the *Control Gear* action to determine when to change gears based on engine speed (i.e., RPM) to optimize performance. The specification of the vehicle controller software can include a state machine diagram that changes state in response to the inputs consistent with the state machine diagram in Figure 3.8.

The specification of the algorithms to realize these actions requires further analysis. A parametric diagram can specify the required fuel and air mixture in terms of RPM and engine temperature to achieve optimum fuel efficiency, and they can be used to constrain the input and output of the actions. The algorithms must implement these constraints by controlling fuel flow rate and air intake, and perhaps other parameters. The algorithms, which consist of mathematical and logical expressions, can be captured in an activity diagram or directly in code. Based on the previous engineering analysis—the details of which are omitted here—the V6 engine is able to satisfy the fuel efficiency requirements and is selected as the preferred vehicle system configuration.

3.4.18 Specifying the *Vehicle* and Its Components

The block definition diagram in Figure 3.10 defined the blocks for the *Vehicle* and its components. The preceding analysis is used to specify the features of the blocks in terms of the functions they perform, their interfaces, and their performance and physical properties. Other aspects of the specification may include a state machine for state-based behavior and definitions of items that are stored by the block, such as fuel.

A simple example is the specification of the *Engine* block shown in Figure 3.17. This block was originally shown in the *Vehicle Hierarchy* block definition diagram in Figure 3.10. In this example, the *Engine* hardware element performs a function called *generate torque*, with ports that specify its interfaces to *air in*, *fuel in*, *control if*, and *out torque*. Selected properties are shown that represent performance and physical properties including its *displacement*, *combustion efficiency*, *max power*, and *weight* along with their **units**. The property values may also be represented as either a single value or a distributed value. Other blocks are specified in a similar way.

3.4.19 Requirements Traceability

The *Automobile System Requirements* were shown in Figure 3.2. Capturing the text-based requirements in the SysML model provides the means to establish traceability between the text-based requirements and other parts of the model.

The requirements traceability for the *Maximum Acceleration* requirement is shown in Figure 3.18. The requirement is **satisfied** by the *Power Subsystem*. The **rationale** refers to the engineering analysis based on the *Vehicle Acceleration Analysis* parametric diagram in Figure 3.14. The *Max Acceleration test case* is

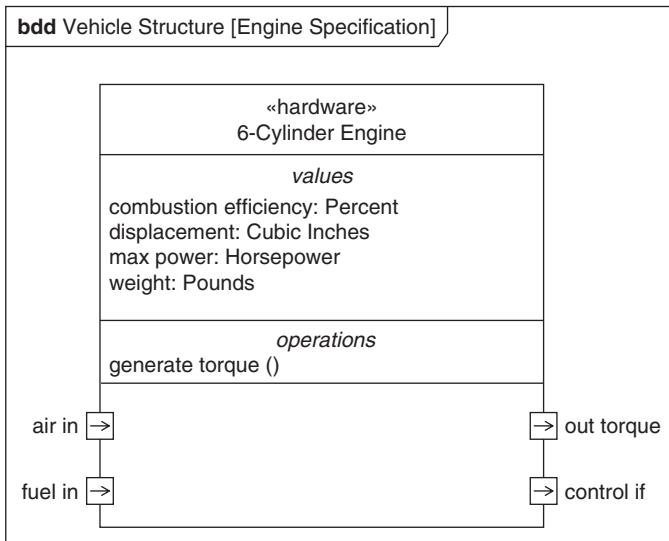


FIGURE 3.17

The block definition diagram shows the *Engine* block and the features used to specify the block. This block was previously shown in the *Vehicle Hierarchy* block definition diagram in Figure 3.10.

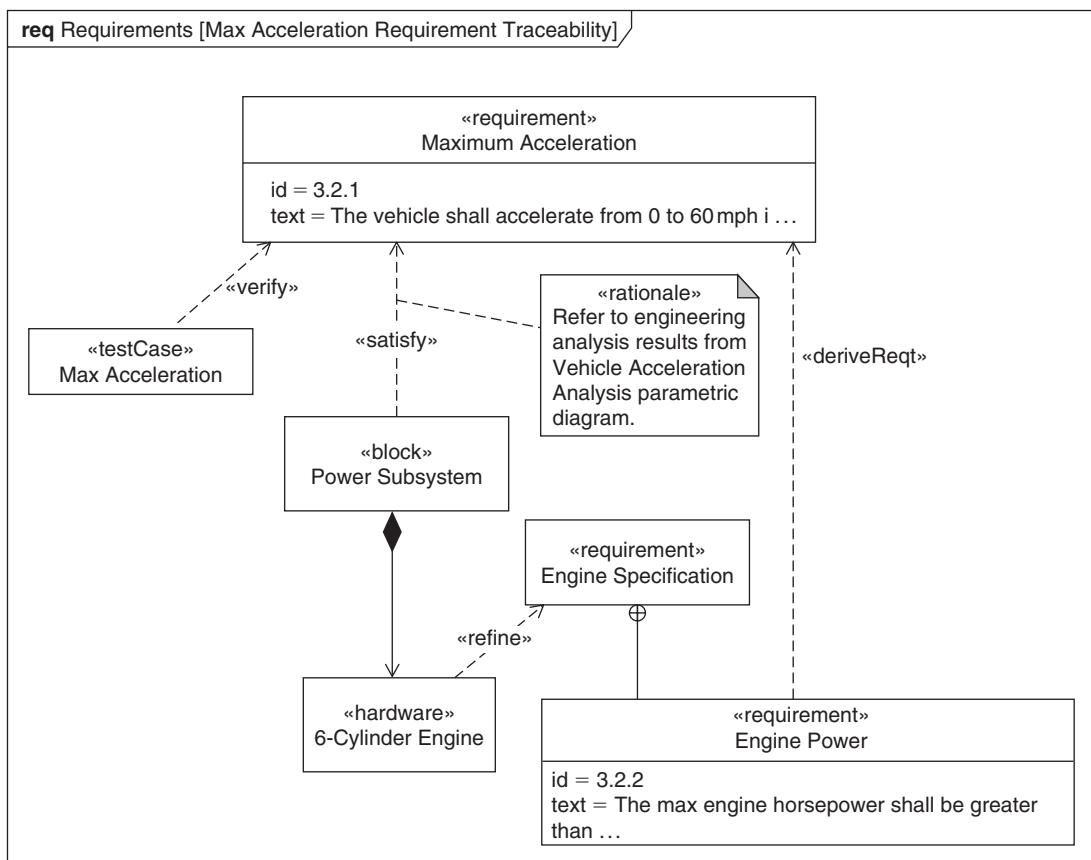
also shown as the method to verify that the requirement is satisfied. In addition, the *Engine Power* requirement is derived from the *Max Acceleration* requirement and contained in the *Engine Specification*. The *Engine* block refines the *Engine Specification* by restating the text requirements in the model. In this way, the system requirements can be traced to the system design and test cases, along with rationale.

The direction of the arrows points from the *Power Subsystem* design, *Max Acceleration* test case, and *Engine Power* requirement to the *Max Acceleration* as the source requirement. This is in the opposite direction that is often used to represent requirements flow-down. The direction represents a dependency from the design, test case, and derived requirement to the source requirement, such that if the source requirement changes, the design, test case, and derived requirement should also change.

As stated previously, there are other requirements relationships for trace and copy. The requirements are supported by multiple notation options including a tabular representation. Details of how SysML requirements and their relationships are modeled are described in Chapter 12.

3.4.20 Package Diagram for Organizing the Model

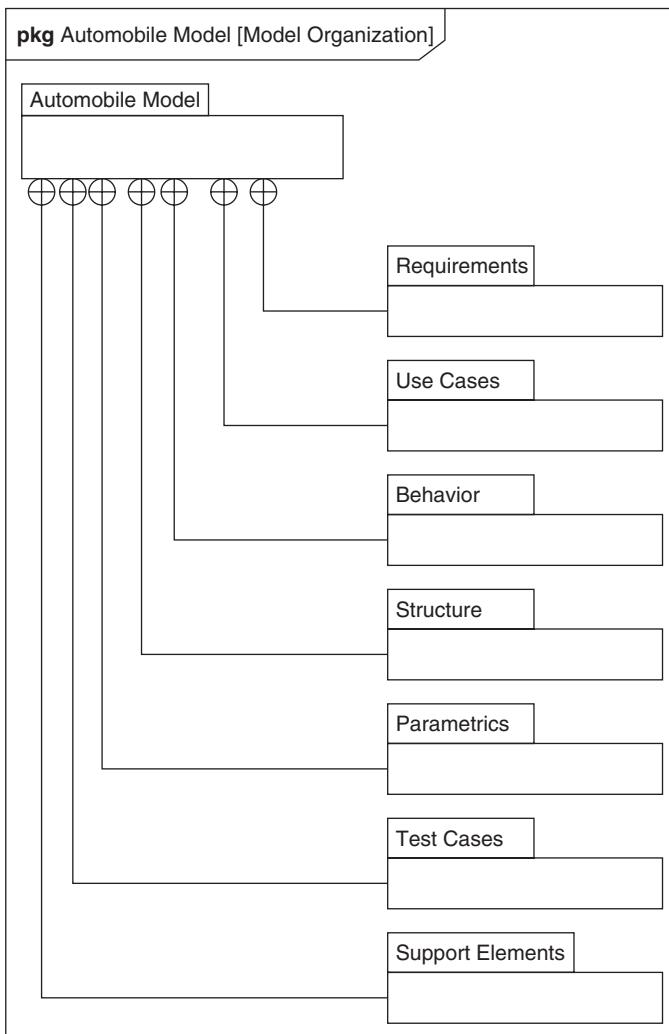
The concept of an integrated system model is a foundational concept for MBSE as described in Chapter 2. The **model** contains all of the **model elements**. The

**FIGURE 3.18**

Requirement diagram showing the traceability of the *Max Acceleration* requirement that was shown in the *Automobile Specification* in Figure 3.2. The traceability to a requirement includes the design elements that satisfy it, other requirements derived from it, and test cases to verify it. Rationale for the traceability relationships is also shown.

model elements and their relationships are captured in a model repository and can be displayed on diagrams. The model elements are integrated such that a model element that appears on one diagram may have relationships to model elements that appear on other diagrams. An example is the *Road* property, such as the *incline* angle that appears as a property of *Road* in the block definition diagram in Figure 3.3, and also is bound to a parameter of a constraint in the parametric diagram in Figure 3.14. The diagrams represent a view into this model.

A model organization is essential to managing the model. A well-organized model is akin to having a set of drawers to organize your supplies, where each supply element is contained in a drawer, and each drawer is contained in a particular cabinet. This facilitates understandability, access control, and change management of the model.

**FIGURE 3.19**

Package diagram showing how the model is organized into packages that contain model elements that comprise the *Automobile Domain*. Model elements in packages are displayed on diagrams. Model elements in one package can be related to model elements in another package.

The **package diagram** in Figure 3.19 shows how the model elements for this example are organized into **packages**. Each package **contains** a set of model elements. Model elements in one package can be related to model elements in another package. However, model organization enables each model element to be uniquely identified by the package that contains it. The model organization is generally similar to the view that is shown in the tool browser. Details on how to organize the model with packages are given in Chapter 5.

3.4.21 Model Interchange

A SysML model that is captured in a model repository can be imported and exported from a SysML-compliant tool in a standard format called **XML metadata** interchange (XMI). This enables other tools to exchange this information if they also support XMI. An example may be the ability to export selected parts of the SysML model to another UML tool to support software development of the controller software, or to import and export the requirements from a requirements management tool, or to import and export the parametric diagrams and related information to engineering analysis tools. The ability to achieve seamless interchange capability may be limited by the quality of the model and how the tool implements the standard, but this capability continues to improve. A description of XMI is included in Chapter 17.

3.5 Summary

SysML is a general-purpose graphical language for modeling systems that may include hardware, software, data, people, facilities, and other elements within the physical environment. The language supports modeling of requirements, structure, behavior, and parametrics to provide a robust description of a system, its components, and its environment.

The semantics of the language enable a modeler to develop an integrated model where model elements on one diagram can be related to model elements on other diagrams. The diagrams enable capturing and viewing the information in the model repository to help specify, design, analyze, and verify systems. The repository information can be imported and exported to exchange model data via the XMI standard and other exchange mechanisms.

The SysML language is a critical enabler of MBSE and can be used with a variety of processes and methods. However, effective use of the language requires a well-defined MBSE method. The automobile example illustrated the use of one such method. Other examples are included in Part III.

3.6 Questions

1. What are some of the aspects of a system that SysML can represent?
2. What is a requirement diagram used for?
3. What is an activity diagram used for?
4. What is a sequence diagram used for?
5. What is a state machine diagram used for?
6. What is a use case diagram used for?
7. What is the primary unit of structure in SysML?
8. What is the block definition diagram used for?
9. What is an internal block diagram used for?
10. What is a parametric diagram used for?
11. What is a package diagram used for?

PART

Language
Description

II

This page intentionally left blank

SysML Language Architecture

4

This chapter sets the stage for the detailed description of the SysML language that follows in the rest of Part II. It contains a discussion on the SysML language architecture and provides an introduction to common concepts that apply to all SysML diagrams. It also includes an introduction to the example used throughout the chapters in Part II to illustrate the language concepts. The remaining chapters in Part II provide the detailed description of the language.

4.1 The OMG SysML Language Specification

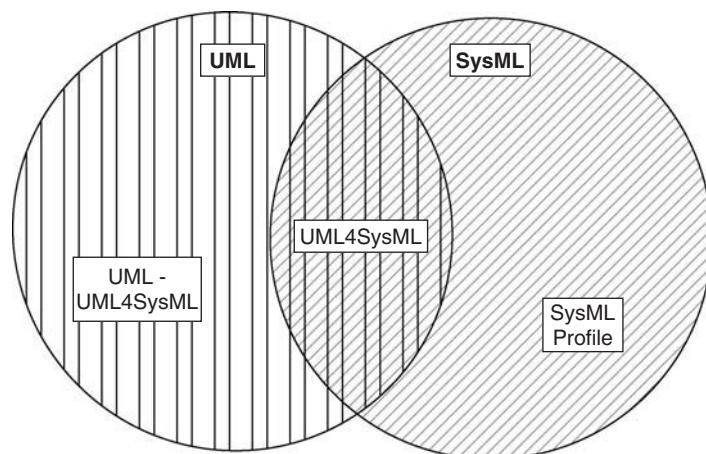
The official OMG SysML specification [1] for the SysML language has been publicly available since September 2007. The specification was developed in response to the requirements specified in the UML for Systems Engineering Request for Proposal (UML for SE RFP) [27]. It was formally adopted by the OMG in 2006 as an extension to the Unified Modeling Language (UML) [28]. The SysML specification is maintained and evolved by the OMG SysML Revision Task Force (RTF).

The SysML specification defines the SysML language concepts used to model systems. The SysML language concepts are described in three parts:

- An **abstract syntax**, or schema, described by a metamodel
- A **concrete syntax**, or notation, described using notation tables
- The **semantics**, or meaning, of the language concepts in the systems engineering domain

SysML is derived from the Unified Modeling Language, which was originally specified as a modeling language for software design but has been extended by SysML to support general-purpose systems modeling. As indicated in the Venn diagram in Figure 4.1, SysML reuses a subset of the UML language and adds extensions to meet the requirements in the UML for SE RFP.

Approximately half of the UML language was reused. The subset of UML reused by SysML is called UML4SysML as indicated in the diagram. The other portion of UML was not viewed as essential to meet the requirements of the UML for SE RFP.

**FIGURE 4.1**

Relationship between SysML and UML.

Limiting the portion of UML that was used reduces the requirements for SysML training and tool implementation, while satisfying the requirements for systems modeling.

The reusable portion of UML was in some cases used as is, without modification such as interactions, state machines, and use cases. Other parts of the reusable portion of UML were extended to address unique systems engineering needs. The profile is the standard UML mechanism used to specify the systems engineering extensions to the language and is described in more detail in Chapter 14. The profile-based approach was chosen over other extension mechanisms because many UML tools can interpret profiles directly. This enables the systems modeling community to leverage widely used UML-based tools for systems modeling. An additional benefit is that a profile of UML can be used in conjunction with UML to help bridge the gap between systems and software modeling.

The SysML profile is organized into the following discrete language units that extend the language:

- *Requirements*—textual requirements and their relationships to each other and to models
- *Blocks*—system structure and properties
- *Activities*—extensions to UML activities to support continuous behavior
- *Constraint blocks*—parametric models
- *Ports and flows*—extensions to the UML structural model to support flow of information, matter, and energy between system elements

The SysML profile is intended to be applied strictly, which means that models authored using the SysML extensions may only use the supported subset of UML, UML4SysML. The SysML language described in the specification is therefore the combination of UML4SysML and the SysML profile as indicated in Figure 4.1.

4.2 The Architecture of the SysML Language

There are typically three levels of concept relevant to a modeling language:

- Domain concepts for the domain being modeled (e.g., for SysML, general-purpose systems modeling concepts such as system and function)
- Mapping of domain concepts to language concepts (e.g., blocks, activities), often called the metamodel
- Instantiation and representation of the language concepts as they apply to a particular system (e.g., a block called airplane), often called the user model

This section describes these levels in more detail.

4.2.1 The General-Purpose Systems Modeling Domain

The goal of a modeling language is to enable the description of some domain of interest. For SysML the domain of interest is the general-purpose modeling of systems such as airplanes, automobiles, and information systems. The domain concepts are defined in the UML for Systems Engineering (SE) RFP that specifies the requirements for SysML. The requirements are organized into concepts needed to model structure, behavior, properties, requirements, and other systems modeling constructs. The following is an example of a requirement under the Structure section:

6.5.1.1. System hierarchy. UML for SE shall provide the capability to model the hierarchical decomposition of a system into lower-level logical or physical components.

Other examples include the requirement to model a system, its environment, functions, inputs/outputs, events, and property values to name a few. These concepts enable the modeler to describe a system such as an airplane.

Figure 4.2 shows a model of an airplane called *Bill's Plane*, and some of its relevant characteristics. In this example, the structure of the airplane is composed of its fuselage, wings, and landing gear. The airplane behavior is described in terms of its interaction with the pilot and the physical environment to support takeoff, flight, and landing. Some of its performance and physical properties include its

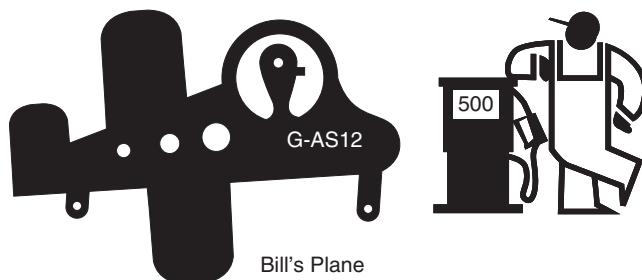


FIGURE 4.2

A typical system.

speed, dry weight, and fuel load. The principal requirement for this airplane is to fly a specified distance with a specified payload in a specified time, and it also needs to meet other requirements such as safety, reliability, and cost.

4.2.2 The Modeling Language (or Metamodel)

At the core of SysML is a **metamodel** that describes the concepts in the language, their characteristics, and interrelationships. This is sometimes called the abstract syntax, and is distinct from the concrete syntax that specifies the notation for the language. The OMG defines a language for representing metamodels, called the Meta Object Facility (MOF) [20]; it is used to define UML and other metamodels.

In a metamodel, the individual concepts in a language are described by **metaclasses** that are related to each other using relationships such as generalizations and associations. Each metaclass has a set of properties that characterize the language concept it represents, and a set of constraints that impose rules on the values for those properties.

The package diagram in Figure 4.3 shows a small fragment of *UML4SysML*, the MOF metamodel on which SysML is based. It shows one of the fundamental concepts of UML, called *Class*, and some of its important relationships. *Class* specializes *Classifier*, which enables it to form classification hierarchies. The figure also shows an association from *Class* to *Property*, which allows classes to have attributes. Another *Classifier*, *Data Type*, is used to describe values of attributes such as integers and real numbers. Finally, the notion of a *Package* is introduced; it can be used to group model elements, called generically *Packageable Elements*. All *Classifiers* are *Packageable Elements* and so are its specializations such as classes and data types.

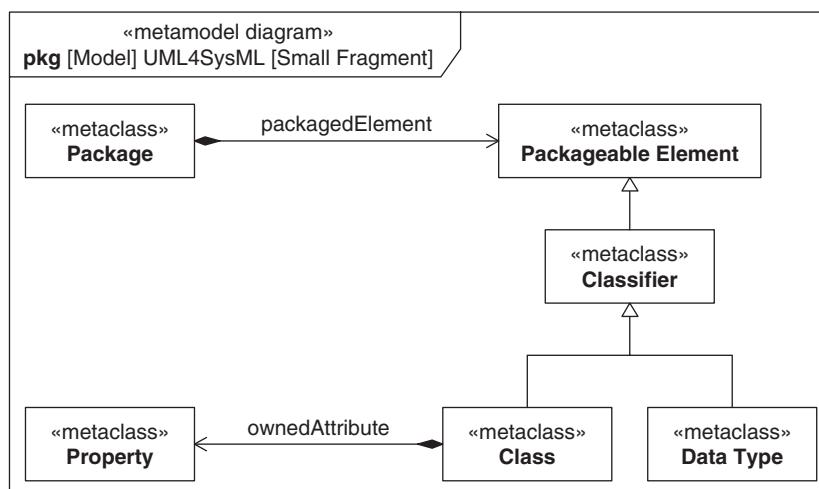
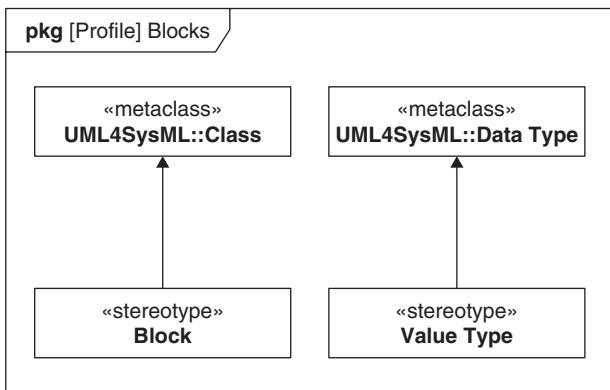


FIGURE 4.3

A fragment of the UML4SysML.

**FIGURE 4.4**

A fragment of the SysML profile for blocks.

A **profile** in UML is the mechanism used to customize the UML language. A profile contains **stereotypes**, which are similar to metaclasses, and are used to extend the metaclasses in UML to create new or modified concepts in the customization. The system engineering extensions to UML in SysML are described using a profile called the SysML profile.

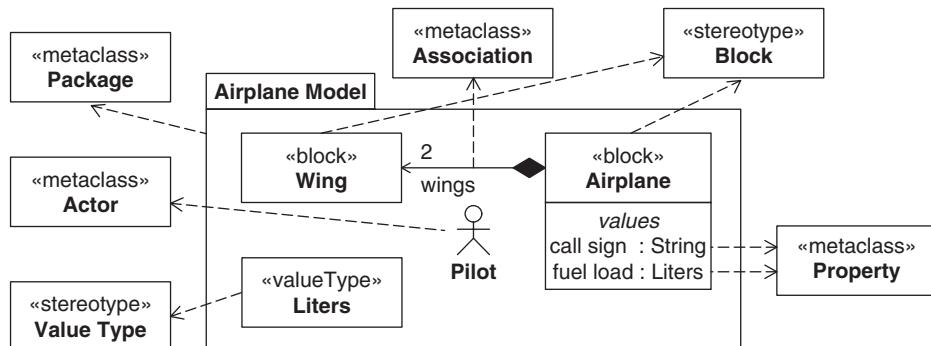
Figure 4.4 shows two SysML concepts in the *Blocks* language unit of the SysML profile and how they relate to UML metaclasses. *Class* and *Data Type* are UML metaclasses from the *UML4SysML* subset. *Block* extends *Class* and is the fundamental structural concept in SysML. *Value Type* extends *Data Type* and adds quantitative features such as unit and dimension.

The semantics of a language describe the meaning of its concepts in the domain of interest. Semantics are described via a mapping between the domain concepts and the language concepts. The domain concepts can be defined in natural language (e.g., English text) or more formally defined mathematically. For SysML, the domain concepts were defined by the requirements in the UML for SE RFP as English text, as described earlier.

The mapping between concepts in the systems modeling domain and the language concepts in SysML is performed by mapping the requirements in the UML for SE RFP to the metaclasses in the SysML metamodel, and it is captured in a requirements traceability matrix [29]. For example, a system and its components map to blocks, a composition relationship maps to a composite association, a function maps to an activity, and a requirement in the domain maps to a requirement in the SysML metamodel.

4.2.3 The System Model (or User Model)

As described in Chapter 2, the user model is a description of a system and its environment for a specific purpose, such as the validation of the requirements for the system or to specify the system's components. A SysML user model consists of

**FIGURE 4.5**

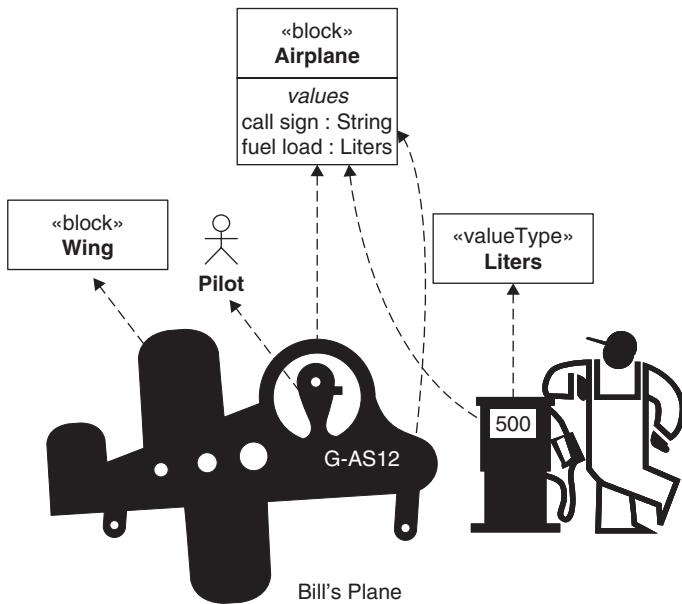
Relationship of metaclasses to model elements.

model elements that are instances of the metaclasses in the SysML metamodel; for example, a SysML block may be instantiated as an airplane, a fuselage, a wing, and a landing gear in the user model. The model elements represented in the user model conform to the metaclass properties, constraints, and relationships defined by the metamodel. These model elements are visualized using a concrete syntax (e.g., symbols on diagrams) as described in Section 4.3. The concrete syntax is mapped to the abstract syntax so that each symbol represents a specific concept. For example, a block and its properties have a specific graphical representation as a box symbol with internal compartments.

Figure 4.5 shows a fragment of a block definition diagram for defining airplanes, along with their mapping to the metaclasses that represent the various concepts. *Airplane Model* is a package containing *Airplane* and *Wing* blocks; *Pilot*, an actor (i.e., external to the system); and *Liters*, a value type. *Airplane* has two properties that describe two of its quantifiable characteristics: *call sign*, whose valid values are described by *String* (a primitive concept defined by SysML), and *fuel load*, with units of *Liters*. *Airplane* has an association to block *Wing*, which describes part of its structure, in this case its (two) *wings*.

As described in Chapter 2, a SysML modeling tool can store a user model as structured data in a model repository. The modeler uses the tool to enter and retrieve this information from the model repository, primarily by using the graphical representation provided by SysML diagrams. A SysML modeling tool that complies with the SysML specification enforces the metaclass properties, constraints, and relationships on the information entered or retrieved from the model.

Figure 4.6 shows how the original concept described in Figure 4.2 relates to the user model fragment described in Figure 4.5. That figure shows the class of airplanes, but in this example, we are referring to a specific airplane. *Bill's Plane* is a specific instance of the *Airplane* block with values for *call sign* and *fuel load* related to the corresponding properties of the block. *Bill* is an instance of *Pilot* and the wings of *Bill's Plane* are instances of the block *Wing*. The value type *Liters* describes how to interpret the value for *fuel load*, which in the case of *Bill's Plane* is 500 liters. Note that the stereotypes and metaclasses referenced by the model elements in Figure 4.5—*Block*, *Value Type*, *Actor*, and *Property*—all represent

**FIGURE 4.6**

Relating real-world concepts to model concepts.

something in the real world of the user. *Package*, however, does not; it is simply used to bring structure to the user model.

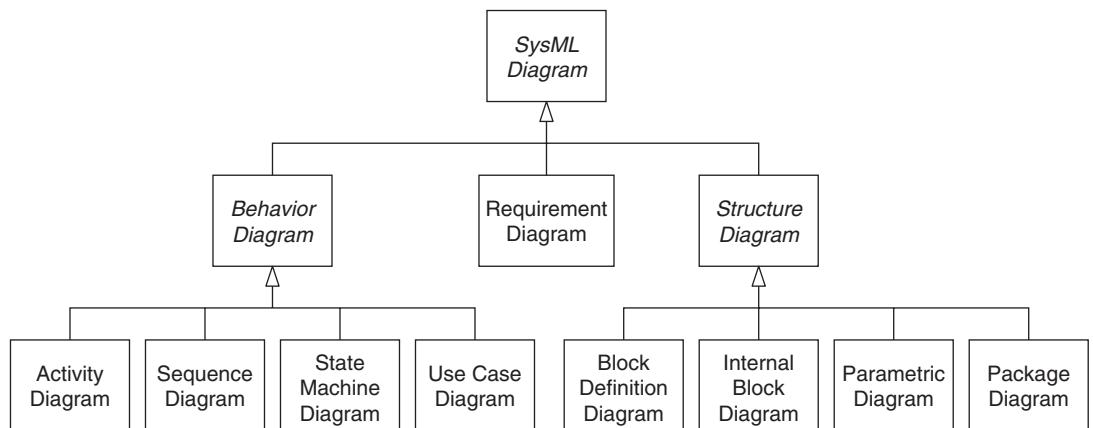
4.2.4 Model Interchange

As well as enabling the reuse of relevant concepts and diagrams from UML, building SysML as a formal extension of UML also enables SysML tools to leverage the data interchange format, called XML metadata interchange or XMI. XMI explicitly states how UML models, including models that use profiles, such as SysML, get converted into XML. The implementation of the XMI specification [21] is intended to enable SysML tools to read and write SysML models so that the modeling information can be exchanged between tools. XMI is summarized in Chapter 17.

4.3 SysML Diagrams

In addition to a metamodel, SysML defines a **notation**, or concrete syntax that describes how SysML concepts are visualized as graphical or textual elements. In the SysML specification, this notation is described in notation tables that map language concepts to graphical symbols on diagrams.

Figure 4.7 shows the SysML diagram taxonomy. SysML notation is based on the notation for UML, although several of the UML diagrams, including the object diagram, collaboration diagram, deployment diagram, communication diagram, interaction overview diagram, and timing diagram, were omitted. SysML includes

**FIGURE 4.7**

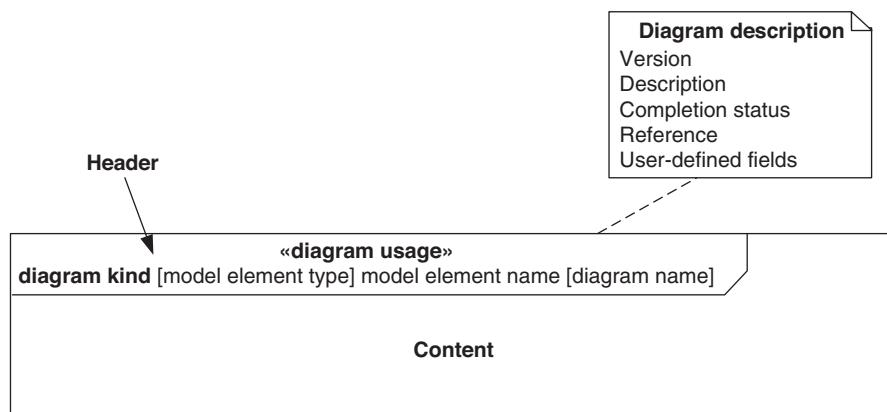
SysML diagram taxonomy.

modifications to other UML diagrams such as the class diagram, composite structure diagram, and activity diagram, and it adds two new diagrams for requirements and parametrics. The detailed notation tables that describe the symbols used on the diagrams can be found in the Appendix.

In addition to the graphical forms of representation used on SysML diagrams, SysML identifies the need for tabular and tree representations of model data, examples of which are included in various chapters in Part II.

4.3.1 Diagram Frames

Every SysML diagram must have a frame, as shown in Figure 4.8. Diagram frames provide a visible context for the diagram. The frame represents a model element

**FIGURE 4.8**

A diagram frame.

that provides the context for the diagram content. In addition, certain diagrams explicitly draw symbols on or to the frame boundary to indicate external interfaces of the model element owning the diagram.

The **diagram frame** is a rectangle with a header, or label, containing standard information in the top left corner. The rest of the rectangle is the content area, or canvas, where the symbols representing diagram content are drawn. An optional diagram description, providing further detail on the status and purpose of the diagram, can be attached to the frame boundary.

4.3.2 Diagram Header

The **diagram header**, or **label**, is a rectangle with its lower right corner cut off. It includes the following information:

- *Diagram kind*—an abbreviation indicating the type of the diagram
- *Model element type*—the type of model element that the diagram represents
- *Model element name*—the name of the represented model element
- *Diagram name*—the name of the diagram, which is often used to say something about its purpose
- *Diagram usage*—a keyword indicating a specialized use of a diagram

An example of a diagram frame with header is shown in Figure 4.3.

Diagram Kind

The **diagram kind** may take one of the following values, depending on the type of diagram:

- Activity diagram—**act**
- Block definition diagram—**bdd**
- Internal block diagram—**ibd**
- Package diagram—**pkg**
- Parametric diagram—**par**
- Requirement diagram—**req**
- Sequence diagram—**sd**
- State machine diagram—**stm**
- Use case diagram—**uc**

Model Element Type

Different diagrams represent different types of model element. The valid permutations are listed here by diagram:

- *Activity diagram*—activity control operator
- *Block definition diagram*—block, constraint block, package, model, model library
- *Internal block diagram*—block
- *Package diagram*—package, model, model library, profile, view
- *Parametric diagram*—block, constraint block

- *Requirement diagram*—package, model, model library, requirement
- *Sequence diagram*—interaction
- *State machine diagram*—state machine
- *Use case diagram*—package, model, model library

The choice of **model element type** is explained further in the following chapters where the diagrams are discussed. Strictly speaking, the model element type only needs to be included in the header to avoid ambiguity if there is more than one allowable model element type that the diagram can represent, although it can help to orient novices in the language. SysML does make provision for the model element type to be a user-defined stereotype where appropriate.

Diagram Name

Since a model can contain considerable amounts of information, the modeler may choose to only highlight selected features in a particular diagram for a given purpose, and hide (elide) other features that may detract from this purpose. The **diagram name** is intended to provide a concise description of the diagram's purpose.

Diagram Usage

The **diagram usage** describes a specialized use for the diagram type. The diagram usage name is included in the header in guillemets. For example, a modeler may specify a context diagram as a usage of a use case diagram. The diagram usage notation does not have any semantic foundation but is thought to be a useful notational extension.

4.3.3 Diagram Description

The **diagram description** is an optional note attached either to the inside or outside of the diagram frame. It is intended to enable the modeler to capture additional information about the diagram. The information includes some predefined fields but also has a provision for user-defined fields. The following are the pre-defined fields.

Version: Version of the diagram.

Completion status: A statement by the diagram author about the completeness of the diagram. It may include a statement, such as “in-process,” “draft,” or “complete,” and may also include a specific description of the information that is still missing from the diagram. A very important use of this field is to indicate whether the diagram is a complete view given its scope. Systems engineers are used to modeling tools that show the complete detail for a given scope, whereas SysML diagrams only show a subset of the possible details. This field can therefore be used by the diagram author to assert its intended completeness of coverage.

Description: Free text description of the diagram content or purpose.

Reference: References to other information about the diagram, or hyperlinks to related diagrams to aid in navigation.

4.3.4 Diagram Content

The **diagram content** area, or **canvas**, contains graphical elements that represent underlying elements in the model. SysML diagrams are composed of two types of graphical elements: nodes and paths. A node is a symbol that can contain text and/or other symbols to represent the internal detail of the represented model element. Paths, also known as edges, are lines that may have multiple additional adornments such as arrows and text strings. The amount of information in the description of many model elements is potentially very large and can lead to diagram clutter. To help mitigate this problem, SysML tools typically offer the user options to hide detail.

Properties and Keywords

SysML includes the notion of a **keyword** that is included in brackets called **guille-mets** as «keyword» before the name of a model element. The keyword identifies the type of model element (i.e., the metaclass) and is typically used to remove ambiguity when a type of graphical element (e.g., rectangle, dashed arrow) represents more than one modeling concept. Users can create their own keywords and associated meanings to further customize the language using stereotypes, as described in Chapter 14.

Symbols display certain commonly used information about their model element, such as name and type, in a formatted string that is often called their name string. Model elements often have additional properties, besides their name string, that also need to be displayed on diagrams. These are shown as a comma-separated list, enclosed in braces, following the name of the model element. A user can also add additional properties, sometimes called tags, to model elements associated with a keyword. To differentiate these from other properties, these properties are displayed before the name and after their associated keyword using the same form (i.e., a comma-separated list in braces).

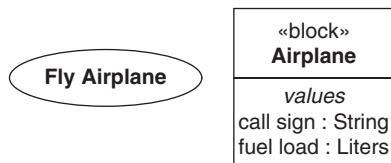
Node Symbols

Node symbols are generally rectangular but may be round-angles, ellipses, and so on. All node symbols have a name compartment that can be used to display the name string of the represented model element, along with any applicable keyword or keywords and properties. Some node symbols, in addition, have extra compartments used to display details of nested elements, either in textual or graphical form. In addition to its predefined nested elements, compartments can be used to display tags added by the user.

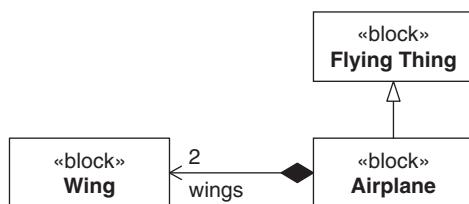
Figure 4.9 shows two examples of node symbols, a use case called *Fly Airplane* and the block *Airplane*. The *Airplane* symbol shows an internal compartment labeled *values* to store value properties.

Path Symbols

All path symbols are some kind of line, but they have different styles and ends depending on the modeling concept they represent. Paths may have a text adornment that will contain their name string, keywords, and additional properties,

**FIGURE 4.9**

Examples of node symbols.

**FIGURE 4.10**

Examples of path symbols.

although this is often hidden. Additional textual information may also be shown on the ends of the lines where the represented model element requires it.

Figure 4.10 shows two examples of path symbols, an association and a generalization. The association symbol indicates that an *Airplane* has exactly two wings. The generalization symbol indicates that an *Airplane* is a kind of *Flying Thing*.

Icon Symbols

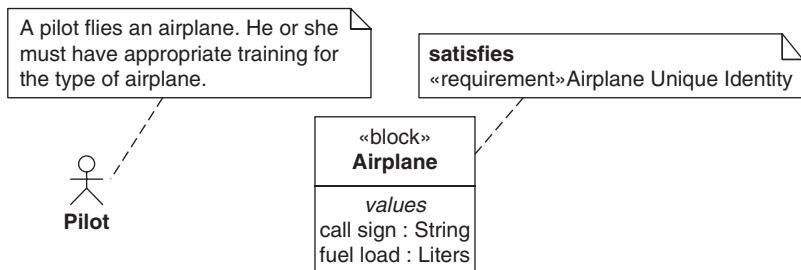
Icons are typically used to represent low-level concepts that do not have further internal detail. However, they can also be used as an alternate representation for most other symbols. In particular, a stereotype may specify an icon that can be used to display a stereotyped element. Where the model element represented by an icon has properties, such as a name, these are displayed in a text string floating near the object. Typically icons appear on the diagram canvas, or inside a node symbol, but icons can also appear on lines. Figure 4.11 shows two examples of icons: a stick figure representing the actor *Pilot* and a small box containing an arrow that represents fuel flowing into the *Airplane* block.

Note Symbols

A **note** symbol can be attached to the symbol for any model element or set of model elements and is used to annotate the model with additional textual information that may include a hyperlink to a reference document. The note symbol

**FIGURE 4.11**

Examples of icon symbols.

**FIGURE 4.12**

Examples of note symbols.

is a rectangular box containing the textual information with a cutoff upper right corner. Often the content of a note is free format textual description of the model element, but notes can also be used to display user-defined tags. They are also used extensively in SysML to display cross-cutting information, such as traceability to requirements (see Chapter 12) and allocations (see Chapter 13).

Figure 4.12 shows two examples of note symbols; one just stores a comment about the *Pilot* and the other represents the claim that *call sign* satisfies, the *Airplane Unique Identity* requirement.

4.3.5 Additional Notations

SysML also includes nongraphical representations of model information that is often useful for efficiently displaying large amounts of information. The forms of nongraphical representation that SysML supports are tables, matrices, and trees.

A **table** can be a highly efficient and expressive way to represent information. Tables have been used traditionally for capturing a wide variety of systems engineering information. For example, *N*-squared (*N*²) charts [30] capture interface information, requirements tables, and many other types of information. SysML allows the use of tabular notation as an alternative diagram notation to represent the modeling information contained in a SysML model repository. Tabular formats may be used to represent properties of model elements and/or relationships among model elements. The details of what information is captured in a table have not been specified to leave this as a flexible capability for a tool vendor to support, but the requirements and allocations chapters describe specific tabular formats for representing requirements and allocations, respectively.

When a table is used, the table is included in a diagram frame with the diagram kind **table** shown in the diagram label. Otherwise, the diagram label format is the same as that for any other kind of diagram. An example of a simple requirements table is shown in Figure 4.13.

Matrices, identified by the diagram kind **matrix**, are very useful for describing relationships, where the rows and columns of the matrix are model elements and its cells describe a relationship between the appropriate row and column

table [Requirement] Capacity [Decomposition of Capacity Requirement]		
id	req't name	req't text
4	Capacity	The Hybrid SUV shall carry 5 adult passengers, along with sufficient luggage and fuel for a typical weekend campout.
4.1	CargoCapacity	The Hybrid SUV shall carry sufficient luggage for 5 people for a typical weekend campout.
4.2	FuelCapacity	The Hybrid SUV shall carry sufficient fuel for a typical weekend campout.
4.3	PassengerCapacity	The Hybrid SUV shall carry 5 adult passengers.

FIGURE 4.13

Example of tabular format in SysML.

elements. **Trees**, identified by the diagram kind **tree**, typically describe hierarchical and other types of relationships that are frequently presented using browser panes in SysML modeling tools.

4.4 The Surveillance System Case Study

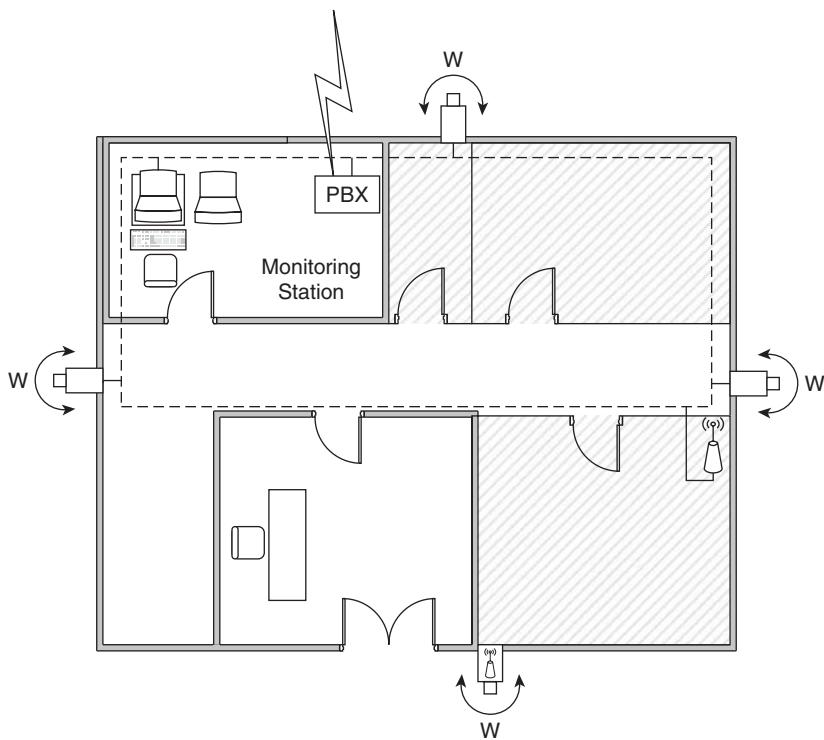
A single case study is used throughout this part of the book to help demonstrate the concepts in the SysML language.

4.4.1 Case Study Overview

A company, called ACME Surveillance Inc., produces and sells surveillance systems. Their range of surveillance systems products is intended to provide security either for homes or small commercial sites. Their systems use sophisticated pan and tilt cameras to produce video images of the surrounding area, and for a fee can be connected to a central monitoring service. ACME also produces the cameras and sells them as separate products for “do-it-yourself” enthusiasts.

The chapters in Part II use selected extracts of the ACME Surveillance Inc. model to highlight the features of SysML. A similar example is used in Chapter 16 to demonstrate the application of a model-based systems engineering method to the development of a residential security system.

Figure 4.14 shows a typical surveillance system setup for a small commercial site. The system has four wall-mounted surveillance cameras, three connected into the company’s Ethernet network and the fourth connected via a wireless access point. One of the offices is used to house the monitoring station for the surveillance system, which is also connected to the office network. This particular monitoring station consists of one workstation and an additional screen. The office has a PBX that the monitoring station uses to communicate to its designated command center.

**FIGURE 4.14**

Depiction of surveillance system example.

4.4.2 Modeling Conventions

Where elements are named in the example model, the names are chosen if possible to be valid English names. Whenever the names have more than one word, the words are separated by spaces. Names of model elements that represent definitions have the first letter of all words in uppercase. Names of features have the first letter of all words in lowercase.

The following chapters contain numerous SysML diagrams used to illustrate the concepts in the language. With few exceptions, each diagram is accompanied by a description, and to better relate the description to the figures, names used in the diagram are presented in italic font. Terms in bold are used to highlight fundamental concepts in the SysML language.

4.5 Chapter Organization for Part II

Chapters 5 through 14 in Part II describe the SysML language concepts and notation and how the language can be used to model a system. The ordering of the chapters is based on the logical development of the language concepts, including

concepts for model organization, structure, behavior, allocation, requirements, and profiles. The ordering is NOT based on a systems engineering process. Part III includes examples of model-based systems engineering methods that show how the language is used as part of a systems engineering process.

Each chapter describes applicable language concepts, diagram notation, and example diagrams to show how to create syntactically correct diagrams and models that conform to the language specifications.

4.6 Questions

1. What does the abstract syntax of a modeling language describe?
2. What are the two parts of the SysML abstract syntax?
3. How are language concepts defined in a metamodel?
4. What is a profile and what does it contain?
5. What do the semantics of a modeling language describe?
6. What is XMI used for?
7. What does the concrete syntax of a modeling language describe?
8. What are the five elements of a diagram header and what are they used for?
9. What are the four kinds of symbols that can appear on a diagram?
10. When is a keyword needed as part of a graphical symbol?

Discussion Topics

SysML could have been described completely as a metamodel, but instead used the UML profiling mechanism. Discuss the relative benefits of these two options.

Traditional engineering modeling tools always show all relevant model elements in any given diagram, whereas SysML allows modelers to selectively hide detail. Discuss the relative benefits of these two approaches.

In addition to graphical representations of the model through diagrams, SysML supports the use of nongraphical representations such as tables and trees. Under which circumstances does it make sense to use these different representations?

Organizing the Model with Packages

5

This chapter addresses the topic of model organization and describes the organizational concepts provided by SysML: models, packages, and views. In SysML, the fundamental unit of model organization is the package. Packages and their contents are shown on a package diagram. Packages are both containers and namespaces, two fundamental concepts in SysML.

5.1 Overview

In SysML, each model element is contained within a single container that is sometimes called its owner or parent. Contained elements are often called the child elements. When a container is deleted or copied, its child elements are also deleted or copied. Some child elements are also containers, which leads to a nested containment hierarchy of model elements.

Packages are one example of a container. The model elements contained within a package are called packageable elements, examples of which are blocks, use cases, and activities. Since packages are also packageable elements, they can support package hierarchies.

In addition to having a place in a containment hierarchy, each model element with a name must also be a member of a namespace. A namespace enables its elements to be uniquely identified within its namespace. A package is a namespace for the packageable elements it contains.

A model is a special type of package that contains a set of model elements that describe a domain of interest. The other chapters in Part II describe the different types of model elements, including structural, behavioral, and cross-cutting, and how they are used to describe a subject area of interest. This chapter describes how those elements are organized to enhance modeling effectiveness.

An effective model organization facilitates reuse of model elements, and also easy access and navigability among model elements. It can also support configuration management of the model, and exchange of modeling information with other tools, as described in Chapter 17. The importance of maintaining a well-defined model organization increases with the size of the model, but even small models

benefit from consistently applied organizational principles. The specific criteria for partitioning the model are methodology dependent, but some examples of model organization principles are included later in this chapter.

Because reuse is so important in modeling, SysML includes the concept of a model library, which is specifically intended to contain model elements that can be shared within and between models. Model libraries are more fully described in Chapter 14.

Views and viewpoints can be used to visualize models according to multiple organizing principles. A view is a kind of package used to show a particular perspective on the model, such as performance or security. A viewpoint represents a particular stakeholder perspective that specifies the contents of a view. A view conforms to a viewpoint.

There are a number of relationships between packages and their contents. An import relationship allows elements contained in one package to be imported into another package so that it can be referenced by its name. SysML also contains a generic concept of dependency between packageable elements, which can be specialized as needed.

5.2 The Package Diagram

The model elements contained within a package can be shown on a **package diagram**. The complete diagram label for a package diagram is as follows:

pkg [package type] package name [diagram name]

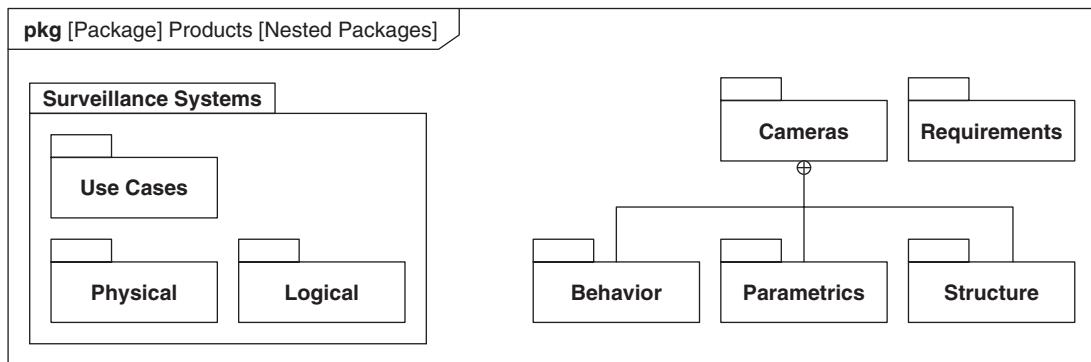
As described in Chapter 4, the diagram frame represents a particular model element that provides the context for the model elements represented on the diagram.

The first field in the diagram label is the diagram kind, **pkg**, which is short for package. The *package type* is the type of package that the diagram represents (e.g., model, package, model library, or view). The third field in the diagram header is the *package name* that identifies the particular package that the diagram frame represents. The last field is a user-defined *diagram name* used to provide further information about the diagram that typically summarizes the diagram purpose.

An example of a package diagram is shown in Figure 5.1. It shows several levels of the package hierarchy for the *Products* package of the ACME Surveillance Systems Inc. model. The notation tables for package diagrams are included in Table A.1 in the Appendix.

5.3 Defining Packages Using a Package Diagram

SysML models are organized into a hierarchical tree of packages that are much like folders in a Windows directory structure. Packages are used to partition elements of the model into coherent units that can be subject to access control, model navigation, configuration management, and other considerations. The most

**FIGURE 5.1**

An example package diagram.

significant types of packages used to organize models in SysML are models, packages, model libraries, and views.

A **package** is a container for other model elements. Any model element is contained in exactly one container, and when that container is deleted or copied, the contained model element is deleted or copied along with it. This pattern of containment means that any SysML model is a tree hierarchy of model elements.

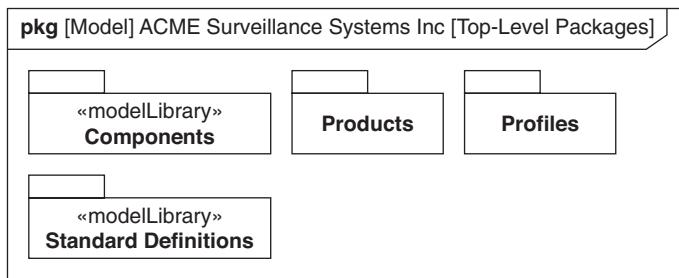
Model elements that can be contained in packages are called **packageable elements** and include blocks, activities, and value types, among others. Packages are themselves packageable elements, which allows packages to be hierarchically nested. The containment rules and other related characteristics of other packageable elements are described in the relevant chapters.

A **model** in SysML is a top-level package in a nested package hierarchy. In a package hierarchy, models may contain other models, packages, and views. The choice of model content and detail—for example, whether to have a hierarchy of models—is dependent on the methodology used. Typically, however, a model is understood to represent a complete description of a system or subject area of interest for some purpose, as described in Chapter 2.

A model has a single primary hierarchy containing all elements whose organizing principle is based on what is most suitable to meet the needs of the project. Views, which are described in Section 5.9, can be used to provide additional perspectives on the model using alternative organizing principles.

Often a package is constructed with the intent that it will be reused in many models. SysML contains the concept of a **model library**—a package that is designated to contain reusable elements. A model library is depicted as a package symbol with the keyword «modelLibrary» above the package name as shown in Figure 5.2 for *Components* and *Standard Definitions*. See Chapter 14 for more details on model libraries.

Relationships, such as dependency and import relationships, can be established between packages and between the packageable elements within those packages. These relationships are described in Sections 5.7 and 5.8.

**FIGURE 5.2**

Package diagram for the surveillance system model.

The diagram content area of a package diagram shows packages and other packageable elements within the package represented by the frame. Packages are displayed using a folder symbol, where the package name and keywords can appear in the tab or the body of the symbol. Where a model appears on a package diagram, which may happen where there is a hierarchy of models, the standard folder symbol includes a triangle in the top right corner of the symbol's body.

The package diagram in Figure 5.2 shows the top-level packages within the corporate model of *ACME Surveillance Systems Inc.*, as specified in the frame label of the diagram. The user-defined diagram name for this diagram is *Top-Level Packages*, indicating that the purpose of this diagram is to show the top level of the model's package structure. In this example, the model contains separate package hierarchies for

- Standard off-the-shelf components
- Standard engineering definitions such as SI units—from the French *Système International d'Unités* (also known as International System of Units)
- The company's products
- Any specific extensions required to support more domain-specific notations and concepts (extensions to SysML, called profiles, are described in detail in Chapter 14)

Each package should contain packageable elements specific to the purpose of the package. These elements can then be represented as needed on different SysML diagrams including structure, behavior, and requirement diagrams, as described in later chapters in this part of the book.

5.4 Organizing a Package Hierarchy

As described previously, a model is organized into a single hierarchical structure of packages. The top-level package is always a model that generally contains packages at the next level of the model hierarchy, as shown in Figure 5.2. These packages in turn often contain subpackages that further partition elements in the model into logical groupings.

Model organization is a critical choice facing the modeler because it impacts reuse, access control, navigation, configuration management, data exchange, and other key aspects of the development process. For example, a package may be the unit of the model assigned write privileges, granting only selected users the ability to modify its contents. In addition, when a particular package is “checked out” to modify its contents, other users may be excluded from making changes until the package is “checked in.” A poorly organized model makes it difficult for users to understand and navigate it.

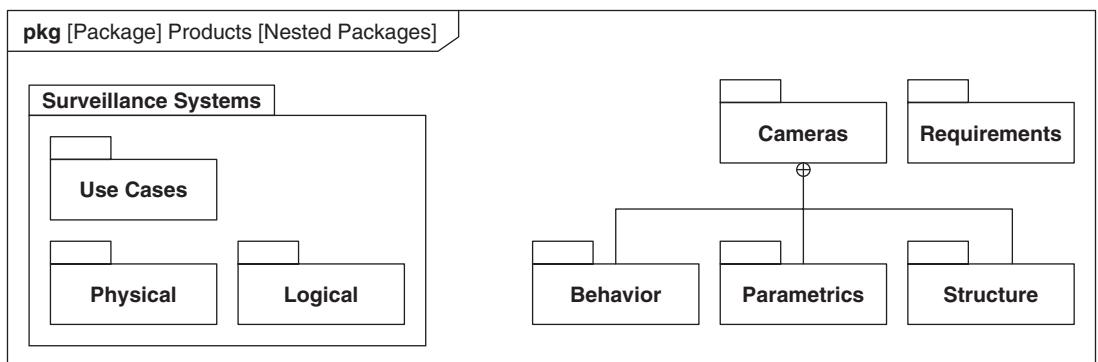
The model hierarchy should be based on a set of organizing principles. The following are some of the possible ways to organize a model:

- By system hierarchy (e.g., system level, subsystem level, component level)
- By process life cycle where each model subpackage represents a stage in the process (e.g., requirements analysis, system design)
- By teams that are working on the model (e.g., Requirements Team, Integrated Product Team (IPT) 1, 2)
- By the type of model elements contained in it (e.g., requirements, behavior, structure)
- By model elements that are likely to change together
- By model elements organized to support reuse (e.g., model libraries)
- By other logical or cohesive groupings of model elements based on defined model-partitioning criteria
- A combination of the preceding

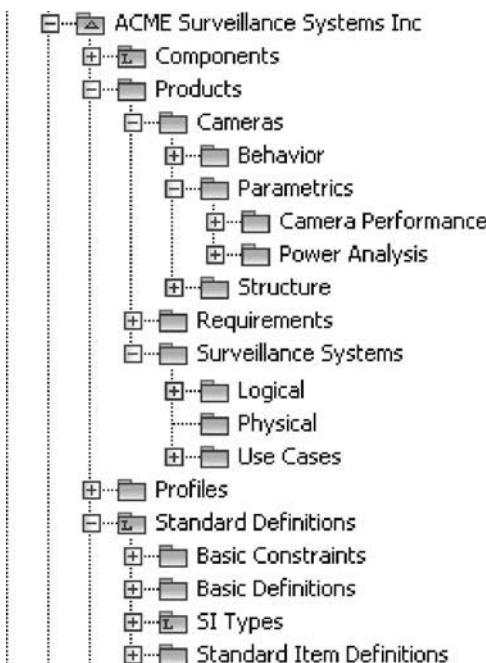
The **containment relationship** relates parents to children within a package hierarchy. Several levels of containment hierarchy can be shown on the package diagram using a containment relationship between container elements and their contained elements. The containment relationship is shown as a line with a cross-hair at the container (parent) end, but with no adornment on the ends associated with the contained elements (children). Each containment relationship can be shown as a separate path, but typically they are shown as a tree with one cross-hair symbol and many lines radiating from it. An alternative representation of the containment relationship is to show the nested model elements enclosed within the body of the package symbol.

Figure 5.3 shows the three packages contained within the *Products* package of the corporate model: *Surveillance Systems*, *Cameras*, and *Requirements*. This example uses both notations for package containment. Different organizational principles are used for the *Products*, *Cameras*, and *Surveillance Systems* packages. The *Products* package is organized to contain packages for the two primary product lines that the company offers and an additional package for all requirements specifications. The *Cameras* package hierarchy is organized by artifact type, and it includes packages to capture the structural, behavioral, and parametric aspects of the camera. The *Surveillance Systems* package hierarchy is organized based on architectural principles that require a *Logical Architecture* package, a *Physical Architecture* package, and a *Use Cases* package.

The containment hierarchy is generally one of the primary browser views visible in a tool. Figure 5.4 provides an example of the expanded browser view

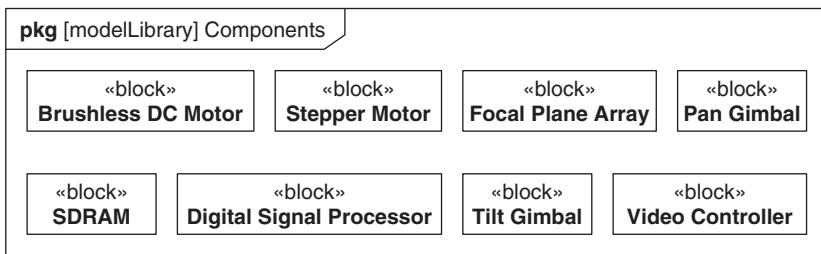
**FIGURE 5.3**

Showing nested packages on a package diagram.

**FIGURE 5.4**

Browser view of the model's package hierarchy.

corresponding to the model organization from Figure 5.3. The containment hierarchy generally expands as the model evolves to include other nested packages with increasing number and type of model elements. A tool generally enables the containment hierarchy and associated content to be viewed in an expanded or contracted form from the browser, similar to the file browser in Windows. Models

**FIGURE 5.5**

Showing the contents of the components package using a package diagram.

and packages form the branches of the containment hierarchy with other model elements appearing as lower-level branches and leaves.

5.5 Showing Packageable Elements on a Package Diagram

In addition to packages, package diagrams are used to show packageable elements. Packageable elements are normally represented by node symbols of various shapes and sizes, although icons can also be used.

The package diagram in Figure 5.5 shows more detail of the *Components* package from Figure 5.2 and contains a set of off-the-shelf components intended for use in building cameras and surveillance systems. The components are blocks, as indicated by the `<block>` keyword, and are shown within the diagram frame that represents the *Components* package. The diagram only shows some of the model elements within the package to reduce clutter. As explained in Chapters 2 and 4, diagrams are simply views of the underlying model and may not show all possible contents of the diagram's context. The diagram name is also elided, but could have been included to highlight the diagram purpose.

5.6 Packages as Namespaces

In addition to acting as a container for packageable elements, a package is a **namespace** for all named elements within it. Most SysML model elements have names although a few, such as a comment, do not.

Any type of namespace defines a set of uniqueness rules to distinguish between the different named elements contained within it. The uniqueness rule for packageable elements in packages is that each element of a particular element type within the package must have a unique name. In practice, however, this can be confusing when the element types are not clearly distinguishable. As a result, many projects require that all packageable elements of a package are uniquely named, even if they have different types. Where this restriction is not used, the presence of the type keyword in the symbol—for example, `<block>` in Figure 5.5—can help to remove potential ambiguity.

As stated earlier, a package hierarchy can include multiple levels of nested packages, meaning that a model element can be contained within a package that is contained in an arbitrarily number of higher-level packages. The containment relationship between a parent and child is unambiguously represented in a browser view of the model. It is also obvious when a model element appears as a symbol with its name on the canvas of a diagram that represents its parent. However, sometimes a model element needs to be shown on a diagram that does not represent its parent. Simply using the model element's name is misleading in this case because it gives a false impression of the containment relationship.

The solution is to show a **qualified name** in the symbol for that model element. If the model element is nested within the containment hierarchy of the package represented by the diagram, then the qualified name shows the relative path from that package to the contained element. If the model element is not nested within the package represented by the diagram, the qualified name contains the full path from the root model to the element.

The qualified name for a model element always ends with the model element name, preceded by a path with each containing namespace in the path delimited by a double-colon symbol “::”, so that when reading the qualified name, the path is resolved from left to right. For example, a model element X that is contained within package B, which in turn is contained within package A, is represented as A::B::X.

Figure 5.6 shows some examples of the use of qualified names in a package diagram that describes the *Standard Definitions* package shown in Figure 5.3. The symbol named *Basic Types::Point* denotes a value type called *Point* within a package called *Basic Types*, within the *Standard Definitions* package. *Point* is used later to specify the scan pattern of a surveillance camera. The other two symbols represent model elements that are external to *Standard Definitions* and therefore have fully qualified names that correspond to the path name from the corporate model, *ACME Surveillance Systems Inc*. In a package hierarchy, each model element is uniquely identified within its type by its qualified name regardless of which namespace it is contained in.

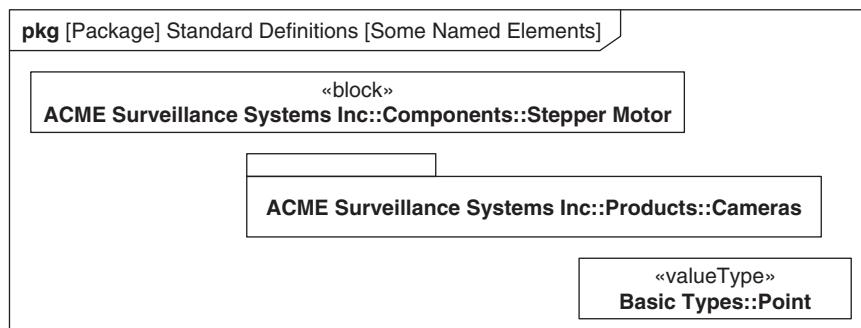


FIGURE 5.6

Using qualified names to represent a model element within a containment hierarchy.

5.7 Importing Model Elements into Packages

Depending on the organization of a model, model elements from different packages are often related to one another and these relationships usually need to be represented on package diagrams. In this case, a given diagram may need to display elements from many packages, so a more scalable alternative than using a qualified name, as shown in Figure 5.6, is needed to avoid diagram clutter.

An import relationship is used to bring an element or collection of elements belonging to a source package into another namespace, called the target namespace. The names of imported element names become part of the target namespace and do not require a qualified name when shown on a diagram that represents the target namespace.

A **package import** applies to an entire package, and all the model elements of the source package are imported into the target namespace. An **element import** applies to a single model element, and may be used when it is unnecessary and possibly confusing to import all the elements of a package.

A name clash occurs when two or more model elements in the target namespace would have the same names as the result of imports. An element import has an alias field that can be used to provide an alternate name for a model element to prevent a name clash in the target namespace. The rules on name clashes are as follows:

- If an imported element name clashes with a child element of the target namespace, that element is not imported, unless an alias is used to provide a unique name.
- If the names of two or more imported elements clash, then neither can be imported into the target namespace.

The named elements recognized within a namespace, whether through direct containment or as a result of being imported, are called **members**. Members have a **visibility**, either public or private, within their namespace. The visibility of a member determines whether it can be imported into another namespace. A package import only imports names with public visibility in the source package into the target namespace. Furthermore, an import relationship can state whether the imported names should be public or private within the target namespace.

When access control on a model is enforced by a modeling tool, an imported element can only be changed in the source package, although any relevant changes made to the element are automatically visible in any diagrams representing the target package.

The import relationship is shown using a dashed arrow, labeled with the keyword «import». The arrow points to the source from which names are being imported and the tail points to the namespace into which the names are to be imported. The arrow points either to an individual model element (element import) or to an entire package (package import). The keyword «access» is used instead of «import» when elements are to be imported as private members of the target namespace.

Figure 5.7 shows three packages, *P1*, *P2*, and *P3*, in package *Parent*. The package called *Model::P1* is not contained in the diagram's context and so its qualified

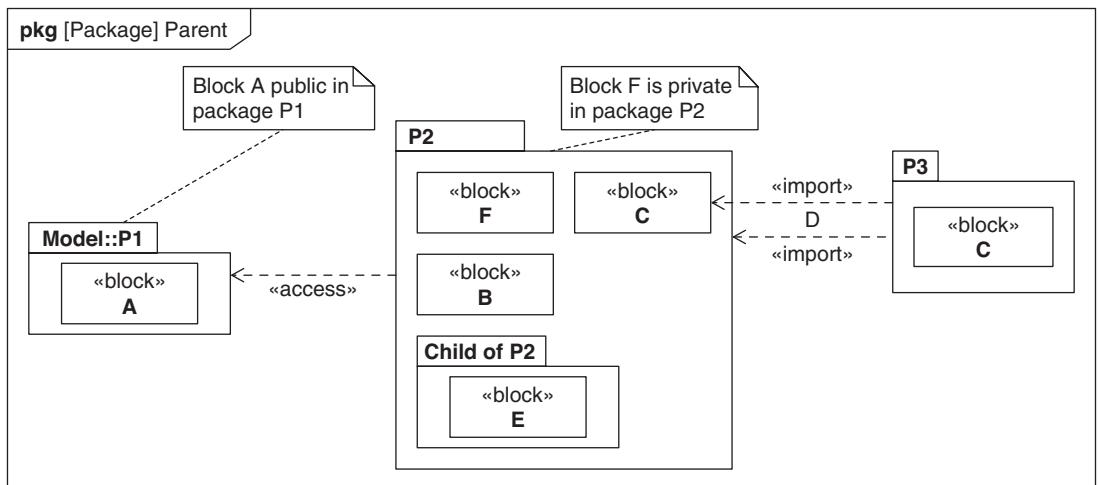
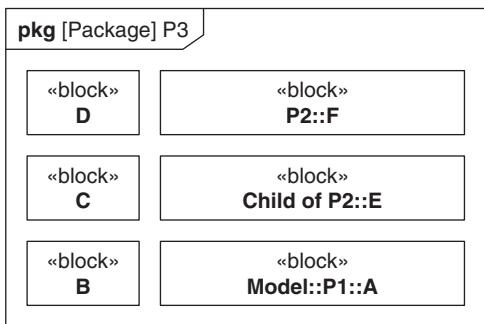
**FIGURE 5.7**

Illustration of «import» and «access».

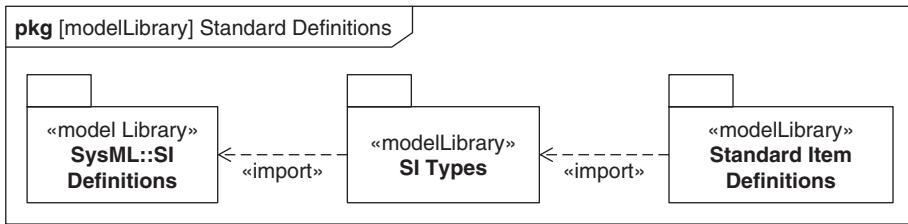
name has to be used. *Model::P1* contains one block, called *A*, with public visibility (SysML does not have a graphical notation for visibility, hence the notes attached to the symbols). Package *P2* privately imports *P1* and contains a set of blocks, *B* and *C*, which are defined with public visibility, and *F*, which is defined with private visibility. *P2* also contains a nested package called *Child of P2*, which in turn contains a single public block, *E*. Package *P3* defines a public block, *C*, and imports the whole package *P2*, but also separately imports block *C* with the alias *D* to avoid a name clash. Note that the alias *D* is annotated on the import relationship.

Figure 5.8 demonstrates the effect of import relationships on naming. It shows a diagram representing package *P3* showing the names of various model elements from Figure 5.7. Blocks *B*, *C*, and *D* (an alias for *P2::C*) can be shown using simple names because they are members of the *P3*, either by direct containment or because they were imported. Block *E* has to be qualified by its parent *Child of P2*, whose name is visible because *P3* has imported *P2*. Block *F* has to be qualified by *P2* because it was defined to be private and so is not imported, but *P2* is visible because it is in the same namespace as *P3*. Block *A* has to be qualified by its parent's fully qualified name, *Model::P1*, because although it was defined with public visibility, *Model::P1* was imported privately into *P2* and was therefore not visible in *P2* and so was not imported into *P3*.

Figure 5.9 shows some of the import relationships within the *Standard Definitions* package. It contains an example of a reusable model library called *SI Definitions*, which is defined within the *SysML* package. (These *SI* definitions are defined as a nonnormative model library in Annex C of the *SysML* specification [1].) *SI Definitions* is imported into the *SI Types* package, which provides a common set of units for use throughout the model. *SI Types* is in turn imported for

**FIGURE 5.8**

Naming in package *P3*.

**FIGURE 5.9**

Importing a library of SI unit types into the Standard Definitions package.

use within many other packages, one of which is the *Standard Item Definitions* package that contains definitions of information, material, and energy flowing through the surveillance systems.

5.8 Showing Dependencies between Packageable Elements

A **dependency** relationship can be applied between packageable elements to indicate that a change in the element on one end of the dependency may result in a change in the element on the other end of the dependency. The model elements at the two ends of the dependency are called client and supplier. The client is dependent on the supplier, such that a change in the supplier will result in a change in the client.

A dependency between packages is used when the content of one package is dependent on the content of another package. For example, the software applications in the application layer of the system software may depend on the software components within the system software's service layer. This may be expressed in a model of the software architecture by a dependency between the package that represents the application layer and the package that represents the service layer.

Dependencies are often used to specify a relationship early in the modeling process that is subsequently replaced or augmented when the precise nature of the relationship is better defined. There are various types of dependency that can be used on the package diagram and selected other diagrams. The following is a list of the more common types of dependencies:

- **Use**—indicates that the client uses the supplier as part of its definition
- **Refine**—indicates that the client represents an increase in detail compared to the specification of the supplier, such as when detailed physical and performance characteristics are included in a component definition. This relationship is often used in requirements analysis, as described in Chapter 12.
- **Realization**—indicates that the client realizes the specification expressed in the description of the supplier, such as when an implementation package realizes a design package
- **Trace**—indicates that there is a linkage between the client and supplier without imposing the more significant semantic constraints of a more precise relationship

A dependency is represented by a dashed line with an open arrow pointing from the client to the supplier. The type of dependency is indicated by a keyword in guillemets.

Figure 5.10 shows some of the types of dependency relationships in the *Camera Performance* view, which can be seen in Figure 5.11. The constraint block *Video Stream Rate* is a more precise representation (refinement) of the *Video Performance* requirement. *Video Stream Rate* uses a definition of megabits per second (Mbps) as part of its definition. The activity *Generate Video Outputs* is traced to the *Video Stream Rate* because if this constraint changes, the performance of the activity may need to be reevaluated. *Generate Video Outputs* is allocated to *Camera* to indicate that the camera is responsible for that activity. Details of these various model elements are described in later chapters.

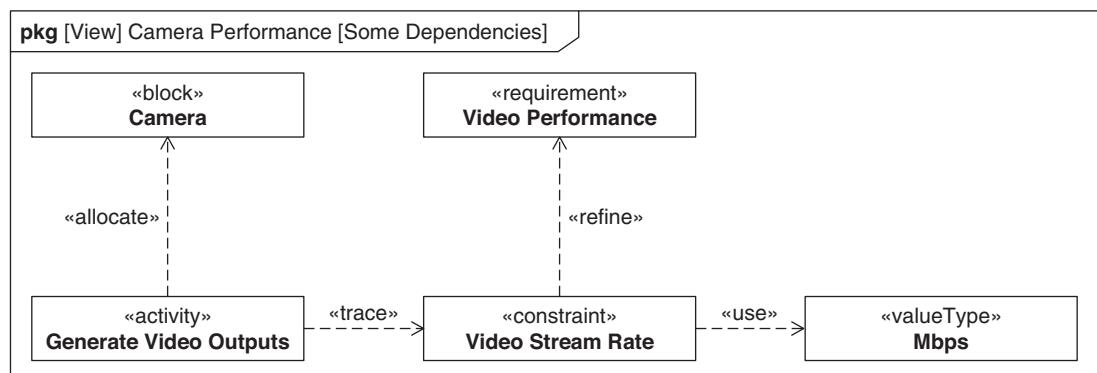
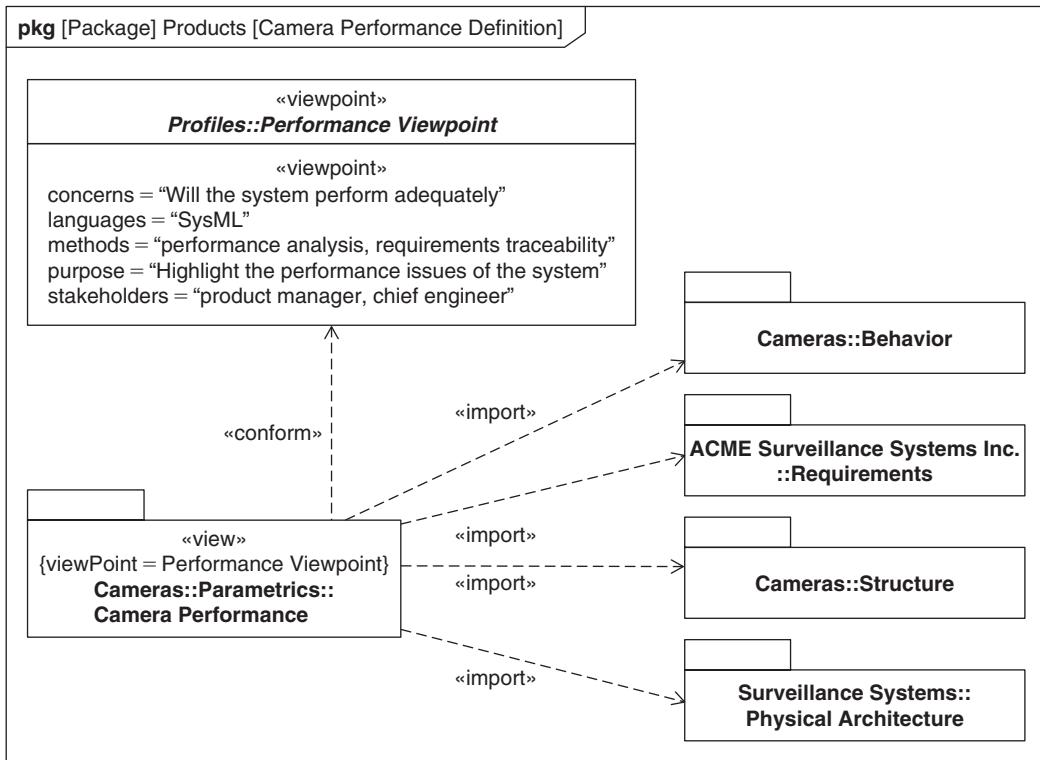


FIGURE 5.10

Example of dependencies in the camera performance view.

**FIGURE 5.11**

Definition of a performance viewpoint and a view that conforms to it.

5.9 Specifying Views and Viewpoints

The package containment hierarchy provides the fundamental organization of a model. However, it is often useful to incorporate a set of model elements that span multiple namespaces into a view of the model that supports a particular stakeholder perspective. SysML introduces the concepts of view and viewpoint to facilitate this. The view and viewpoint terminology in SysML is generally consistent with the IEEE 1471 standard, “Recommended Practice for Architectural Description of Software-Intensive Systems” [17].

A **viewpoint** describes a perspective of interest to a set of stakeholders that is used to specify a view of a model. A viewpoint includes a set of properties that identify:

- The purpose or reason for taking this perspective
- The stakeholders who have an interest in this perspective
- The concerns that the stakeholders wish to address
- The languages used to present the view
- The methods used to establish the view

A **view** is a type of package that conforms to a viewpoint. The view imports a set of model elements according to the viewpoint methods and is expressed in the viewpoint languages to present the relevant information to its stakeholders. The view is intended to provide the model information that addresses the stakeholder concerns. The properties of a viewpoint are often specified informally, as guidance to view builders, but can in theory be specified precisely enough to allow automated construction and evaluation of a view.

A viewpoint is represented as a rectangle symbol with the keyword «viewpoint» and the viewpoint name in the name compartment. The viewpoint properties are shown in a separate compartment headed «viewpoint». A view is represented as a package symbol with the keyword «view», along with the view name and a reference to its viewpoint. A conformance relationship between a view and viewpoint is shown as a dashed arrow pointing from the view to the viewpoint, adorned with the keyword «conform».

A viewpoint that is often important from the perspective of a system architect is one that emphasizes those aspects of the model that affect system performance. In Figure 5.11 the *Performance Viewpoint* highlights those aspects of the model that focus on performance. The *Camera Performance* view conforms to the *Performance Viewpoint*. The *Camera Performance* view imports the *Structure* and *Behavior* packages of *Cameras* because they contain elements whose performance is being assessed. It also imports the *Requirements* package to refer to the performance requirements. Finally, it imports the *Surveillance Systems::Physical Architecture* package to enable the system architect to assess factors in the camera environment that may affect camera performance.

5.10 Summary

A well-defined model organization is essential to ensure that the model is partitioned into model elements that support reuse, access control, navigability, configuration management, and data exchange. Different organizing principles can be applied to establish a consistent package hierarchy with nested packages, each of which contains logical groupings of packageable elements. The following list summarizes the important aspects of model organization.

- The principal SysML organizing construct is called a package. Package diagrams are used to describe this model organization in terms of packages, their contents, and relationships.
- A model is a type of package that represents a description of a system for a given purpose. Models are the roots of package hierarchies. If the area of interest is sufficiently complex then it may have submodels.
- Package hierarchies are based on the concept of containment or ownership of packageable elements. An essential aspect of containment is that the packageable elements in a package get deleted or copied with their container. Examples of packageable elements are blocks, activities, and value types.

- Packages are also namespaces for a set of named elements called members. A namespace defines a set of rules for uniquely identifying an individual member. The namespace rule for packages is that a member must have a unique name within all members of its type (e.g., block, activity).
- The names of the diagram's symbols must allow a viewer to explicitly understand where the represented element is within the model containment hierarchy. If a symbol represents a member of the package that the diagram represents, then its name (and sometimes keyword) is all that is required. Otherwise a qualified name is required, which is a concatenation of the member's name and a path of all the namespaces between the member and the root model or diagram context.
- Package (and other) diagrams can get very cluttered with qualified names. To avoid this, SysML provides a mechanism to import the members from a package into a namespace, either as a whole package or as individual model elements. The visibility of the member in its source package governs whether it is a member of the target namespace.
- Model elements depend on each other in various ways. The dependency relationship between a supplier and a client element indicate that the supplier element is subject to change if the client element changes. Different types of dependencies are identified with a keyword and are used for specific purposes such as refinement, allocation, and traceability.
- A model has a single containment hierarchy, which therefore imposes a single organizational perspective on the model. A viewpoint is a mechanism designed to allow the modeler to view a model from a particular perspective. A given view conforms to a single viewpoint that identifies both how it should be constructed and its purpose. Views typically do not contain elements but instead import model elements in order to collect them into a common namespace for viewing via package and other diagrams.

5.11 Questions

1. What is the diagram kind that appears in the frame label of a package diagram?
2. Which kinds of model element can be represented by a package diagram?
3. What is the generic term for model elements that can be contained in packages?
4. Where does a model appear in a package hierarchy?
5. Name three potential organizing principles that might be used to construct the package hierarchy of a model.
6. How can you show on a package diagram that one package contains another?
7. Which rule does a package enforce for the named elements that are its members?
8. How can you tell by looking at a package diagram that a model element represented on the diagram is a member of the package that owns the diagram?

9. Write down the qualified name for a block B1 contained in a package P1, which in turn is contained in a model M1.
10. A package P1 contains three elements—block B1, block B2, and block B3—all with public visibility, and a package P4 with private visibility. Another package P2 contains a package called B1 and two blocks called B2 and B4. If package P2 imports package P1 with public visibility, list all the members of P2.
11. If an empty package P9 imports P2 with public visibility, list all the members of P9.
12. What is an alias used for?
13. Name three common kinds of dependency.
14. How are dependencies shown on a package diagram?
15. Name three properties of a viewpoint.
16. How do you represent a view V1, which conforms to a viewpoint VP1, on a package diagram?

Discussion Topic

For a model that you are trying to build, discuss the type of model organization that is appropriate for it.

Modeling Structure with Blocks

6

This chapter addresses modeling the structure of systems in terms of their hierarchy and interconnection. It describes blocks—the principle structural construct—and the two types of diagrams used to represent structure—the block definition diagram and the internal block diagram.

6.1 Overview

The block is the modular unit of structure in SysML that is used to define a type of system, system component, or item that flows through the system, as well as conceptual entities or logical abstractions. The block describes a set of uniquely identifiable instances that share the block's definition.

The block definition diagram is used to define block characteristics in terms of their structural and behavioral features, and the relationships between the blocks such as their hierarchical relationship. The internal block diagram is used to describe the internal structure of a block in terms of how its parts are interconnected.

Properties are the primary structural feature of blocks. Part properties describe the decomposition hierarchy of a block and provide a critical mechanism to define a part in the context of its whole. Value properties describe quantifiable physical, performance, and other characteristics of a block such as its weight or speed. Value properties are defined by value types that describe the valid range of values, along with its dimension (e.g., length) and its units (e.g., feet or meters). Value properties may be related using parametric constraints as discussed in Chapter 7.

Ports are structural features that describe the points at which a block interacts with other blocks and are used to connect the parts of a block. The two types of SysML ports are flow ports, which specify what can flow in and out of blocks, and standard ports, which specify the types of services that a block either requires or provides. An item flow describes what flows on the connectors between ports and parts.

The behaviors associated with a block defines how the block responds to a stimuli. The different behavioral formalisms, including activities, interactions, and state machines, are discussed in Chapters 8 through 10, respectively. The behavioral features of a block, which include operations and receptions, provide a mechanism for external stimuli to invoke these behaviors.

In addition to decomposition hierarchies, blocks can be organized into classification hierarchies that allow blocks to be defined in terms of their similarities and differences. Within a classification hierarchy, a block can specialize another more general block that allows it to inherit features from the general block and to add new features specific to it. Blocks in a classification hierarchy can also be used to describe a unique design configuration such as a system under test.

6.1.1 Block Definition Diagrams

The **block definition diagram** or “bdd” is used to define blocks in terms of their features, and their structural relationships with other blocks. The block definition diagram header is depicted as follows:

bdd [model element type] model element name [diagram name]

A block definition diagram can represent a package, a block, or a constraint block, as indicated by the *model element type* in square brackets. The *model element name* is the name of the package, block, or constraint block, and the *diagram name* is user defined and is often used to describe the purpose of the diagram.

Figure 6.1 shows an example block definition diagram containing the most common symbols. The diagram shows several levels of the composition hierarchy

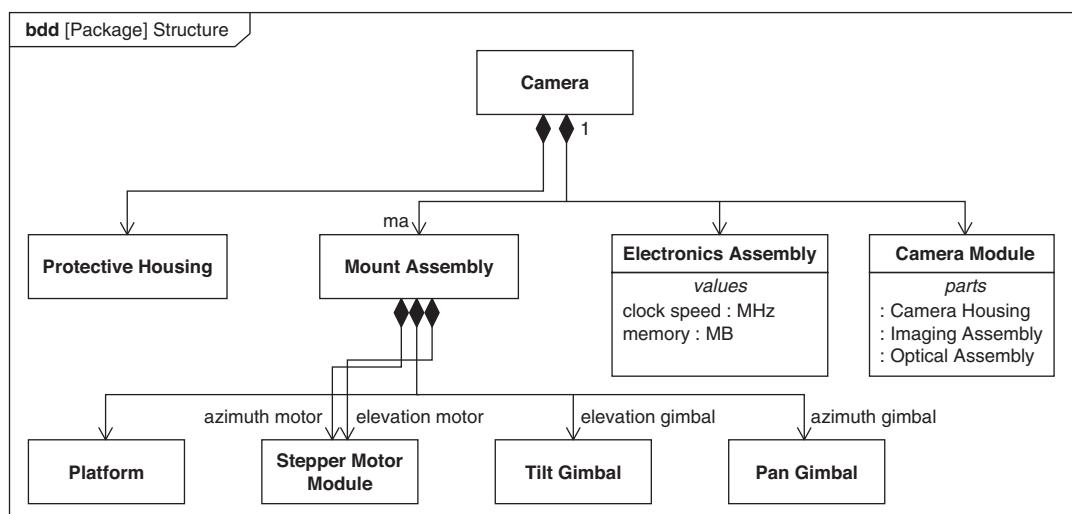


FIGURE 6.1

Example block definition diagram.

of an ACME camera. The notation used in the block definition diagram to describe blocks and their relationships is shown in the Appendix, Tables A.3 through A.5.

6.1.2 Internal Block Diagram

The **internal block diagram** or “ibd” resembles a traditional system block diagram and shows the connections between parts of a block. The internal block diagram header is depicted as follows:

ibd [Block] block name [diagram name]

The frame of an internal block diagram always represents a block, so the model element type is often elided in the diagram header. The *block name* is the name of the block that is designated by the frame, and the *diagram name* is user defined to represent a description of the diagram purpose.

Figure 6.2 shows an example internal block diagram containing some common symbols. The diagram describes part of the internal structure of the *Camera*, and how light flows in and through various intermediate parts to become video (MPEG4) output.

The notation used in the internal block diagram to describe the usage of blocks, called parts, and their interconnections is shown in the Appendix, Tables A.8 and A.9.

6.2 Modeling Blocks on a Block Definition Diagram

The **block** is the fundamental modular unit for describing system structure in SysML. It can define a type of a logical or conceptual entity, a physical entity (e.g., a system); a hardware, software, or data component; a person; a facility; an entity that flows through the system (e.g., water); or an entity in the natural environment (e.g., the atmosphere or ocean). Blocks are often used to describe reusable components that can be used in many different systems. The different categories of block features used to define the block are described later and are generally classified into structural features, behavioral features, and constraints.

A block is a type, that is, a description of a set of similar **instances**, or **objects**, each of which exhibits the features defined by it. An example of a block is an automobile that might have a set of features including its physical, performance, and other properties (e.g., its weight, speed, odometer reading), and vehicle registration number, as well as its behavioral features that define how it responds to stimuli. Each instance of the automobile block will include these features and be uniquely identified by the value of some of its properties. So, for example, a Honda CR-V might be modeled as a block, a particular Honda CR-V is an instance of a Honda CR-V with vehicle registration “A1F R3D” and an odometer reading “150,010” miles. An instance of a block can be modeled explicitly in SysML as a unique design configuration, as described in Section 6.6.5. An instance can also include value properties the values of which change over time, such as the speed and odometer reading.

The block symbol is notated as a rectangle that is segmented into a series of compartments. The name compartment appears at the top of the symbol and

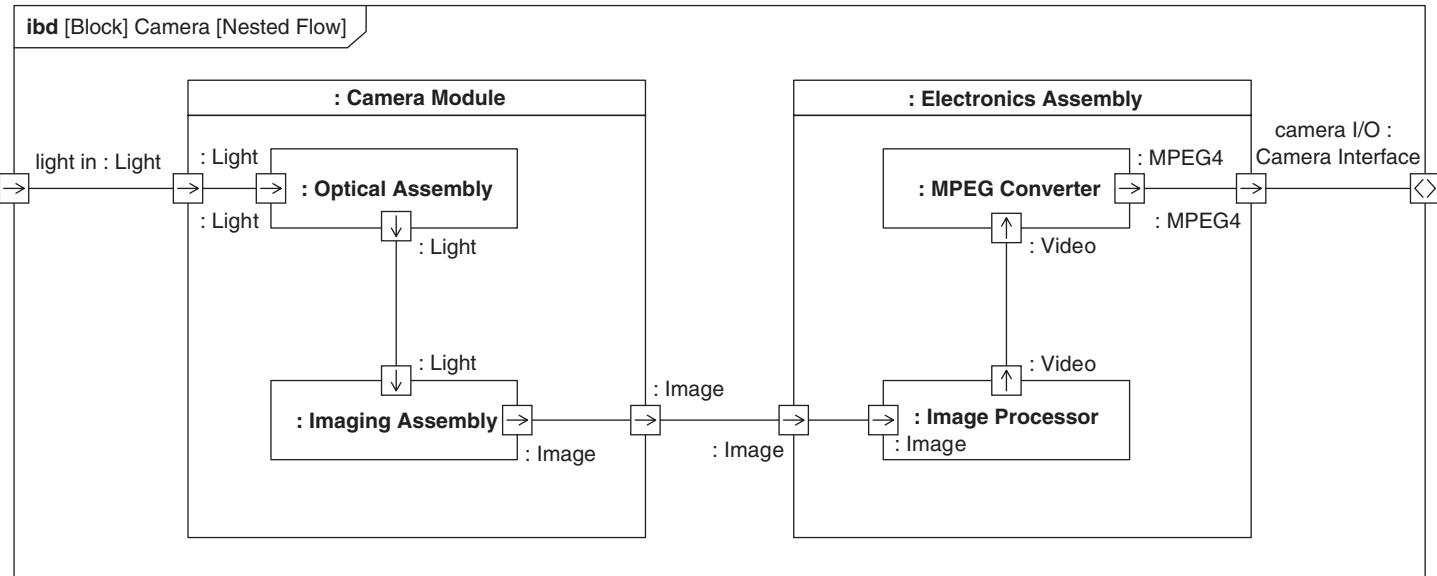
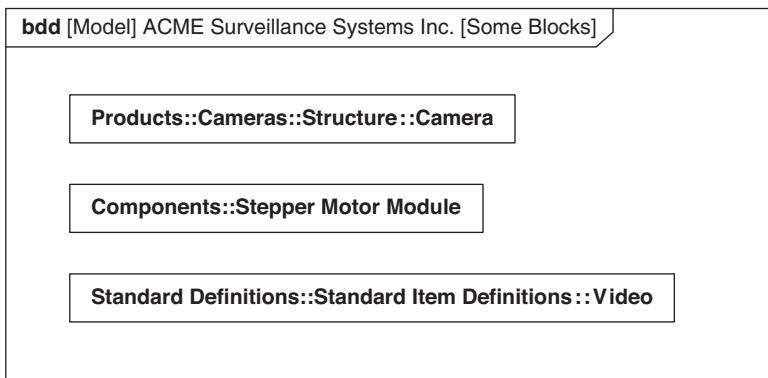


FIGURE 6.2

Example internal block diagram.

**FIGURE 6.3**

Blocks on a block definition diagram.

is the only mandatory compartment. Other categories of block features, such as parts, operations, and ports, can be represented in other compartments of the block symbol. All compartments, apart from the name compartment, have labels that indicate the category of feature they contain.

Names on block definition diagrams follow the same convention as on package diagrams. Model elements that are either directly contained in or imported into the namespace represented by the diagram are designated just by their names. Other model elements must be designated by their full qualified names in order to clearly identify their location in the model hierarchy.

A rectangular symbol on a block definition diagram is interpreted by default as representing a block, but the optional keyword «block» may be used, preceding the name in the name compartment, if desired. To reduce clutter, the convention used in this chapter is that the «block» keyword is only used where blocks appear on the same block definition diagram as other model elements represented by rectangles.

Figure 6.3 shows a block definition diagram that has three blocks in the company's corporate model, called *ACME Surveillance Systems Inc*. The names of the blocks are fully qualified with their path to show where they are located within the package hierarchy of the model. The blocks shown cover a range of uses: *Camera* is a description of an ACME product; *Stepper Motor Module* is an off-the-shelf component used in ACME's cameras; and *Video* is used to describe the video images that the cameras produce.

6.3 Modeling the Structure and Characteristics of Blocks Using Properties

Properties are one category of features of a block. They are used to capture the structural relationships and values of a block. A property has a type that may be

another block, or some more basic concept such as an integer value. This section describes the three categories of property and their uses.

- *Part properties* (parts for short) describe the decomposition of a block into its constituent elements. These are described in Section 6.3.1.
- *Reference properties* describe weaker relationships between blocks than the composition relationship represented by part properties. These are described in Section 6.3.2.
- *Value properties* describe the quantifiable characteristics of a block, such as its weight or velocity. These are described in Section 6.3.3.

Later sections address more advanced topics related to properties, including:

- *Property derivation* is described in the Derived Properties, subsection of Section 6.3.3.
- *Property redefinition and subsetting* is defined in Section 6.6.5.
- *Property ordering and uniqueness* is defined in Chapter 7, Section 7.3.

6.3.1 Modeling Block Composition Hierarchies Using Part Properties

Part properties, sometimes shortened to just **parts**, describe composition relationships between blocks. This type of hierarchical composition of blocks is often seen in a breakdown of equipment, or bill of materials. A composition relationship is also called a whole-part relationship, where a block represents the whole and the part property represents the part. A part property is always typed by a block.

A part property uniquely identifies the usage of its type in the context of another block. The key distinction between a part and an instance of a block is that the part describes an instance or instances of a block *in a particular context* of an instance of the composite block.

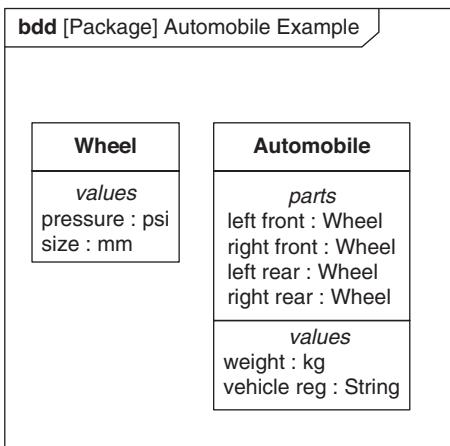
An instance of a whole may include multiple instances of a part property. The potential number of instances is specified by the multiplicity of the part property, which is defined as follows:

- A lower bound (minimum number of instances) that may be 0 or any positive integer. The term *optional* is often used for multiplicities where the lower bound is 0 because an instance of the whole can include zero instances of the part.
- An upper bound (maximum number of instances) that may be 1, many (denoted by “*”), or a specific positive number.

A part property is a feature of a block, and as such can be listed in a separate parts compartment within a block. The parts compartment is headed by the keyword **parts** and contains one entry for each part in the block. Each entry has the following format:

part name: block name [multiplicity]

The upper and lower bounds of a multiplicity are typically combined into one expression like this: “lower bound .. upper bound,” except where they have the

**FIGURE 6.4**

An automobile with four wheels described as separate parts.

same value, in which case just the upper bound is shown. Where no multiplicity is shown, a value of 1..1 is assumed.

Figure 6.4 shows a simple example of an automobile with four wheels, where each usage of *Wheel* is uniquely identified by a part property. In this case, the *Automobile* is the whole and the wheels are represented as parts. Each of the four wheels has a common block definition, *Wheel*, with certain characteristics (e.g., *size*, *pressure*, and so on), but each wheel can have a unique **usage** or **role** in the context of a particular automobile. The front wheels have a different role from the rear wheels and may have different values for their pressure. Each wheel may also behave differently when the car is accelerating and decelerating and be subject to different constraints. Similarly, the front wheels on a front wheel-drive vehicle may have a different role than front wheels on a rear wheel-drive vehicle.

A part property defines a set of instances that belong to an instance of the whole or composite block. If a block is at the part end of more than one composite block, the SysML semantics are that an instance of the block at the part end will be part of at most one block instance at the whole end at any point in time. An example is an engine that can be part of two different types of vehicle, such as an automobile and a truck. However, any given instance of engine can only be part of one vehicle instance at a time. This rule implies that at the instance level, the composition hierarchy is a strict tree, which means there can be no whole-part cycles leading from any given instance back to itself.

Typically, a whole-part relationship means that certain operations that apply to the whole also apply to each of the parts. For example, if a whole represents a physical object, a change in position of the whole could also change the position of each of its parts. A property of the whole, such as its mass, could also be implied by its parts.

A particular application domain may establish its own interpretation of the whole-part relationship for the blocks defined in a particular model. When

blocks represent components of physical systems, the whole-part relationships generally can be thought of as an assembly relationship, where an instance of the block on the whole end is made from instances of the block on the part end. The implications of whole-part relationships for software relate to creating and returning memory locations for computation. This may also apply to the definition of operations that apply to the parts and the whole. For software objects, a typical interpretation is that delete, copy, and move operations apply to all parts of a composite whole. The whole-part semantics specify that when an instance at the whole end is destroyed, the instances at the part end will also be destroyed.

Composite Associations

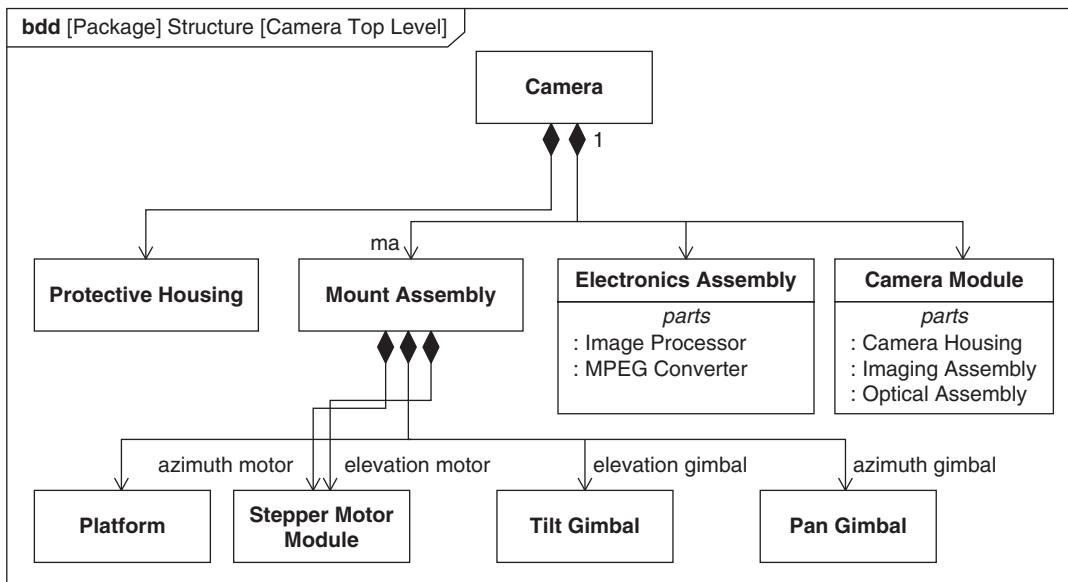
While a part property does state the number of instances that can be included in an instance of its whole, it does not state whether instances of the part must *always* exist as part of an instance of some whole. For example, an engine may physically exist on its own, as well as in an instance of an automobile or truck.

This information must be specified using a special relationship called a **composite association** that relates part and whole with a multiplicity at both its ends. The upper bound of the multiplicity at the whole end of a composite association is always 1 because an instance of a part property may only exist in one whole at any one time. The lower bound of the multiplicity at the whole end, however, may be 0 or 1. A value of 1 means that instances of the block at the part end must always be composed within instances of the block at the whole end. A value of 0 means that an instance of the block at the part end can exist if no whole exists. Specifying a lower bound of 0 enables a block to be part of more than one composite association. In this case, it is still mandated that an instance of a block at the part end is only part of a single instance at the whole end at any given time.

A composite association is shown as a line between two blocks with various adornments at its ends. The whole end of a composite association is adorned by a black diamond. A shorthand notation can be used to represent a block that has many composite associations by showing a single black diamond with a series of lines connecting to the part ends. The part end of an association is typically adorned with an arrowhead. The lack of an arrowhead indicates that the block at the part end has a reference property that can be used to reference the block at the whole end. Reference properties are described in the next section.

Each end of the association may optionally show a name, often called a role name, and a multiplicity. The term *role name* is used to convey the idea that the instances at the part end are playing some role (c.f., an actor) in the overall whole. When the multiplicity for an end is not shown, the default interpretation is a whole end multiplicity of 0..1 and a part end multiplicity of 1. The name of the part property can appear as the role name at the part end of the association although, often, part properties are not named.

A block can include a parts compartment that contains the part properties represented at the part end of an association, but typically on a given diagram the part property is shown either in a parts compartment or as an association end.

**FIGURE 6.5**

Showing a block composition hierarchy on a block definition diagram.

Figure 6.5 shows a portion of the top two levels of the composition hierarchy for a *Camera*. Any number of levels of decomposition can be shown on a single block definition diagram, although here only the decomposition for *Camera* and *Mount Assembly* is shown. The parts of *Electronics Assembly* and *Camera Module* are shown in compartments. Multiple levels of decomposition can be shown on a single diagram, but this can increase the clutter for even relatively simple systems. Note that even though the block definition diagram shows blocks, the diagram frame represents the package *Structure*, as indicated in the diagram header.

Different projects have different philosophies on which part properties should have names. In this chapter, except where stated, the following naming philosophy is used:

1. A role name at the part end of the association is provided to distinguish two part properties with the same type (block). An example of this is the use of role names for *Stepper Motor Module* to distinguish the two roles of *elevation motor* and *azimuth motor*.
2. A role name is provided when the name of the type does not adequately describe the role the part plays. Examples of this are the role names *elevation gimbal* and *azimuth gimbal*. The block names *Tilt Gimbal* and *Pan Gimbal* do not explicitly describe the plane in which the gimbals move in the *Camera* application.

3. A role name is not provided when the block name provides sufficient information on the role of the part. Examples of this are *Protective Housing*, *Camera Module*, and *Electronics Assembly*. This is often the case when a block has been explicitly created to represent this part. This should also apply to *Mount Assembly*, but a name was required to illustrate an additional notational form in Figure 6.8.

Where a part name exists it is used in the figure description, otherwise the block name is used.

The lack of multiplicity adornments on all part ends in this figure indicate that there is exactly one instance of each in the composition hierarchy of *Camera*. The single multiplicity adornments on the whole end indicate that the *Electronics Assembly*, *ma*, and the *Camera Module* are always part of a *Camera*, whereas the block *Protective Housing* may be used in other blocks. All the parts of *ma* are typed by reusable blocks that have uses in many other contexts. *Camera Module* is shown with a parts compartment that lists its three part properties. None of them has a name, and they all have the default multiplicity of 1.

Modeling Parts on an Internal Block Diagram

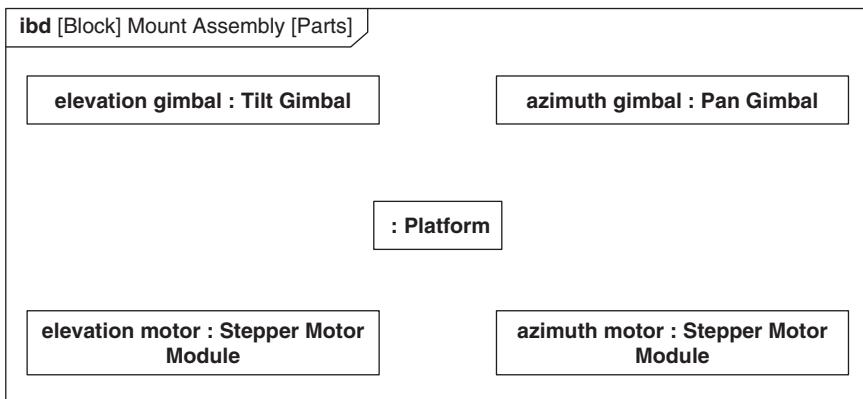
In addition to appearing on a block definition diagram, part properties can be shown on another diagram called the internal block diagram that presents a different visualization of block composition. The internal block diagram enables parts to be connected to one another using connectors and ports as described later.

The relationship between composition, as expressed on a block definition diagram and on an internal block diagram, is as follows:

- The whole end or composite (block) is designated by the diagram frame on the internal block diagram with the block name in the diagram header. It provides the context for all the diagram elements on the diagram.
- Each name on the part end of a composite association whose whole end is the context block, or an entry in the parts compartment of the context block, appears as a box symbol with a solid boundary within the frame of the internal block diagram. The name string of the box symbol is composed of the part name followed by a colon followed by the type of the part. Either the part name or the type name can be elided.

The multiplicity of each part property may be shown in the top right corner of the part symbol or in square brackets after the type name. If no multiplicity is shown, then a multiplicity of 1 is assumed.

Figure 6.6 is an internal block diagram derived from the composite associations whose whole end is *Mount Assembly*. The diagram header identifies the *Mount Assembly* as the enclosing block that provides the context for the five parts shown in the diagram. In this case, the multiplicities are not shown, indicating that the multiplicity is the default value of 1. (See Figure 6.28 for an example of nondefault multiplicity.) Note that this is a simplified form of internal block diagram for illustration only. A modeler would seldom if ever build an internal block diagram without connectors between the parts or other structural information present.

**FIGURE 6.6**

An internal block diagram for the *Mount Assembly*.

Connecting Parts on an Internal Block Diagram

An internal block diagram can be used to show connections between the parts of a block, something that cannot be shown in a block definition diagram. A **connector** is used to bind two parts and provides the opportunity for those parts to interact, although the connector says nothing about the nature of the interaction. Connectors can also connect ports, as described later in the *Connecting Flow Ports on an Internal Block Diagram* section. The interaction between the parts of a block is specified by the behavior of the parts, as described in the Chapters 8, 9, and 10 about behavior.

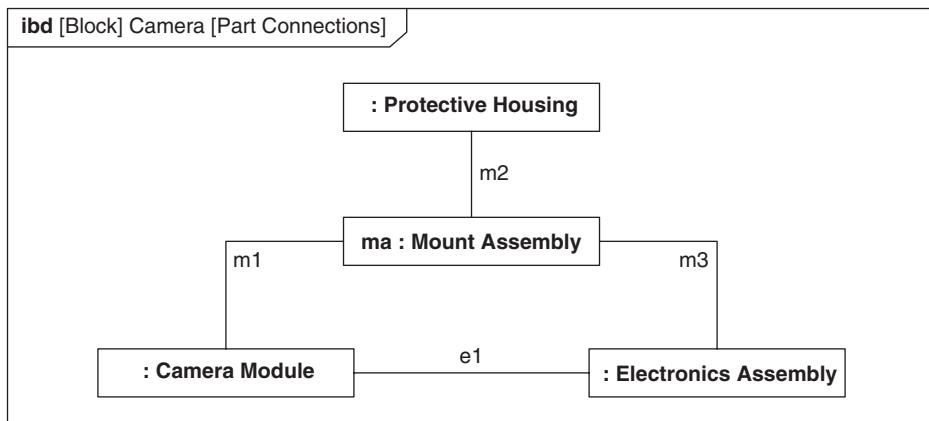
This interaction may include the flow of inputs and outputs between parts, the invocation of services on parts, the sending and receiving of messages between parts, or constraints between properties of the parts on either end. Where appropriate, the nature and direction of items flowing on a connector can be shown using item flows, as described in Section 6.4.2.

The ends of a connector can include multiplicities that describe the relative number of instances that can be connected by **links** described by the connector. (Note: A link connects instances whereas a connector connects parts.) A connector may be typed by an association that allows further definition of the characteristics of the connection. Where a connector is typed, its type is normally a reference association, although composite associations may be used.

On an internal block diagram, the connector between two parts is depicted as a line connecting two part symbols. A part can connect to multiple other parts, but a separate connector is required for each connection. The full form of the connector name string is as follows:

connector name: association name

The ends of a connector can include an arrowhead, which means that the association that typed the connector had the equivalent adornment. The ends of the

**FIGURE 6.7**

Connecting parts on an internal block diagram.

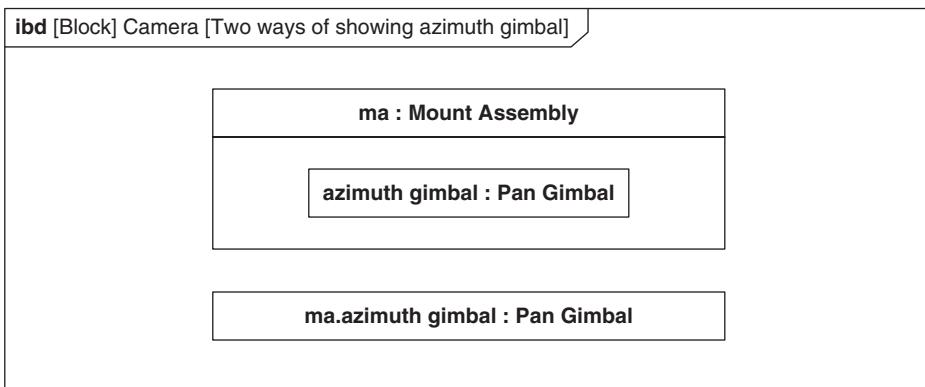
connector can be adorned with the name and multiplicity of the connector ends. If no multiplicity is shown, then a multiplicity of 1 is assumed. When connectors cross one another, the intersection can be designated by a contour to indicate that they are not related in any way.

The internal block diagram for the *Camera* is shown in Figure 6.7. The *Protective Housing* that protects the camera internals is mechanically connected to the *Mount Assembly* (*ma*). The *Mount Assembly* provides the platform for the *Camera Module* and *Electronics Assembly*, which are connected to pass electrical signals that allow the camera to function. The connectors in this example have names, indicating that they are mechanically connected (*m1* to *m3*) or electrically connected (*e1*), but the names have no semantic implications. Meaningful keywords can be added using a domain-specific profile as described in Chapter 14. All connectors have default multiplicity implying one-to-one connections.

Modeling Deeply Nested Structures and Connectors

Sometimes it is necessary to show multiple levels of nested parts within a system hierarchy on an internal block diagram. The nested parts can be represented by showing part symbols within part symbols, as shown in Figure 6.8. SysML also introduces an alternative notation to designate a nested part, also shown in the figure, where each level of nesting of the part is separated by a period (i.e., dot) within the name string of a single part symbol. The symbol's name string, with dot notation, represents the path from the level of the context block for the diagram down to the nested part.

Figure 6.8 shows two ways to display the nested part *azimuth gimbal* within an internal block diagram whose context is the *Camera*. The *azimuth gimbal* can be represented as a nested rectangle within the *ma:Mount Assembly* symbol. It can also be represented using the dot notation with the higher-level part name, *ma*, and a dot preceding the part name, *azimuth gimbal*.

**FIGURE 6.8**

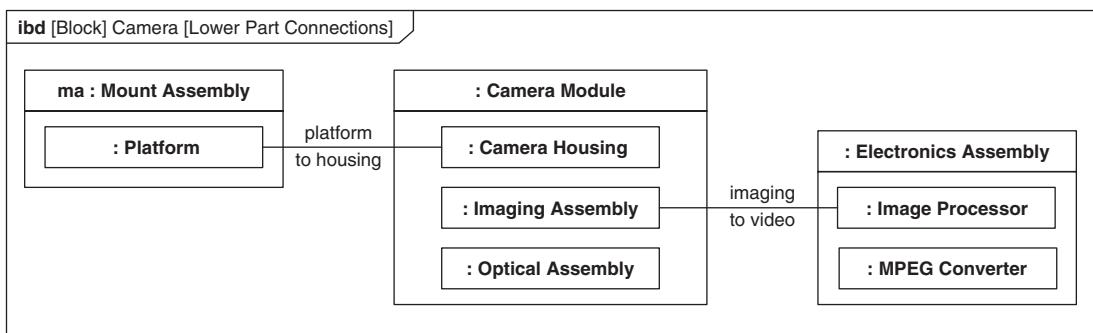
Showing deep-nested parts on an internal block diagram.

Connectors can connect parts at different levels of nesting without directly connecting to the intermediate levels of nested parts. For example, a tire can be connected directly to a road without having to connect the road to the vehicle, the vehicle to the suspension, the suspension to the wheel, and the wheel to the tire with intermediate connectors at each level of nesting. The connector simply crosses the nested part boundaries in order to directly connect the tire to the road. It is often the case that connections are initially specified between top-level parts, and then as the internal details of the parts become known, connectors are specified between lower-level elements. It is a modeling choice as to whether the outer connectors are removed or kept. Blocks have a special Boolean property called *encapsulated*, which if true prohibits connectors from crossing boundaries without connecting to any intermediate nested parts.

Connectors with nested ends are shown in the same way as normal connectors except that they cross the boundaries of part symbols. The encapsulated property on a block is shown if true and not shown if false. Where shown, it appears in the name compartment in braces before the block name.

Figure 6.9 includes a more detailed look at the connections within the subassemblies in Figure 6.7. After further investigation, connector *m1* has been augmented with a nested connector, called *platform to housing*, that connects the *Platform* of *ma* to the *Camera Housing* in the *Camera Module*. The electrical connector, *e1*, has been augmented with a nested connector, called *imaging to video*, that connects the *Imaging Assembly* of the *Camera Module* to the *Image Processor* in the *Electronics Assembly*.

When a connector at one level of the structure is used to add more detail about a connector at some higher level, there are potential issues with maintaining the resulting model. For example, if the *m1* connector from Figure 6.7 is removed from the model, should *platform to housing* be removed as well? If this type of relationship is important, then an association block can be used to show decomposition of the connector in a similar way that blocks show the decomposition of parts. Association blocks are described in the next section.

**FIGURE 6.9**

Nested connectors on an internal block diagram.

6.3.2 Modeling Noncomposite Relationships between Blocks Using Reference Properties

Reference properties, sometimes shortened to just **references**, indicate that there is some relationship between the instances of the block that owns the reference property and the instances of the block that type the reference property. The composition semantics of whole-part relationships, as described by part properties, define a specific relationship between an instance of the block at the whole end and an instance of the block at the part end, as described in the previous section. An example of this is destruction semantics, where destroying an instance of the block at the whole end also destroys the instances of the blocks at the part ends. For reference properties, the destruction semantics associated with composition do not apply. There is also no constraint on the number of blocks that can have reference properties that reference the same instance. This is a particularly important point that provides significant utility as described next.

Reference properties can be used to describe a logical hierarchy that references blocks that are part of other composition hierarchies. Reference properties can thus be used to cut across the tree structure of a composition hierarchy, which allows additional views besides the primary system whole-part hierarchy. This logical hierarchical organization can be represented on both the block definition diagram and internal block diagram. Allocations, discussed in Chapter 13, can be used to establish the relationship between the reference property in a logical hierarchy and the corresponding part in a composition hierarchy. Another use of reference properties is to model stored items (e.g., water stored in a tank). The water is not part of the tank in the same way that a valve is a part of the tank. For this case, the water may be owned by another block and shown as a reference property of the tank.

Reference Associations

The composite association was discussed earlier in this chapter to represent a hierarchy of blocks. **Reference associations** are used on a block definition diagram

to capture a different relationship between blocks, where the block on one end of the association is referenced by the block on the other end. A reference association can specify a reference property on the blocks at one or both ends.

A reference association is represented as a line between two blocks. The black diamond that represents a composite association is not used. When there is a reference property on only one end, the line has an open arrowhead pointing toward the type of the reference property and away from the owner of the reference property. If the association is bidirectional (i.e., has reference properties at both ends), then there are no arrowheads. Multiplicities on the ends of reference associations have the same form as for composite associations.

One end of a reference association may be represented by a white diamond. SysML assigns the same meaning to the association whether the white diamond is present or not. However, the white diamond symbol is intended to be used with an applied stereotype that may specify unique semantics for a particular domain.

Like part properties, reference properties can be listed in a separate compartment within a block. The references compartment is headed by the keyword *references* and contains one entry for each reference property in the block, with the same presentation as part properties.

Figure 6.10 shows a block called *Mechanical Power Subsystem* that uses reference associations to reference the power supply of the *Camera*, its powered mechanical components, including the motors in the various assemblies, and the *Distribution Harness*. The *Distribution Harness* itself has references to other harnesses that are part of the different assemblies in the *Camera*. In the composition hierarchy for the *Camera*, the components are part of a number of different assemblies, some of which are shown in Figure 6.5. The *Mechanical Power Subsystem* represents a logical aggregation of these components that interact to provide

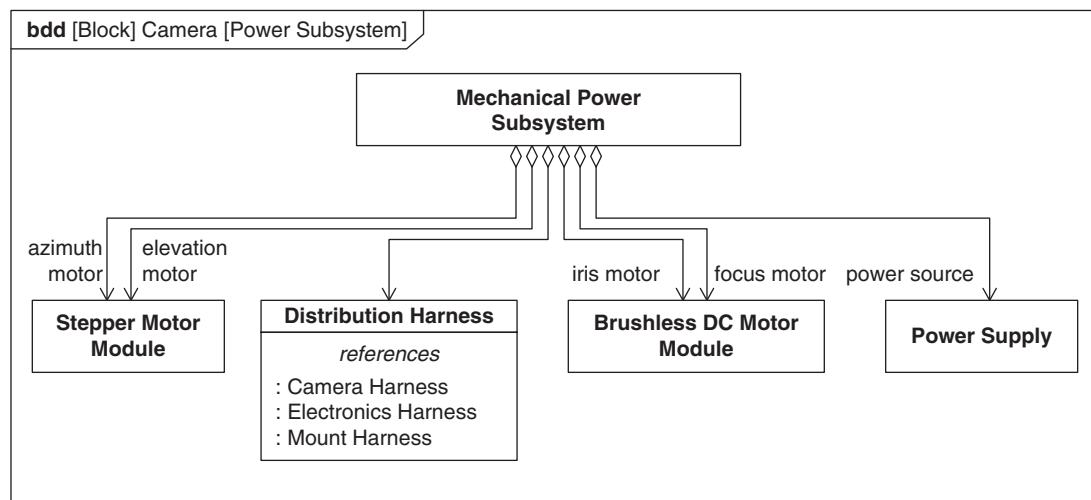


FIGURE 6.10

A reference association on a block definition diagram.

power to the rest of the camera. The white diamond adornment is used in this example to emphasize the hierarchical nature of the *Mechanical Power Subsystem*, but this emphasis is strictly notational and has no semantic implications.

Different model-based methods may include a block such as the *Mechanical Power Subsystem* in different parts of the model structure. Here it is contained in the *Camera* block itself, but it could just as easily have been placed in a special package of similar subsystems (refer to Chapter 16 for another example). An instance of *Mechanical Power Subsystem* would not show up in the equipment tree for the *Camera*, but is more like a cross-cutting view of a portion of the equipment tree.

Modeling Reference Properties on Internal Block Diagrams

Reference properties are depicted in a similar fashion to parts when shown on the internal block diagram, except that their box symbol has a dashed instead of solid boundary. Otherwise they have similar adornments and can be connected in the same way as any part symbol.

Figure 6.11 shows the connections between the reference properties in the *Mechanical Power Subsystem* used to support power transfer within the subsystem. In this case a single *power source* provides all the power needs of the mechanical parts of the *Camera*, through the *Distribution Harness*.

Using Associations to Define Common Features of Connectors

A connector can be typed by an **association** to define its characteristics in more detail. An association is defined between two blocks. For the connector to be typed by an association, the connected parts or references on the connector ends must have the same types as the association ends. An association defines the multiplicity of block instances on each of its ends. Although connectors may have their own multiplicities, their lower and upper bounds are constrained to be within the multiplicity defined for the ends of the association that types it.

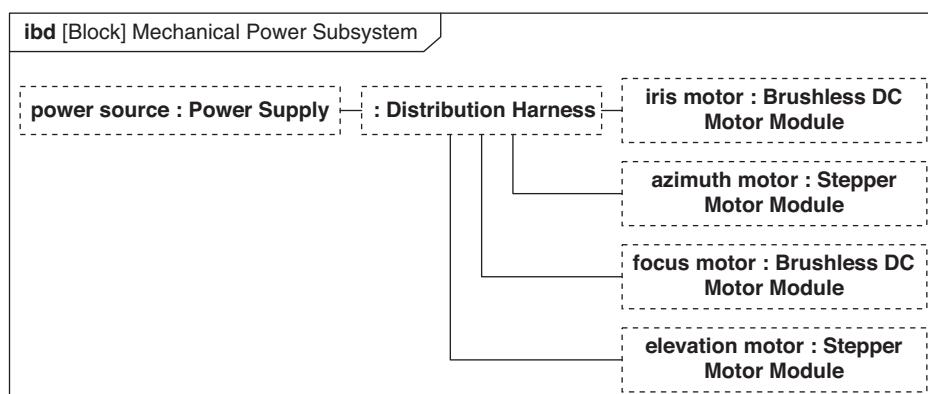


FIGURE 6.11

Reference properties and their interconnections on an internal block diagram.

An Asynchronous Digital Subscriber Line (ADSL) connection is used to connect a *Surveillance System* and *Command Center*, as shown by the association *ADSL Connection* in Figure 6.12. The ends of *ADSL Connection* are named *adsl dte* and *adsl dce*, indicating the respective roles of the related blocks. A *Surveillance System* is a data terminator and thus has higher download than upload and must be related, via its reference property *adsl dce*, to exactly one *Command Center*. A *Command Center* is related, via its reference property *adsl dte*, to zero or more *Surveillance Systems*.

Figure 6.13 shows the part of the *ACME Surveillance Network* that deals with residential users. It has a *residential center* connected to two residences, *residence A* and *residence B*. The two connectors, *ADSL A* and *ADSL B*, are typed by the *ADSL Connection* and so must conform to its defined multiplicities, which they do.

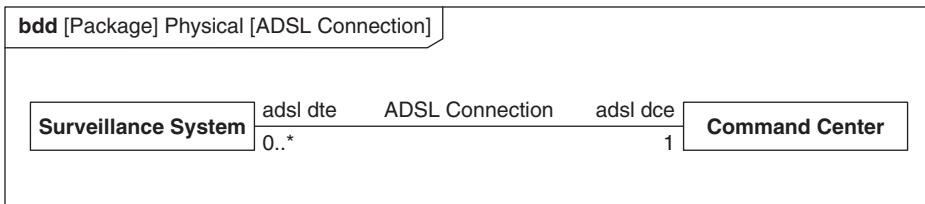


FIGURE 6.12

A reference association between two blocks.

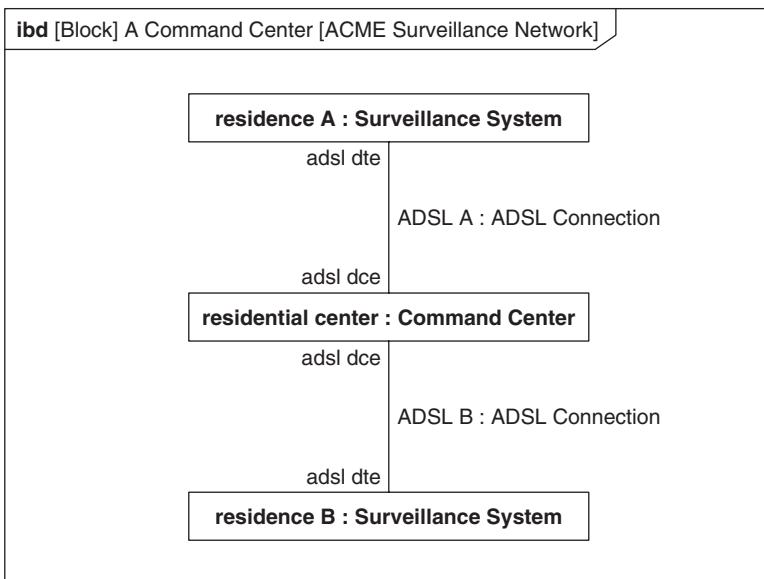


FIGURE 6.13

ADSL connection in use.

Association Blocks

More detail can be specified for connectors by typing them with **association blocks**. An association block, as the name implies, is a combination of an association and a block, so it can relate two blocks together but can also have internal structure and other features of its own.

Association blocks are shown on block definition diagrams as an association path with a block symbol attached to it via a dashed line. The name of the association block is shown in the block symbol rather than on the association path.

Figure 6.14 shows a refinement to Figure 6.12 where *ADSL Connection* is now an association block and is joined by another association block, *SDSL Connection*. The figure also shows additional internal structure inside *Surveillance System* and *Command Center*, an *ADSL Modem* and an *ADSL Gateway*, respectively. These new parts are used to handle the ADSL communication between them, which is specified in *ADSL Connection*, as shown in Figure 6.15. *SDSL Connection* represents the use of a Synchronous Digital Subscriber Line (SDSL) between *Command Centers*, but the parts required to support SDSL are not shown.

The internals of an association block are specified in an internal block diagram using the notation described earlier in this chapter but with one addition. The ends of the association block are represented by participant properties, shown on the diagram using a dashed box, like a reference property, but distinguished from other properties by the keyword «participant». They may also show an end property in braces indicating the association end that the participant property represents.

Figure 6.15 shows the internal detail of the *ADSL Connection* association block. Its two participant properties—*adsl dte* and *adsl dce*—are shown using the «participant» keyword. In this case the end property is not shown because the participant properties have the same names as the association's ends. The nested parts of *adsl dte* and *adsl dce* are shown in order to describe how an *ADSL Connection* is achieved, in this case via a connector, called *adsl link*, between an *ADSL Modem* and an *ADSL Gateway*. It is now implicit that every connector

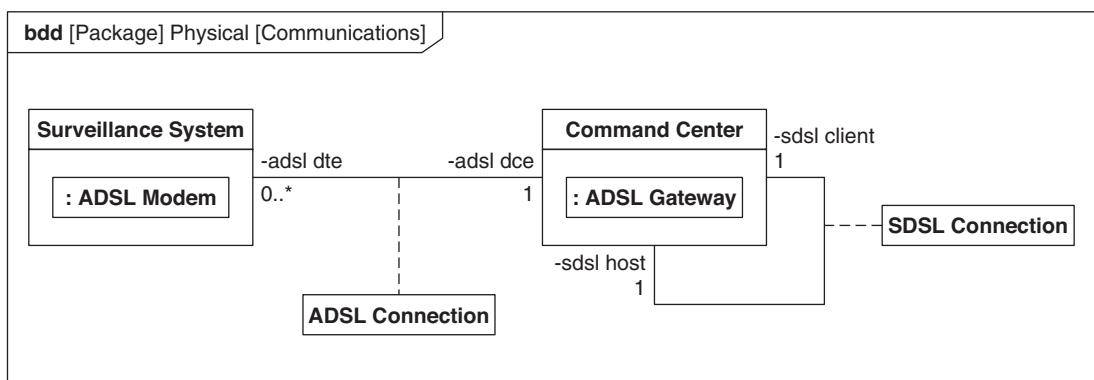
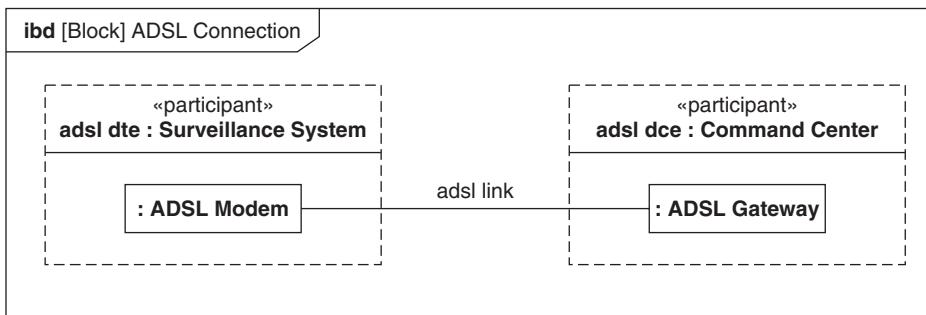


FIGURE 6.14

Using association blocks to relate blocks.

**FIGURE 6.15**

The internal structure of an association block.

typed by *ADSL Connection* ensures that the *ADSL Modem* of its *adsl dte* and the *ADSL Gateway* of its *adsl dce* are connected via a connector called *adsl link*. Note that the connector *adsl link* is not typed and so there is no additional detail on the link's nature. If further internal detail, such as the nature of the physical details of the ADSL connection, were required, an association could have been added.

Figure 6.16 shows both the *ADSL Connection* and *SDSL Connection* in use. This ACME Surveillance System has two command centers, one for corporate clients and the other for residential clients. The command centers communicate through an *SDSL Connection* and to their clients through *ADSL Connections*.

6.3.3 Modeling Quantifiable Characteristics of Blocks Using Value Properties

Value properties are used to model the quantitative characteristics, such as its weight or speed, associated with a block. They can also be used to model vector quantities such as position or velocity. Whereas the definition of a part or reference property is based on a block, the definition of a value property is based on a value type that specifies the range of valid values the property can take when describing an instance of its owning block. SysML defines the concepts of unit and dimension that can be used to further characterize a value type, although value types do not need to have dimensions or units. Value properties can have initial values associated with them, and they can also define a probability distribution for their values.

Modeling Value Types on a Block Definition Diagram

Value types are used to describe the values for quantities. For example, a value properties called *total weight* and *component weight* might be typed by a value type called *kilograms* (kg). The intent of the value type is to provide a uniform definition of a quantity that can be shared by all value properties. Value type definitions can be reused by typing multiple value properties with the same value type.

A value type describes the data structure for representing a quantity and specifies its allowable set of values. This is especially important when relying on computers to operate on the values to perform various computations. A value type can be based on the fundamental types that SysML provides—*Integer*, *String*,

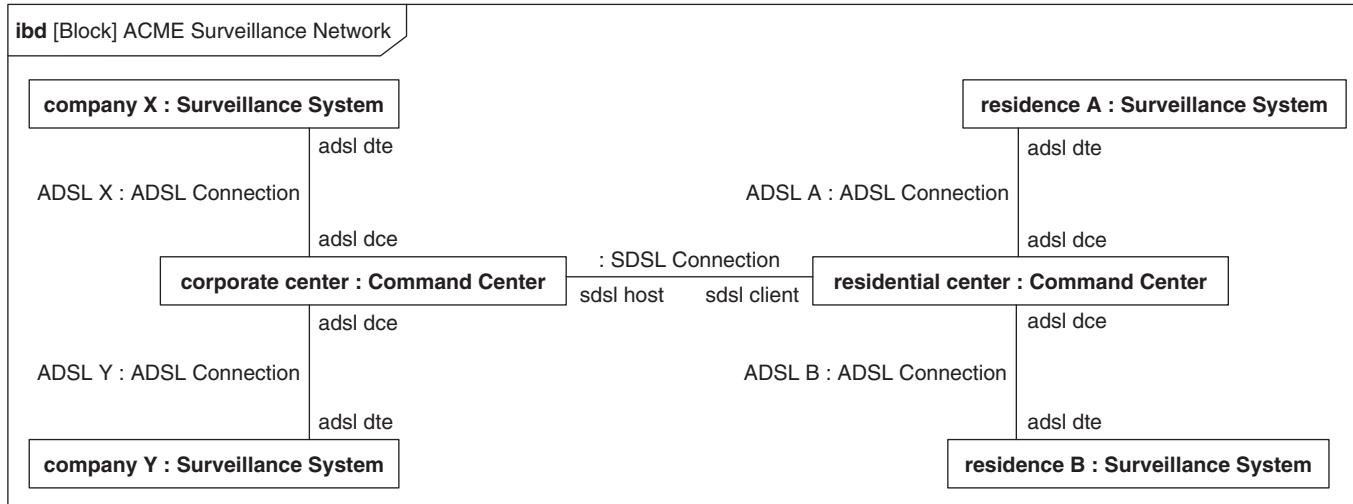


FIGURE 6.16

Example of an ACME surveillance network with two command centers.

Boolean, *Real*, or *Complex*—or it can be defined more generally. The following are the different categories of value type:

- A primitive type supports the definition of scalar values. *Integer*, *String*, *Boolean*, and *Real* are primitive types.
- An enumeration defines a set of named values called literals. Examples of enumerations are colors and days of the week.
- A structured type represents a specification of a data structure that includes more than one data element, each of which is represented by a value property. *Complex* is a structured type provided by SysML.

A value type can take any of these basic forms but extends the definition of a quantifiable characteristic to include units and dimensions. What they all have in common is that they represent values, not entities, and so unlike blocks they have no concept of identity. Two instances of a value type are deemed to be identical if they have the same values.

Value types are represented on a block definition diagram by a box symbol with a solid boundary. The name compartment of a value type has the keyword «valueType» preceding its name. The symbol representing an enumeration has a single compartment listing all the literals of the enumeration and the keyword «enumeration» preceding its name in the name compartment. The symbol representing a structured type also has a single compartment labeled *values* that lists the subelements of the data type, using the same compartment notation as shown for other properties.

Figure 6.17 shows some value types in the *Basic Definitions* package. *Size* is a structured type, with three subelements: *width*, *height*, and *length*; they are typed

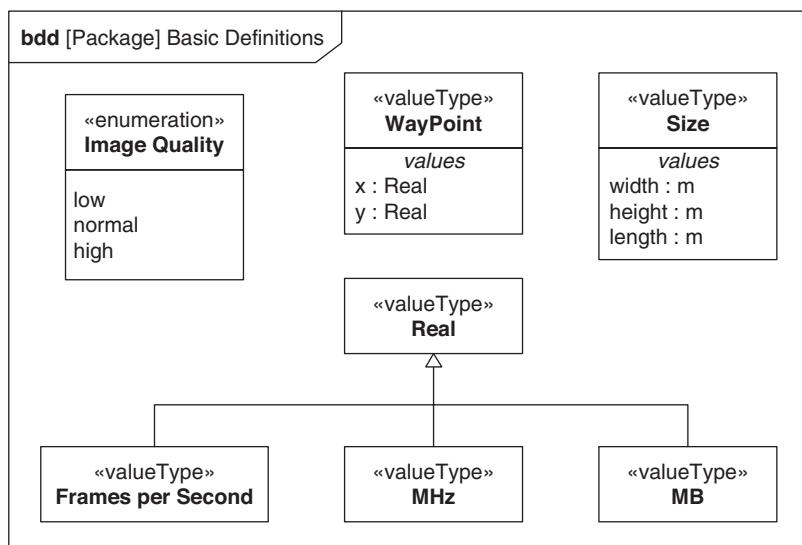


FIGURE 6.17

Definition of basic types in a block definition diagram.

by another value type *m* (for meters). The definition of *m* includes its unit and dimension and is shown later in Figure 6.19. *Image Quality* is an enumeration used to specify the quality of image captured by the camera, which can be used to control how much data are required to capture each video frame. The other data types are all real numbers, so specialize the SysML value type *Real*. In this case the specialization is simply stating that the values for *MHz*, *MB*, and *Frames per Second* are real numbers. See Section 6.6 for further discussion on the meaning and notation for specialization.

Adding Units and Dimensions to Value Types

SysML defines the concepts of unit and dimension to enable their use as shareable definitions that can be used consistently across a model, or captured in a model library that can be reused across a set of models. A **dimension** identifies a physical quantity such as length, whose value may be stated in terms of defined units (e.g., meters or feet). A **unit** must always be related to a dimension, but a dimension need not have any associated units, and often equations can be expressed in terms of quantities that include dimensions without specifying units.

A value type that represents a physical quantity may reference a dimension and/or unit as part of its definition, and thus assign units and dimensions to any value property that it types.

The SI Standard for Units and Dimensions

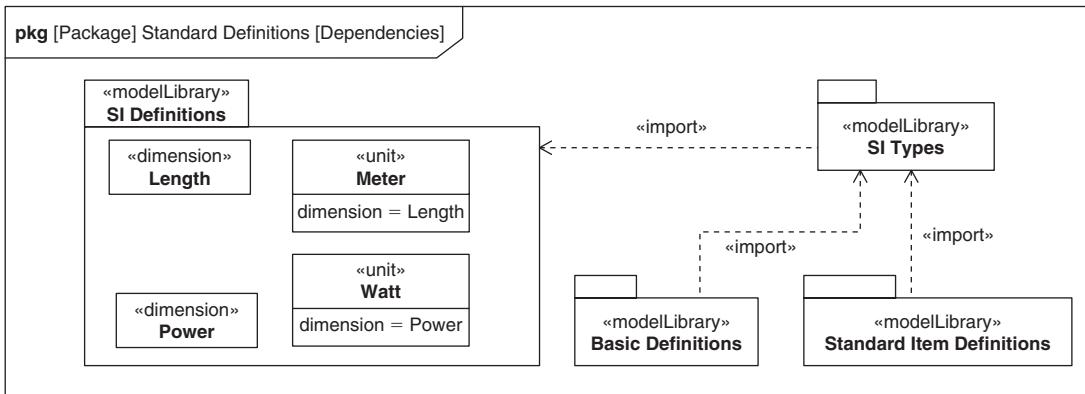
The International System of Units (**SI**) is a standard for units and dimensions published by the International Standards Organization (ISO). The complete set of SI dimensions and units are described in a model library in Annex C of the OMG SysML Specification. This model library can be imported into any model to allow the SI definitions to be used as is, or to use them as the basis for defining more specialized units and dimensions. Although this model library is not formally a part of the SysML specification, it is anticipated that many SysML modeling tools will include this library and possible extensions.

Figure 6.18 shows some of the definitions in the *SI Definitions* model library of SysML. Although SysML provides descriptions of the SI units and dimensions, it does not define standard value types because value types are often customized for the application based on the needs of data representation and accuracy. *SI Types* is a locally defined model library that imports *SI Definitions* in order to define a set of SI value types for this application based on the SI units and dimensions.

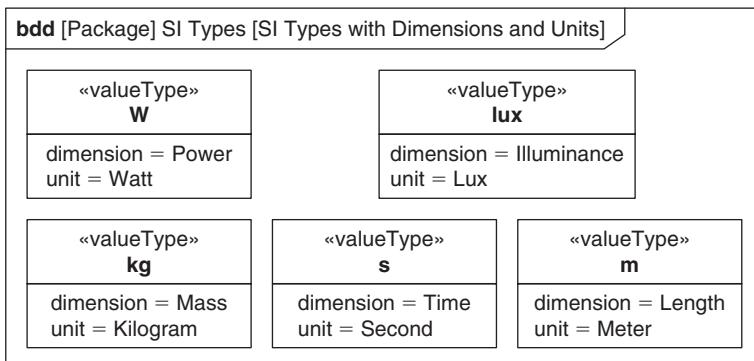
Some of the types in the *SI Types* model library are shown in Figure 6.19, using unit and dimension definitions imported from the SysML *SI Definitions* package. This enables a consistent representation of quantities that can be checked for compatibility of dimensions and consistency of units. Although not shown here, all the value types in this figure are defined to be real numbers.

Adding Value Properties to Blocks

Once a set of value types have been defined, they can be used to type the value properties of blocks. Value properties have the same features as other properties

**FIGURE 6.18**

Importing the SI definitions defined by SysML.

**FIGURE 6.19**

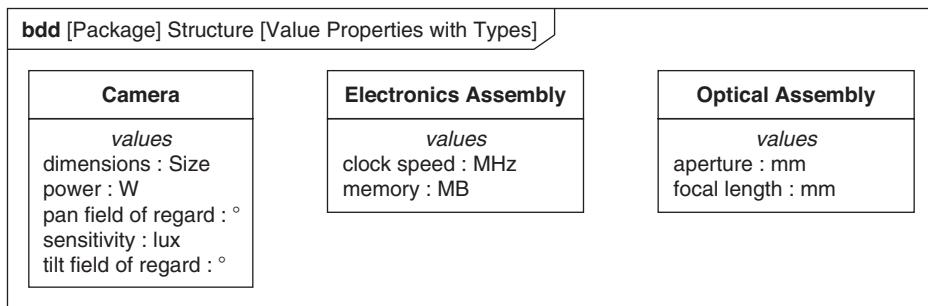
Using dimensions and units in the definition of value types.

such as multiplicity, and like other properties, are shown in a compartment of their owning block. The values compartment has the label *values*.

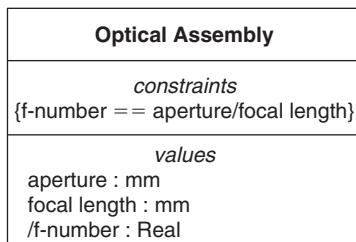
Figure 6.20 shows a block definition diagram containing three blocks with value properties: *Camera*, *Electronics Assembly*, and *Optical Assembly*. Some of the value properties are typed with the value types specified in Figure 6.17, such as the *clock speed* and *memory* of *Electronics Assembly*. Others are typed with value types shown in Figure 6.19. For example, the *sensitivity* of the *Camera* is typed by *lux*, which measures luminosity. The names of value types are not limited to alphanumeric characters. For example, *pan field of regard* in *Camera* is typed by the symbol “°”, which stands for degrees.

Derived Properties

Properties can be specified as derived, which means that their values are derived from other values. In software systems, a **derived property** is typically calculated by the software in the system. In physical systems, a property is typically marked as

**FIGURE 6.20**

Use of a value type to type a value property on an internal block diagram.

**FIGURE 6.21**

Example of derived property.

derived to indicate that the values of derived properties are calculated based on analysis or simulation, and may well be subject to constraints as described in Chapter 7. By definition, constraints express noncausal relationships between properties, but derived properties can be interpreted as dependent variables, and thus allow the equations expressed in constraints to be treated as mathematical functions.

A derived property is indicated by placing a forward slash (/) in front of the property name.

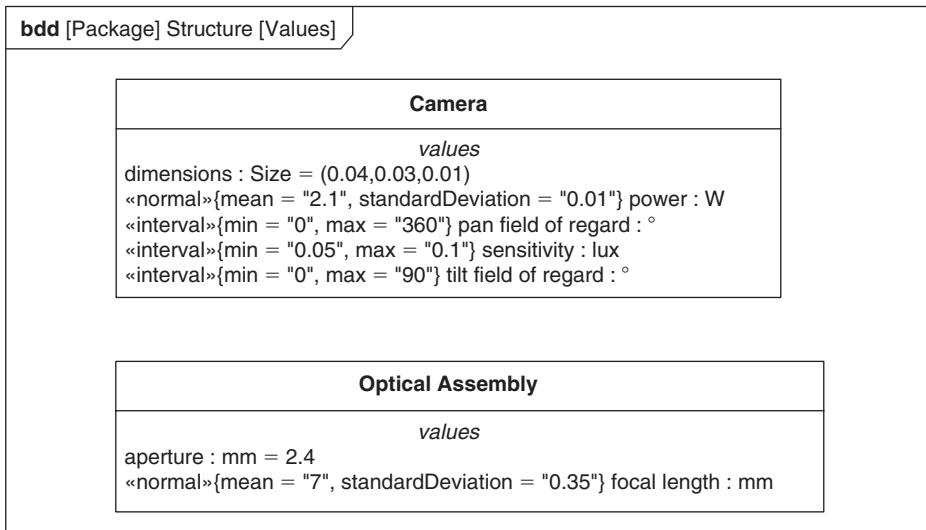
Figure 6.21 shows *Optical Assembly* with an additional property, *f-number*, that is marked as derived. It also shows a constraint between *focal length*, *aperture*, and *f-number* that can be used, given *focal length* and *aperture*, to calculate the value of *f-number*.

Modeling Property Values and Distributions

An **initial** can be assigned to a property. Initial values for value properties can be specified as part of their property string in an initial values compartment, for a block using the following syntax:

property name: type name=initial value

The initial values for a part can be specified using a dedicated compartment labeled *initialValues*. Quantities may need to be represented by a **probability**

**FIGURE 6.22**

Examples of property values and distributions.

distribution rather than a single value. SysML allows a modeler to describe the probability distribution of the range of values for a value property. Annex C of the OMG SysML specification defines a small set of commonly used distributions in a model library that can be reused. The following notation is used to represent a distributed property:

«distributionName» {p1=value, p2=value ...}property name:type name

The properties *p1*, *p2*, and so on are properties that characterize the probability distribution. For example, this may be a *mean* and *standard deviation* for a normal distribution, or a *min* and *max* value for a uniform distribution.

Figure 6.22 shows a number of distributed properties, including *Camera.pan field of regard* and *Optical Assembly:focal length*. *Camera.pan field of regard* is the size of the arc that the camera can cover while panning. It is defined as an interval distribution with a minimum of 0° and a maximum of 360° because the actual field of regard will depend on where the camera is installed. The focal length of the *Optical Assembly* is defined as a normal distribution with a mean of 7 millimeters and a standard deviation of 0.7 millimeters. This is intended to accommodate differences arising from the combination of minor deviations in the placement of lenses and mirrors during manufacturing.

The distributions of both *pan field of regard* and *focal length* are distributions over the whole population of cameras and optical assemblies. The *Camera.dimensions* and *Optical Assembly:aperture* have initial values, a simple scalar value for *aperture*, and a value for each of the constituent value properties of *dimensions*.

6.4 Modeling Interfaces Using Ports and Flows

A **port** represents an interaction point on the boundary of a block and on the boundary of any part typed by that block. The port enables the behavior of a block or part to be accessed. A block may have many ports that specify different interaction points. Even though ports are defined on blocks, ports can only be connected to one another by connectors on an internal block diagram to support the interaction between parts.

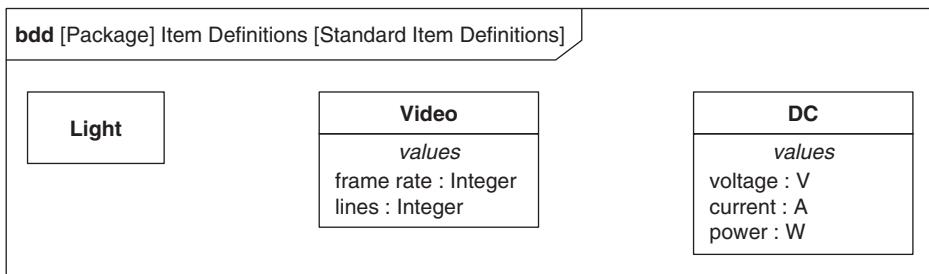
There are two kinds of ports in SysML to specify two different types of interaction. The first kind is a flow port that specifies what can flow in or out of the block at the interaction point. The flow port may represent physical flow; for example, a water pump might have a port that specifies that water can flow in and water can flow out. The pump might have another flow port that specifies that electrical power can flow in. Often, in electronic systems, flow ports describe the flow of information and/or control, such as a signal that a radar system has detected a target, or that a button has been pressed on a keyboard. Flow ports are described in Section 6.4.3.

The second kind of port is a standard port that specifies the services required or provided by the block at this interaction point. Standard ports are often used either as high-level descriptions of system capabilities or to describe the interface with software-intensive systems, such as command and control. An example of a standard port might be one that provides access to a target-tracking subsystem, where a client can request information on current targets and threats, or perhaps historical data on recent levels of activity. Typically, a single standard port describes a set of features related to some specific service, such as tracking or navigation, but the allocation of the services offered by a block to its ports is a methodological question. Standard ports are described in Section 6.5.3.

The selection of which port type to use is a methodological question that often relates to how the behavior is expressed. In general, flow ports are well suited for representing continuous flows of physical entities, or representing other continuous or discrete flows sent from one process to another. Standard ports are better suited for a system whose behavior is described by the invocation of services and are commonly used to represent interfaces between software components. A combination of flow ports and standard ports can be used in any given model, but standard ports cannot be directly connected to flow ports or vice versa.

6.4.1 Modeling Items that Flow

An **item** is used to describe a type of entity that flows through a system; it may be a physical flow, which includes matter and energy, as well as a flow of information. Items may be blocks, value types, or signals. Physical items may be modeled as blocks, which typically include value properties that describe physical quantities of the item, such as the water temperature for a block that represents water. A flow may also be simplified to represent just a quantifiable property (e.g., water temperature) in which case the item can be represented as a value type instead

**FIGURE 6.23**

Items that flow in the *Camera* system.

of a block. An item may have a complex internal structure, such as an automobile that flows through an assembly line.

The flow of information can be represented by signals. Signals may also be used to control the behavior of a part that waits for the signal to be received.

Items can be defined at different levels of abstraction and may be refined throughout the design process. For example, an alert flowing from a security system to an operator may be represented as a signal at a high level of abstraction. However, in exploring the nature of how that alert is communicated in detail, the item may be redefined. If the alert is communicated as an audio alarm, for example, it may be redefined as a block representing the amplitude and frequency of the sound.

Figure 6.23 shows part of the *Standard Item Definitions* model library that covers the items that flow in cameras. The items are modeled as blocks and contain value properties that describe their characteristics.

6.4.2 Modeling Flows between Internal Block Diagram Parts

An **item flow** is used to specify the items that flow across a connector in a particular context. An item flow specifies the type of item flowing and the direction of the flow. It may also be associated to a property, called an **item property**, of the enclosing block to identify a specific usage of an item in the context of the enclosing block. For example, water may flow in one port and out of another. The type is the same for both; the property is different to correspond to different usages. Item properties can be constrained in parametric equations, as described in Chapter 7. A connector can have more than one item flow attached to it, either flowing in the same or different directions.

Item flows are represented as black-filled arrowheads on a connector where the direction of the arrowhead indicates the direction of flow. All the item flows in a given direction are shown in a comma-separated list of item flow descriptions floating near the arrow for the appropriate flow direction. Each item flow description has a type name, and if related to an item property, includes the property name as well.

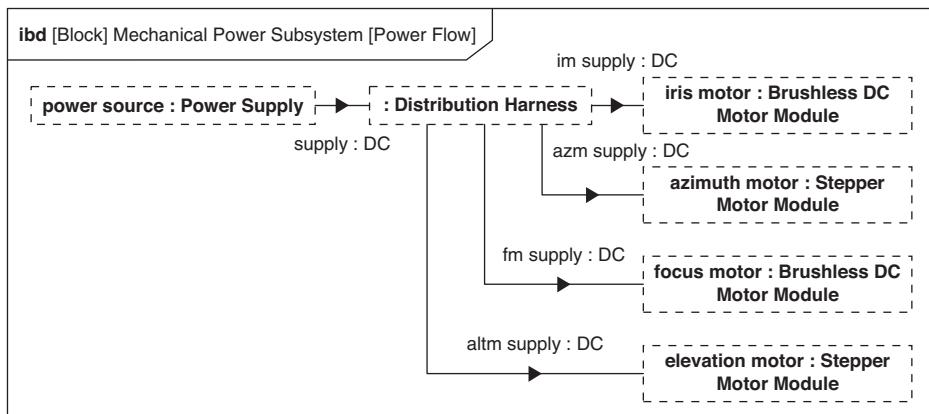


FIGURE 6.24

Item flows on an internal block diagram.

Figure 6.24 shows the flow of electricity (represented by the block *DC*) through *Mechanical Power Subsystem*. The overall flow, as one might expect, is from *power source* through the *Distribution Harness* to the various motors. In this case, each item flow is represented by a corresponding item property owned by the *Mechanical Power Subsystem*.

6.4.3 Modeling Flow-Based Interfaces Using Flow Ports

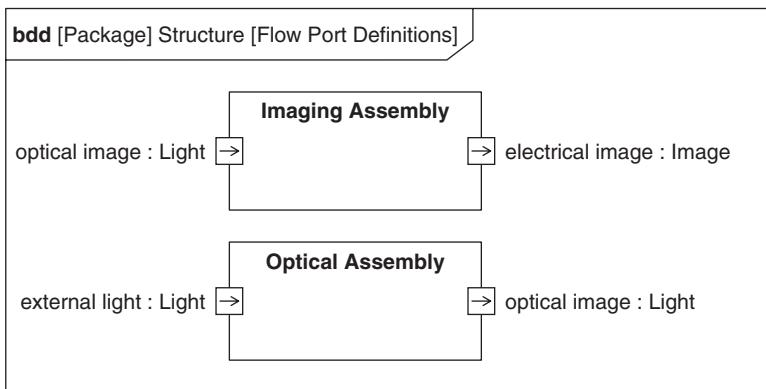
A **flow port** is used to describe an interaction point for items flowing in or out of a block. It specifies what input items can be received by the block and what output items can be sent by the block. Like other structural features of a block, a port can have a multiplicity that indicates how many instances of the port are present on an instance of its owning block.

Flow ports specify what can flow. What actually does flow in a particular context, as represented by item flows, may be different. Item flows on connectors between ports must be compatible with the port definitions but may be more specific. For example, a flow port on a pump may be typed by fluid, but in a given context, the specific fluid that flows through the port may be typed by water.

Atomic Flow Ports

A port that specifies only a single type of input or output flow is modeled as an **atomic flow port**. An atomic flow port specifies the flow direction (in, out, or inout). An atomic flow port is typed by the item that can flow in and/or out of the block, which may be a block, value type, or signal. Examples include a block (e.g., water), a value type (e.g., current in units of amperes), or a signal (e.g., an alert message).

Atomic flow ports are shown as small squares on the boundary of the block with an arrow inside the symbol representing the direction of the port. A two-headed

**FIGURE 6.25**

A block with atomic flow parts.

arrow represents a direction of inout. The name, type, and multiplicity of the port are shown in a string floating near the port in the form:

port name: item name[multiplicity]

Alternatively, the port strings can be included in a separate *flow ports* compartment, using the syntax:

direction port name: item name[multiplicity]

On the block definition diagram in Figure 6.25, the atomic flow ports on *Optical Assembly* specify that it can accept *Light* as an input and produces a focused optical image, which is still *Light*, as an output. The atomic flow ports on *Imaging Assembly* specify that it accepts an *optical image* and produces an *electrical image* for further processing.

Nonatomic Flow Ports

Where an interaction point has a complex interface with many items flowing, the port is modeled as a **nonatomic flow port**. In this case a flow specification must type the port. A **flow specification** is defined on a block definition diagram. The flow specification includes flow properties that correspond to individual specifications of input and/or output flow. Each **flow property** has a type and a direction (in, out, or inout). Like an atomic flow port, the type of the flow property can be a block, value type, or signal depending on the specification of what can flow.

When two blocks interact, they may exchange similar items but in opposite directions. Rather than creating two separate flow specifications for the nonatomic flow ports on the interacting blocks, SysML provides a mechanism called a **conjugate port** to reuse a single flow specification for both ports. One port is set to be the conjugate of the other, which indicates that the direction of all flow properties in the flow specification is reversed with respect to this port.

A flow specification is shown as a box symbol with the «flowSpecification» keyword above the name in the name compartment. The flow properties of a flow specification are shown in a special compartment labeled *flowProperties*, with each flow property shown in the format:

direction property name: item name.

A nonatomic flow port is indicated by two angle brackets facing each other (<>) drawn inside the port symbol. A conjugate flow port is indicated by inverting the fill and line color in the symbol. Nonatomic flow ports can be listed in the flow ports compartment of their owning block, although they do not have a direction. If a noncomposite flow port is «conjugated», then its entry ends with the keyword conjugated in braces.

In Figure 6.26, data are communicated from the *Monitoring Station* to one or more *Cameras*. A single flow specification, *Camera Interface*, describes the data that can be passed between the communication ports—*camera I/O* and *station I/O*. Video flows out and commands in. Given that the two blocks are intended to be connected, they share the same flow specification for their ports. The *station I/O* port on the *Monitoring Station* is conjugated with the dark shading to indicate that flow specification is reversed from the *camera I/O* port on the *Camera*; that is, the video flows in and the commands flow out.

Connecting Flow Ports on an Internal Block Diagram

When a block has ports, the parts that are typed by this block also feature these ports, which can then be connected on an internal block diagram using connectors. When an output port, or nonatomic flow port with an output flow property,

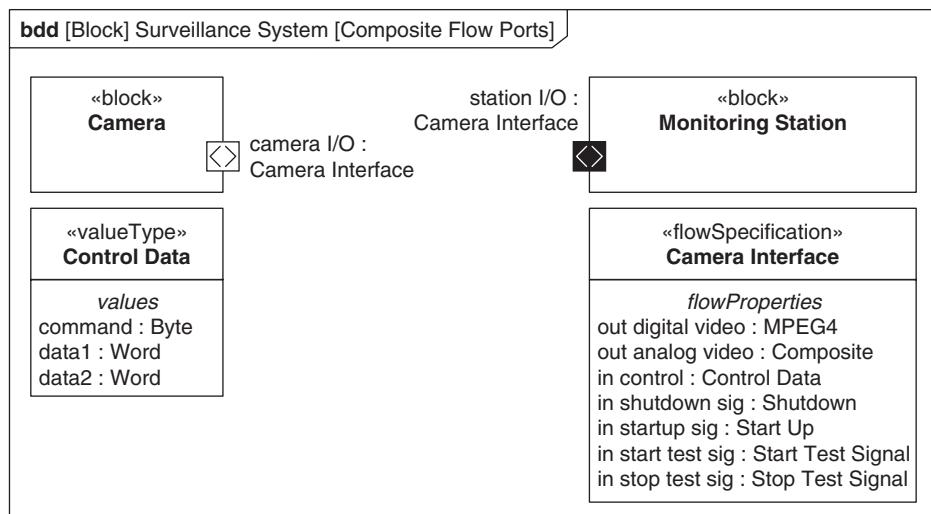


FIGURE 6.26

A block with a nonatomic flow port.

is connected to more than one other port, the items sent from that port are broadcast along all connectors.

When attempting to connect two ports, their compatibility needs to be assessed. Whether ports are compatible or not depends on both the kind of port (flow ports can only be connected to flow ports and standard ports can only be connected to standard ports) and more significantly on the compatibility of the specification of the port types.

The simplest form of compatibility between two connected flow ports is where:

- *Atomic flow ports* have matching types and the direction of one port is in and the other out, or both are inout.
- *N nonatomic flow ports* have matching flow specifications and one flow port is the conjugate of the other.

As discussed in Section 6.6, the type of a port may be specialized, perhaps to add new features. For the purposes of compatibility, a port with a given type will always be compatible with a port that specializes that type.

Atomic flow ports may be connected to nonatomic flow ports as long as the flow specification that types the nonatomic flow port contains a flow property of a matching type and direction. In Figure 6.27, two atomic flow ports on the *Camera Module* and *Electronics Assembly* of the *Camera* are connected to show that images can flow out of the *Camera Module* and into the *Electronics Assembly*. The connection is valid since both ports have a compatible type, *Image*, and they have compatible directions.

In Figure 6.28, two parts of the *4-Camera Surveillance System—Monitoring Station* and the set of *cameras*—are connected to enable communications

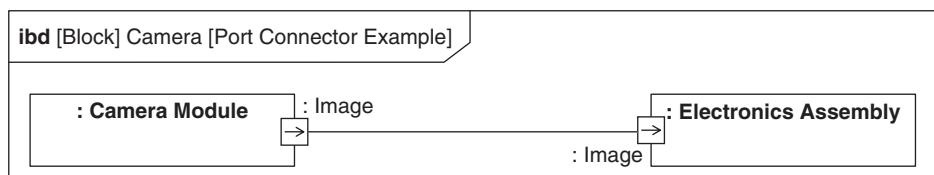


FIGURE 6.27

Connecting ports on an internal block diagram.

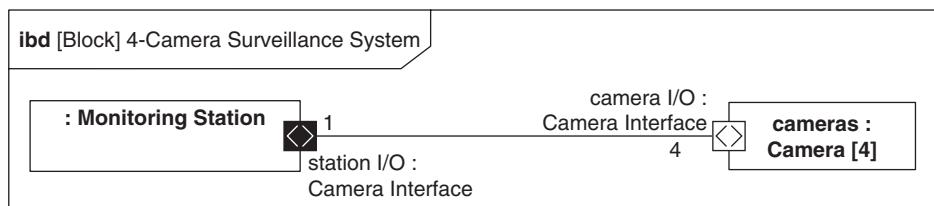


FIGURE 6.28

A connector that connects two nonatomic flow ports with the same specification.

between them. Ports on both parts use the flow specification *Camera Interface* to define their interface. The port on the *Monitoring Station* is conjugated, and the port on the *cameras* is not, so they are compatible and can be connected. The multiplicities on the ends indicate how their instances can be connected in a *4-Camera Surveillance System*. A camera instance must be connected to exactly one monitoring station and a monitoring station must be connected to exactly four cameras. Given that the *Camera* part has multiplicity 4 and the *Monitoring Station* part has a multiplicity of 1, the instance of *Monitoring Station* is connected to all four instances of *cameras*.

Delegating Responsibility between Ports

There are two cases to consider for how a block handles the interactions taking place at its ports. Either it handles the interactions directly itself, or it delegates the handling to a part or parts. If a block handles the port interactions itself, then the port is called a **behavior port**. Where a flow port is a behavior port, the flowing items must be relayed either to/from some feature of its owning block, or a parameter of the block's main (or classifier) behavior. The mechanism used to specify the mapping is not stated in SysML, allowing modelers in different domains and situations to take different approaches. See Section 6.5.1 for a description of the classifier behavior for a block.

The other case is when the block delegates the behavior to its parts. In this case the block presents a port on its interface but it delegates responsibility for handling the interaction at that port to a nested part or parts, via ports on their interfaces. This type of port is called a **delegation port**.

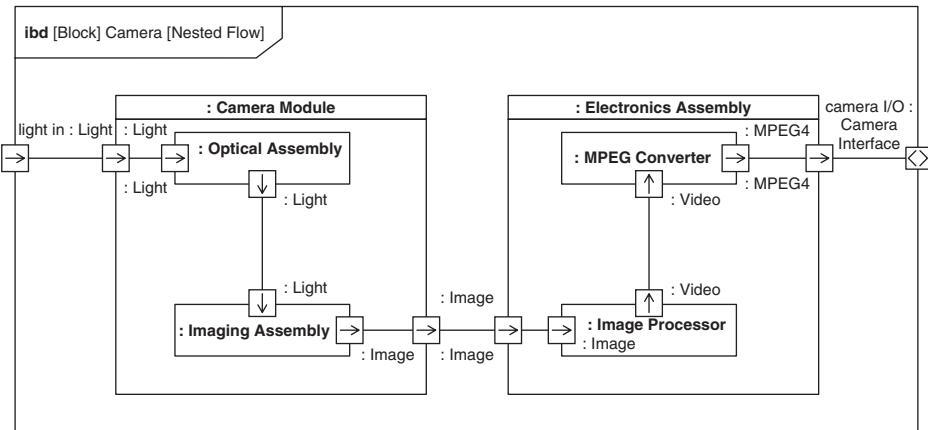
The delegation is carried by a connector from a delegation port on the parent to a port on one of its parts. To distinguish the two cases, a connector from a parent to a part is called a **delegation connector** and a connector between parts is called an **assembly connector**. Delegation connectors are similar in most ways to assembly connectors except that the compatibility rules are different.

In the case of delegation connectors, the specifications of the connected ports, instead of being complementary, must be similar, so:

- For an atomic flow port, the types of ports must be compatible and their directions match.
- For nonatomic flow ports, they must be typed by the same flow specification and either both must be conjugate or both not be conjugate.

The compatibility rules in the presence of specialization of port types are the same as for assembly connectors.

Ports shown on the diagram frame of the internal block diagram represent the ports on the enclosing block that is designated by the diagram frame as shown in Figure 6.29. SysML at present does not provide a notation to distinguish behavior ports, although this information is stored in the model repository. Depending on circumstances, it may be useful to add information in the diagram description about whether a given part on an internal block diagram delegates port interactions to its own internal structure.

**FIGURE 6.29**

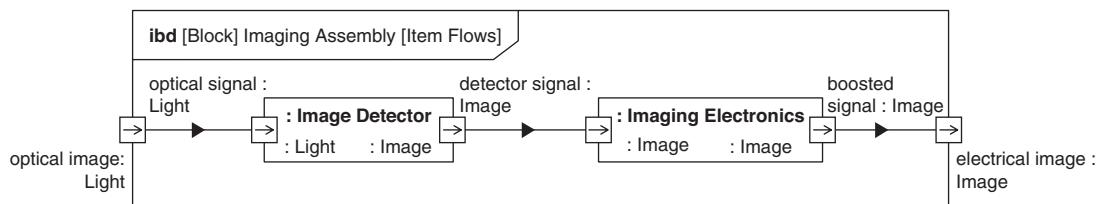
Examples of delegation ports.

In Figure 6.29, the conversion of light by the *Camera Module* into an electrical signal that represents an image is actually performed by two of its parts, the *Optical Assembly* that focuses the light and the *Imaging Assembly* that scans the result and converts it to an analog signal representing the image. The *MPEG4* atomic output port of the *Electronics Assembly* is connected to *camera I/O* on *Camera* that is a composite. However, this connector is valid because *Camera Interface* contains an output flow property typed by *MPEG4*. Note that the names of the internal flow ports have been elided to reduce clutter on the diagram.

Modeling Flows between Ports

As noted earlier, item flows can be shown on connectors between parts. Item flows are also shown on port-to-port connectors. SysML includes compatibility rules to confirm that the item flow is compatible with the type of the ports on either side of the connector. For an item flow to be valid in this context, it must be attached to a connector between two flow ports that support types compatible with the type of the flowing item. Specifically, the type of item that flows must be compatible with the type of connected atomic flow ports, or be compatible with the type of one of the flow properties in the flow specification that types the connected nonatomic flow ports.

Figure 6.30 shows the two parts of *Imaging Assembly*, the *Image Detector* that creates an analog electrical image from the incoming light and the *Imaging Electronics* that amplifies the electrical image so that it can be read by the image processor in the *Electronics Assembly*. There are three item flows shown in the figure, representing the flow of light into the *Image Detector*, the flow of electrical signals from the *Image Detector* to the *Imaging Electronics*, and the flow of boosted electrical signals from the *Imaging Electronics*. All the item flows are associated to an item property and they are all compatible with their associated connector. For example, the type of *detector signal* is *Image*, which is compatible

**FIGURE 6.30**

A port-to-port connector with item flows.

with the two ports at either end of its associated connector, and it is flowing in a compatible direction, from an out port to an in port.

6.5 Modeling Block Behavior

Blocks provide the context for behaviors, which is a SysML term covering any and all descriptions of how the block deals with inputs and outputs and changes its internal state. The two ways to specify the behavior of a block are referred to as the main behavior (called classifier behavior) and by methods that provide the specification of how service requests are handled. These in turn may invoke other behaviors within the block. All behaviors have parameters that are used to pass items into or out of the behavior before, after, or sometimes during execution.

As Chapters 8 through 10 describe, there are three main behavioral formalisms in SysML: activities, state machines, and interactions:

- Activities transform inputs to outputs, including matter, energy, and information.
- State machines are used to describe how the block responds to events.
- Interactions describe how the parts of a block interact with each other using message passing.

SysML recognizes two other forms of behavior that are not formalized within the language. An opaque behavior is represented as a textual expression and defines the language in which the expression is written. A function behavior is similar to an opaque behavior with the added restriction that it is not allowed to directly affect the state of its owning block and may only communicate using parameters. Function behaviors are often used to define mathematical functions.

6.5.1 Modeling the Main Behavior of a Block

The **main behavior** (sometimes called **classifier behavior**) of a block starts at the beginning of the block's lifetime and generally terminates at the end of its lifetime. Depending on the nature of the block, the choice of formalism for the classifier behavior is between state machines, if the block is largely event-driven (e.g., part of a service-based architecture), and activities, if the block is largely used to transform input items to output items. A popular hybrid approach is to use a state machine to describe the states of a block and to specify an activity that executes when a block is in a given state.

When a block has a classifier behavior and also has parts with classifier behaviors, the modeler should ensure that the behaviors between the whole and the parts are consistent at each system hierarchy level. A classifier behavior may act as a controller that plays an active role in coordinating the behaviors of its parts. In this case, the behavior of the block is a combination of its classifier behavior and the classifier behavior of all its parts. Another approach is for the classifier behavior of the block to be some alternative abstraction of the behavior of its parts. In this case, part-level behavior is a refinement of the block's classifier behavior.

6.5.2 Specifying the Behavioral Features of Blocks

Along with structural features, blocks can also own **behavioral features** that describe which requests a block can respond to. There are two types of behavioral features, operations and receptions. A **reception** represents an asynchronous request; that is, where the requester does not wait for a response. Each reception is associated with a **signal** that defines a message with a set of attributes that represent the content of the message. Receptions in different blocks can respond to the same signal type, so frequently used messages can be defined once and reused in many blocks. The attributes of the signal in turn define the set of arguments passed in with the asynchronous request.

An **operation** is a behavioral feature that typically represents a synchronous request; that is, where the requester waits for a response. Operations may also be asynchronous but then are largely equivalent to receptions. Operations define a set of **parameters** that describe the arguments passed in with the request, or are passed out once a request has been handled, or both.

A behavioral feature may have an associated method that is a behavior invoked when the block handles a request for the feature. Behavioral features are discussed later in this section and in more detail in the activity, interaction, and state machine chapters—Chapters 8 through 10, respectively.

Operations are shown in a separate compartment labeled *operations* and are described by their signature, a combination of their name along with parameters, and optional return type as follows:

operation name (parameter list):return type

The parameter list is comma-separated with the format:

direction parameter name: parameter type

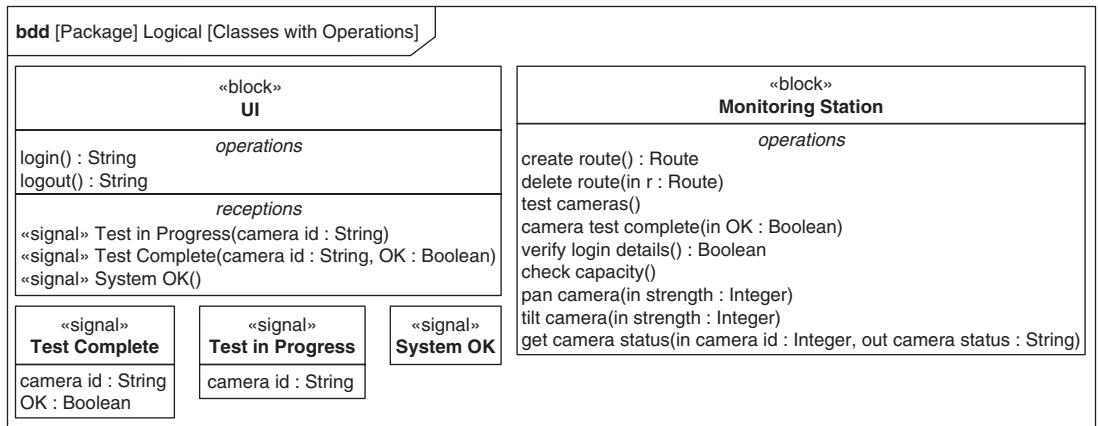
Parameter direction may be in, out, or inout.

Receptions are shown in another compartment labeled *receptions* and are described by their signature of name and list of attributes as follows:

«signal» reception name (attribute list)

The attribute list is comma-separated with the format:

attribute name: attribute type

**FIGURE 6.31**

Showing block operations on a block diagram.

Signals are defined using a box symbol with a solid outline and the keyword «signal» before the signal name. A signal symbol has a single unlabeled compartment that contains the attributes with the form:

attribute name: attribute type [multiplicity]

Figure 6.31 shows operations and receptions for the blocks *UI* and *Monitoring Station*. *UI* has operations to support user login and a set of receptions so that it can be informed of the progress of camera tests. The diagram also shows the definition of the signals related to the receptions. *Monitoring Station* has a set of operations that support route management, user management, and test management.

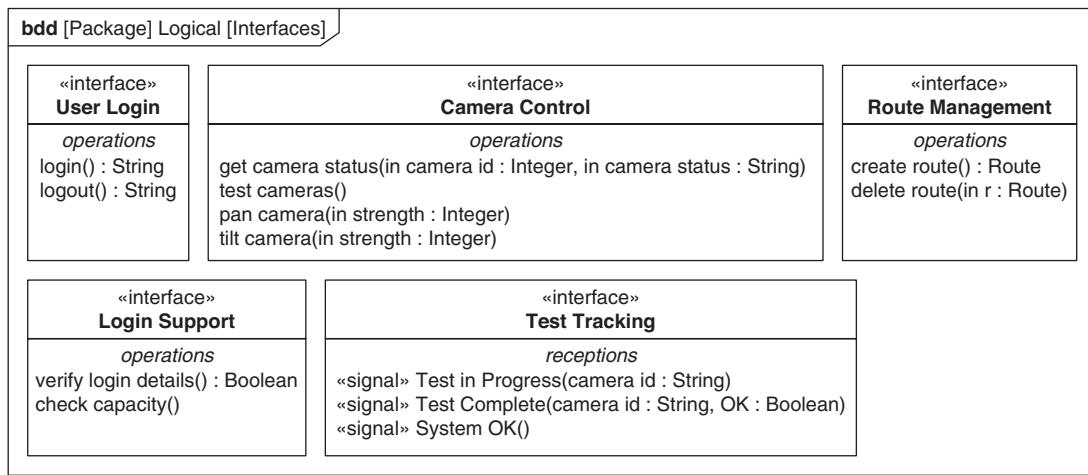
6.5.3 Modeling Service-Based Interfaces Using Standard Ports

An alternative method for describing a service-based interface to a block is to define the behavioral features the block can support at a given interaction point. A standard (service-based) port uses a definition called an **interface** to specify the set of behavioral features either required or provided at this interaction point.

Modeling Interfaces

Interfaces that type a standard port are defined on a block definition diagram as box symbols with the keyword «interface» before their name. Interface symbols have operation and reception compartments like block symbols.

Figure 6.32 shows five interfaces that describe different logical groupings of services. *Camera Control* contains a set of operations that provide support for controlling the camera. *Test Tracking* contains a set of receptions that allow the reporting of progress during camera testing. The other interfaces support other services (e.g., user and route management).

**FIGURE 6.32**

A set of interfaces used to define provided or required services.

Adding Interfaces to Standard Ports

A **required interface** on a port specifies one or more operations required by the block (or its parts) to realize its behavior. A **provided interface** on a port specifies one or more operations that a block (or one or more of its parts) provides. A part that has a port with a required interface generally requests another part to provide an input it needs to perform its function. The other part then provides the service by returning a value to the requester. Interface definitions can be reused as needed to define the interfaces of standard ports on many blocks.

When a standard port is a behavior port, the owner of the port must include all the operations and receptions specified by its provided interfaces. The typical approach to modeling this is to add a **realization dependency** from the block to each provided interface, which asserts that the block will declare a behavioral feature for each behavioral feature in that interface. A block can assert that it requires a set of behavioral features by adding a **uses dependency** to an interface, although this has no affect on the features of the block.

The uses dependency is represented by a dashed arrow with an open head pointing toward the interface. The realization dependency is represented by a dashed arrow with an unfilled triangular head pointing toward the interface.

The required and provided interfaces of a port may be represented by the dependency notation described earlier, but a more popular notation called "ball and socket" is often used instead. An interface is represented by either a ball or socket symbol with the name of the interface floating near it. The ball depicts a provided interface, and the socket depicts a required interface. A solid line attaches the interface symbol to the port that requires or provides the interface.

A standard port can have one or more required interfaces and one or more provided interfaces, and hence can be connected to multiple interface symbols.

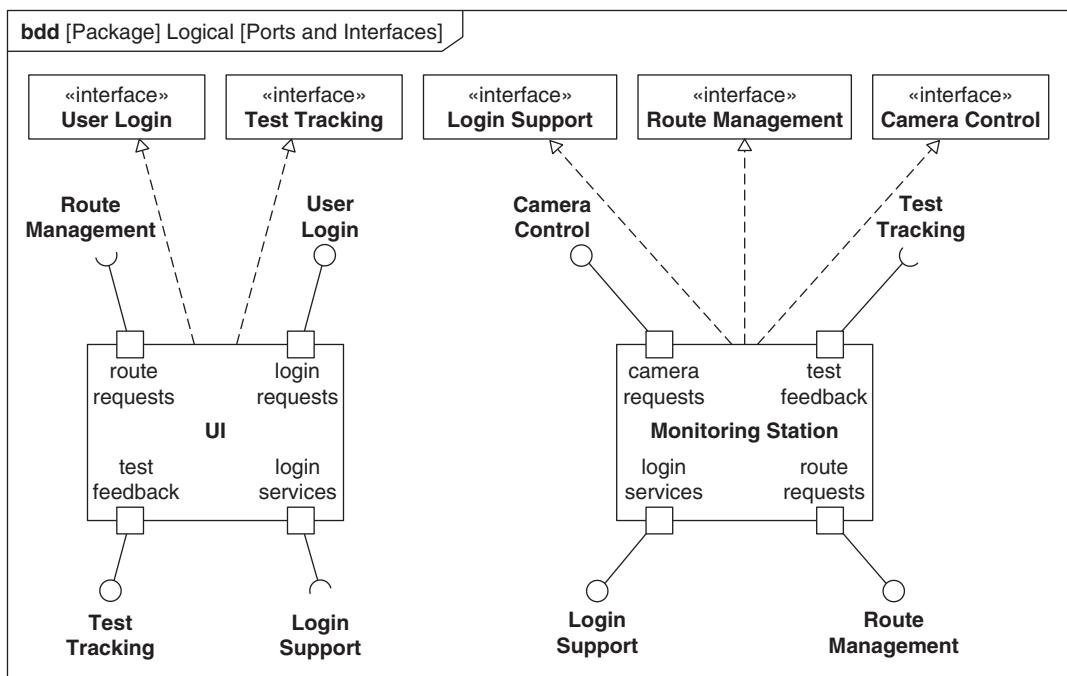


FIGURE 6.33

Defining a service-based interface using standard ports.

Alternatively, the port strings can be included in a separate *standard ports* compartment, using the following syntax:

port name: interface name[multiplicity]

Figure 6.33 shows the set of standard ports that define interface points on the blocks *UI* and *Monitoring Station*. *UI* has four ports, two that provide services and two that require services. The port *test feedback* provides the services defined by the interface *Test Tracking*. The port *route requests* requires the services defined by the interface *Route Management*. *Monitoring Station* also has four ports, three that provide services and one that requires services. Whereas the port *route requests* on *UI* requires the interface *Route Management*, port *route requests* on *Monitoring Station* provides the interface *Route Management*.

Login requests and *test feedback* on block *UI* are behavior ports, and so *UI* has realization dependencies to the *Test Tracking* and *User Login* interfaces. Similarly, *Monitoring Station* realizes *Camera Control*, *Route Management*, and *Login Support* because *login services*, *route requests*, and *camera requests* are all behavior ports.

6.5.4 Connecting Standard Ports on an Internal Block Diagram

On an internal block diagram, standard ports, like flow ports, can be connected to multiple other ports through connectors. A request made through a standard port

is communicated along any connector that connects it to a port with a compatible interface.

A tool can verify the compatibility of the interfaces of connected standard ports by verifying the compatibility of the operations on the interfaces. For assembly connectors, compatibility is ensured if the provided interfaces of one port are matched one-for-one with the required interfaces of the other port and vice versa. For delegation connectors, the connected ports are compatible if they have the same required and provided interfaces. In many cases, however, this is overly restrictive because a connector may not be used for all the possible requests implied by the interfaces of the connected ports, and so SysML leaves the precise details of port compatibility for standard ports undefined.

Required and provided interfaces can be shown on an internal block diagram using the ball-and-socket notation introduced earlier if required, although this often adds clutter to the diagram. If the ball-and-socket notation is used, it is easy to perform a quick visual check on the compatibility of connected ports. Ports connected by delegation connectors should have the interface symbols with the same name and shape. Ports connected by assembly connectors should have interface symbols with the same name and different shapes.

Figure 6.34 displays an internal block diagram for *Surveillance System* showing two of its parts, the *UI* and the *Monitoring Station*. The ports of these two parts are connected both to the ports on their parent and to each other's ports. *Surveillance System* delegates the handling of requests on its *user login* port to the *UI*, and the handling of requests on its *camera cmds* port to the *Monitoring Station*. The *UI* uses *Login Support* services of the *Monitoring Station*, via its *login services* port, to provide data on current users, and also passes through route management requests. The *Monitoring Station* provides *Test Tracking* services to the *UI*.

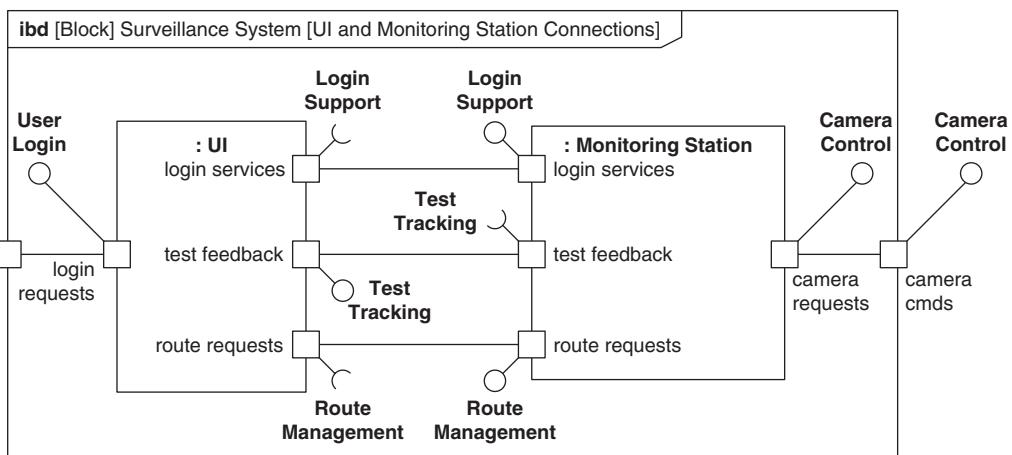


FIGURE 6.34

Connecting standard ports on an internal block diagram.

6.5.5 Modeling Block-Defined Methods

Some behaviors owned by the block only execute in response to a particular stimulus, either a request for a service provided by an operation or a signal request tied to a reception. Such a behavior is called a **method**, and it is related to the behavioral feature describing the request.

Unlike the main block behavior, methods typically have a limited lifetime, starting their execution following the stimulus, performing their allotted task, and then terminating, perhaps returning some results. Methods are usually specified using activities, opaque behaviors, or function behaviors.

It should be mentioned that not all behavioral features require methods. Requests associated with behavioral features can be handled directly by behaviors using the specialized constructs described in Chapters 8 through 10.

SysML supports the notion of **polymorphism**, where many different blocks may respond to the same service request or message, but each may do so in a specific way; that is, by invoking a specific method. Polymorphism is strongly associated with classification, as described in the next section.

6.6 Modeling Classification Hierarchies Using Generalization

All the definitions that can appear on a block definition diagram are **classifiers**, which means that they can be organized into a classification hierarchy. The classifiers so far encountered in this chapter are blocks, value types, interfaces, flow specifications, and signals. In a classification hierarchy each classifier is described as being more general or more specialized than another. Typically a general classifier includes a set of features common to a set of more specialized classifiers that also include additional features. The relationship between the general classifier and specialized classifier is called **generalization**. Different terms are used to identify the classifiers at the end of a generalization relationship. In this chapter, the general classifier is called the **superclass**, and the more specialized classifier is called the **subclass**.

Classification can facilitate reuse where a subclass reuses the features of a superclass and adds its own features. The benefits of such reuse can be substantial when the superclass has significant detail.

This section deals initially with the classification of structural features of a block, covering both the addition of features and the redefinition of existing features in subclasses. Although the focus for this section is blocks, other elements with structural features, such as flow specifications and value types, can also be classified in the same fashion. Subclasses of flow specifications can add or redefine flow properties. Subclasses of value types may add additional characteristics such as units and dimensions. Subclasses of an interface can add new operations and receptions, and subclasses of a signal can add new attributes.

In addition to classification for reuse, classification can also be used to describe specific configurations of a block, to identify unique configurations for testing, or as the input to simulations or other forms of analysis.

Classification also applies to behavioral features and can be used to control the way blocks respond to incoming requests. Classification of behavioral features and the semantics implied by the use of classification are covered by numerous texts on object-oriented design and so will not be dealt with in any detail here.

Generalization is represented by an arrow between two classifiers with a hollow triangular arrowhead on the superclass end. Generalization paths may be displayed separately, or a set of generalization paths may be combined into a tree, as shown later in Figure 6.36.

Figure 6.35 shows two subclasses of *Camera*, *Wired Camera* and *Wireless Camera*. Both of the subclasses require all the characteristics of *Camera* but add their own specialized characteristics as well. *Wired Camera* has both a mains connection and a wired Ethernet connection. The *Wireless Camera* uses WiFi

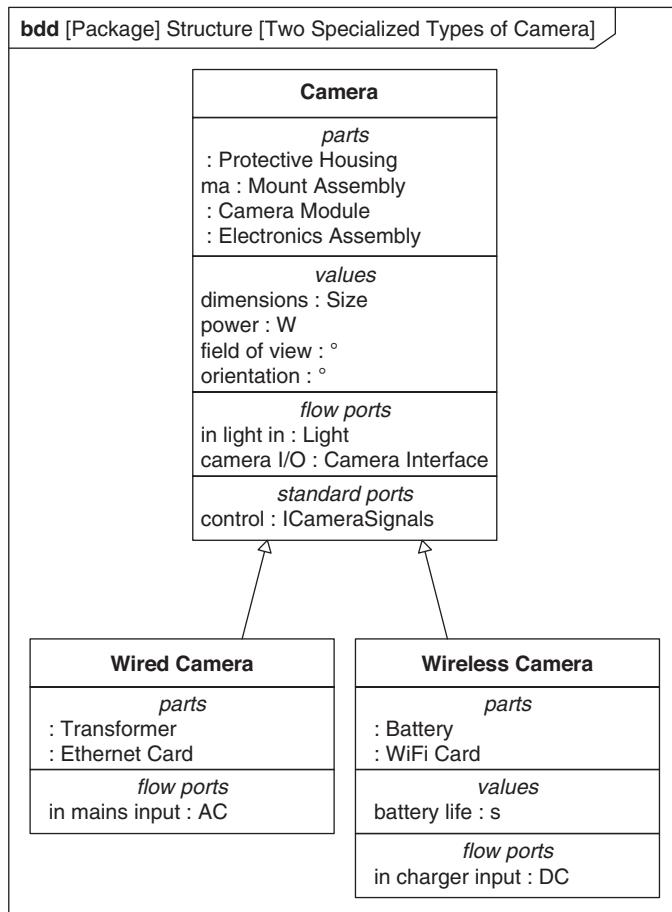


FIGURE 6.35

Example of block specialization.

to communicate and is battery-driven. It has a DC charger input and a measure of the current battery life.

6.6.1 Classification and the Structural Features of a Block

Different blocks in a classification have different structural features, with subclasses adding features not present in their superclasses. Not all features added in subclasses are new; some are introduced to override or otherwise change the definition of an existing feature, which is called **redefinition**. A feature in a subclass may also be defined to represent a **subset** of a feature in a superclass.

The fundamental consequence of redefining a property in a subclass is to prevent further use of that property in the subclass. However, on top of this the redefining property is typically intended to be used in place of the redefined property, so often has the same name. When used instead of the redefined property, the redefining property may:

- Restrict its multiplicity—for example, from $0..*$ to $1..2$ in order to reduce the number of instances or values that the property can hold.
- Add or change its initial value.
- Provide a new distribution or change an existing distribution.
- Change the type of the property to a more restricted type—in other words, a type that is a subclass of the existing type.

Redefinition is shown in braces after the name string of the redefining property using the keyword *redefines* followed by the name of the redefined property.

In the *Components* package, two motor modules are described for use in the system. Both motor modules share a number of features; for example, they both have some common value properties, such as *weight*, *power consumption*, and *torque*. In Figure 6.36 a general concept of *Motor Module* has been introduced to capture the common characteristics of the two motor modules.

In addition to value properties, *Motor Module* defines a common concept of a *control input* using a flow port. The *Brushless DC Motor Module* and the *Stepper Motor Module* are represented as subclasses of this common concept with special features of their own, such as the *step size* and *position* output for the *Stepper Motor Module*. In addition, the common properties from *Motor Module* have been redefined in the subclasses in order to place bounds on their values that are appropriate to the type of motor. The value properties are described by an «interval» probability distribution to represent the range of values properties can have in their given subclass.

6.6.2 Modeling Overlapping Classifications Using Generalization Sets

Sometimes a subclass may include features from multiple superclasses. This is called multiple generalization, or sometimes **multiple inheritance**. The subclasses of a given class may also be organized into groupings based on how they

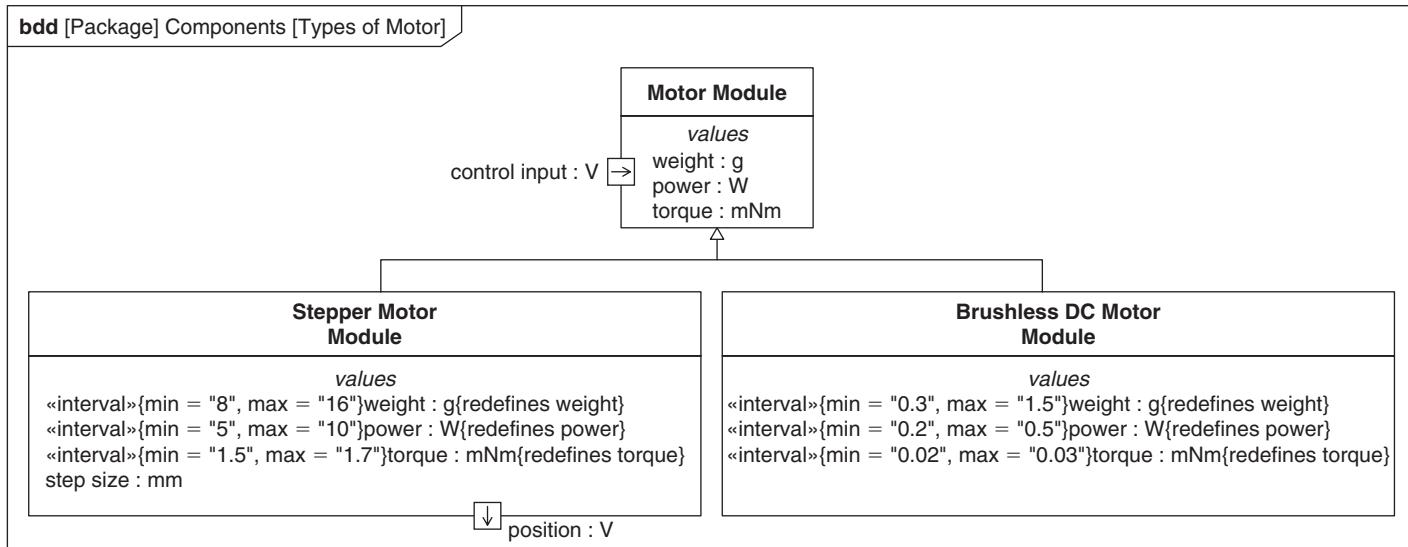
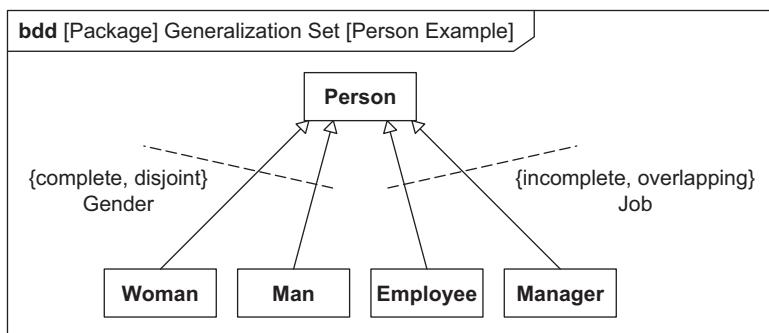


FIGURE 6.36

Showing a classification hierarchy on a block definition diagram.

**FIGURE 6.37**

Showing a generalization set on a block definition diagram.

can be used for further classification. For example, a superclass *Person* may have subclasses that represent the characteristics of an *Employee* OR a *Manager* in their job AND subclasses that represent the characteristics of a *Woman* OR a *Man* as their gender. This situation can be modeled using generalization sets, as shown in Figure 6.37. **Generalization sets** have two properties that can be used to describe coverage and overlap between their members.

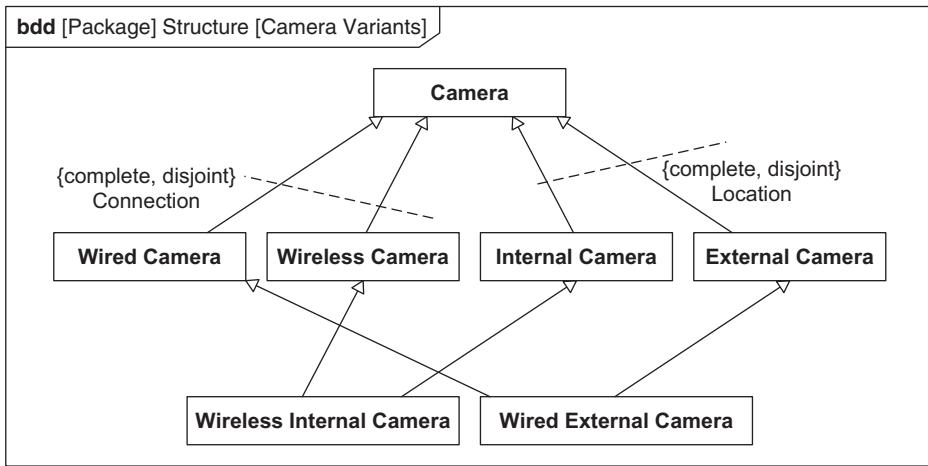
The **coverage** property specifies whether all the instances of the superclass are instances of one or another of the members of the generalization set. The two values of the coverage property are complete and incomplete. The **overlap** property specifies whether an instance of the superclass can only be an instance of at most one subclass in the generalization. The two values of the property are disjoint and overlapping.

A generalization set may be displayed on a block definition diagram by a dashed line intersecting a set of generalization paths. The name of the generalization set and the values of the overlap and coverage properties, shown in braces, are displayed floating near the line that represents the generalization set. Alternatively, where the tree form of generalization notation is used, a generalization set may be represented by a tree with the generalization set name and properties floating near the triangle symbol at its root. Figure 6.37 shows the dashed-line variant and Figure 6.40 the tree variant.

Figure 6.37 shows the example of generalization sets described earlier. *Person* is subclassed by four subclasses in two generalization sets. *Gender* has two members, *Woman* and *Man*, and is both disjoint and completely covered because all instances of *Person*, must be an instance of either *Woman* or *Man* but not both. *Job* has two members, *Employee* and *Manager*, and is overlapping and incompletely covered because an instance of *Person* may be an instance of both *Employee* and *Manager*, or neither.

6.6.3 Modeling Variants Using Classification

The description and organization of product variants is a large and complex topic and requires solutions that cover many different disciplines, of which modeling

**FIGURE 6.38**

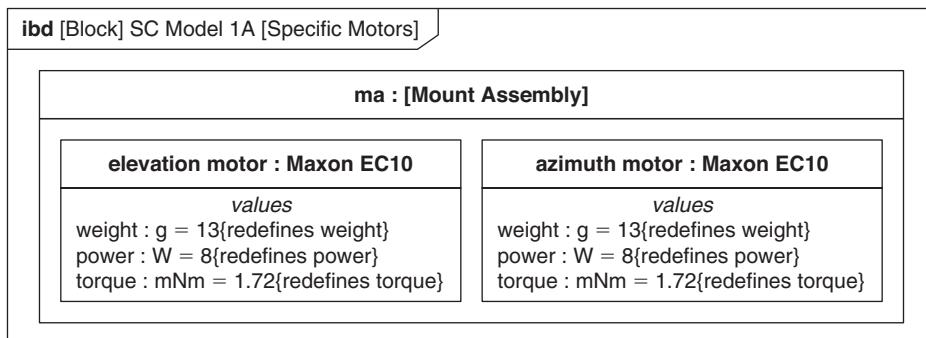
Modeling variant configurations on a block definition diagram.

is just one. Nonetheless, SysML contains concepts like classification that can be used to capture some of the details and relationships needed to model variants. For example, classification can be used to model different variants of a block definition that represent alternative designs being evaluated in a trade study. This can be achieved by describing several specialized variants of a block as subclasses of the original, grouped into generalization sets. Note that multiple subclasses of a superclass can be recombined using multiple generalizations in subsequent levels of classification, but these must obey the specified overlap and coverage of their superclasses.

Figure 6.38 shows two mutually exclusive characterizations of the *Camera*: its intended location and the way that it connects with a controller. Each characterization in this case has two variants. There are two intended locations, indicated by the generalization set *Location*, served by either an *Internal Camera* or an *External Camera*. There are also two intended modes of connection, indicated by the *Connection* generalization set, served by the *Wired Camera* and *Wireless Camera* blocks originally shown in Figure 6.35. Two further variants, *Wireless Internal Camera* and *Wired External Camera*, are created by multiple generalization from these four. The features of the blocks are hidden to reduce clutter.

6.6.4 Using Property-Specific Types to Model Context-Specific Block Characteristics

A **property-specific type** is used to designate parts or value properties that are further specialized for localized use within an internal block diagram. This might happen, for example, when one or more properties of a part have different distributions than in the original of their type. The property-specific type implicitly

**FIGURE 6.39**

Property-specific types.

creates a subclass of the block that types the part property to add the unique characteristics. The presence of a property-specific type is indicated by including the type name of a property in brackets. Compartments can be used to depict the unique features for each part-specific property, such as the value properties for the different motors' weights in the following example.

Figure 6.39 shows a small fragment of a particular model of surveillance camera, the *SC Model 1A*, that specializes *Camera*. In the *SC Model 1A*, the generic *Stepper Motor Module* used in the *Mount Assembly* (*ma*) of *Camera* has been replaced by a specific motor module containing the *Maxon EC10*. To do this replacement, rather than specifically create a block that represents this variant of *Mount Assembly*, a property-specific type is used. Significant properties of the *Maxon EC10* are shown in the *values* compartments of the parts.

6.6.5 Modeling Block Configurations as Specialized Blocks

A **block configuration** describes a specific structure and specific property values intended to represent a unique instance of a block in some known context. For example, a block configuration may be used to identify a particular aircraft in an airline company's fleet by its call sign and to provide other characteristics specific to that aircraft. In that example, the call sign is intended to consistently identify the same aircraft even though the values of other properties may change over time. Block configurations can also be used to identify the state of some entity at a given point in time. Extending the example of the aircraft, it might be important for an air-traffic control simulation to describe a snapshot of an aircraft's position, velocity, fuel load, and so on at certain critical analysis stages.

It is important to note that because a block configuration can only describe a finite set of features and values, there may be many actual instances in the physical domain that match that description. It is up to the modeler to ensure that the context is understood and that any ambiguity does not compromise the value of the model. Typically the block contains a value property whose value can be used

to identify a single instance within the context. For example, a car number plate may be unique within a given country but not over all countries.

Modeling a Configuration on a Block Definition Diagram

A block configuration is constructed using the generalization relationship described earlier. The configuration becomes a subclass of the block for which it is a configuration. There is no specific notation for designating a unique configuration. However, a block is often defined with a property that represents a unique identifier such as the vehicle identification number that can be used when modeling configurations. Often it is useful to introduce a generalization set for block configurations to distinguish them from other specializations of that block.

A useful characteristic of the SysML property concept is the ability to state that a property in a subclass may subset one or more properties in one of its superclasses. This means that the set of instances or values of the subsetting property are also in the set of instances or values for a subsetted property. Whereas a redefining property replaces the redefined property in the subclass, a subsetting property sits alongside its subsetted property and holds a subset of its values and instances.

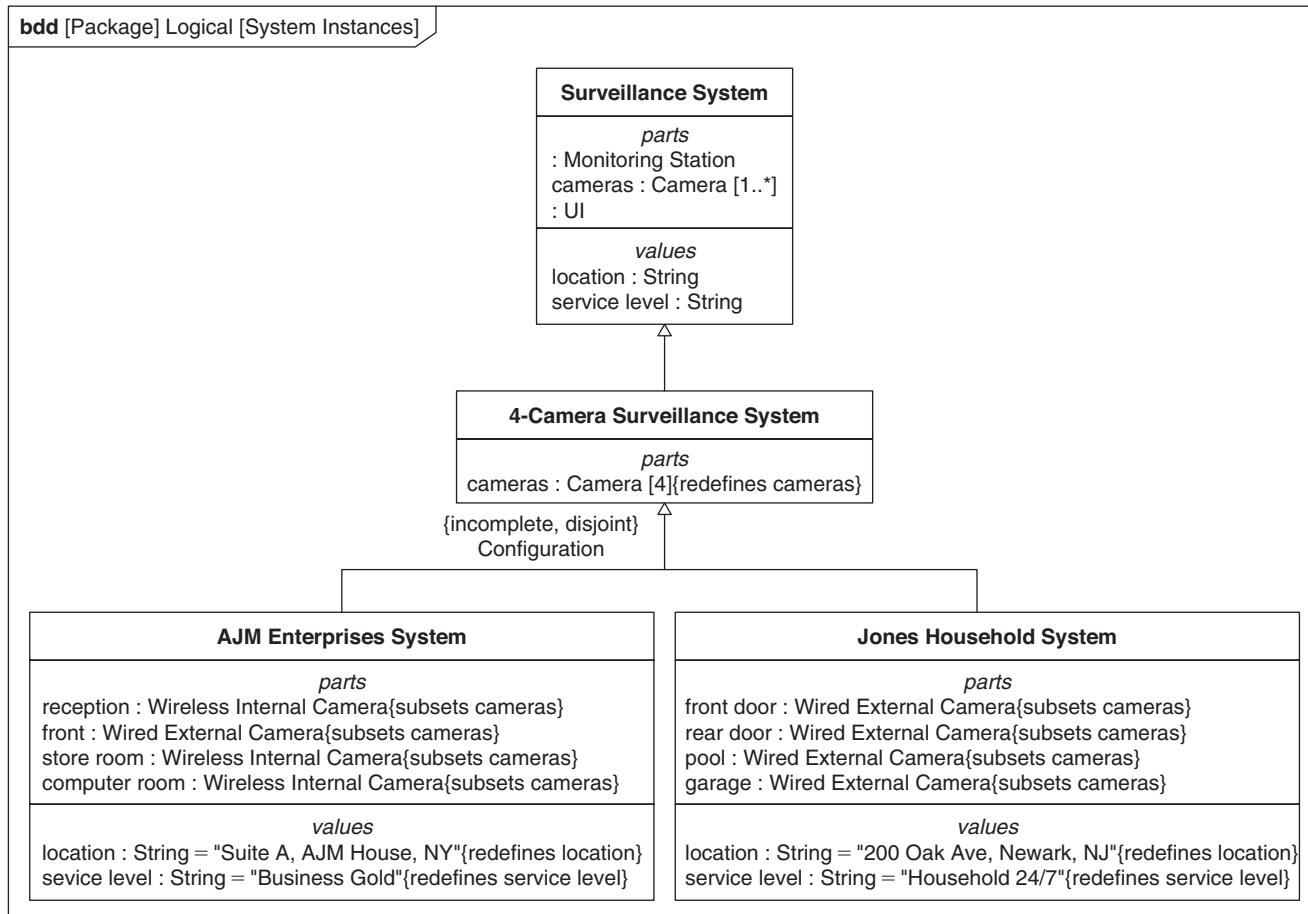
Where a property has an upper bound of greater than 1, subsetting can be used to explicitly identify one of the set of instances held by the property in order to define its specific characteristics. Subsetting is shown in braces after the name string of the subsetting property using the keyword *subsets* followed by the names of the subsetted properties.

Two configurations of the company's popular *4-Camera Surveillance System* are shown in Figure 6.40. The values for *location* in each case give the addresses of the installations. It is intended that within the context of the ACME business, the specific values for *location* are enough to uniquely identify the instance of one of their surveillance systems. The company also offers an optional service package and the *service level* provides details of the level of service offered. *Business Gold* includes hourly visits by a security agent outside office hours. *Household 24/7* ensures a response to any alert within 30 minutes, 24 hours and 7 days a week.

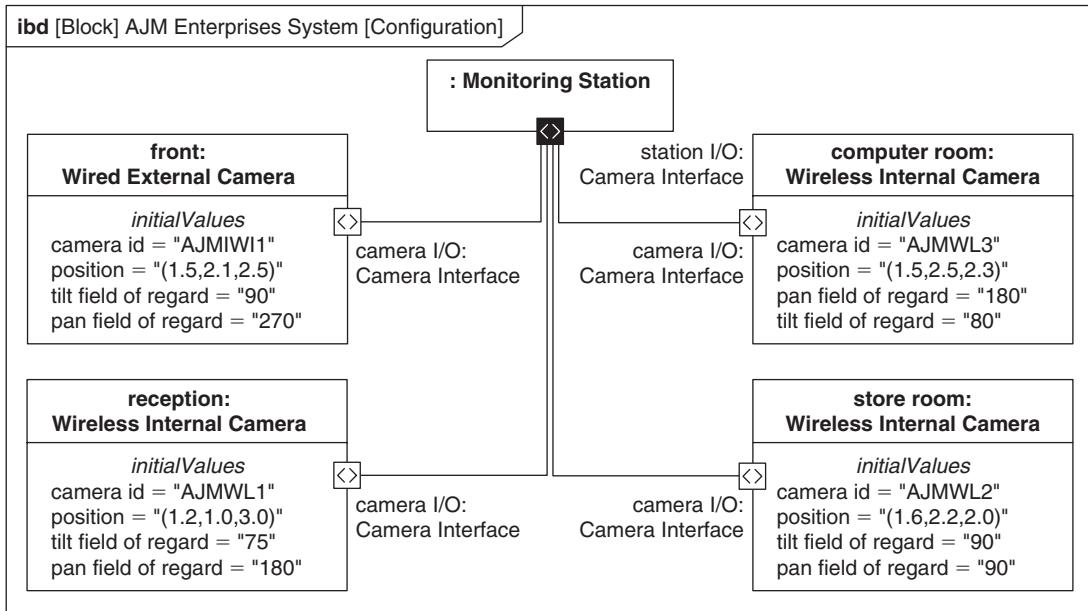
The *4-Camera Surveillance System* specializes *Surveillance System* and redefines its *camera*'s part property with a new property, also called *cameras*. The new property has a multiplicity of 4 that restricts the upper number of instances held by *cameras* to 4 from the original upper bound of “*”, and also raises the lower bound to 4.

To describe specific configurations, *AJM Enterprises System* and *Jones Household System* specialize the *4-Camera Surveillance System* and redefine or subset some of its properties. Two value properties, *location* and *service level*, are redefined in order to provide specific values for them. The *camera*'s part property is subsetted by part properties that represent individual cameras in the configuration. In *AJM Enterprises*, the new parts are called *front*, *reception*, *store room*, and *computer room*, based on their location within the company's building.

The set of configurations of the *4-Camera Surveillance System* is grouped by a generalization set called *Configuration*. *Configuration* is disjoint because each subclass is intended to describe a separate instance, and is incomplete because there may be other instances of the superclass than just these.

**FIGURE 6.40**

Modeling an instance of a block on a block definition diagram.

**FIGURE 6.41**

Showing the configuration of a block instance on an internal block diagram.

Modeling Configuration Details on an Internal Block Diagram

When a block has been used to describe a configuration, the internal block diagram for that block can be used to capture the specific internal structural (e.g., precise multiplicities and connections) and values unique to configuration properties. In particular, this should include the value of a property that uniquely identifies the entities in the configuration (e.g., name, serial number, call sign). A unique design configuration, such as a configuration with a specific serial number, can be created by defining an identification property for each part in the block that corresponds to the unique identification of the enclosing block.

Given that *AJM Enterprises System* is a subclass of *4-Camera Surveillance System*, it has four cameras. Figure 6.41 identifies a number of camera variants, including the *Wireless Internal Camera* and *Wired External Camera*, used here to satisfy the installation requirements. The *camera id* property of each camera provides a unique identifier for the cameras in the system and the four cameras have their own values, also stenciled on the casing of the camera. The configuration also describes the *position* and *field of regard (pan and tilt)* of each camera to facilitate coverage analysis as part of a security viewpoint.

6.6.6 Classification and Behavioral Features

Just as the properties of blocks can be organized into classification hierarchies, the behavioral features of blocks can be treated in a similar fashion. A summary

description of the classification of behavioral features and corresponding behaviors is included here; however, more complete discussions are beyond the scope of this book and can be found in many object-oriented design books.

General services are described at an abstract level (as operations or receptions) in the classification hierarchy and more specific services are described in more specialized blocks. As with structural features, the behavioral features of superclasses may be redefined in subclasses to modify their signature.

Interfaces can also be classified and their behavioral features specialized in the same fashion as blocks. When a block realizes an interface, it must support the behavioral features of not just that interface but also all its superclasses.

The response of a block to a request for a behavioral feature may also be specialized. Although a behavioral feature is defined in a general block, the method for that feature in a given specialization of the block may be different. In software engineering, this phenomenon is called polymorphism—from the Greek “many forms”—because the response to a request for a given behavioral feature may be different depending on the method that actually handles the request.

In object-oriented programming languages, polymorphism is handled by a dispatching mechanism. If a behavior sends a request to a target object, it knows what the type (e.g., block) of the target object is and that it can support the request. However, due to specialization, the target object may validly be a subclass of the type that the requester knew about, and that subclass may implement a different response to the request. The dispatching mechanism can “look behind the scenes” and make sure the method of the appropriate type is invoked to handle the request.

6.7 Summary

SysML structure is primarily represented on block definition diagrams and internal block diagrams. The following are key concepts related to modeling structure.

- The block is the fundamental unit of structure in SysML and is represented on both the block definition diagram and the internal block diagram. Blocks describe types of entities defined by their features. A block provides the description for a set of uniquely identified instances that all have the features defined by the block. A block definition diagram is used to define a block, its characteristics, and its relationship to other blocks. An internal block diagram is used to describe internal details of a block.
- Blocks have a number of structural and behavioral features that comprise its definition. Properties describe its structural aspects in terms of its relationship to other blocks and quantifiable characteristics. Ports describe a block's interface as a set of interaction points on its boundary. Behavioral features specify the behaviors that may be invoked by requests for service.
- A part property is used to describe the hierarchical composition (sometimes called whole-part relationships) of block hierarchies. Using this terminology, the

owner of the property is the whole and the type of the part property is the part. Any given instance of the block that types a part property may only exist as part of one instance of a whole. Composite associations are used to express the relationship of the part to the whole; in particular, whether blocks of the part type always exist in the context of an instance of the whole or may exist in isolation.

- Reference properties are used to describe other relationships between blocks. A reference property does not imply any exclusive relationship between related instances, but does allow blocks to keep references to others.
- Value properties represent quantifiable characteristics of a block such as its physical and performance characteristics. Value properties are typed by value types. A value type provides a reusable description of some quantity and may include units and dimensions that further characterize the quantity. A value property may have an initial value and has extensions for capturing probability distributions.
- SysML has two different types of ports: a flow port and a standard port. The flow port specifies what can flow in or out of a block and the standard port specifies what behavioral features are required or provided by the block.
- A block can have two types of behavioral features, operations and receptions. Operations describe synchronous interactions where the requester waits for the request to be handled; receptions describe asynchronous behaviors where the requester continues immediately. Behavioral features may be related to methods, which are the behaviors that handle requests for the features. Requests for behavioral features may also be handled directly by the main behavior, typically an activity or state machine, as described in Chapters 8 and 10.
- The concepts of classification and generalization sets describe how to create classification hierarchies of blocks and other classifiers such as value types and flow specifications. Classifiers specialize other classifiers in order to reuse their features and add new features of their own. Generalization sets group the subclasses of a given superclass according to how they partition the instances of their superclass. Subclasses may overlap, which means that a given instance can be described by more than one, or not. Subclasses may have complete coverage of the superclass, which means that all instances are described by one of the subclasses in the set, or not.
- Blocks can be used to describe configurations, where the features of the block are defined in enough detail to identify a specific instance of the block in the real world of the system.

6.8 Questions

1. What is the diagram kind of a block definition diagram, and which model elements can it represent?
2. What is the diagram kind of an internal block diagram, and which model elements can it represent?

3. How is a block represented on a block definition diagram?
4. Name the three categories of block properties.
5. Which type of property is used to describe composition relationships between blocks?
6. What is the commonly used term for properties with a lower multiplicity bound of 0..1?
7. What is the default interpretation of the multiplicity for both ends of an association when it is not shown on the diagram?
8. Draw a block definition diagram, using composite associations, for blocks “Boat,” “Car,” and “Engine” showing that a “Car” must have one “Engine,” and a “Boat” may have between one and two “Engines.”
9. Give two situations in which the use of role names for the part end of a composite association should be considered.
10. How are parts shown on an internal block diagram?
11. What does the presence of a connector between two parts imply?
12. Draw an internal block diagram for the “Boat” from Question 8, but with an additional part “p” of type “Propeller.” Add a connector between the “Engine” part (using its role name from Question 8 if you provided one) and “p,” bearing in mind that one “Propeller” can be driven by only one “Engine.”
13. What are the two graphical mechanisms that can be used to represent properties nested more than one level deep on an internal block diagram?
14. What is the major difference between parts and references?
15. What is the difference in representation between the symbol for composite association and reference association on a block definition diagram?
16. What is an association block?
17. How are the quantitative characteristics of blocks described?
18. What are the three categories of value types?
19. Apart from the definition of a valid set of values, what can value types describe about their values?
20. A block “Boat” is described by its “length” and “width” in “Feet” and a “weight” in “Tons.” Draw a block definition diagram describing “Boat,” with definitions of the appropriate value types, including units and dimensions.
21. What is a derived property?
22. How are probability distributions, say an interval distribution, for a property represented in the values compartment on a block definition diagram?
23. Which SysML concepts can be used to represent items (i.e., things that flow)?
24. What does an item flow define?
25. What does a flow port specify?
26. A block “Boat” takes “fuel” and “cold water” as inputs and produces “exhaust gases” and “warm water” as outputs. Show “Boat” on a block definition diagram with inputs and outputs as atomic flow ports. Demonstrate the use of both port icons and the “flow ports” compartment.
27. What is the difference between atomic and nonatomic flow ports?
28. What is the rule for assessing the compatibility of an item flow on a connector between two atomic flow ports?
29. What is a delegation port on a block used for?

30. Name all five types of behavioral specification supported by SysML.
31. What are the behavioral features of blocks used for?
32. What is a method?
33. What is a standard port used to describe?
34. What do the required interfaces of standard ports specify?
35. What do the provided interfaces of standard ports specify?
36. Describe the ball-and-socket representation for the interfaces of ports.
37. Name four types of classifier encountered in this chapter.
38. Name three aspects of a redefined property that a redefining property can change.
39. How is a generalization relationship represented on a block definition diagram?
40. When specifying a generalization set, what is the coverage property used to define?
41. How are generalization sets represented on a block definition diagram?
42. Where one property is defined to be a subset of another, what is the relationship between the elements of the subsettled property and the elements of the subsetting property?

Discussion Topic

Discuss the benefits of enforcing encapsulation of block structure using the encapsulated properties.

This page intentionally left blank

Modeling Constraints with Parametrics

7

This chapter describes SysML support for modeling constraints on the performance and physical properties of systems and their environment to support a wide array of engineering analyses.

7.1 Overview

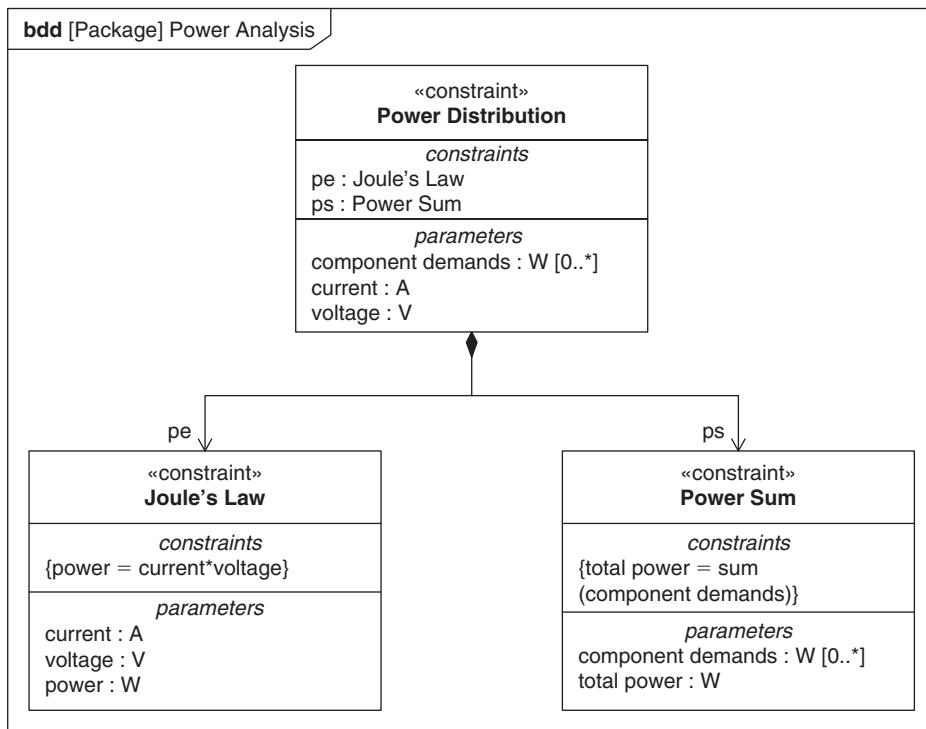
A typical design effort includes the need to perform many different types of engineering analyses to support trade-off studies, sensitivity analysis, and design optimization. It may include the analysis of performance, reliability, and physical properties to name a few. SysML supports this type of analysis through the use of parametric models.

Parametric models capture the constraints on the properties of the system, which can then be evaluated by an appropriate analysis tool. The constraints are expressed as equations whose parameters are bound to the properties of a system. Each parametric model can capture a particular engineering analysis of the design. Multiple engineering analyses can then be performed on each design alternative to support trade-off analysis.

SysML introduces a constraint block to support the construction of parametric models. A constraint block is a special kind of block used to define equations so that they can be reused and interconnected. Constraint blocks have two main features: a set of parameters and an expression that constrains the parameters. Constraint blocks follow a similar pattern of definition and use that applies to blocks and parts as described in Chapter 6. A use of a constraint block is called a constraint property. The definition and use of constraint blocks is represented on a block definition diagram and parametric diagram, respectively.

7.1.1 Defining Constraints Using the Block Definition Diagram

Block definition diagrams are used to define constraint blocks in a similar way to which they are used to define blocks. An example of a block definition diagram containing constraint blocks is shown in Figure 7.1.

**FIGURE 7.1**

Example block definition diagram with constraint blocks.

This figure shows three constraint blocks. *Joule's Law* and *Power Sum* are leaf constraint blocks that each define an equation and its parameters. *Power Distribution* is a constraint block composed of *Joule's Law* and *Power Sum* to build a more complex equation.

The diagram elements for defining constraint blocks in the block definition diagram are shown in the Appendix, Table A.5.

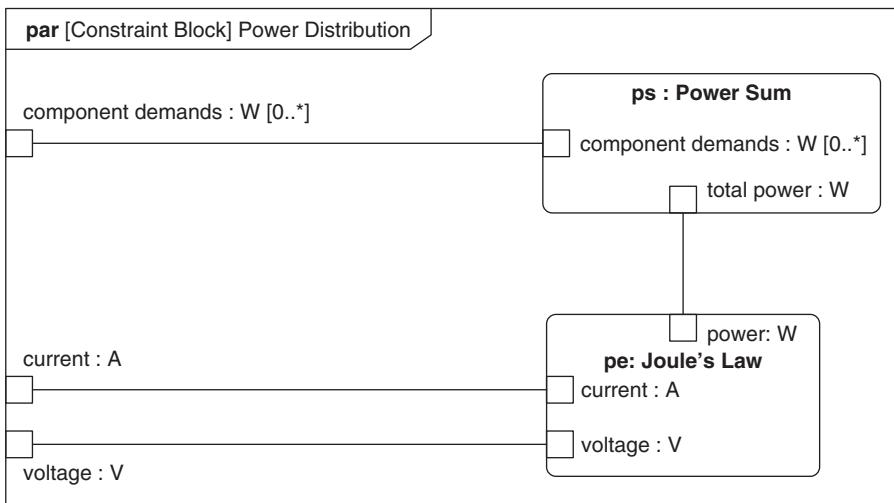
7.1.2 The Parametric Diagram

Parametric diagrams are used to create systems of equations that can constrain the properties of blocks. The parametric diagram header is depicted as follows:

par [model element type] model element name [diagram name]

The diagram kind is **par**, short for parametric diagram. The diagram frame of a parametric diagram represents either a block or a constraint block. The diagram name as usual is user defined and is intended to emphasize the purpose of the diagram.

Figure 7.2 shows a parametric diagram for the constraint block *Power Distribution* from Figure 7.1. The constraint properties *ps* and *pe* are usages of

**FIGURE 7.2**

A parametric diagram used to construct systems of equations.

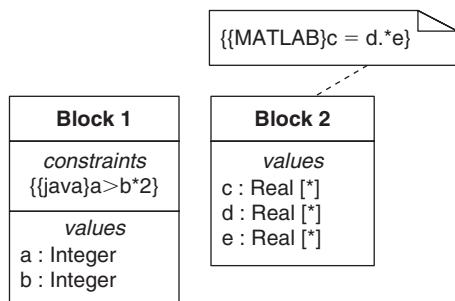
the constraint blocks *Power Sum* and *Joule's Law*, respectively. The parameters of the constraint properties *ps* and *pe* are bound to each other and to the parameters of *Power Distribution*, which are shown flush with the diagram frame. The diagram elements of the parametric diagram are shown in the Appendix, Table A.10.

Sections 7.2 through 7.7 describe the definition of constraint blocks and their use in constraining the properties of blocks. Section 7.8 deals with constraining time-dependent properties. Section 7.9 deals with the application of constraints to item flows. Sections 7.10 and 7.11 deal with more sophisticated analysis scenarios, including defining an analysis context and the application of parametrics to trade studies.

7.2 Using Constraint Expressions to Represent System Constraints

SysML includes a generic mechanism for expressing constraints on a system as text expressions that can be applied to any model element. SysML does not provide a built-in constraint language because it was expected that different constraint languages, such as OCL, Java, or MathML, would be used as appropriate to the domain. The definition of a **constraint** can include the language used to enable the constraint to be evaluated.

Constraints may be owned by any element that is a namespace, such as a package or block. If the element that owns the constraint is shown as a symbol with compartments, such as a block, the constraint can be shown in a special compartment labeled *constraints*. A constraint can also be shown as a note symbol attached to the model element(s) it constrains, with the text of the constraint

**FIGURE 7.3**

Example of the two notations for showing constraints.

shown in the body of the note. The constraint language is shown in braces before the text of the expression, although it may be and often is elided to reduce clutter.

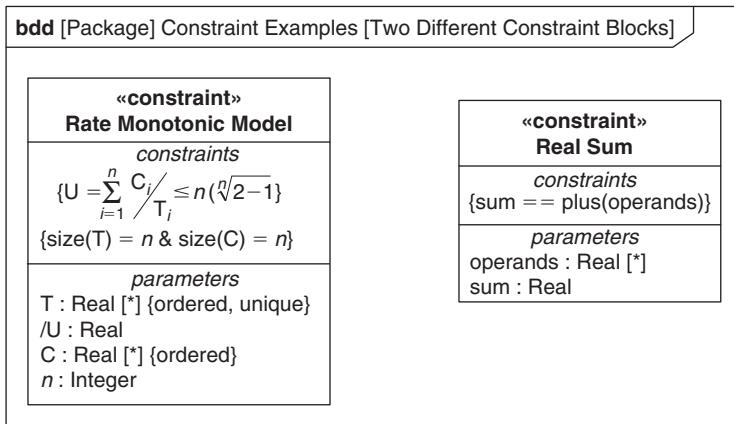
Figure 7.3 shows examples of the different constraint notations used in SysML that constrain the properties of a block. *Block 1* has an explicit compartment for the constraint, which in this case is expressed using Java. *Block 2* has a constraint that is shown in an attached note and is expressed in the constraint language of a specialized analysis tool called MATLAB.

7.3 Encapsulating Constraints in Constraint Blocks to Enable Reuse

SysML also features a constraint block that extends the generic constraint concept. A **constraint block** encapsulates a constraint to enable it to be defined once and then used in different contexts, similar to the way parts represent usages of blocks in different contexts. The equivalent concept to the part is called a **constraint property**.

The constraint expression can be any mathematical expression and may have an explicit dependency on time, such as a time derivative in a differential equation. In addition to the constraint expression, a constraint block defines a set of **constraint parameters**—a special kind of property used in the constraint expression. Constraint parameters are bound to other parameters and properties of the blocks where they are used. Constraint parameters do not have direction to designate them as dependent or independent with respect to the constraint expression. The interpretation of the dependencies between parameters is based on the semantics of the language used to specify the constraint expression. So, for example, in the C programming language, the expression $a = b + c$ implies that a is dependent on the value of b and c , whereas the expression $a == b + c$ does not.

Like other properties, each parameter has a type that defines the set of values that the parameter can take. Typically parameters are scalars, vectors, or a structured data type such as complex. Through its type, the parameter can also be constrained to have a specific unit and dimension. Parameters can also support probability distributions like other properties.

**FIGURE 7.4**

Two reusable constraint blocks expressed on a block definition diagram.

7.3.1 Additional Parameter Characteristics

Properties have two characteristics that are useful when defining collections; that is, properties whose multiplicity has an upper bound greater than 1. Modelers can specify whether the collection is **ordered** and whether the values in the collection have to be **unique**. Ordered in this case simply means that the members of the collection are mapped to the values of a positive integer: member 1, member 2, and so on. The means by which the order is to be determined would have to be specified by a constraint, or using a behavior that builds the collection. These two characteristics are useful in specifying constraint parameters.

Another useful characteristic of properties is that they can be marked as derived (see Derived Properties section in Chapter 6). If a property is marked as derived it means that its value is derived, typically from the values of other properties' values. This characteristic has two obvious uses in specifying parametrics. First, where the calculation underlying an equation is known to be implemented as a function, a derived parameter can be used to indicate which value is calculated. An example of this can be seen in Figure 7.4. Second, where the modeler wishes to guide the equation solver, derived properties can indicate which values in a given analysis need to be solved for. An example of this can be seen later in Figure 7.18.

A constraint block is defined in a block definition diagram as shown in Figure 7.4. The diagram header is the same as any other block definition diagram specifying the package or block that owns the constraint block. The name compartment of the constraint block includes the keyword «constraint» above the name to differentiate it from other elements on a block definition diagram. The constraint expression is defined in the *constraints* compartment of the constraint block and the constraint parameters are defined in the *parameters* compartment using a string with the following format:

parameter name: type[multiplicity]

Indications of ordering and uniqueness appear as keywords in braces after the multiplicity. The ordering indication is either “ordered” or “unordered”; the uniqueness indication is either “unique” or “nonunique.” In practice, unordered and nonunique are often indicated by the absence of a keyword. A derived property is shown with a forward slash (/) before its name.

Figure 7.4 shows two constraint blocks, *Real Sum* and *Rate Monotonic Model*. *Real Sum* is a simple reusable constraint where one parameter, *sum*, equals the sum of a set of *operands*, as expressed in the constraint in the constraints compartment. *Rate Monotonic Model* is also reusable but more specialized; it describes the equations underlying the rate monotonic analysis approach to scheduling periodic tasks on a processing resource. *T* represents the periods of the tasks, *C* represents the computation load of the tasks, and *U* represents the utilization of the processing resource. The constraint language is not shown in either case, but it can be seen that the constraint for *Real Sum* is expressed in a “C”-like syntax. The utilization constraint for *Rate Monotonic Model* is expressed using a more sophisticated equation language, which has the capability to be rendered using special symbols. Both mechanisms are equally acceptable in a SysML constraint block.

Both *T* and *C* are ordered collections, as indicated by the ordered keyword. The values of *T* are required to be unique because each task must have a different rate for the analysis to be correct. Parameter *n* specifies the number of tasks and an additional constraint is used to constrain the size of both *T* and *C* to be *n*. *U* is always the dependent variable in the underlying calculation and so is marked as derived.

7.4 Using Composition to Build Complex Constraint Blocks

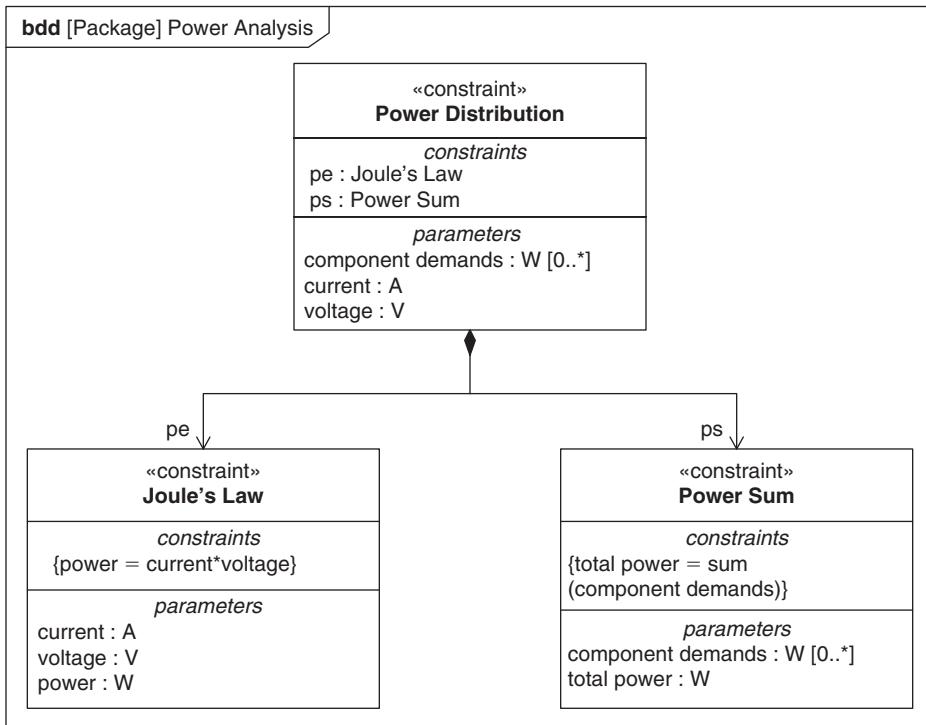
Modelers can compose complex constraint blocks from existing constraint blocks on a block definition diagram. In this case, the composite constraint block describes an equation that binds the equations of its child constraints. This enables complex equations to be defined by reusing simpler equations.

The concept of definition and usage that was described for blocks in Chapter 6 applies to constraint blocks as well. A block definition diagram is used to define constraint blocks. The parametric diagram represents the usage of constraint blocks in a particular context. This is analogous to the usage of blocks as parts in an internal block diagram. The usages of constraint blocks are called constraint properties.

Composition of constraint blocks is described using composite associations between constraint blocks. The associations are depicted using the standard association notation introduced in Chapter 6 to represent composition hierarchies. A constraint block can also list its constraint properties in its *constraints* compartment using the following syntax:

constraint property : constraint block[multiplicity]

Figure 7.5 shows the decomposition of a *Power Distribution* constraint block into two other constraint blocks, *Joule’s Law* and *Power Sum*. The role names on

**FIGURE 7.5**

A hierarchy of constraints on a block definition diagram.

the component end of the compositions correspond to constraint properties. *Pe* is a usage of the *Joule's Law* constraint block, which describes the standard power equation. *Ps* is a usage of the *Power Sum* constraint block, which equates the *total power* demand to a set of *component demands*. *Power Distribution* uses these equations to relate the demands of a set of components to the required *current* and *voltage* of a power supply.

The *Joule's Law* and *Power Sum* constraint blocks feature their equations in their *constraints* compartments, whereas *Power Distribution* lists its constituent constraint properties. Note that, in this example, the constituent constraints of *Power Distribution* are represented both in its *constraints* compartment and as association symbols. However, typically, in a given diagram, only one form of representation is used.

7.5 Using a Parametric Diagram to Bind Parameters of Constraint Blocks

As with blocks and parts, the block definition diagram does not show all the required information needed to interconnect its constraint properties. Specifically, it does not show the relationship between the parameters of constraint properties

and the parameters of their parent and siblings. This additional information is provided on the parametric diagram using **binding connectors**. Binding connectors express equality relationships between their two ends.

Constraint properties show the same parameters as their types. Two constraint parameters can be bound directly to each other on a parametric diagram using a binding connector, which indicates that the values of the two bound elements must be the same. This enables a modeler to connect multiple equations to create complex sets of equations where a parameter in one equation is bound to a parameter in another equation.

Just as the parameters of a constraint block say nothing about causality, similarly binding connectors express an equality relationship between their bound elements, but say nothing about the causality of the equation network. When an equation is to be solved, it is assumed that the dependent and independent variables are identified or deduced, including the specification of initial values. This is typically addressed by a computational equation solver, which is generally provided in a separate analysis tool, as discussed in Chapter 17. As stated earlier, derived parameters or properties can be used to guide equation solvers where parts of the solution order are known.

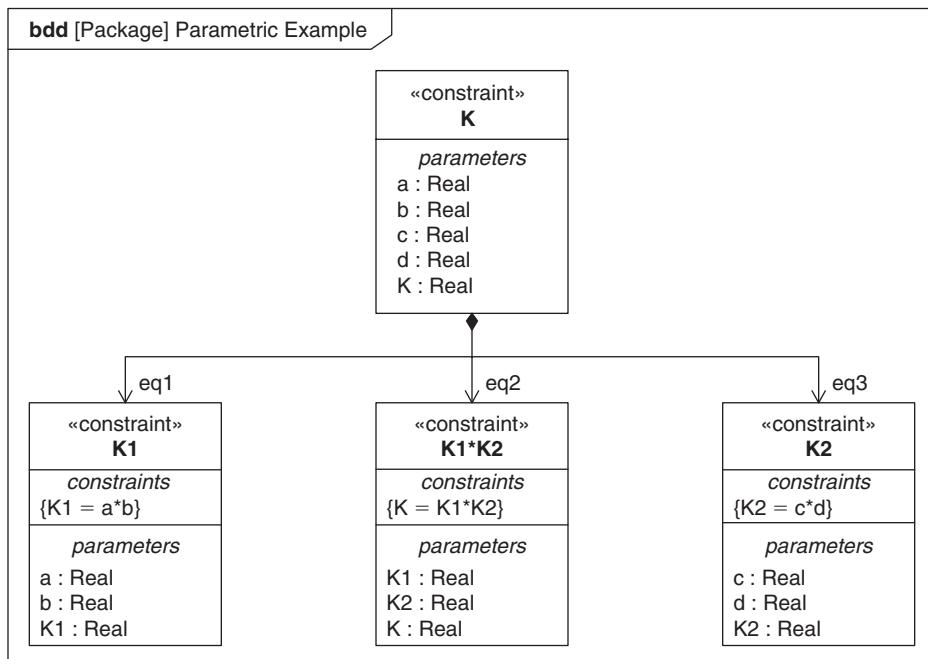
Just as with the internal block diagram, the notation for constraint properties in a parametric diagram relates back to their definition on the block definition diagram as follows:

- A constraint block, or block on a block definition diagram that owns constraint properties, can be represented as the diagram frame of a parametric diagram with the constraint block or block name in the diagram header.
- A constraint property, or a constraint block on the component end of the composite association on the block definition diagram, may appear as a constraint property symbol within a frame representing the constraint block on the composition end. The name string of the symbol is composed of the constraint property name and its type. Where a composite association was used, the constraint property name corresponds to the role name on the component end of the association. The type name corresponds to the name of the constraint block on the component end of the association.

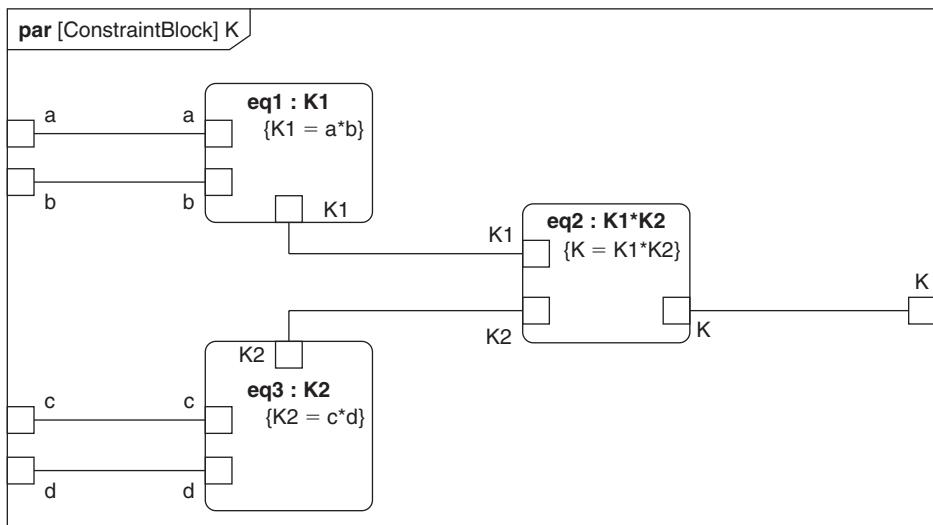
A simple example of constraint block composition is shown using Figure 7.6 and Figure 7.7.

Figure 7.6 shows a block definition diagram with a constraint block K composed from three other constraint blocks: $K1$ with three parameters, $K1, a$, and, b , and the constraint $\{K1 = a * b\}$; $K2$ with three parameters, $K2, c$, and d , and the constraint $\{K2 = c * d\}$; and finally constraint block $K1 * K2$ with three parameters, $K, K1$, and $K2$, and constraint $\{K = K1 * K2\}$. K itself has five parameters, K, a, b, c , and d , that will be bound to parameters of its constituents' constraint properties as shown in Figure 7.7.

The frame of a parametric diagram corresponds to a constraint block or a block as described in the next section. If the parametric diagram represents a constraint block, then any parameters are shown as small rectangles flush with the inner surface of the frame. The name, type, and multiplicity of each parameter are shown in a textual label floating near the parameter symbol.

**FIGURE 7.6**

A block definition diagram for a composite constraint block.

**FIGURE 7.7**

Binding two equations to each other.

On a parametric diagram, a constraint property is shown as a round-cornered rectangle (round-angle) symbol with the name of the property and its type inside the box. Either the property name or the type name can be elided if desired. The constraint equation itself can be elided, but if shown, may appear either inside the round-angle or attached via a comment symbol to the round-angle. The parameters of the constraint property are shown flush with the inside surface of the constraint property symbol.

A binding connector is depicted as a restricted form of the connector used on an internal block diagram. It is simply a solid line with no arrows or other annotations.

Figure 7.7 shows the corresponding parametric diagram for constraint block K in Figure 7.6. As stated earlier, the names in the constraint property symbols are produced from the component ends of the associations on the block definition diagram. The bindings in this case connect parameters with similar names, which produces an equation for K that could be simplified as $\{K = a * b * c * d\}$.

It should be noted that although this is just a trivial example it does highlight an important point. Parametric diagrams can be quite bulky compared to textual expressions but are useful when constructing more complex equations from reusable constraint blocks. This is an artificial example to demonstrate the concept of composing constraint blocks. For an actual model using this simple example, K would probably be expressed directly with the expression $\{K = a * b * c * d\}$ and would have no internal structure.

Figure 7.8 shows an example from the Surveillance System, where the *Power Distribution* composite constraint block, originally introduced in Figure 7.5, is depicted as the frame of a parametric diagram.

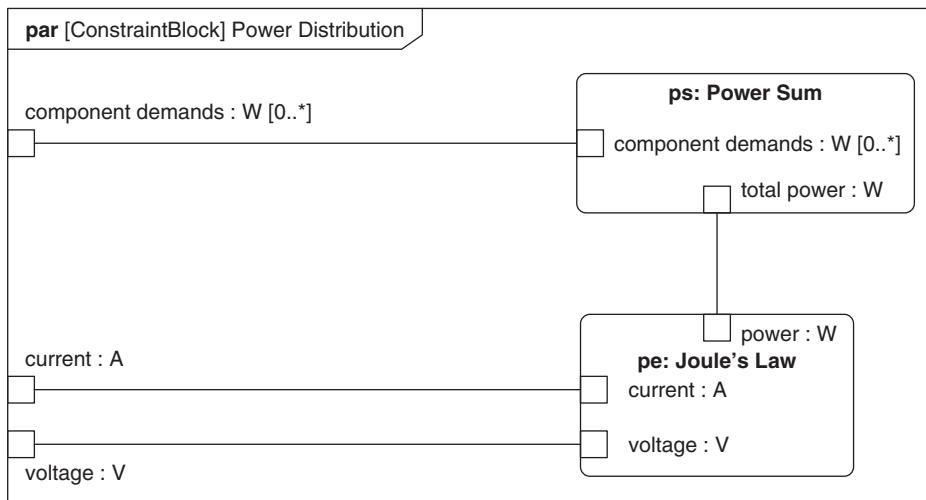


FIGURE 7.8

Internal details of the power distribution equation using a parametric diagram.

The diagram shows how the parameters of constraint properties *ps*, a usage of *Power Sum*, and *pe*, a usage of *Joule's Law*, are bound together. The *voltage* and *current* parameters of *pe* are bound to the *voltage* and *current* parameters of the block *Power Distribution* (hence shown on the frame boundary). The *power* parameter of *pe* is bound to the total cumulative power of all the powered equipment, calculated by *ps*, from the set of *component demands* (also a parameter of *Power Distribution* and shown on the frame boundary).

7.6 Constraining Value Properties of a Block

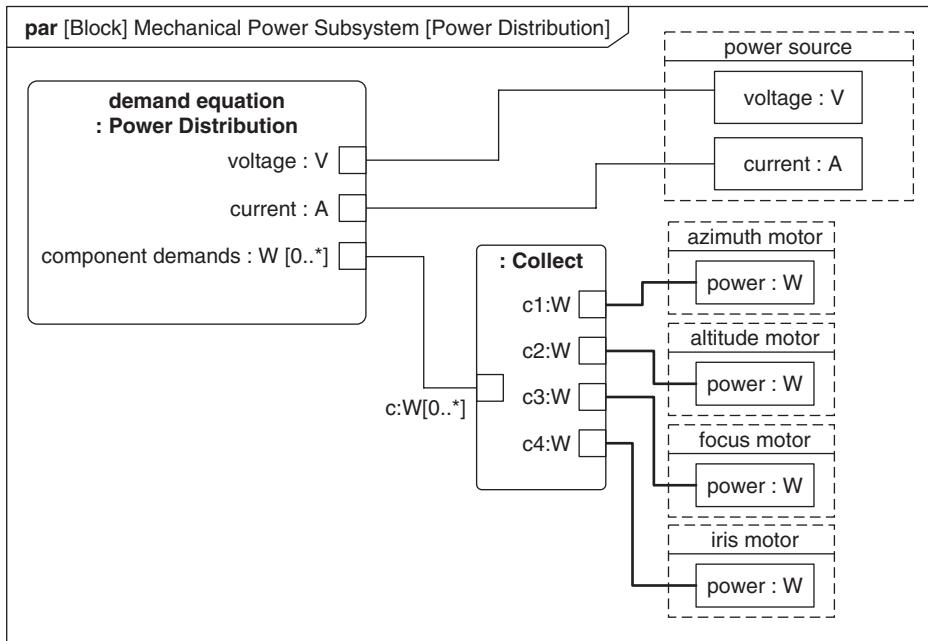
The value properties of a block can be constrained using constraint blocks. This is achieved on a block definition diagram by drawing composite associations between the block whose values are being constrained and the required constraint blocks. In a parametric diagram, the block represents the enclosing frame and the constraint properties represent usages of the constraint blocks. The parameters of the constraint properties can be bound to the value properties of the block using binding connectors. For example, if the equation $F = w * a/g$ is specified as the constraint of a constraint block with parameters *F*, *w*, and *a*, the weight property of a block that is subject to the force can be bound to parameter *w* of a constraint property typed by that constraint block. This enables the equation to be used to explicitly constrain the properties of interest.

In a parametric diagram for a block, a value property is depicted as a rectangle displaying its name, type, and multiplicity. A nested value property within a part hierarchy can be shown nested within its containing part symbol or can be shown using the dot notation that was described in Chapter 6. An example of binding nested value properties using the part hierarchy notation is shown in Figure 7.9, and an example using the dot notation is shown in Figure 7.10.

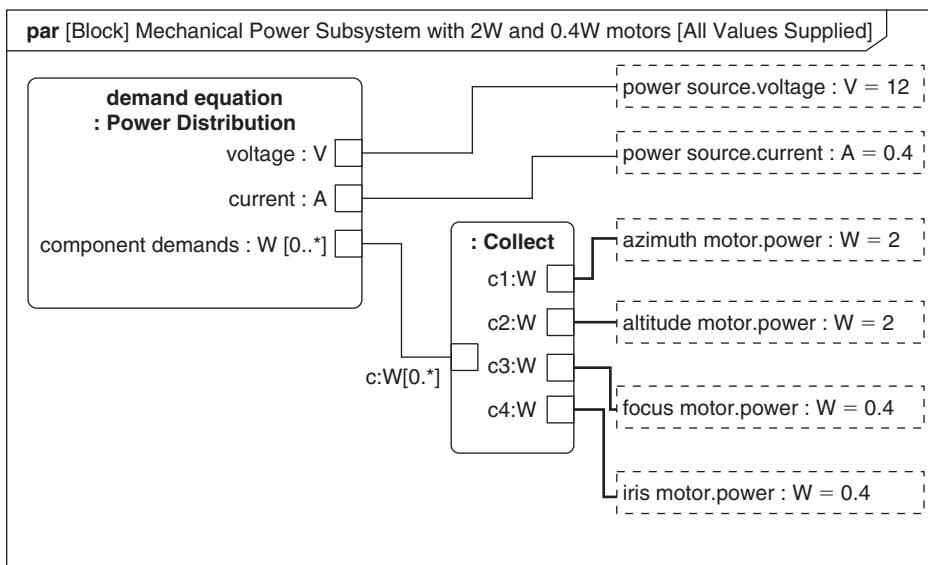
Figure 7.9 shows the constraints on the power supply for the *Mechanical Power Subsystem* described by the internal block diagram in Figure 6.11. The *Power Distribution* constraint block is used, via a constraint property *demand equation*, to relate the current and voltage of the power source for the *Mechanical Power Subsystem* to the load imposed on the power source by the various motors. An additional constraint block, *Collect*, is used to collect the power demand values of all the powered devices into one collection for binding to the *component demands* parameter of *demand equation*.

7.7 Capturing Values in Block Configurations

To allow an analysis tool to evaluate blocks containing constraint properties, at least some of the value properties of the block under analysis need to have specific values defined. Often, these values are provided during analysis using the interface of the analysis tool, but they can be specified using block configurations—that is, by creating a specialization of the block for a given analysis.

**FIGURE 7.9**

Binding constraints to properties on a parametric diagram.

**FIGURE 7.10**

Describing a specific analysis configuration.

Although the block in Figure 7.9 contains all the relationships required to perform an analysis of the *Mechanical Power Subsystem* block, the related properties do not have values, and so no calculation of required power can be performed. Figure 7.10 shows a configuration of the *Mechanical Power Subsystem* block, specified as a specialization of the original block and called *Mechanical Power Subsystem with 2W and 0.4W motors*.

Even though there are no mandatory naming standards for configurations it is often useful to include information about the configuration, as part of its name. Note that, in this case, all the values for the related properties are shown and so the *demand equation* constraint property simply acts as a check that the values are consistent. In other analysis scenarios, one or more properties may not have a value, in which case an equation-solving tool (often a human being) would be used to rearrange the constraint expression to compute the missing value or values, or to report an error if a value cannot be determined.

7.8 Constraining Time-Dependent Properties to Facilitate Time-Based Analysis

A value property is often a time-varying property that may be constrained by ordinary differential equations with time derivatives, or other time-dependent equations. There are two approaches to representing these time-varying properties. The first, as illustrated in Figure 7.11, is to treat time as implicit in the expression. This can help reduce diagram clutter and is often an accurate representation of the analysis approach with time provided behind the scenes by the analysis tool.

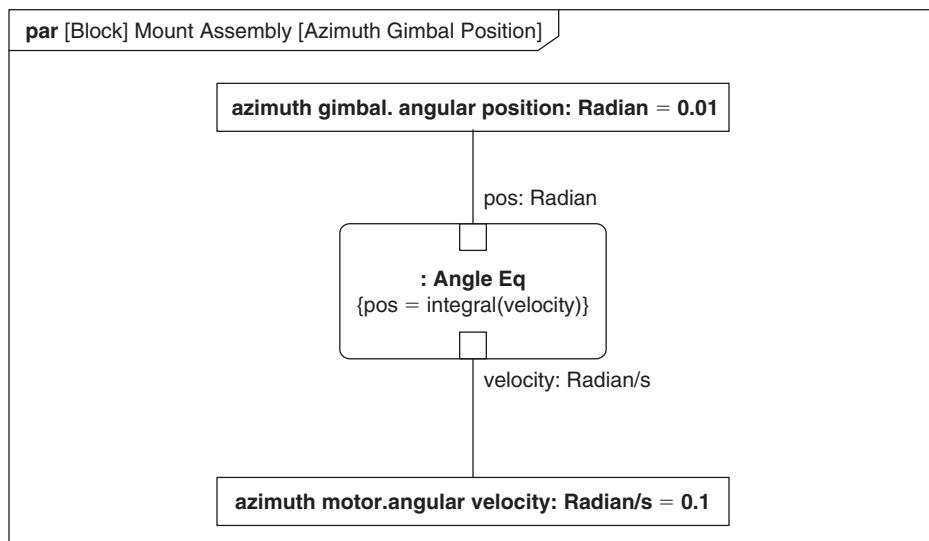


FIGURE 7.11

Using a time-dependent constraint.

Figure 7.11 shows the calculation of the *angular position*, in *Radians*, of the *azimuth gimbal* over time. The equation simply integrates the *angular velocity* of the *azimuth motor* over time to establish the angular position, *pos*. The initial values of azimuth gimbal *angular position* and *azimuth motor angular velocity* in this case could be interpreted as initial or constant values depending on the semantics of the analysis.

Another approach to the representation of time is to include a separate time property that explicitly represents time in the constraint equations. The time property can be expressed as a property of a reference clock with specified units and dimension. The time-varying parameters in the constraint equations can then be bound to the time property. Local clock errors, such as clock skew or time delay, can also be introduced by defining a clock with its own time property that is related to a reference clock through additional constraint equations.

In Figure 7.11, time was implicit and initial conditions were defined by the default values of the *position* and *velocity* properties. Figure 7.12 shows an example of the alternate approach of explicitly showing time, and uses constraints on values to express conditions at time zero.

The figure shows the standard distance equation bound to the values of an object under acceleration. The block *Accelerating Object* contains a reference to a *Reference Clock*, whose *time* property is bound to *t*, a value property of *Accelerating Object* that records passage of time as experienced by the object. The acceleration *a*, initial velocity *u*, and distance traveled *s* are bound to the *Distance Equation* along with time *t*. An additional constraint, *Distance at T0*, is used to specify the initial distance of the object (i.e., at time zero), which in this

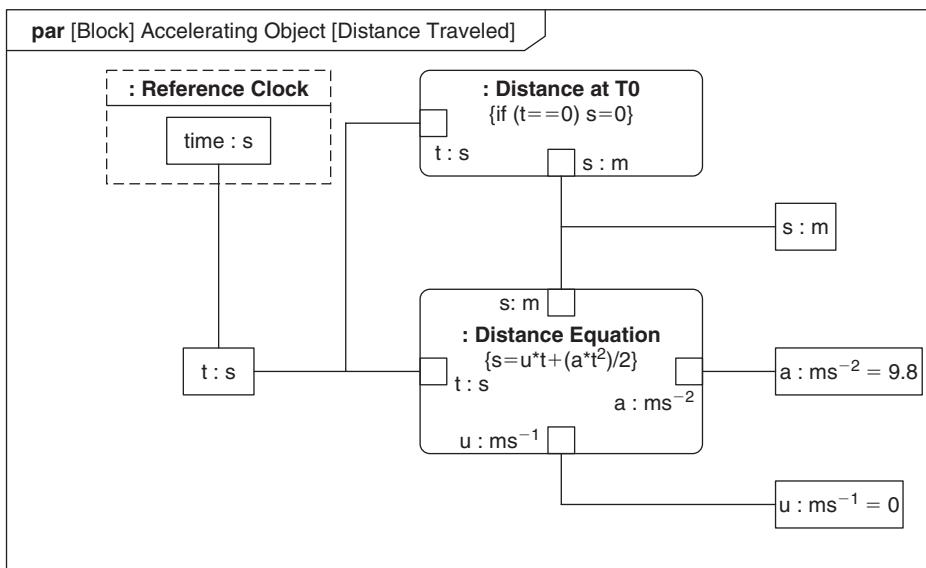


FIGURE 7.12

Explicitly representing time in a parametric diagram.

case is 0. The value of property *a* is specified with an initial value that represents the constant value of acceleration due to gravity.

7.9 Using Constraint Blocks to Constrain Item Flows

A powerful use of constraint blocks is to show how properties associated with the flow of matter, energy, or information is constrained. To achieve this, item flows (or more accurately the item properties corresponding to item flows) can be shown on parametric diagrams and bound to constraint parameters.

Figure 7.13 shows the amplitudes of the item flows shown on the internal block diagram in Figure 6.30. *Detector signal* is the item flow from the image detector to the *Imaging Electronics*, and *boosted signal* is the item flow from the *Imaging Electronics* to the boundary of the imaging assembly and therefore to the electronics assembly. The low-level electrical signal from the image detector must be amplified before it leaves the camera module to reduce its sensitivity to noise. The required amplification is specified using a gain equation to constrain the amplitude of the input and output signals of the *Imaging Electronics*. The *gain* parameter in the gain equation, *Gain Eq*, is bound to the *gain* property of the *Imaging Electronics*.

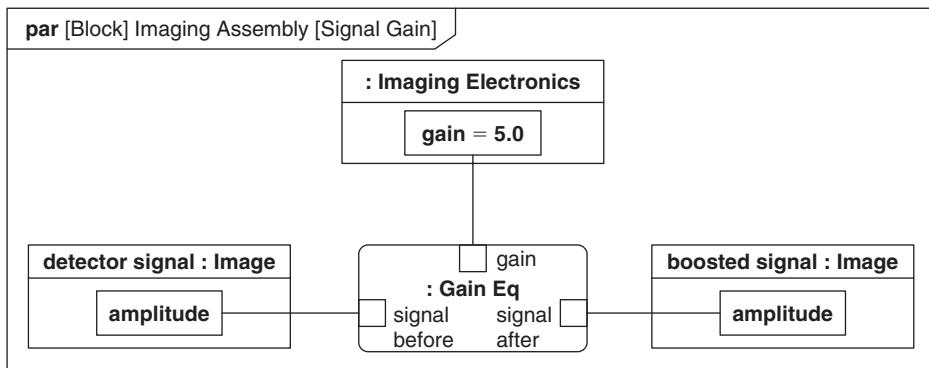


FIGURE 7.13

Constraining item flows.

7.10 Describing an Analysis Context

A constraint property that constrains the value properties of a block can, as discussed earlier, be part of a block's definition and thus shown in its constraint compartment. This works well when the constrained properties are intrinsically related in this way in all contexts. What often occurs, however, is that the constraints on block properties may vary based either on current context or analysis

requirements. For example, a different fidelity of analysis may be applied to the same system block depending on the required accuracy of the value of key properties. This type of scenario requires a more flexible approach where the properties of the block can be constrained without the constraint being part of the block's definition. This approach effectively decouples the constraint equations from the block whose properties are being constrained, and thus enables the constraint equations to be modified without modifying the block whose properties are being constrained.

To follow the approach described earlier, a modeler creates an **analysis context**, which contains both the block whose properties are being analyzed and all constraint blocks required to perform the analysis. Libraries of constraint blocks may already exist for a particular analysis domain. These constraint blocks are often called **analysis models** and may be very complex and supported by sophisticated tools. The general analysis models in these libraries may not precisely fit a given scenario and the analysis context may contain other constraint blocks to handle transformations between the properties of the block and the parameters of the analysis model.

An analysis context is modeled as a block with associations to the block being analyzed, the chosen analysis model, and any intermediate transformations. By convention, the block being analyzed is referenced by the analysis context because it is really part of the system being built, rather than part of the analysis context. The choice of using a white diamond symbol or a simple association with no end adornment to represent a reference is arbitrary. Composite associations are used between the analysis context and the analysis model and any other constraint blocks, however. An example of an analysis context is shown in Figure 7.14.

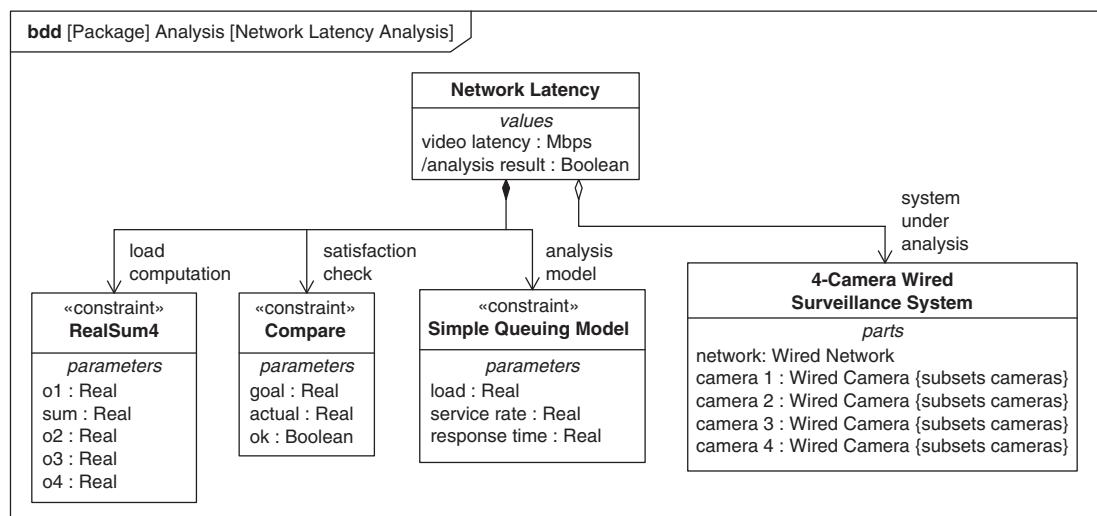
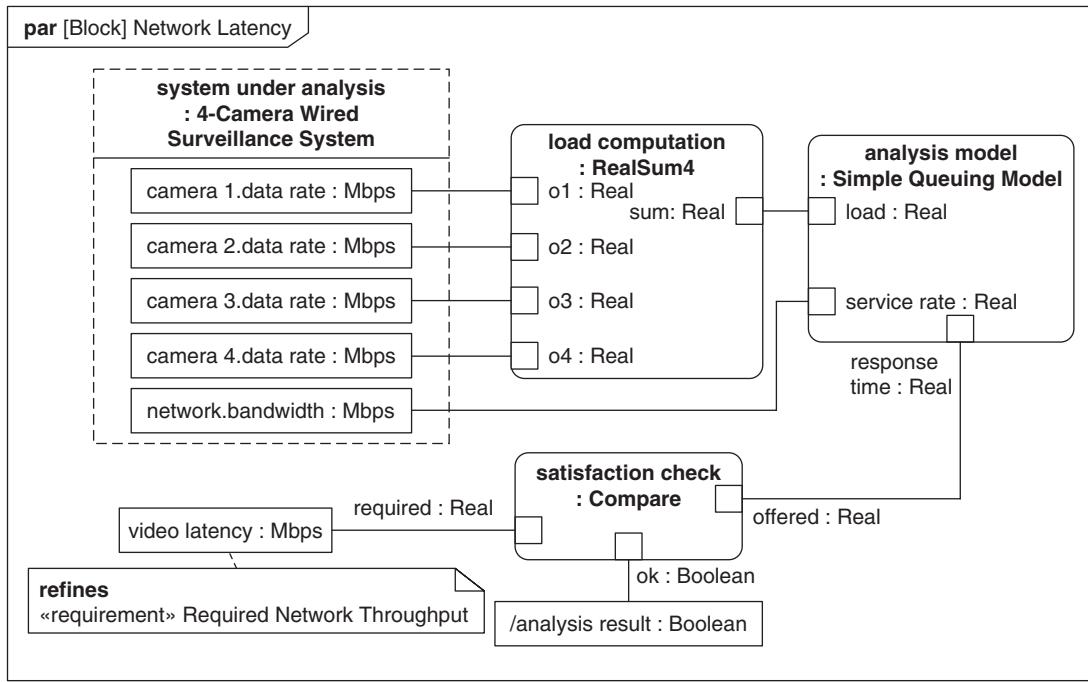


FIGURE 7.14

An analysis context shown on a bdd.

Figure 7.14 shows the analysis of network throughput for a *4-Camera Wired Surveillance System*. The analysis context is called *Network Latency*, which references the *system under analysis*, a *4-Camera Wired Surveillance System*. The analysis context also contains an *analysis model*, in this case a *Simple Queuing Model*, and uses a couple of basic constraints, *Real Sum* and *Compare*, to perform a *load computation* and a *satisfaction check*, respectively. *Network Latency* contains two value properties, *video latency*, specified in *Mbps*, and *analysis result*, which is intended to be a computed value and hence is derived.

In Figure 7.15, the bindings needed to perform the analysis are shown. The parameters of the analysis model are bound to the properties of the block under analysis. The loads on the system from all four cameras in the *system under analysis* are summed to establish the total *load* using *load computation*. The *network bandwidth* of the *system under analysis* is used to establish the *service rate* for the *analysis model*. The *response time*, calculated using the *analysis model*, is then compared, using the *satisfaction check*, to the required *video latency*, itself a refinement of a network throughput requirement, to establish the analysis result (see Chapter 12 for a discussion of requirements). If the *analysis result* is true, then the network satisfies the requirement.

**FIGURE 7.15**

Binding values in an analysis context.

7.11 Modeling Evaluation of Alternatives and Trade Studies

A common use of constraint blocks is to support “trade studies.” A **trade study** is used to compare a number of alternative solutions to see whether they satisfy a particular requirement. Each solution is characterized by a set of **measures of effectiveness** (often abbreviated “moes”) that have a calculated value or value distributions. The moes for a given solution are then evaluated using an **objective function** (often called a cost function or utility function), and the results for each alternative are compared to select a preferred solution.

Annex C of the SysML specification introduces some concepts to support the modeling of trade studies. A moe is a special type of property. An objective function is a special type of constraint block that expresses an objective function whose parameters can be bound to a set of moes using a parametric diagram. A set of solutions to a problem may be specified as a set of blocks that each specialize a general block. The general block defines all the moes that are considered relevant to evaluating the alternatives, and the specialized blocks provide different values or value distributions for the moes.

A moe is indicated by the keyword «moe» in a property string for a block property. An objective function is indicated by the keyword «objectiveFunction» on a constraint block or constraint property.

Figure 7.16 shows two variants of a *Camera* intended to provide a solution to operate in low-light conditions. These variants are shown using specialization, as described in Chapter 6, and are called *Camera with Light* and *Low-Light Camera*.

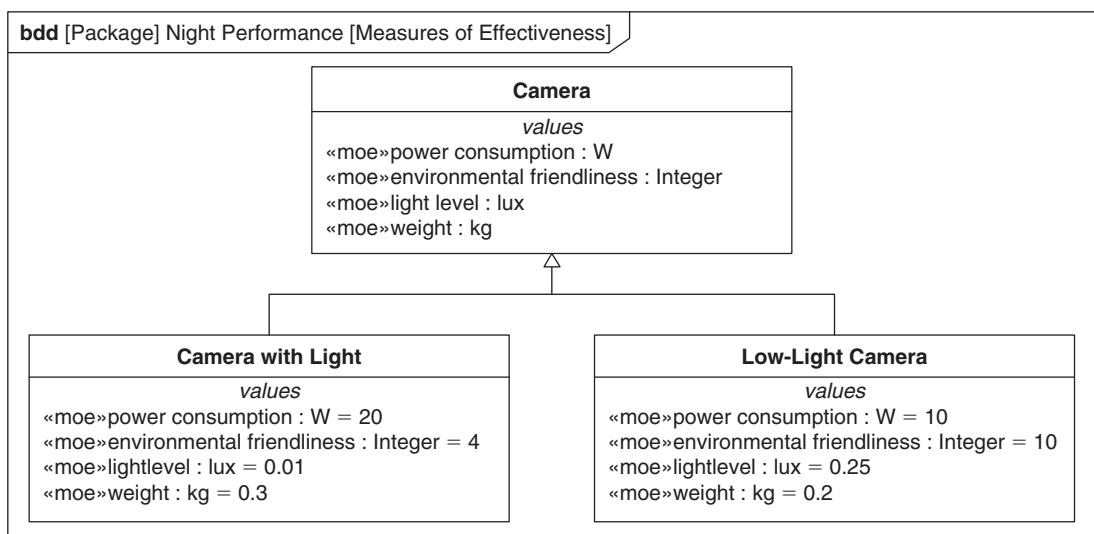
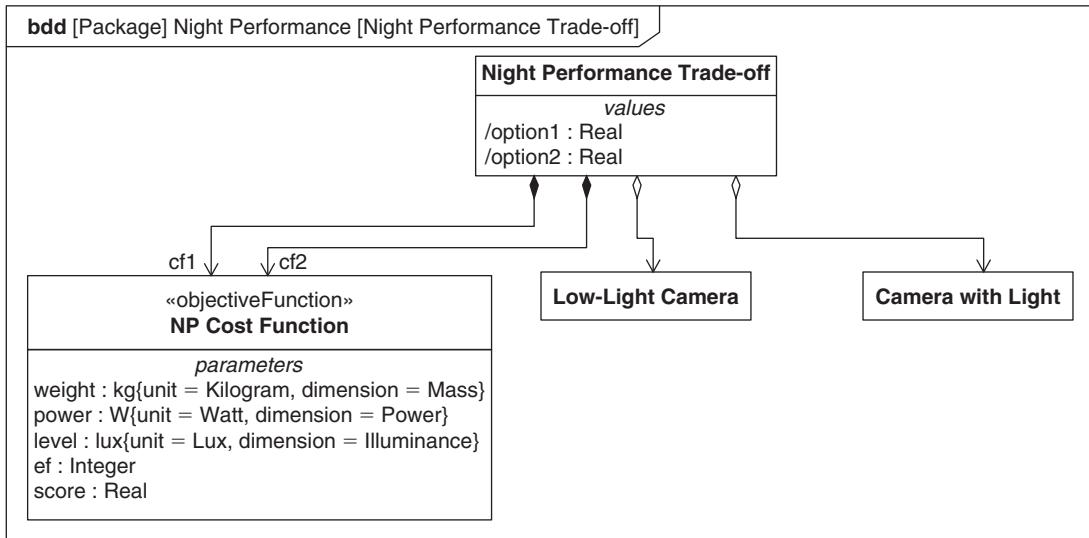


FIGURE 7.16

Two variants of a camera for handling low-light conditions.

**FIGURE 7.17**

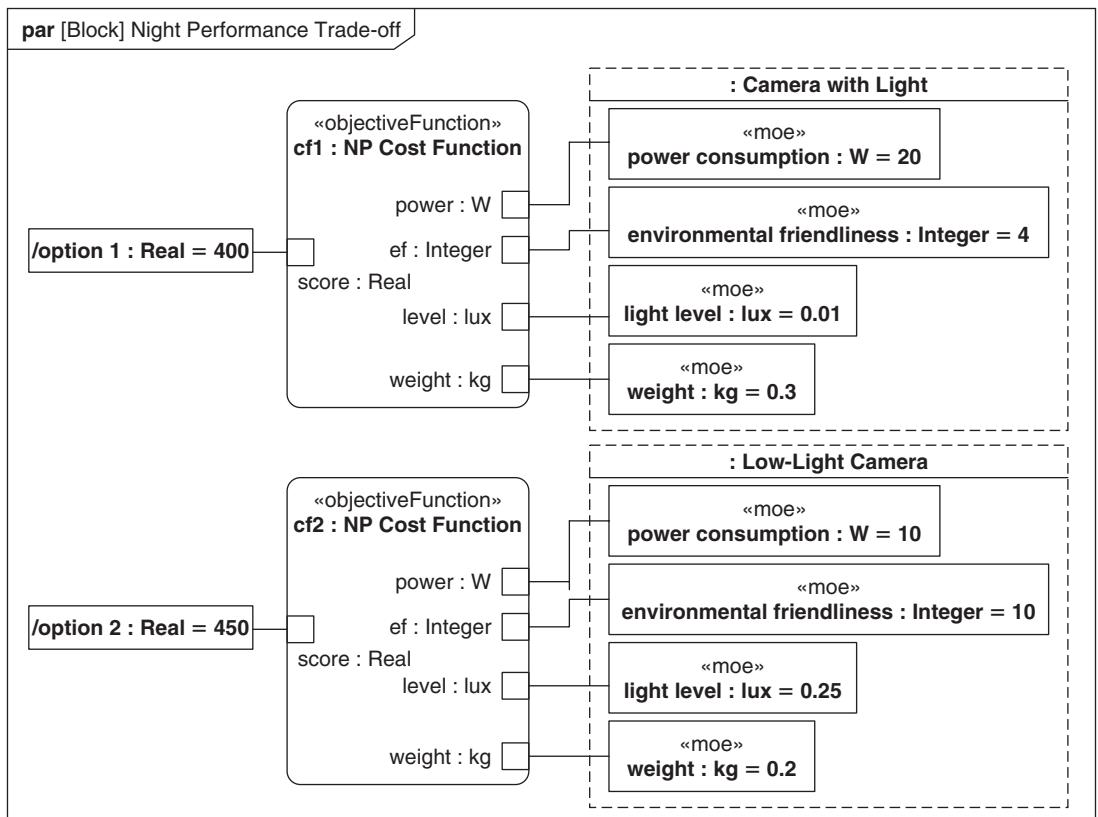
A trade study represented as an analysis context.

Four relevant measures of effectiveness, indicated by the keyword «moe», are used to conduct the trade studies.

A trade study is typically described as a type of analysis context, which references the blocks that represent the different alternatives. It also contains constraint properties for the objective function (or functions) to be used to evaluate the alternatives, and a means to record the results of the evaluation, typically value properties that capture the score for each alternative.

Figure 7.17 shows the definition of *Night Performance Trade-off*—a trade study for evaluating the nighttime performance of two camera variants. As indicated by its associations, *Night Performance Trade-off* contains two constraint properties, both typed by objective function *NP Cost Function* and two reference properties, one typed by *Low-Light Camera* and the other by *Camera with Light*. It is intended in the analysis that the equations are solved for *option 1* and *option 2* and so they are shown as derived. The bindings between the various properties of *Night Performance Trade-off* are shown in Figure 7.18.

Figure 7.18 shows the internal bindings of the trade-off study *Night Performance Trade-off*. One use of the objective function *NP Cost Function*, *cf1*, is bound to the value properties of the *Low-Light Camera*, and the other, *cf2*, is bound to the *Camera with Light*. The *score* parameters of *cf1* and *cf2* are bound to two value properties of the context called *option 1* and *option 2*, which are the dependent variables in this particular analysis. In this case, using the values provided in Figure 7.16 for the measures of effectiveness of the two solutions, the scores are 400 for *option 1* and 450 for *option 2*, indicating that the *Low-Light Camera* is the preferred solution.

**FIGURE 7.18**

Trade-off results between the two low-light camera variants.

7.12 Summary

Constraint blocks are used to model constraints on the properties of blocks to support engineering analyses, such as performance, reliability, and mass properties analysis. The following are key aspects of constraint blocks and their usages.

- SysML includes the concept of a constraint that can correspond to any mathematical or logical expression, including time-varying expressions and differential equations. SysML does not specify a constraint language but enables the language to be specified as part of the definition of the constraint.
- SysML provides the ability to encapsulate a constraint in a constraint block so that it can be reused and bound with other constraints to represent complex sets of equations. A constraint block defines a set of parameters related to each other by the constraint expression. Parameters may have types, units, dimensions, and probability distributions. The block definition diagram is used to define constraint

blocks and their interrelationships. In particular, a composite association can be used to compose constraint blocks to create complex equations. Constraint blocks can be defined in model libraries to facilitate specific types of analysis (performance, mass properties, thermal, etc.). Constraint blocks can be used by blocks to constrain the values of their properties.

- Constraint properties are usages of constraint blocks. The parametric diagram shows how constraint properties bind to one another and to the value properties of blocks through their parameters. They are bound using binding connectors that express equality between the values of the parameters or properties at their ends. The specific values needed to support the evaluation of the constraints for a block are typically specified by a configuration of that block.
- An analysis context is a block that provides the context for a system or component that is subject to analysis. The analysis context is composed of the constraint blocks that correspond to the analysis model and references the system being analyzed. A parametric diagram, whose frame represents the analysis context, is used to bind the relevant properties of the block and the parameters of the analysis model. The analysis context can be passed to an engineering analysis tool to perform the computational analysis, and the analytic results can be provided back as properties of the analysis context.
- A common and useful form of analysis used by systems engineers is the trade study, which is used to compare alternative solutions for a given problem based on some criteria. A moe (short for measure of effectiveness) is used to define a property that needs to be evaluated in a trade study and a specialization of constraint block, called an objective function, is used to define how the solutions are evaluated.

7.13 Questions

1. What is the diagram kind of a parametric diagram?
2. If a constraint parameter is ordered, what does that imply about its values?
3. If a constraint parameter is unique, what does that imply about its values?
4. How are constraint parameters represented on a block definition diagram?
5. How is the composition of constraints represented on a block definition diagram?
6. How are constraint properties represented on a parametric diagram?
7. How are constraint parameters represented on a parametric diagram?
8. What are the semantics of a binding connector?
9. How can constraint blocks be used to constrain the value properties of blocks?
10. A block “Gas” has two value properties, “pressure” and “volume,” that vary inversely with respect to each other. Create an appropriate constraint block to represent the relationship and use it in a parametric diagram for “Gas” to constrain “pressure” and “volume.”
11. What are the two approaches to specifying parametric models that include time-varying properties?

12. How are composite associations and reference associations typically used in an analysis context?
13. What is a measure of effectiveness and what is it used for?
14. What is an objective function and how is it represented on a block definition diagram and a parametric diagram?

Discussion Topics

Under what circumstances is it useful or necessary to use derived properties or parameters in parametric models?

What are the relative merits of making parametric equations part of the definition of blocks, or applying an externally defined parametric model to an existing block?

Modeling Flow-Based Behavior with Activities

8

This chapter describes concepts needed to model behavior in terms of the flow of inputs, outputs, and control using an activity diagram. An activity diagram is similar to a traditional functional flow diagram with many additional features to precisely specify a behavior.

8.1 Overview

In SysML, an activity is a formalism for describing behavior that specifies the transformation of inputs to outputs through a controlled sequence of actions. The activity diagram is the primary representation for modeling flow-based behavior and is analogous to the functional flow diagram that has been widely used for modeling systems. Activities provide enhanced capabilities over traditional functional flow diagrams, such as the inherent capability to express their relationship to the structural aspects of the system (e.g., blocks, parts), and the ability to model continuous flow behaviors. The semantics of activities are precise enough to enable them to be mapped to executable constructs in an execution environment. However, the mapping itself has not been standardized as of this time, although there are current efforts under way to do this.

Actions are the building blocks of activities and describe how activities execute. Each action can accept inputs and produce outputs, called tokens, on their pins. These tokens can correspond to anything that flows such as information or a physical item (e.g., water). Although actions are the leaf or atomic level of activity behavior, a certain class of actions, termed call actions, can invoke other activities that can be further decomposed into other actions. In this way, call actions can be used to compose activities into activity hierarchies.

The concept of object flow specifies how the input and output items transformed by an activity flow between its constituent actions. Object flows can connect the output pin of one action to the input pin of another action to enable the passage of tokens. Flows can be discrete or continuous, where continuous flow represents the situation when the time between tokens is effectively zero. Complex routing of object tokens between actions can be specified by control nodes.

The concept of control flow provides additional constraints on when, and in which order, the actions within an activity will execute. A control token on an incoming control flow enables an action to start execution, and a control token is offered on an outgoing control flow when an action completes its execution. When a control flow connects one action to another, the action at the target end of the control flow cannot start until the source action has completed. Control nodes, such as join, fork, decision, merge, initial, and final nodes, can be used to control the routing of control tokens to further specify the sequence of actions.

The sending and receiving of signals is one mechanism for communicating between activities executing in the context of different blocks, and for handling events such as timeouts. Signals are sometimes used as an external control input to initiate an action within an activity that has already started. Activities also include more advanced modeling concepts, such as extensions to flow semantics to deal with interrupts, flow rates, and probabilities.

Activities can depict behavior without explicit reference to which elements are responsible for performing the behavior. Alternatively, activities can depict behavior performed by specific blocks or parts, such as a system or its components. SysML provides several mechanisms to relate activities to the blocks that perform them. Activity partitions allow the modeler to partition actions in an activity according to the blocks that have responsibility for executing them.

An activity may be specified as the main behavior of a block that describes how inputs and outputs of the block are processed. The activity can also be specified as the method for an operation of the block that is invoked as a result of a service request for that operation. When the behavior of a block is specified using a state machine, activities are often used to describe what happens when the state machine transitions between states, or what happens when in a state.

Other traditional systems engineering functional representations are also supported in SysML. Activities can be represented on block definition diagrams to show activity hierarchies similar to functional hierarchies. Activity diagrams can also be used to represent Enhanced Functional Flow Block Diagrams (EFFBDs).

8.2 The Activity Diagram

The principal diagram used to describe activities is called an **activity diagram**. On an activity diagram, the frame represents an activity, and the content of the diagram defines the actions along with the flow of input/output and control. The frame label for an activity diagram has the following form:

act [Activity] activity name [diagram name]

The diagram kind for an activity diagram is designated as **act** (for activity). The frame always represents an activity, and therefore may be elided. The *activity name* is the name of the represented activity, and the *diagram name* is user defined and is intended to describe the purpose of the diagram. Figure 8.1 shows an example activity diagram.

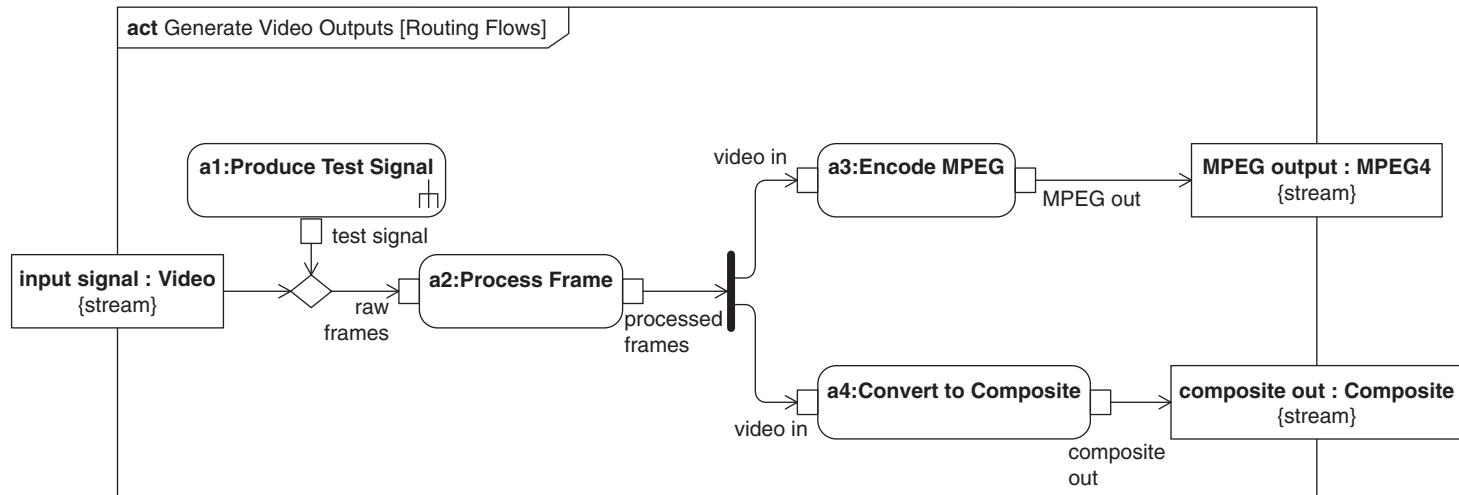
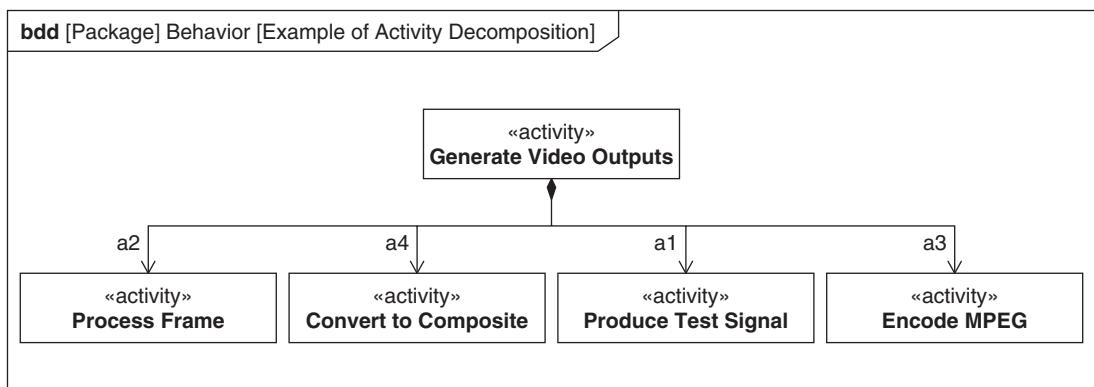


FIGURE 8.1

An example activity diagram.

**FIGURE 8.2**

An example of an activity hierarchy in a block definition diagram.

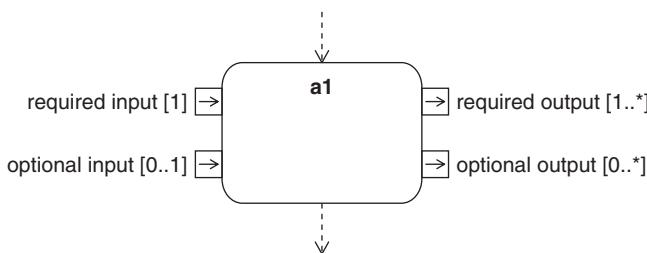
Figure 8.1 shows an activity diagram for the activity *Generate Video Outputs* and some of the basic activity diagram symbols. *Generate Video Outputs* includes call actions that invoke other activities, such as the action *a2* that invokes the *Process Frame* activity. Actions have input and output pins, shown as small rectangles, to accept tokens that may represent units of information, matter, or energy. Pins are connected using object flows. Actions can also be connected with control flows although none are shown in this figure. The notation for activity diagrams is shown in the Appendix, Tables A.11 through A.14.

Figure 8.2 shows an example of an activity hierarchy that can be represented on a block definition diagram. The activity hierarchy includes an alternative view of the actions and invoked activities included in the activity *Generate Video Outputs* shown in Figure 8.1; however, it does not include the flows between the actions and other activity constructs such as control nodes. The structure of the hierarchy is shown using composite associations from the parent activity to other activities such as *Process Frame*. The role names on the associations, such as *a2*, correspond to the names of the actions used to invoke the activities. The notation required to show activity hierarchies on block definition diagrams is described in the Appendix, Table A.7.

8.3 Actions—The Foundation of Activities

As described previously, an activity decomposes into a set of **actions** that describe how the activity executes and transforms its inputs to outputs. There are a number of different categories of action in SysML described during this chapter, but this section provides a summary of the fundamental behavior of all actions. SysML activities are based on token-flow semantics related to Petri-Nets [31, 32]. **Tokens** correspond to values of inputs, outputs, and control that flow from one action to another.

An action processes tokens placed on its **pins**. Tokens on input pins are consumed, processed by the action, and placed on output pins for other actions to

**FIGURE 8.3**

An action with input and output pins and input and output control flow.

accept. Each pin has a multiplicity that describes the minimum and maximum number of tokens that the action consumes or produces in any one execution. A pin acts as a buffer where input and output tokens to an action can be stored prior to or during execution. If a pin has a minimum multiplicity of zero, then it is said to be optional, otherwise it is said to be required.

The action symbol varies depending on the type of action, but typically it is a rectangle with round corners. The pin symbols are small boxes flush with the outside surface of the action symbol and may contain arrows indicating whether the pin is an input or output. Typically once a pin is connected to a flow and the direction becomes obvious, arrow notation is discarded.

Figure 8.3 shows a typical action, called *a1*, with a set of input and output pins. One input pin and one output pin are required; that is, they have a lower multiplicity bound greater than zero. The other two pins are optional; that is, they have a lower multiplicity bound of zero. The action also has one incoming control flow and one outgoing control flow; see Section 8.6 for a detailed description of control flows. An action will begin execution when tokens are available on all its required inputs, including its control inputs as described next.

The following rules summarize the requirements for actions to begin and end:

- The first requirement is that the action's owning activity must be executing.
- Given that, the basic rules for whether an action can execute are as follows:
 - The number of tokens available at each required input pin is equal to or greater than its lower multiplicity bound.
 - A token is available on each of the action's incoming control flows.
- Once these prerequisites are met, the action will start executing and the tokens at all its input pins are available for consumption.
- For an action to terminate, the number of tokens it has made available at each required output pin must be equal to or greater than its lower multiplicity bound.
- Once the action has terminated, the tokens at all its output pins are available to other actions connected to those pins. In addition, a control token is placed on each outgoing control flow.
- Regardless of whether an action is currently executing or not, it is terminated when its owning activity terminates.

The preceding paragraphs describe the basic semantics of actions, but the following additional semantics are discussed later in this chapter:

- Different types of actions perform different functions, and some, particularly the call actions discussed in Section 8.4.2, introduce additional semantics such as streaming.
- Object and control tokens are routed using control nodes that can buffer, copy, and remove tokens. For more information, see Section 8.5 for object flow and Section 8.6 for control flow. SysML allows control tokens to disable as well as enable actions, but actions need control pins to support this, as described in Section 8.6.2.
- SysML also includes continuous flows that are addressed in Section 8.8.2.
- Actions can be contained inside an interruptible region, which when interrupted will cause its constituent actions to terminate immediately. Interruptible regions are described in Section 8.8.1.

The relationship between the semantics of blocks and activities is discussed in Section 8.9.

8.4 The Basics of Modeling Activities

Activities provide the context in which actions execute. Activities are used, and more important reused, through call actions. Call actions allow the composition of activities into arbitrarily deep hierarchies that allows an activity model to scale from descriptions of simple functions through to very complex algorithms and processes.

8.4.1 Specifying Input and Output Parameters for an Activity

An activity may have multiple inputs and multiple outputs called **parameters**. Note that these parameters are not the same as the constraint parameters described in Chapter 7. Each parameter may have a type such as a value type or block. Value types range from simple integers to complex vectors and may have corresponding units and dimensions. Parameters can also be typed by a block that may correspond to a structural entity such as water flow or an automobile part flowing through an assembly line. Parameters have a direction that may be in or out or both.

Parameters also have a multiplicity that indicates how many tokens for this parameter can be consumed as input or produced as output by each execution of the activity. The lower bound of the multiplicity indicates the minimum number of tokens that must be consumed or produced by each execution. As with pins, if the lower bound is greater than zero, then the parameter is said to be **required**; otherwise, it is said to be **optional**. The upper bound of the multiplicity specifies the maximum number of tokens that may be consumed or produced by each execution of the activity.

Activity parameters are represented on an activity diagram using **activity parameter nodes**. During execution an activity parameter node holds tokens that represent the arguments of its corresponding parameter. An activity parameter node is related to exactly one of the activity's parameters and must have the same type as its corresponding parameter. If a parameter is marked as inout, then it needs at least two activity parameter nodes associated with it, one for input and the other for output.

A 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 tokens when the activity first starts executing, 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. Streaming parameters add significant flexibility for representing certain types of behavior. Parameters have a number of other characteristics described later in this chapter.

Activity parameter node symbols are rectangles that straddle the activity frame boundary. Each symbol contains a name string, composed of the node name, parameter type, and parameter multiplicity, thus:

parameter name: parameter type[multiplicity]

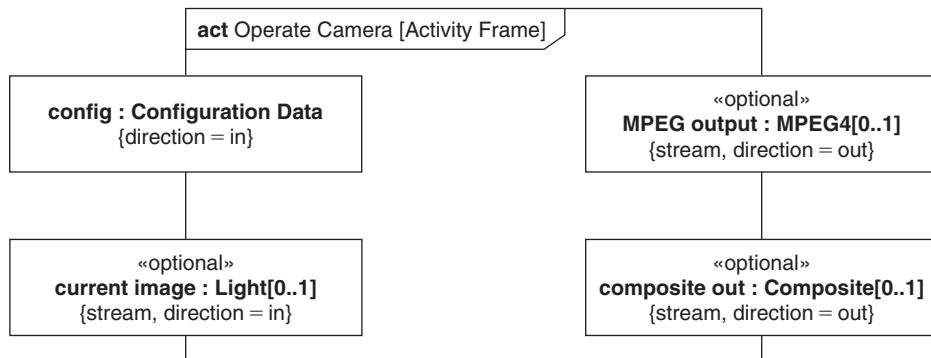
If no multiplicity is shown, then the multiplicity “1..1” is assumed. The node’s name is typically the same as the name of its related parameter. An optional parameter is shown by the keyword «optional» above the name string in the activity parameter node. Conversely, the absence of the keyword «optional» indicates that the parameter is required.

There is no specific graphical notation to indicate the direction of an activity parameter node, although the direction can be shown using property notation. Some methodologies suggest that input parameters are shown on the left of the activity and outputs on the right. Once activity parameter nodes have been connected by flows to nodes inside the activity, the parameter direction is explicit.

Additional characteristics of the parameter, such as its direction and whether it is streaming, are shown in braces either inside the parameter node symbol after the name string or floating close to the symbol.

Figure 8.4 shows the inputs and outputs of the *Operate Camera* activity that is the main behavior of the camera. As can be seen from the notation in the parameter nodes, *Light* from the camera’s environment is available as input using the *current image* parameter and two types of video signal are produced as outputs using the *composite out* and *MPEG output* parameters. The input parameter, *config*, is used to provide configuration data to the camera when it starts up.

The activity consumes and produces a stream of inputs and outputs as it executes, as indicated by the {stream} annotation on the main parameter nodes. The other parameter, *config*, is not streaming because it has a single value that is read when the activity starts. As stated earlier, when the multiplicity is not shown, for instance, on parameter *config*, this indicates a lower bound and upper bound of

**FIGURE 8.4**

Specifying an activity using a frame on an activity diagram.

one. The other parameters are streaming and there is no requirement to consume or produce tokens, so they are shown as «optional».

8.4.2 Composing Activities Using Call Behavior Actions

A significant type of action is a **call behavior action**, which invokes a behavior when it executes. The call behavior action owns a set of pins that must match, in number and type, the parameters of the invoked behavior. The called behavior is assumed to be an activity in this chapter, although it can be other types of SysML behavior.

A call behavior action has a pin for each parameter of the called behavior and the characteristics of those pins must match the multiplicity and type of their corresponding parameters on the invoked behavior. If an activity parameter on the invoked activity is streaming, then the corresponding pin on the call behavior action has streaming semantics. As stated earlier, tokens on normal or nonstreaming pins, such as those shown in Figure 8.3, can only be available to the action for processing at the start of (in the case of input pins) or at the end of (in the case of output pins) the action execution. By comparison, tokens continue to be available through streaming pins while their owning action is executing, although the number of tokens consumed or produced by each execution is still governed by its upper and lower multiplicity bounds.

The call behavior action symbol is a round-cornered box containing a name string, with the name of the action and the name of the called behavior (e.g., activity), separated by a colon as follows: action name : behavior name. The default notation is to include only the action name and not the colon. When the action is shown but is not named, the colon is included to differentiate this notation from the default.

Different naming philosophies exist for call behavior actions. Names are almost always used to differentiate two calls to the same activity. They are also often used to provide a name for an action used in an allocation (see Chapter 13 on allocations for more detail).

The name string of a pin on a call behavior action has the same form as the name string for an activity parameter node symbol, but floats outside the pin symbol. The name may include characteristics, such as streaming, of the corresponding parameter. A rake symbol in the bottom right corner of a call behavior action symbol indicates that the activity being invoked is described on another diagram.

To transform light into video signals, the *Operate Camera* activity invokes other activities that perform various subtasks using call behavior actions, as shown in Figure 8.5. The action name strings take the form “: Activity Name,” indicating that actions do not have names. This figure shows just activity parameter nodes and actions with their inputs and outputs. Figure 8.6 shows how their input and output pins are connected. Note that the types of the pins have been elided here to reduce clutter.

All the invoked activities consume and produce streams of input and output tokens, as indicated by the {stream} annotation on the pins of the actions. *Collect Images* is an analog process performed by the camera lens. *Capture Video* is where the images from the outside world are digitized to a form of video output. *Generate Video Outputs* takes the internal video stream and produces MPEG and composite outputs for transmission to the camera’s users.

8.5 Using Object Flows to Describe the Flow of Items between Actions

Object flows are used to route input/output tokens that may represent information and/or physical items between object nodes. Activity parameter nodes and pins are two examples of object nodes. Object flows can be used to route items from the parameters nodes on the boundary of an activity to/from the pins on its constituent actions, or to connect pins directly to other pins. In all cases, the direction of the object flow must be compatible with the direction of the object nodes at its ends (i.e., in or out), and the types of the object nodes on both ends of the object flow must be compatible with each other.

An object flow is shown as a line connecting the source of the flow to the destination of the flow, with an arrowhead at the destination. When an object flow is between two pins that have the same characteristics, an alternative notation can be used where the pin symbols on the actions at both ends of the object flow are elided and replaced by a single rectangular symbol called an object node symbol. In this case, the object flow connects the source action to the object node with an arrowhead on the object node end, and then connects the object node to the destination action, with an arrowhead at the destination end. The object node symbol can have the same annotations as a pin symbol.

In Figure 8.6, the subactivities of *Operate Camera* shown in Figure 8.5 are now interconnected by object flows to establish the flow from light entering the camera to the output of video images in the two required formats. The incoming light on the parameter called *current image* flows to the *Collect Images* activity; its output, *captured image*, is the input to *Capture Video* (note the use of the

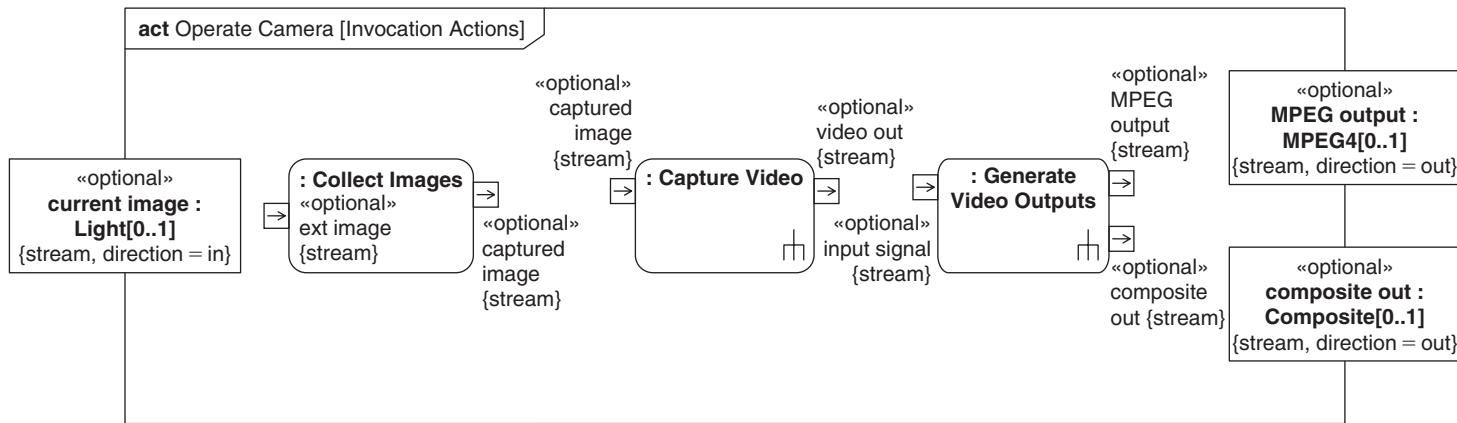


FIGURE 8.5

Invocation actions on an activity diagram.

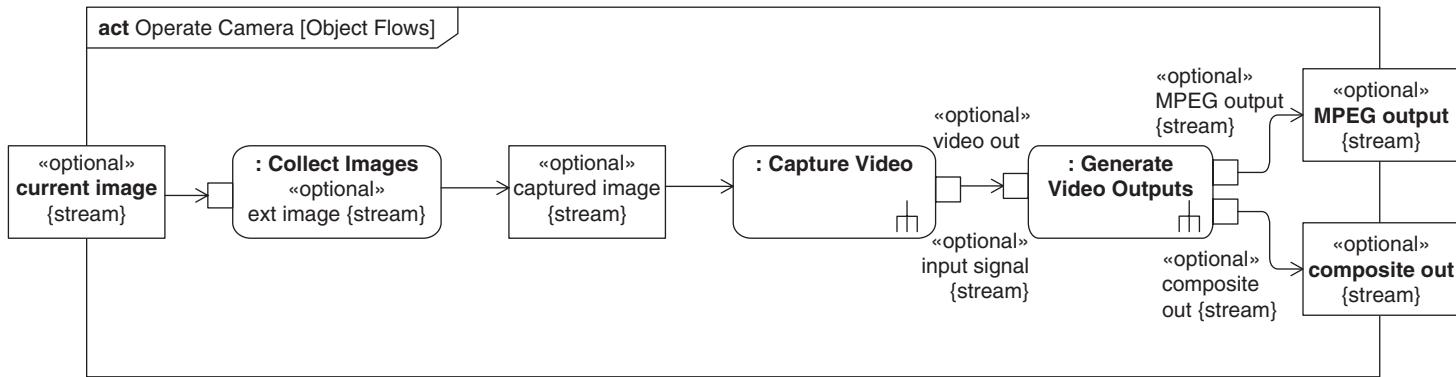


FIGURE 8.6

Connecting pins and parameters using object flows.

“object node” notation). *Capture Video* produces video images, via its *video out* pin, which in turn becomes the input for *Generate Video Outputs*. *Generate Video Outputs* converts its input video signal into MPEG and composite outputs that are then routed to corresponding output parameter nodes of *Operate Camera*.

In Figure 8.6, the names of the actions have been elided, which is indicated by the absence of a colon in the name string of the action symbols. See Figure 8.8 later for an example where the actions are named.

8.5.1 Routing Object Flows

There are many situations where simply connecting object nodes using object flows does not allow an adequate description of the flow of tokens through the activity. SysML provides a number of mechanisms for more sophisticated expressions of flow control. Each object flow may have a guard expression that specifies a rule to govern which tokens are valid for the object flow.

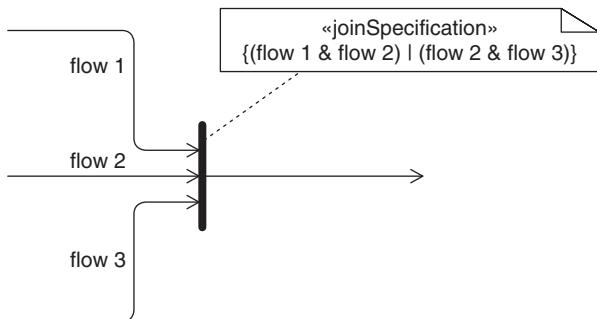
In addition, there are several constructs in SysML activities that provide more sophisticated flow mechanisms, including:

- A **fork node** has one input flow and more than one output flow—it replicates every input token it receives onto each of its output flows. The tokens on each output flow may be handled independently and concurrently. Note that the tokens merely represent the items flowing, and the replication of tokens does not imply that the represented items are replicated. In particular, if the represented item is physical, replication of that physical object may not even be possible.
- A **join node** has one output flow and more than one input flow—its default behavior for object flows is to produce output tokens only when an input token is available on each input flow. Once this occurs, it places all input object tokens on the output flow. This has the important characteristic of synchronizing the flow of tokens from many sources. Note that this applies to object flow, but the handling of control tokens is different, as described in Section 8.6.

The default behavior of join nodes can be overridden by providing a join specification that specifies a logical expression for matching token arrival on different flows.

Figure 8.7 shows an example of a join specification. The join node has three input flows—*flow 1*, *flow 2*, and *flow 3*—and the join specification states that output tokens are produced if input tokens are received on both *flow 1* and *flow 2*, or on both *flow 2* and *flow 3*. The expression uses the names of flows, so the flows must be named in this situation. Another use of flow names is to support flow allocation (see Chapter 13).

- A **decision node** has one input and more than one output flow—an input token can only traverse one output flow. The output flow is typically established by placing mutually exclusive guards on all outgoing flows and offering the token to the flow whose guard expression is satisfied. The guard expression “else” can be used on one of the node’s outgoing flows to ensure that there is always one flow that can accept a token. If more than one outgoing object flow can

**FIGURE 8.7**

Example of a join specification.

accept the token, then it cannot be determined which of the flows will receive the token.

A decision node can have an accompanying decision input behavior that is used to evaluate each incoming object token and whose result can be used in guard expressions.

- A **merge node** has one output and more than one input flow—it routes each input token received on any input flow to its output flow. Unlike a join node, a merge node does not require tokens on all its input flows before offering them on its output flow. Rather, it offers tokens on its output flow as soon as it receives them.

Fork and join symbols are shown as solid bars, typically aligned either horizontally or vertically. Decision and merge symbols are shown as diamonds. Where forks and joins, or decisions and merges, are adjacent (i.e., would be connected by just a flow with no guards), they can be shown as a single symbol with the inputs and outputs of both connected to that symbol. Figure 8.12, later in the chapter, contains an example of this. Join specifications and decision input behaviors are shown in notes attached to the relevant node.

In Figure 8.8, the activity *Generate Video Outputs* accepts an input video signal and outputs it in appropriate formats for external use, in this case, *Composite video* and *MPEG4*. The *Produce Test Signal* activity allows *Generate Video Outputs* to generate a test signal if desired. See the specification of *Produce Test Signal* later in Figure 8.14 to see how the activity knows when to generate the signal. The test signal, when generated, is merged into the stream of video frames using a merge node, and this merged stream is then converted into video frames by *Process Frame*. Note that if tokens are produced on both the *input signal* parameter node and the *test signal* pin, then they will be interleaved into the *raw frames* pin by the merge node. In this case that is the desired behavior, but if not, then additional control would be needed to ensure that incoming token streams were exclusive.

Once processed, the tokens representing the processed frames are then forked and offered to two separate actions: *Convert to Composite* that produces the *composite out* output and *Encode MPEG* that produces the *MPEG* output. These two actions can continue in parallel, each consuming tokens representing frames and performing a suitable translation. Note that the fork node does not imply that

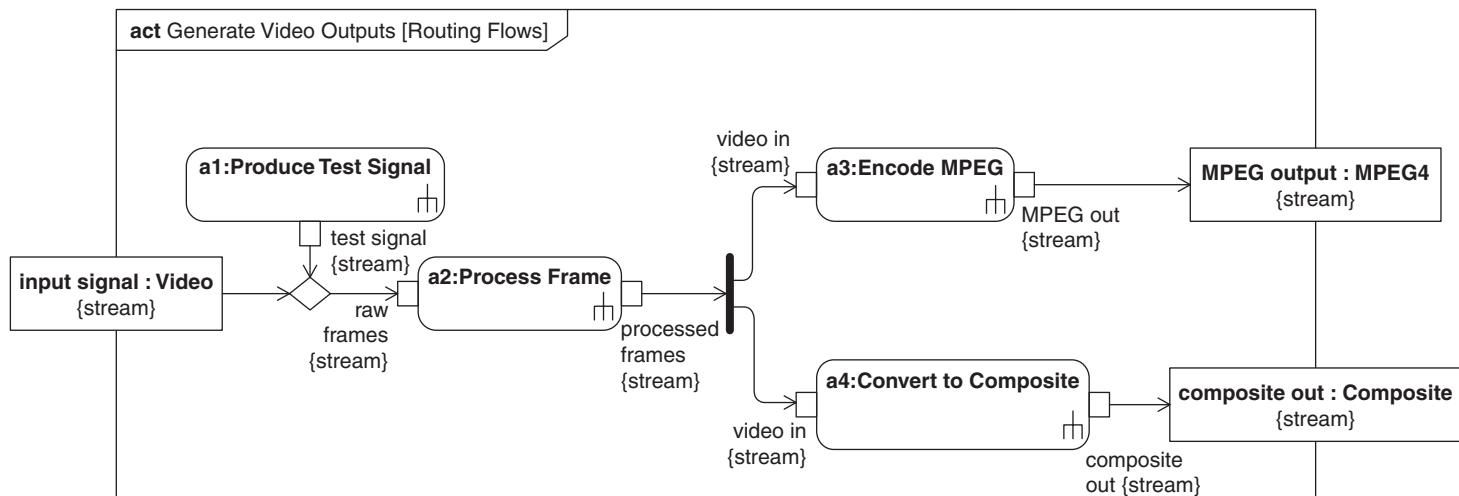


FIGURE 8.8

Routing object flows between invocations.

the frame data are copied (although they may be), but merely that both *Encode MPEG* and *Convert to Composite* have access to the data via their input tokens.

In this example, the name strings of the call behavior actions include both the action name and activity name, when arguably the actions need not be named. This helps to demonstrate the mapping from activities on this activity diagram to the same activities represented on the block definition diagram in Figure 8.25 in Section 8.10.1.

8.5.2 Routing Object Flows from Parameter Sets

The parameters of an activity can be grouped together into **parameter sets**, where a parameter set must have either all input or all output parameters as members. When an activity that has input parameter sets is invoked, the parameter nodes corresponding to at most one input parameter set can contain tokens. When an activity that has output parameter sets has completed, the parameter nodes corresponding to at most one output parameter set can contain tokens. A given parameter may be a member of multiple parameter sets.

Each set of parameters is shown by a rectangle, on the outer boundary of the activity, that partially encloses the set of parameter nodes that correspond to parameters in the set. These rectangles can overlap to reflect the overlapping membership of parameter sets.

Figure 8.9 shows an activity called *Request Camera Status* with two distinct sets of outputs. When presented with a *camera number* as input, *Request Camera Status* will return an *error* and a *diagnostic* if there is a problem with the camera, or a *power status* and *current mode* if the camera is operational.

If an invoked activity has parameter sets, then the groupings of pins corresponding to the different parameter sets are shown on the call behavior action, using similar notation to parameter sets on activities.

Figure 8.10 shows the object flow for an activity *Handle Status Request* that reads a *camera id* and writes a *camera status*. It invokes *Request Camera Status* with a *camera number* and expects one of two sets of outputs that correspond

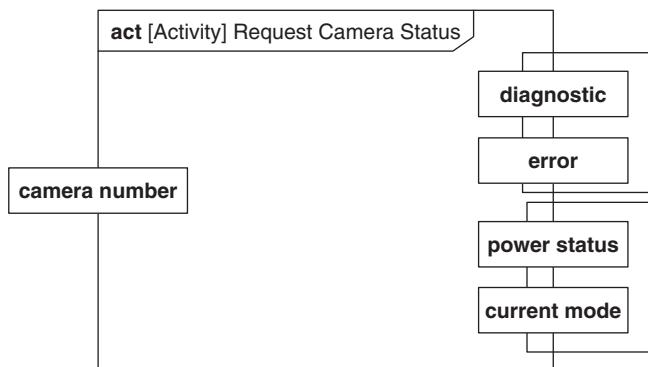
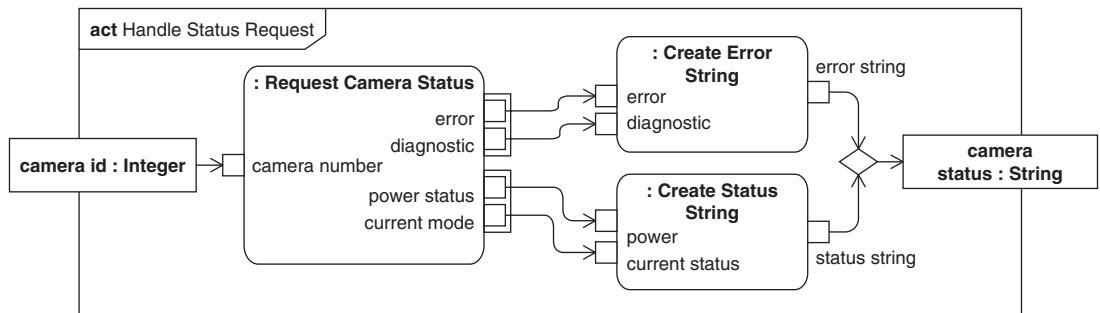


FIGURE 8.9

An activity with parameter sets.

**FIGURE 8.10**

Invoking an activity with parameters sets.

to two parameter sets: an *error* and a *diagnostic*, or a *power status* and *current mode*. These two sets of outputs are used by two different string-formatting functions, *Create Error String* and *Create Status String*. Whichever formatting function receives inputs produces an output string that is then conveyed via a merge node to the *camera status* output parameter node.

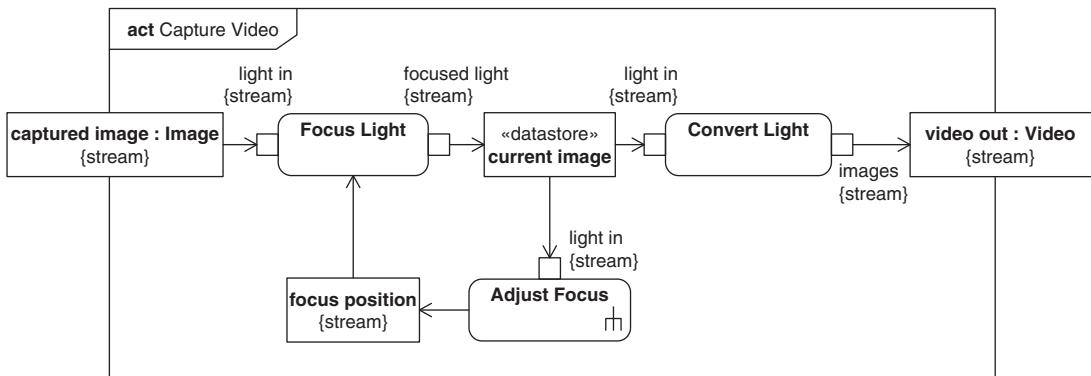
8.5.3 Buffers and Data Stores

Pins and activity parameter nodes are the two most common types of object node, but there are cases where additional constructs are required. A **central buffer node** provides a store for object tokens outside of pins and parameter nodes. Tokens flow into a central buffer node and are stored there until they flow out again. It is needed when there are multiple producers and consumers of a single-buffered stream of tokens at the same time; pins and activity parameter nodes have either a single producer or single consumer.

Sometimes activities require the same object tokens to be stored for access by a number of actions during execution. A type of object node called a **data store node** can be used for this. Unlike a central buffer node, a data store node provides a copy of a stored token rather than the original. When an input token represents an object that is already in the store, it overwrites the previous token. Data stores can provide tokens when a receiving action is enabled, thus supporting the pull semantics of traditional flow charts.

Data store nodes and central buffer nodes only store tokens while their parent activity is executing. If the values of the tokens need more permanent storage, then a property should be used. There are primitive actions, described in Section 8.12.1, that can be used to read and write properties.

Both central buffer nodes and data store nodes are represented by a rectangle with a name string, with the keywords «centralBuffer» and «datastore» above the name string. Their names have the same form as pins, except without multiplicity: buffer or store name : buffer or store type. An example of a central buffer node is shown in Figure 8.19 in Section 8.8.4.

**FIGURE 8.11**

Using a data store node to capture incoming light.

Figure 8.11 describes the internal behavior of the *Capture Video* activity. Light entering the camera lens is focused by the activity *Focus Light*, which produces an image that is stored in a data store node called *current image*. The image stored in *current image* is then used by two other activities: *Convert Light* that samples the images to create video frames and *Adjust Focus* that analyzes the current image for sharpness and provides a *focus position* to *Focus Light*. The use of a data store node here facilitates the transition between the analog nature of the incoming light from the lens and the digital nature of the video stream. (See Figure 8.17 in the Flow Rates subsection of 8.8.2 for an enhanced version of this diagram, including flow rate information.) In this case, the data store may be allocated to the focal plane array of the camera (see Chapter 13 for a description of allocation).

The object node called *focus position* is input to *Focus Light*, whereas *Convert Light* and *Adjust Focus* receive their input from a data store node. The notation for the object node representation of flows and the representation of buffer nodes is quite similar, but buffer nodes always have the keyword «datastore» or «centralBuffer» above their name.

Sections 8.8.2 and 8.8.3 discuss other mechanisms to specify the flow of tokens through data store and central buffers nodes, as well as other object nodes.

8.6 Using Control Flows to Specify the Order of Action Execution

As mentioned previously, there are control semantics associated with object flow, such as when an action waits for the minimum required number of tokens on all input pins before proceeding with its execution. However, sometimes the availability of object tokens on required pins is not enough to specify all the execution constraints on an action, in which case **control flows** are available to provide further control using control tokens. Although object flows have been described first in this chapter, the design of an activity need not necessarily start with the specification of

object flows. In traditional flow charts, it is often the control flows that are established first and the routing of objects later.

In addition to any execution prerequisites established by required pins, an action also cannot start execution until it receives a control token on all input control flows. When an action has completed its execution, it places control tokens on all outgoing control flows. The sequencing of actions can thus be controlled by the flow of control tokens between actions using control flows.

An action can have more than one control flow input. This has the same semantics as connecting the action to the outgoing control flow of a join with multiple incoming control flows. Similarly, if an action has more than one control flow output, it can be modeled by connecting the action via an outgoing control flow to a fork with multiple control flow outputs. As will be seen in Section 8.6.2, control tokens can be used to disable actions as well as enabling them.

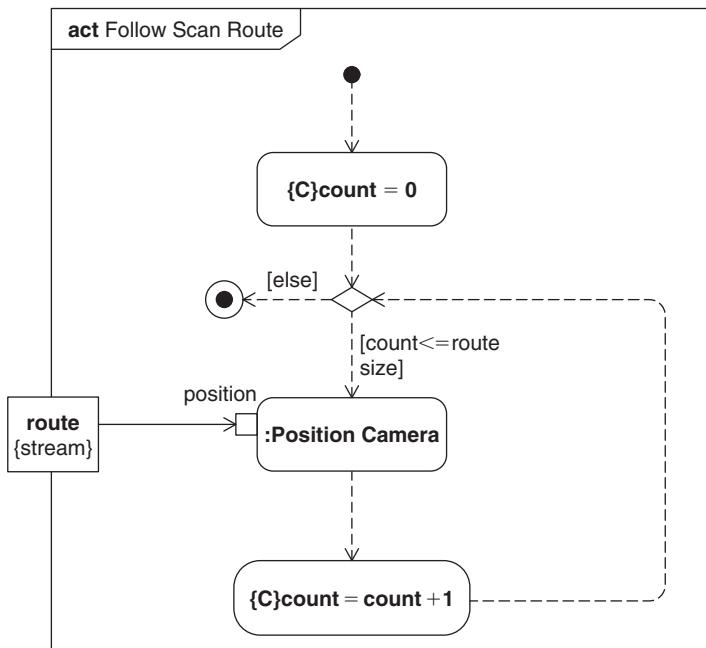
8.6.1 Depicting Control Logic with Control Nodes

All the constructs used to route object flows can also be used to route control flows to represent control logic. A join node has special semantics with respect to control tokens. Even if it consumes multiple control tokens, it emits only one control token once its join specification is satisfied. Join nodes can also consume a mixture of control and object tokens, in which case once all the required tokens have been offered at the join node, all the object tokens are offered on the outgoing flow along with one control token. In addition to the constructs described in Section 8.5.1, there are some special constructs that provide additional control logic:

- *Initial node*—when an activity starts executing, a control token is placed on each initial node in the activity. The token can then trigger the execution of an action via an outgoing control flow. Note that although an initial node can have multiple outgoing flows, a control token will only be placed on one. Typically guards are used where there are multiple flows in order to ensure that only one is valid, but if this is not the case, then the choice of flow is arbitrary.
- *Activity final node*—when a control or object token reaches an activity final node during the execution of an activity, the execution terminates.
- *Flow final node*—control or object tokens received at a flow final node are consumed but have no effect on the execution of the enclosing activity. Typically they are used to terminate a particular sequence of actions without terminating an activity. This may occur when processing in an activity is proceeding concurrently, after a fork node and only one processing route terminates the activity.

A control flow can be represented either by using a solid line with an arrowhead at the destination end like an object flow or, to more clearly distinguish it from object flow, by using a dashed line with an arrowhead at the destination end.

An initial node symbol is shown as a small solid black circle. The activity final node symbol is shown as a “bulls-eye,” and the flow final node symbol is a hollow circle containing a crosshair symbol, rotated 45° from the horizontal/vertical axis. Examples of the initial and activity final nodes are shown in Figure 8.12. Figure 8.20 in Section 8.9.1 contains an example of a flow final node.

**FIGURE 8.12**

Control flow in activities.

The console software provides the capability to drive a camera through a pre-set scan route, as shown in Figure 8.12. The activity *Follow Scan Route* will follow a route, that is a set of positions, for the camera defined in terms of pan-and-tilt angles. It has one input parameter, the *route* as a fixed-length stream of positions with size *route size*. When started, the activity resets its *count* property, then iterates over all points in the route—incrementing *count* for every point—and finally terminates when the guard `[count <= route size]` is false, indicating that the last point in the route is reached. The *Position Camera* activity is invoked for each position token offered on the *route* parameter. Control flows dictate the order in which the activity executes.

Note that in this case there is a combined merge and decision symbol that accepts two input control flows and has two output control flows: one leads to an activity final node and the other leads into another iteration of the algorithm. The property *count* is initialized and incremented using actions `count = 0` and `count = count + 1`; these are opaque actions; that is, their function is expressed in some language (in this case the C programming language). As with constraints, the language used to specify the action can be added in braces before the expression.

8.6.2 Using Control Operators to Enable and Disable Actions

An action with nonstreaming inputs and outputs typically starts once it has the prerequisite incoming tokens and terminates execution when it completes the

production of its outputs. However, particularly if the action is a call action with streaming inputs and/or outputs, the completion of the action execution may need to be controlled externally. To achieve this, a value can be sent via a control flow to the action to enable or disable its invoked activity. SysML provides a specific control enumeration for this called **ControlValue**, with values **enable** and **disable**. For an action to receive this control input, it needs to provide a control pin that can receive it. A control value of *enable* has the same semantics as the arrival of a control token, and a control value of *disable* will terminate the invoked activity.

A special behavior called a **control operator** produces control values via an output parameter, typed by **ControlValue**. A control operator can include complex control logic and can be reused, via a call behavior action, in many different activities. A control operator is also able to accept a control value on an appropriately typed input parameter and will treat it as an object token rather than a control token.

The control value type could be extended in a profile (see Chapter 14) to include other control values in addition to *enable* and *disable*. A control operator could then output these new values. A control value of *suspend* might, for example, not terminate execution of the action like *disable*. The action would allow execution to resume where it left off when it received a *resume* control value.

The definition of a control operator is indicated by the presence of the keyword **controlOperator** as the model element type in the diagram label on the activity diagram frame.

Figure 8.13 shows a simple control operator, called *Convert Bool to Control*, that takes in a *Boolean* parameter called *bool in* and, depending on its value, either outputs an enable or disable value on its *control out* output parameter. The values are created using primitive actions, called value specification actions, whose purpose is to output a specified value. By convention, the input and output pins of these actions are elided. (See Section 8.12.1 for a discussion of primitive actions.) *Convert Bool to Control* is a generally useful control operator that can be reused in many situations.

A control operator is a kind of behavior and so may be invoked using a normal call behavior action. A call behavior action that invokes a control operator has

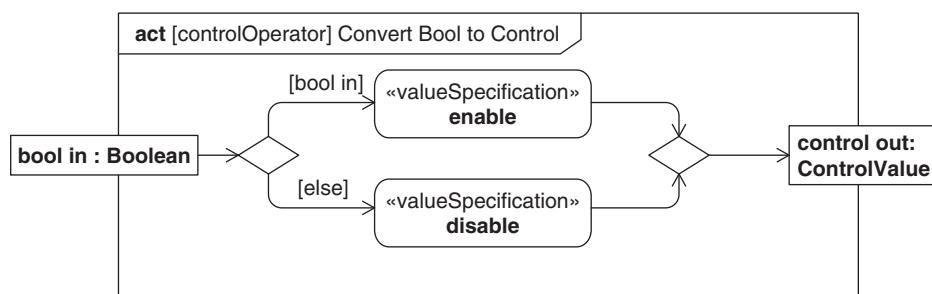
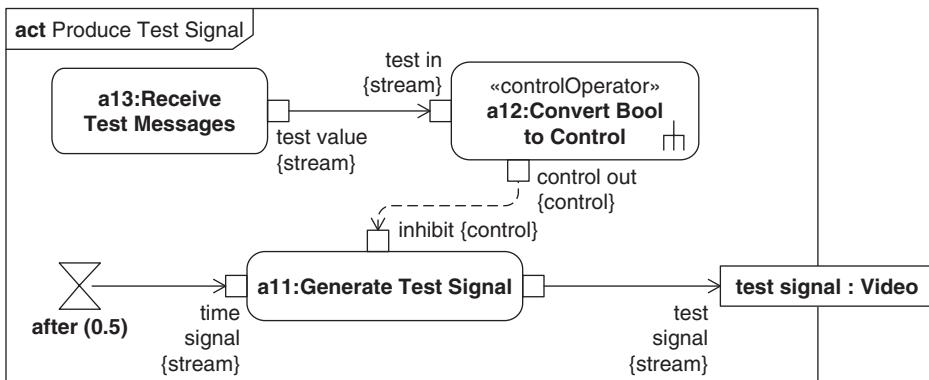


FIGURE 8.13

Using a control operator to generate a control value.

**FIGURE 8.14**

Using a control operator to control the execution of an activity.

the keyword «controlOperator» above its name string. A control pin symbol is a standard pin symbol with the addition of the property name “control” in braces floating near the pin symbol.

A test signal, by definition, is not always wanted on the video output. A mechanism to inhibit test signal production is shown in Figure 8.14. The *Convert Bool to Control* control operator shown in Figure 8.13 reads a Boolean flag, *test in*, from the activity *Receive Test Messages* and uses that to output an enable or disable value on a control pin called *control out*. This pin in turn is connected via a control flow to the *inhibit* pin of the *Generate Test Signal* activity. *Generate Test Signal* interprets this input as a control value because *inhibit* is a control pin, as indicated by the annotation “[control].” When *Generate Test Signal* is enabled, it reads the time at 2Hz from the accept time event action (see Section 8.7 for a discussion of time events). The activity *Receive Test Messages* is defined in Figure 8.23 (see page 204).

8.7 Handling Signals and Other Events

In addition to obtaining inputs and producing outputs using its parameters, an activity can accept signals using an **accept event action** for a signal event (commonly called an **accept signal action**) and send signals using a **send signal action**. Communication can then be achieved between activities by including a send signal action in one activity and an accept signal action for a signal event representing the same signal in another activity. More typically signals are sent from or received by the instances of the blocks that own and execute the activities, as described in Section 8.9.2. Communication via signals takes place asynchronously; that is, the sender does not wait for the signal to be accepted by the receiver before proceeding to other actions.

An accept signal action can output the received signal on an output pin. A send signal action has one input pin per attribute of the signal to be sent and one pin to specify the target for the signal.

The accept event action can accept others kinds of event, including:

- A time event, which 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, which 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.

An accept event action with no incoming control flows is enabled as soon as its owning activity (or owning interruptible region, see Section 8.8.1) starts to execute. However, unlike other actions, it remains enabled after it has accepted an event and so is ready to accept others.

A send signal action is represented by a rectangle with a triangle attached on one end, and an accept event action is represented by a rectangle with a triangular section missing from one end. When the event accepted is a time event, the accept event action may be shown as an hourglass symbol (see Figure 8.14).

Figure 8.15 shows how MPEG frames get transmitted over the surveillance camera network. The *Transmit MPEG* activity first sends a *Frame Header* signal to indicate that a frame is to follow. It then executes *Send Frame Contents*, which splits the frame into packets and sends them. When *Send Frame Contents* finishes, it outputs a *packet count* and two signaling actions are performed, a *Frame Footer* signal is sent, and then an accept signal action waits for a *Frame Acknowledgment* signal. Finally, the *Check Transmission* activity is invoked once the *Frame Acknowledgment* signal has been received, to check the packet count

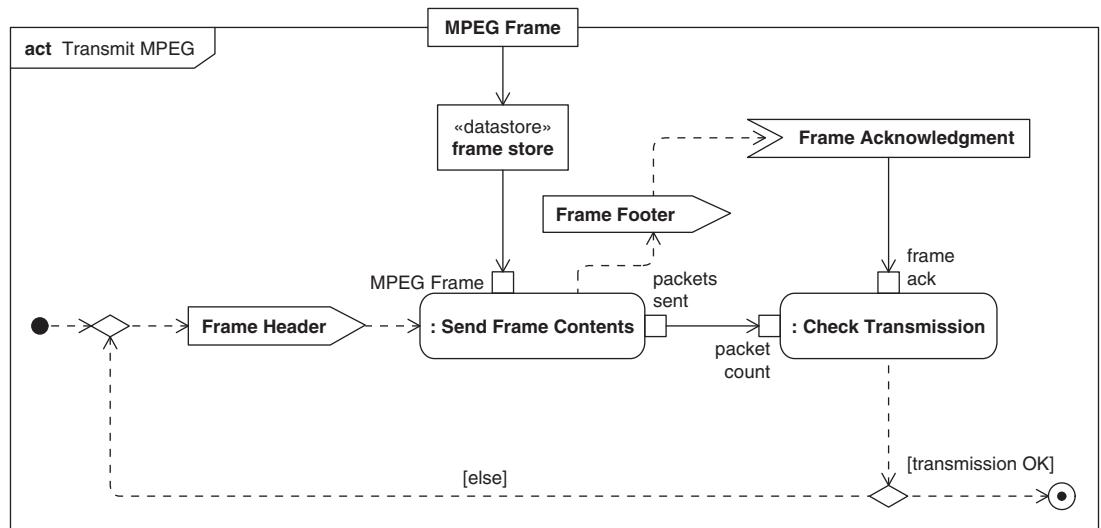


FIGURE 8.15

Using signals to communicate between activities.

returned with the acknowledgment against the count provided as an output of *Send Frame Contents*. If the packet counts match, then transmission is deemed to have succeeded and the variable [*transmission OK*] is set to true. This variable is then tested on the outgoing guards of a decision node, and if true, then the activity terminates; otherwise, the frame is resent, having previously been stored.

Note that this description is incomplete because it does not provide a target object for the send signal actions.

8.8 Advanced Activity Modeling

This section addresses several advanced activity modeling concepts.

8.8.1 Interruptible Regions

All the action executions within an execution of an activity are terminated when the activity is terminated. However, there are some circumstances where the modeler wants a subset of the action executions to be terminated but not all.

An **interruptible region** can be used to model this situation. An interruptible region groups a subset of actions within an activity and includes a mechanism for interrupting execution of those actions, called an **interrupting edge**, whose source is a node inside the region and whose destination is a node outside it. Both control and object flows can be designated as interrupting edges. Normal (i.e., noninterrupting) flows may have a destination outside the region as well; tokens sent on these flows do not interrupt the execution of the region.

An interruptible region is entered when at least one action within the region starts to execute. An interruption of an interruptible region occurs whenever a token is accepted by an interrupting edge that leaves the region. This interruption causes the termination of all actions executing within the interruptible region, and execution continues with the activity node or nodes that accepted the token from the interrupting edge. (It can be more than one node because the interrupting edge can connect to a fork node.)

A token on an interrupting edge often results from the reception of a signal by the activity containing the interruptible region, or its context object, if it has one. In that case, the signal is received by an accept signal action within the interruptible region that offers a token on an outgoing interrupting edge to some activity node outside the region. Because this is a common case, there are special semantics associated with accept event actions contained in interruptible regions. As long as they have no incoming edges, the accept event action does not start to execute until the interruptible region is entered, as opposed to the normal case where the accept event action starts when the enclosing activity starts.

An interruptible region is notated by drawing a dashed round-cornered box around a set of activity nodes. An interrupting edge is represented either by a lightning bolt symbol or by a normal flow line with a small lightning bolt annotation floating near its middle.

Figure 8.16 shows a more complete definition of the overall behavior of the camera, *Operate Camera*, previously shown in Figure 8.6. After invoking the *Initialize* activity, the camera waits for a *Start Up* signal to be received by an accept signal action before proceeding simultaneously with the primary activities that the camera performs: *Collect Images*, *Capture Video*, and *Generate Video Outputs*. These actions are triggered, following the acceptance of the *Start Up* signal, using a fork node to copy the single control token emerging from the accept signal action into control flows terminating on each action.

The actions are enclosed in an interruptible region and continue to execute until a *Shut Down* signal is accepted by an accept event action. When a *Shut Down* signal has been accepted, an interrupting edge leaves the interruptible region, all the actions within it terminate, and control transitions to the action that invokes the *Shutdown* activity. Once the *Shutdown* activity has completed, a control token is sent to an activity final node that terminates *Operate Camera*. Note that there are other flows leaving the interruptible region, but because they are not interrupting edges, they do not cause its termination.

8.8.2 Modeling Flow Rates and Flow Order

There is a default assumption that tokens flow at the rate dictated by the executing actions and that tokens flowing into an object node flow out in the same order. SysML offers constructs to deal with situations where these assumptions are not valid.

Flow Rates

Any streaming parameter may have a rate attached to it that specifies the expected rate at which tokens flow into or out of a related pin or parameter node. The expected rate at which tokens can arrive at or leave a pin or parameter node is specified by the rate property on a parameter. Note that this is only the statistically expected rate value. The actual value may vary over time, only averaging out to the expected value over long periods. Continuous flow is a special case that indicates that the expected rate of flow is infinite, or conversely the time between token arrivals is zero. In other words, there are always newly arriving tokens available at whatever rate the tokens are read.

When no rate information is displayed, an arbitrary discrete rate is assumed, which can be explicitly indicated by marking the parameter as discrete. In addition to parameters, flows can be notated as continuous or discrete or with a rate. When a rate is provided for a flow, it specifies the expected number of objects and values that traverse the edge per time interval; that is, the expected rate at which they leave the source node and arrive at the target node. It does not refer to the rate at which a value changes over time.

Continuous and discrete rates are indicated by the appearance of the corresponding keywords «continuous» and «discrete» above the name string of the corresponding symbol, or if a specific expected rate is required, by the property pair, rate = rate value, in braces either inside or floating alongside the corresponding symbol.

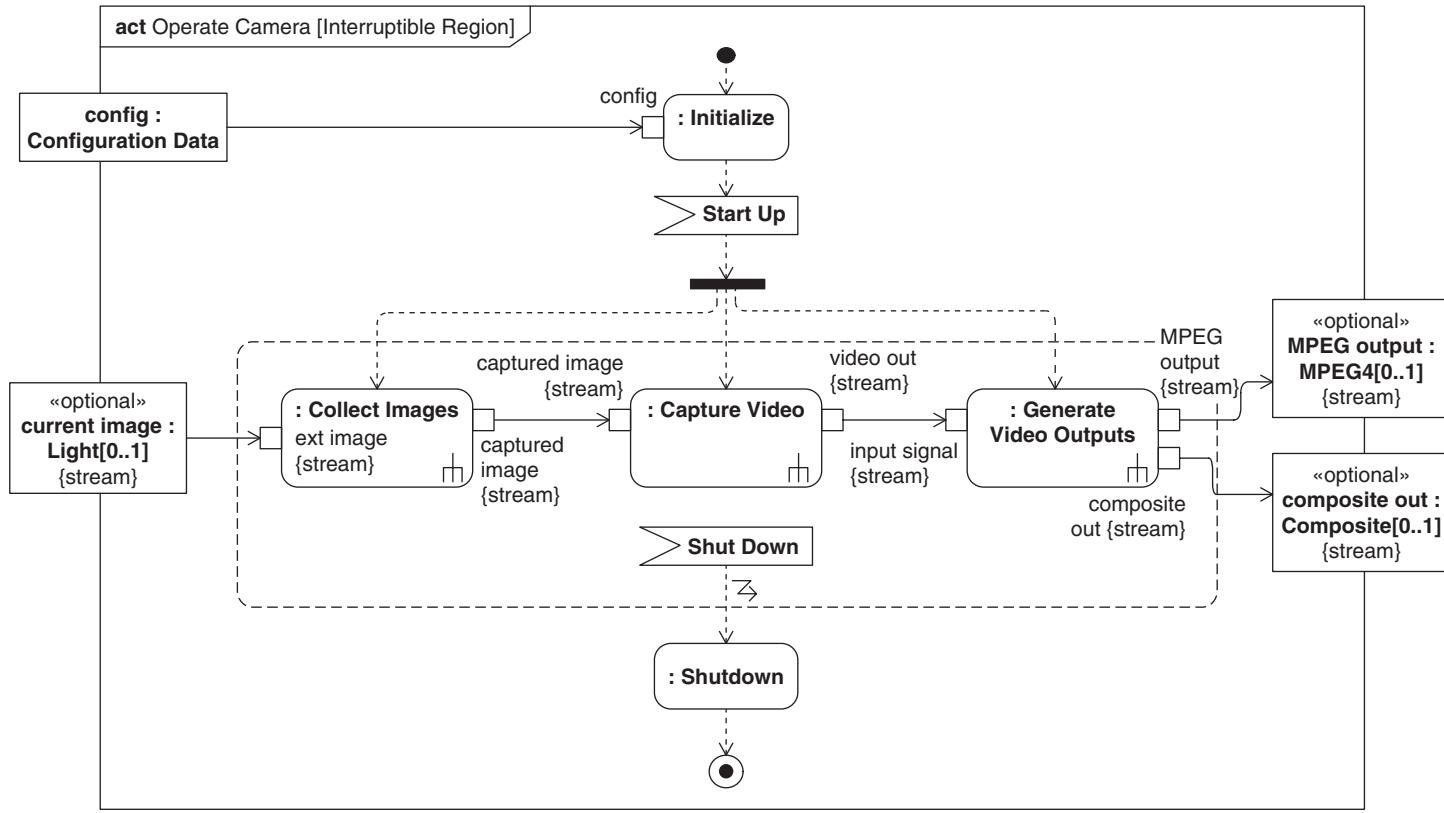
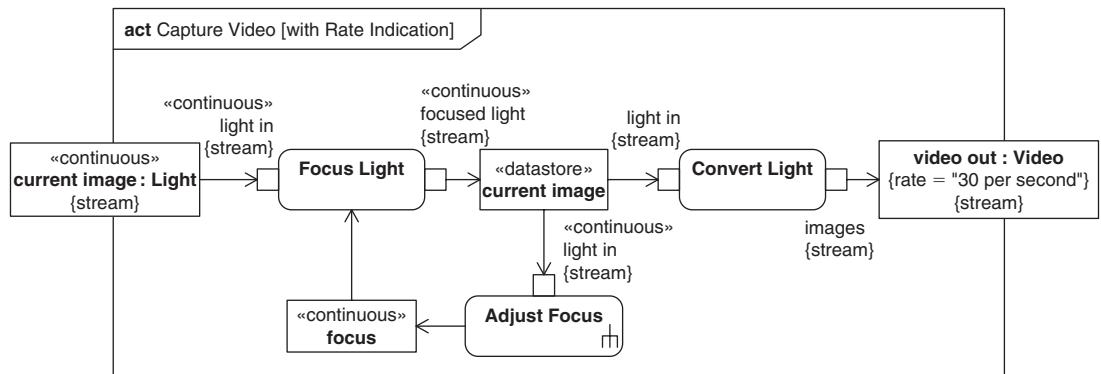


FIGURE 8.16

An interruptible region.

**FIGURE 8.17**

Use of continuous flows and discrete flows with rate information.

In Figure 8.17, the object flows associated with light in the *Capture Video* activity are continuous. The *Focus Light* and *Adjust Focus* actions invoke analog processes with continuous inputs and outputs, as indicated by the appearance of the keyword «continuous» on object nodes associated with those actions, including the *current image* parameter node. However, the images generated by the *Convert Light* action must be produced at a rate of 30 frames per second, as indicated on the *video out* parameter node.

Flow Order

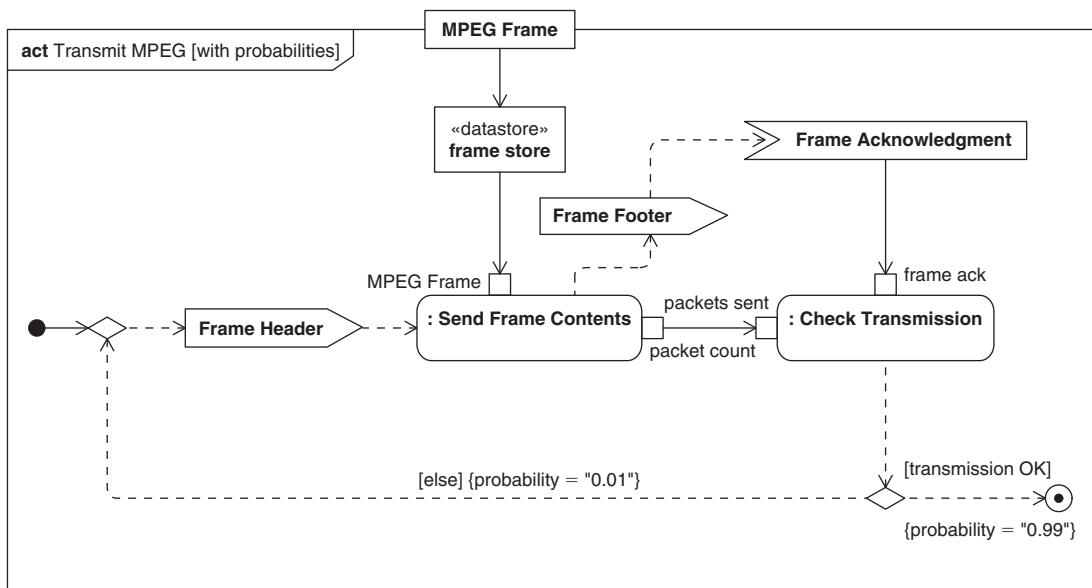
As described earlier in this chapter, tokens can be queued at pins or other object nodes as they await processing by the action, subject to a specified **upper bound**. Where the upper bound of an object node is greater than one, the modeler can specify the order in which its tokens are read using the **ordering property** of the node that can take values of ordered, First-In/First-Out (FIFO), Last-In/First-Out (LIFO), or unordered. Where the ordering property is specified as ordered, the modeler must provide an explicit selection behavior that defines the ordering. This mechanism can be used to select the token based on some value, such as priority, of the represented object.

In the case where an offered token would cause the number of tokens to exceed the upper bound of the object node, a modeler can choose to **overwrite** tokens already there, or to **discard** the newly arrived tokens.

The notation for ordering is the name value pair `ordering = ordering value`, placed in braces near or inside the object node. If no ordering is shown, then the default FIFO is assumed. The keyword «overwrite» is used to indicate that a token arriving at a full node replaces the last token in the queue according to the “ordering” property for the node. Alternatively, the keyword «noBuffer» can be used to discard newly arriving tokens that are not immediately processed by the action.

8.8.3 Modeling Probabilistic Flow

Where appropriate, a flow can be tagged with a probability to specify the likelihood that a given token will traverse a particular flow among available alternative

**FIGURE 8.18**

Probabilistic flow.

flows. This is typically encountered in flows that emanate from a decision node, although probabilities can also be specified on multiple edges going out of the same object node (including pins). Each token can only traverse one edge, with the specified probability. If **probabilistic flows** are used, then all alternative flows must have a probability and the sum of the probabilities of all flows must be 1.

Probabilities are shown either on activity flow symbols, or parameter set symbols, as a property/value pair, probability = probability value enclosed in braces floating somewhere near the appropriate symbol.

Figure 8.18 shows the activity diagram for *Transmit MPEG*, first introduced in Figure 8.15. In this example, the probability of successful transmission has been added. The two flows that correspond to successful and unsuccessful transmission have been labeled with their relative probability of occurrence.

8.8.4 Modeling Pre- and Postconditions and Input and Output States

An action is able to execute when all of the prerequisite tokens have been offered at its inputs, and similarly may terminate when it has offered the postprerequisite tokens on its outputs. However, sometimes additional constraints apply, which are based on the values of those tokens or conditions currently holding in the execution environment. These constraints can be expressed using **pre-** and **postconditions** on the actions, and in the case of call actions, on the behaviors they invoke. In the specific case when an object represented by a token has an associated

state machine, an object node may explicitly specify the required current state or states of that object in a **state constraint**.

The display of pre- and postconditions depends on whether they are specified against the behavior or the action. Pre- and postconditions on behaviors (in this case activities) are specified as text strings placed inside the activity frame, preceded by either the keyword «precondition» or «postcondition». Pre- and postconditions on actions are placed in note symbols attached to the action, with the keyword «localPrecondition» or «localPostcondition» at the top of the note, preceding the text of the condition.

A state constraint on an object node is shown by including the state name in square brackets underneath the name string of the symbol for that object node. This is equivalent to a local precondition or postcondition on the owning action requiring the specified state.

Although ACME Surveillance Systems Inc. does not manufacture the cameras, they do want to have some say in the production process. Figure 8.19 shows their preferred process. The optimal path for the production process is through *Assemble Cameras* and *Package Cameras*. However, their experience is that some assembled cameras do not work properly but can be repaired, at reasonable cost, and sold as reconditioned.

The repair process is modeled as the activity *Repair Cameras*. Some cameras are unfixable, but even then the camera can be cannibalized (through activity *Cannibalize Cameras*) for spare parts that can be fed back into the assembly process. A camera in production progresses through a number of states (see Chapter 10 for a description of state machines) as it moves through production, and different activities require or provide cameras in specific states. *Assemble Cameras* may produce cameras faster than they can be packaged or repaired, so they are placed in a buffer called *assembled cameras*. From there they either progress directly to *Package Cameras* if their state is *operational*; otherwise, they progress to *Repair Cameras* if their state is *damaged*. *Repair Cameras* accepts cameras in the *damaged* state, and they are either *repaired* or deemed *unfixable* when the activity has finished with them.

Note that the activity *Build Cameras* merely models the process of building cameras, using tokens to represent cameras. In this example, the flow of tokens could mirror quite closely the flow of physical cameras through a production system; the central buffer node might be allocated to a storage rack, for example. However, the physical production system may be quite different, and it's only when these activities are allocated to physical processing nodes that the physical meaning of the token flow is understood.

The previous discussion described how the input and output states could be used to specify preconditions and postconditions, respectively. A constraint on the input and output relationship can also be specified, in effect by combining a precondition and postcondition. These constraints might, for example, express the relationship between the pressure of some incoming gas and the temperature readings provided by some outgoing electrical signal. Alternatively, this could be used to express an accuracy or time constraint associated with the action or activity.

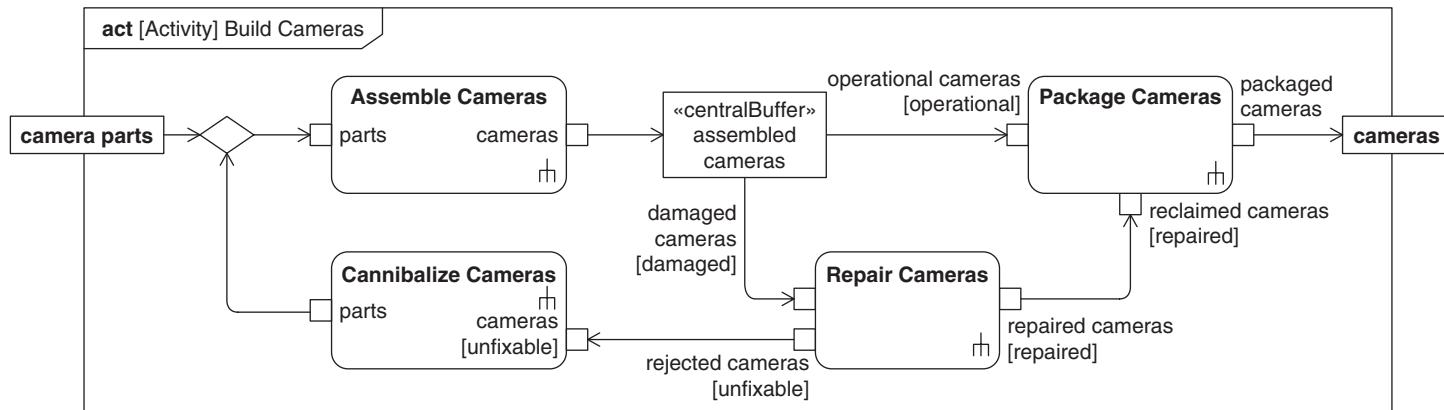


FIGURE 8.19

Example of using states on pins.

8.9 Relating Activities to Blocks and Other Behaviors

Activities are often specified independently of structure (i.e., blocks), and their execution semantics do not depend on the presence of blocks. However, as the system design progresses, the relationship between the behaviors of a system, expressed in this case using activities, and the structure of a system, expressed using blocks, does eventually need to be established.

Different methods approach this in different ways. A classical systems engineering functional decomposition method allocates the functions to components, as discussed in Chapter 13. Other methods approach this somewhat differently by establishing a block hierarchy and driving out the scenarios between the blocks. Examples of these different methods are included in Chapters 15 and 16.

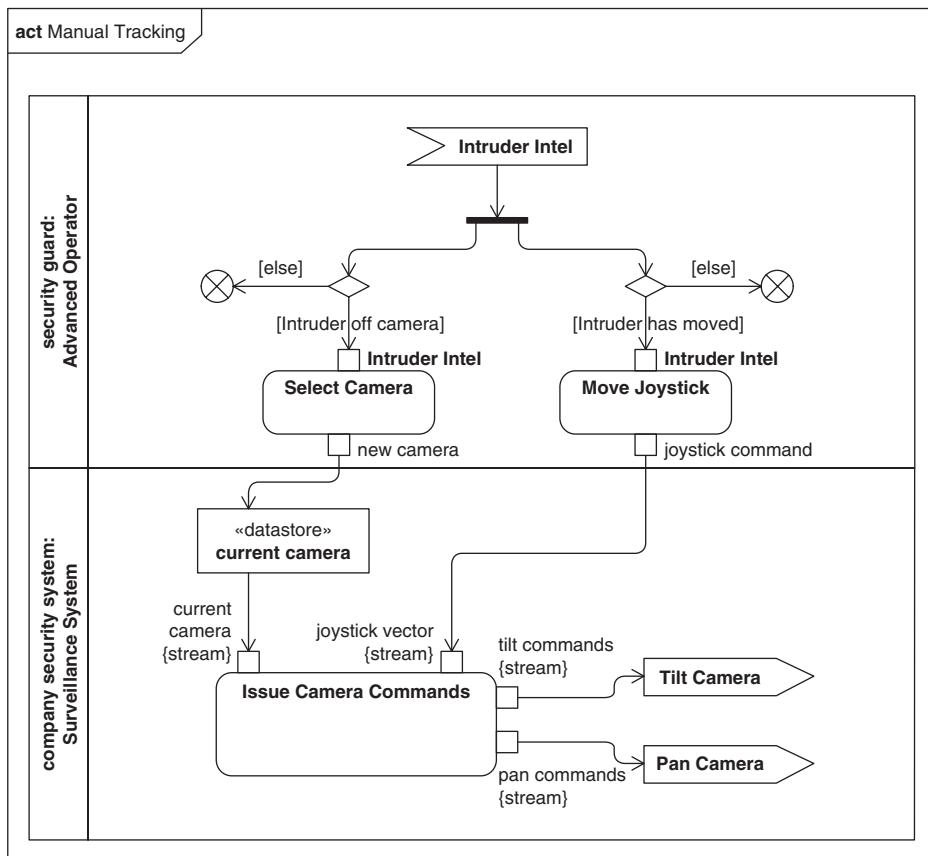
SysML also has two other mechanisms to relate blocks and activities. The first is the use of an activity partition to assert that a given block (or part) is responsible for the execution of a set of actions. The second is for a block to own an activity and use this as a basis for specifying the block's behavior.

8.9.1 Linking Behavior to Structure Using Partitions

A set of activity nodes, and in particular call actions, can be grouped into an **activity partition** (also known as a **swimlane**) that is used to indicate responsibility for execution of those nodes. A typical case is where an activity partition represents a block or a part and indicates that any behaviors invoked by call actions in that partition are the responsibility of the block or the block that types the part. The use of partitions to indicate which behaviors are the responsibilities of which blocks is an important communication mechanism to specify the functional requirements of a system or component defined by the block.

Activity partitions are depicted as rectangular symbols that physically encompass the action symbols and other activity nodes within the partition. Each partition symbol has a header containing the name string of the model element represented by the partition. In the case of a part or reference, the name string consists of the part or reference name followed by the type (block) name, separated by a colon. In the case of a block, the name string simply consists of the block's name. Partitions can be aligned horizontally or vertically to form rows or columns, or optionally can be represented by a combination of horizontal and vertical rows to form a grid pattern. An alternative representation for an activity partition for call actions is to include the name of the partition or partitions in parentheses inside the node above the action name. This can make the activity easier to lay out than when using the swimlane notation.

Figure 8.20 contains an example of partitions taken from the model of an ACME surveillance system. It shows how new intruder intelligence is analyzed and dealt with by the *security guard* and the *company security system* within some overall system context. Once the security guard has received new intelligence (signal *Intruder Intel*), he or she may need to address two concerns in parallel, so the token representing the signal is forked into two object flows. If the

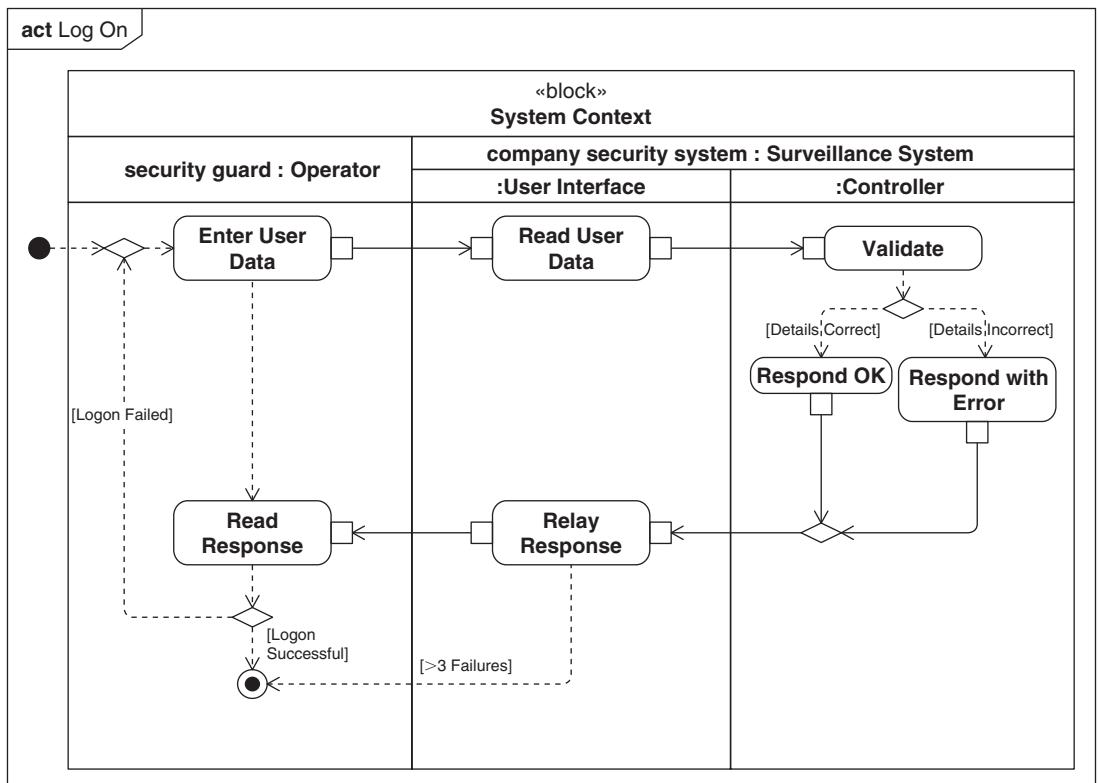
**FIGURE 8.20**

Activity partitions.

intruder has moved, then a *Move Joystick* action is performed to follow him or her. If the intruder is deemed to have moved out of range of the current camera, then a *Select Camera* activity is performed to select a more appropriate camera. In both cases, a flow final node is used to handle the tokens referencing the signal data where no action is required.

The *company security system* stores the currently selected camera in a data store node. It uses this information when it reacts to joystick commands by sending *Pan Camera* and *Tilt Camera* commands to the selected camera. *Security guard* and *company security system* are parts, as indicated by the name strings in the partition headers. Partitions themselves can have subpartitions that can represent further decomposition of the represented element.

Figure 8.21 shows the process for an *Operator (security guard)* logging in to a *Surveillance System (company security system)*. The *security guard* enters his or her details that are read by the *User Interface*, part of the *company security system*, and validated by another part, *Controller*, that then responds appropriately.

**FIGURE 8.21**

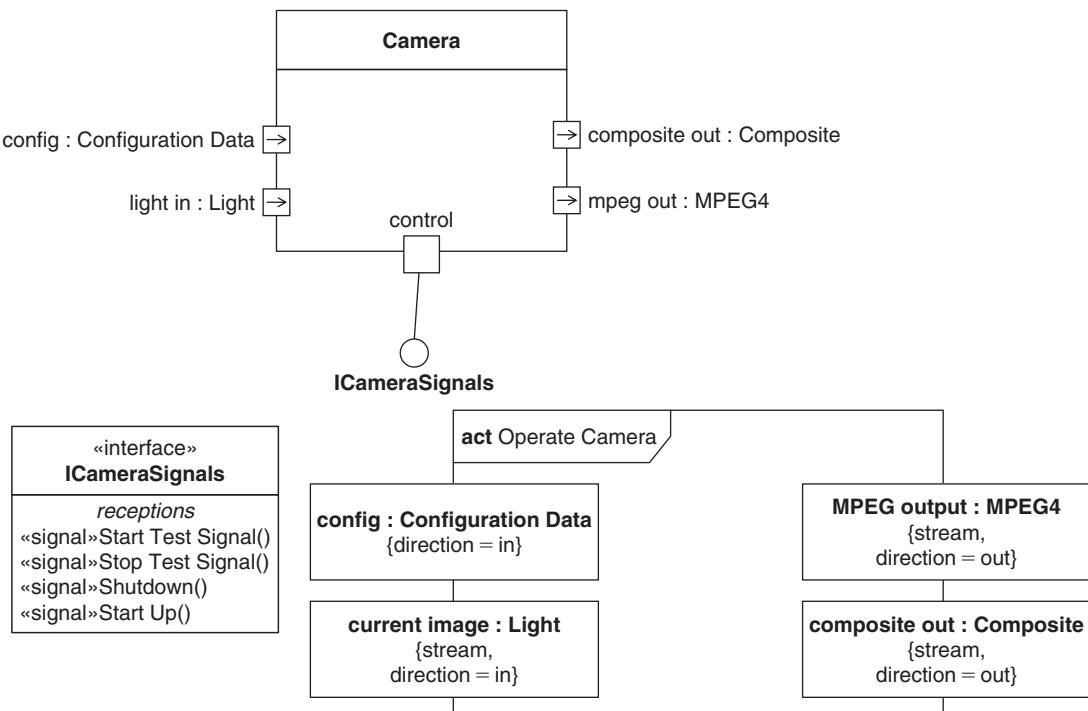
Nested activity partitions.

The *User Interface* and the *Controller* are represented by nested partitions with *company security system*. In this case, the *security guard* and the *company security system* are themselves shown as nested partitions of a block representing the context for both the surveillance system and its users.

An allocate activity partition is a special type of partition that can be used to perform behavioral allocation, as described in Chapter 13.

8.9.2 Specifying an Activity in a Block Context

In SysML, activities can be owned by blocks in which case an instance of the owning block executes the activity. For a block, an activity may either represent the implementation of some service, which is termed a method, or it may describe the behavior of the block over its lifetime, which is termed the classifier behavior or the main behavior. During execution of an activity, an instance of its owning block provides its execution context. The execution of the activity can access stored state information for the instance and has access to its queue of requests.

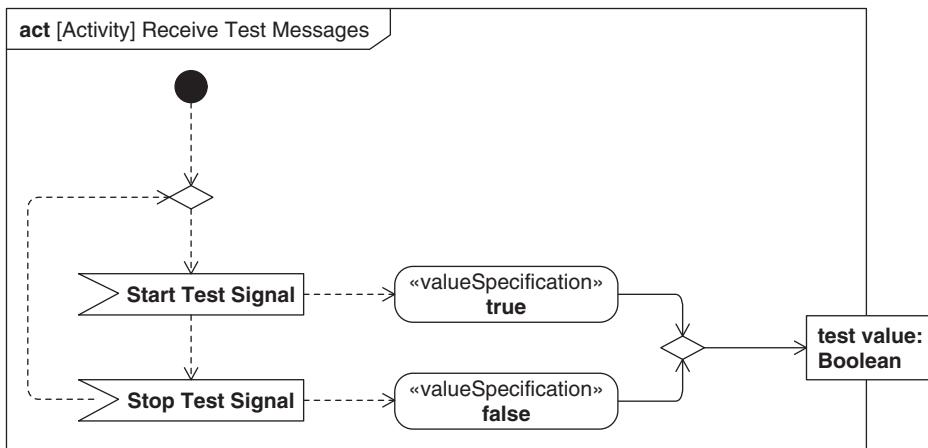
**FIGURE 8.22**

A block with flow ports and a block behavior.

Activities as Block Behaviors

When an activity serves as a main behavior, parameters of the activity may be mapped to flow ports on the owning block. The mapped ports must be behavior ports; that is, their inputs and/or outputs must be consumed and/or produced by the block behavior rather than being delegated to parts of the block. SysML does not explicitly say how flow ports are matched to parameters because there are many different approaches, depending on methodology and domain. An obvious strategy is to match parameters to ports based on at least type and direction, but where this still results in ambiguity, the names can also be used to confirm a match. Allocation can also be used to express the mapping.

Figure 8.22 shows a block called *Camera* that describes the design for one of ACME's surveillance cameras. It has four flow ports, three of which allow light to flow into the camera and video to flow out in either composite or MPEG4 format. The fourth allows configuration data to be passed to the camera. It also has a standard (client/server) port with a provided interface that supports a set of control signals used to control the operation of the camera. The block behavior of the camera is the activity *Operate Camera* that has appeared in a number of previous figures, most recently Figure 8.16. In Figure 8.22, the parameters of the activity match the flow ports of the *Camera* block in terms of their direction and type

**FIGURE 8.23**

Using signals to control activity flow.

but not in all cases their names. However, in this case matching by type and direction is sufficient to resolve any ambiguity. The parameters of *Operate Camera* can therefore be mapped one to one with the flow ports of *Camera*.

In Figure 8.22, there is no direct correspondence between the *control* port on *Camera* and a parameter or parameters on its block behavior *Operate Camera*. However, when an activity acts as the block behavior for a block, it can accept signals received through standard ports on the block. These signals can be accepted using an accept event action, and then used within the activity.

Figure 8.23 shows the specification of the activity *Receive Test Messages* that is invoked as part of *Produce Test Signal*, as shown on Figure 8.14. Once the activity starts, it simply waits for a *Start Test Signal* signal using an accept signal action, then a *Stop Test Signal* signal, and then repeats the sequence. The accept signal actions trigger value specification actions via control flows that create the right Boolean value and these values are merged into a *test value* output. Because *Operate Camera* executes *Receive Test Messages* (albeit several levels deep in the activity hierarchy), the execution has access to signals received by the owning context that is, in this case, an instance of *Camera*. The other two signals recognized by the *control* port in Figure 8.22 are *Shutdown* and *Start Up*, whose use is shown in Figure 8.16.

Activities as Methods

When used as a method, an activity needs to have the same signature (i.e., same parameter names, types, multiplicities, and directions) as the associated service definition, called in SysML a behavioral feature. There are two types of behavioral feature. An operation supports synchronous requests (i.e., the requester waits for a response) and asynchronous requests (i.e., the requester does not wait for a response). A reception only supports asynchronous requests. A reception indicates that the object can receive signals of a particular kind, as the result of a

send signal action (see Section 8.7). A method is invoked when the owning block instance (object) consumes a request for its associated behavioral feature. The activity executes until it reaches an activity final node, when the service is deemed to be handled, and if the request is synchronous, any output arguments are passed back to the initiator of the request.

SysML has a specific action to invoke methods via operations, called a **call operation action**. This has pins matching the parameters of the operation, and one additional input pin used to represent the target. When the action is executed, it sends a request to the target object that handles the request by invoking the method for the feature, passing it the input arguments, and passing back any output arguments.

If an activity invoked as the result of a call operation action has streaming parameters, then the pins of the call operation action may consume and produce tokens during execution of the activity. However, in a typical client/server approach to system design, all parameters are nonstreaming to fit more easily into a client/server paradigm.

Figure 8.24 shows the *Surveillance System* block with one of its ports, called *status*. The status port provides an interface *Camera Status* that includes an operation called *get camera status* as shown, with an input parameter called *camera id* and an output parameter called *camera status*. The activity *Handle Status Request*, shown originally in Figure 8.10, is designated to be the method of *get camera status*, so it has the same parameters. A call operation action for *get camera status* will result in the invocation of *Handle Status Request* with an argument for *camera id*, and it will expect a response on *camera status*. A call operation action for *get camera status* is shown, with pins corresponding to the two parameters and also a pin to identify the *target*; that is, the *Surveillance System* to which the request must be sent.

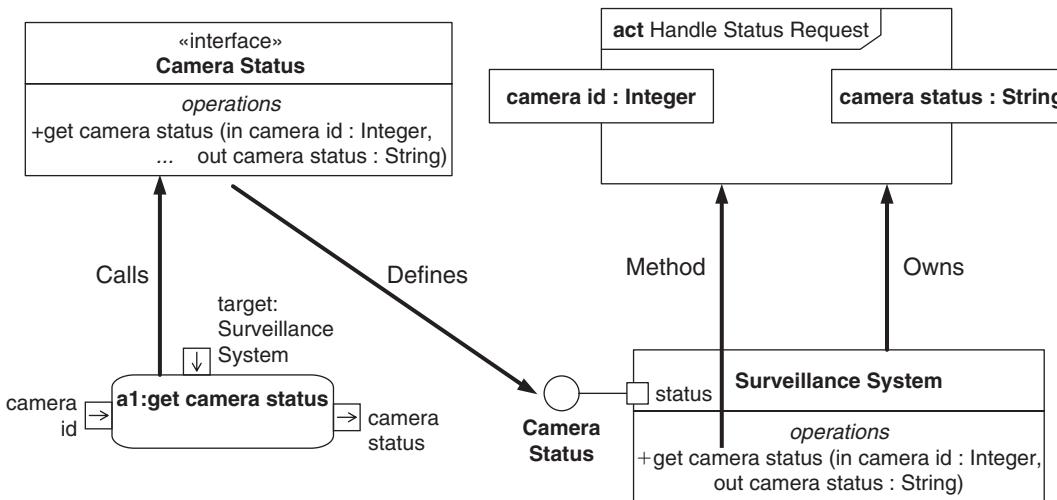


FIGURE 8.24

A block with behavioral features and associated methods.

8.9.3 Relationship between Activities and Other Behaviors

SysML has a generic concept of behavior that provides a common underlying base for its three specific behavioral formalisms: activities, state machines, and interactions. This provides flexibility to select the preferred behavioral formalism for the modeling task. For example, a call behavior action or call operation action in an activity can be used to invoke any type of behavior. However, the design and analysis method must further specify the semantics and/or constraints for a call action to call a state machine or an interaction from an activity since this is not currently fully specified. We expect future versions of SysML, and perhaps domain-specific extensions, to provide more precise semantics.

State machines use the general SysML concept of behavior to describe what happens when a block is in certain states and when it transitions between states. In practice, activities are often used to describe these behaviors as follows:

- What happens when a state machine enters a state (called an entry behavior).
- What happens when a state machine exits a state (called an exit behavior).
- What happens while a state machine is in a state (called a do activity).
- What happens when a state machine makes a transition between states (called a transition effect).

State machines are discussed in Chapter 10.

8.10 Modeling Activity Hierarchies Using Block Definition Diagrams

Activities can be represented as hierarchies in a very similar way to blocks using a block definition diagram. When represented this way, the **activity hierarchies** resemble traditional functional decomposition hierarchies.

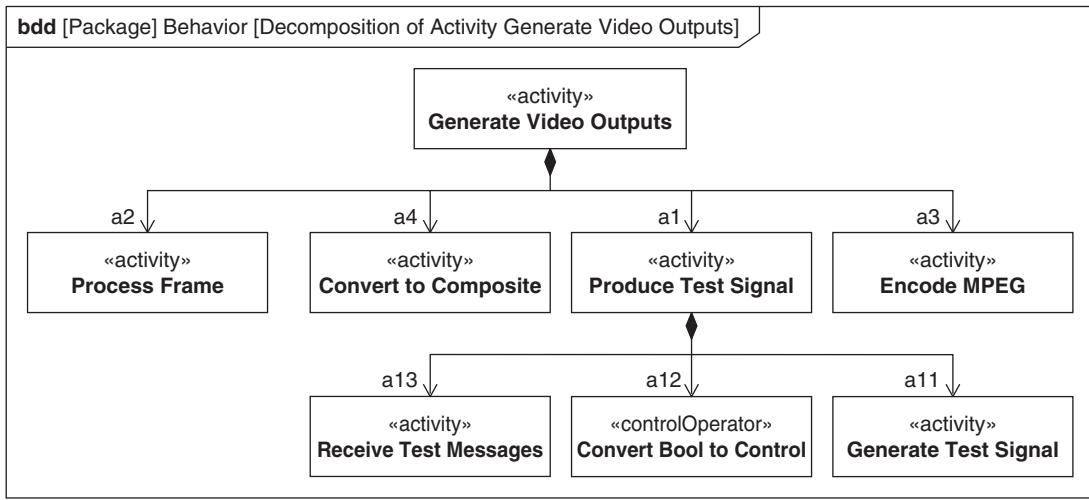
8.10.1 Modeling Activity Invocation Using Composite Associations

Invocation of activities via call behavior actions is modeled using the standard composition association where the calling activity is shown at the black diamond end and the called activity is at the other end of the association. On a block definition diagram, activities are shown using a block symbol with the keyword «activity». The role name is the name of the call behavior action that performs the invocation.

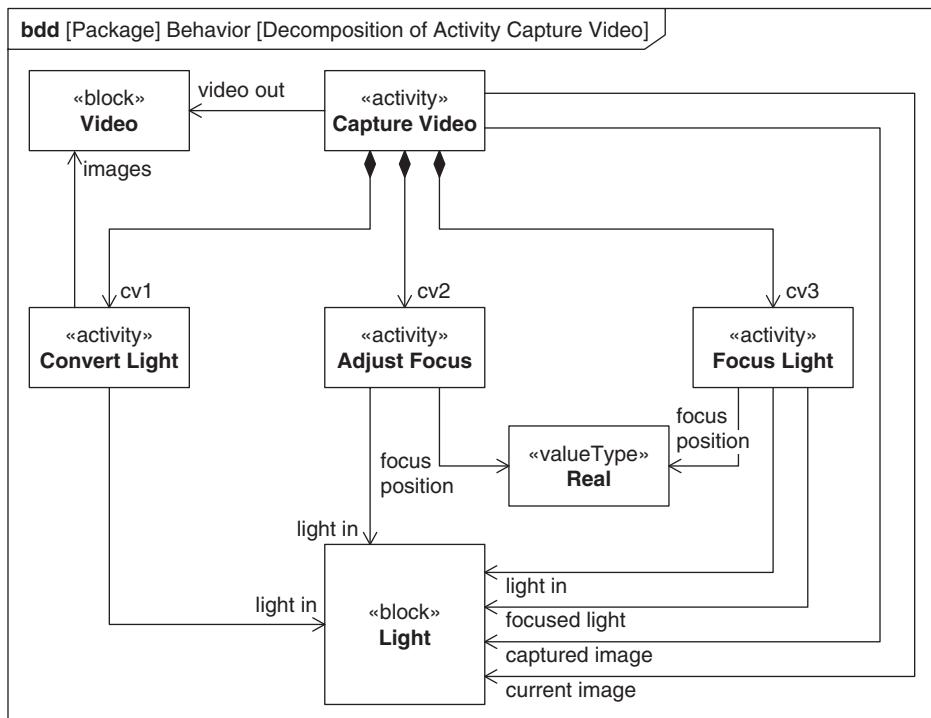
Figure 8.25 shows the block definition diagram equivalent of the activity hierarchy for *Generate Video Outputs*, as described in Figures 8.8 and 8.16. The block definition diagram does not represent the flows on the activity diagram but can include the parameters and object nodes, as shown in Figure 8.26.

8.10.2 Modeling Parameter and Other Object Nodes Using Associations

Parameters and other object nodes can also be represented on the block definition diagram. However, by convention, the relationship from activities to object

**FIGURE 8.25**

An activity hierarchy modeled on a block definition diagram.

**FIGURE 8.26**

Activity hierarchy with parameters.

nodes is represented with a reference association because the tokens contained within the object nodes are references to entities that are not “part” of the executing activity, and they are not necessarily destroyed when the execution of the activity terminates. However, composition can be used when composition semantics between the activity and the referenced objects apply. If the white diamond notation is used, then the activity is shown at the white diamond end and the object node type at the other end, and the role name at the part end is the name of the object node. Properties of the object node may be shown floating near the corresponding role name.

Figure 8.26 shows the hierarchy of activities for the *Capture Video* activity, originally shown in Figure 8.11, including its own parameter nodes and the parameter nodes of its various subactivities. The data store, *current image*, is also shown.

8.11 Enhanced Functional Flow Block Diagram

The Enhanced Functional Flow Block Diagram (EFFBD) or variants of it have been widely used in systems engineering to represent behavior. A function in an EFFBD is analogous to an action in an activity. The EFFBD does not include the distinction between an invocation action and an activity.

Most of the functionality of an EFFBD can be represented as a constrained use of a SysML activity diagram. The constraints are documented in Annex C of the SysML specification [1]. Using the keyword «effbd» in the diagram header of an activity indicates that the activity conforms to the EFFBD constraints. These constraints preclude the use of activity partitions and continuous and streaming flows, as well as many other features within activity diagrams.

Some EFFBD semantics are not explicitly addressed by the activity diagram. In particular, a function in an EFFBD can only be executed when all triggering inputs, the control input, and the specified resources are available to the function. In addition, a “resource” is not an explicit construct in SysML but can be modeled using constraints. Triggering inputs in EFFBDs correspond to “required inputs” in activity diagrams, nontriggering inputs correspond to “optional inputs,” and control inputs correspond to control flow in activity diagrams. The detailed mapping between EFFBD and activity diagrams, along with an example of the mapping in use, is described in Bock [33].

8.12 Executing Activities

Models can be used to specify systems, as discussed in Chapter 2. Often these models are used simply to facilitate communication among project teams; but sometimes models are intended to be interpreted by machines or computer programs to simulate the system. This latter category of model is often called an executable specification because it contains all the information necessary for a machine to “execute.” The construction of executable specifications imposes a

burden of completeness on the modeler, but it also requires the modeling formalism (SysML in this case) to have semantics defined precisely enough to allow execution of the model. This section describes how SysML supports the execution semantics of activities in the context of existing technologies.

To execute activities, the details of the processing must be specified, such as the transformation of property values, using mathematical operations. SysML has a set of primitive actions that support basic object manipulation such as creation, deletion, access to properties, object communication, and others; some of them have already been described in this chapter. This section describes some of the primitive actions and their intended use.

SysML further depends on other executable formalisms for its execution semantics, either to provide an executable definition of its own primitives or as direct expressions of functionality defined in so-called “opaque” constructs. These executable formalisms are normally accompanied by technologies for performing executions, as discussed in Chapter 17.

8.12.1 Primitive Actions

SysML includes a set of actions and a precise, although informal, definition of them. Some of these actions have been described previously in this chapter:

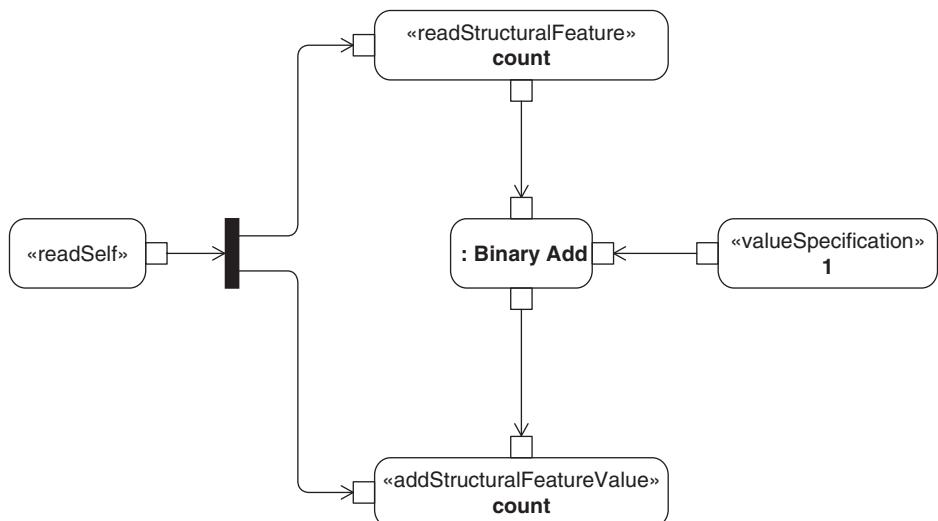
- Accept event actions respond to events in the environment of the activity.
- Send signal actions support communication between executing behaviors using messages.
- Call actions allow an activity to trigger the invocation of another behavior and to provide it with inputs and receive outputs from it.

In addition, there are a number of actions that have a more localized effect, such as updating properties and creating or destroying objects. These actions can be broadly categorized as:

- Object access actions allow properties of blocks and the variables of activities to be accessed.
- Object update actions allow those same elements to be updated or added to.
- Object manipulation actions allow objects themselves to be created or destroyed.
- Value actions allow the specification of values.

Note that the set of actions in SysML does not include fundamental operations such as mathematical operators. These have to be provided as libraries of opaque behaviors, or more likely function behaviors, suitable for the domain.

Primitive actions do have a corresponding notation, although because they are so low level, it is often a very painstaking job to construct activities from them using graphical notation. The intention behind their inclusion in SysML is that different textual notations for specifying processing would be mapped to these primitive actions and a tool would translate the textual notation to actions. To date there is no standard textual notation, but Figure 8.27 shows an example of how such a translation to actions might appear.

**FIGURE 8.27**

Example of primitive actions.

Figure 8.27 shows an alternate representation of the expression $count = count + 1$ in the algorithm in Figure 8.12, but using primitive actions instead of an opaque action. The resulting activity fragment first has to execute a read self action to obtain a reference to itself so that it can access its own internal structural features properties. The resulting object is passed to a read structural feature action that accesses the *count* property of the executing activity. The value of the *count* property is then passed to a call of the *Binary Add* function behavior. The other input is provided by a value specification action that outputs the value 1. The result of *Binary Add* is then offered to an add structural feature value action that uses the identity of the activity, obtained from the read self action, to update its *count* property. Needless to say, most modelers prefer the short form!

8.12.2 Executing Continuous Activities

Where a model is used as a blueprint for a system, it is expected that continuous activities will be implemented by physical devices such as motors, sensors, or humans. In this case the specification of the activity may be a set of equations, or it may simply be allocated to some component that is already known to provide the appropriate behavior.

However, sometimes it is important to simulate these continuous activities prior to building the system itself. There are simulation systems that can take physical equations and use those to construct a simulation of the implied system. Alternatively, there are other simulation systems that execute discrete activities at different execution rates, in the expectation that a sufficiently fast rate will approximate the continuous behavior to serve the purposes of the modeler. These technologies impose restrictions on the constructs that can be used in the

activity's definition (no token buffering, for example) and have their own specialized libraries of functions that need to be integrated into the model.

Invariably, the use of technologies to simulate continuous activities requires additional constructs and semantics that have to be provided using a profile. More information on profiles can be found in Chapter 14.

8.13 Summary

The concepts for flow-based behavior are based on activities, which are represented on both the activity diagram and the block definition diagram.

- An activity represents a controlled sequence of actions that transform its inputs to its outputs. The inputs and outputs of an activity are called parameters.
- Activities are composed of actions that represent the leaf level of behavior. An action consumes input tokens and produces output tokens via its pins.
- Actions are connected by flows. There are two types of flow:
 - Object flows route object tokens between the input and output pins of actions. On occasion, the flowing tokens may need to be queued or stored for later processing. The input and output pins on actions can queue tokens along with specialized nodes capable of storing tokens. Depending on the domain, flows may be identified as continuous, which is particularly useful for describing physical processes.
 - Control flows transfer control from one action to other actions using control tokens.
- Where the routing of flows is complex, intermediate constructs called control nodes, including join, fork, decision, and merge, allow flows to be split and merged in various ways. There are also specialized control nodes that describe what happens when an action starts and stops, called the initial node, activity final node, and flow final node, respectively.
- Actions come in many different categories from primitive actions, such as updating variables, to the invocation of entire behaviors.
 - Call actions are an important category of action because they allow one activity to invoke the execution of another (or in principle any type of behavior). The pins of call actions correspond to the parameters of the called entity. A call behavior action allows an activity to include the execution of another activity as part of its processing. A call operation action allows an activity to make a service request on another object that can trigger the execution of some activity to handle the request. Operation calls make use of the dispatching mechanism of SysML blocks to decouple the caller from knowledge of the invoked behavior.
 - Send signal actions and accept event actions allow the activity to communicate via signals rather than just through its parameters. When the activity is executing in the context of a block, the activity can accept signals sent either to the block or sent directly to the activity.

- Activity partitions provide the capability to assign responsibility for actions in an activity diagram to blocks or parts that the partitions represent.
- Block definition diagrams are used to describe the hierarchical relationship between activities, and the relationship of activities to their inputs and outputs. As such, only a limited form of the block definition diagram is used. The use of a block definition diagram for this purpose is similar to a traditional functional hierarchy diagram.
- Activity diagrams can be used to describe Enhanced Functional Flow Block Diagrams (EFFBDs), although special constraints on the semantics of activities must be imposed to ensure compliance with them.
- Activities may be described as stand-alone behaviors independent of any structure, but they often exist as the main behavior of a block. Activities within a block often communicate using signals, accepting signals that arrive at the block boundary and sending signals to other blocks. The parameters of a main behavior may also be mapped directly to the flow ports of its parent block. In this case flows to and from activity parameter nodes are routed directly through the flow ports.
- An activity can also be used to implement the response to a service request, where the arguments of the request are mapped to the activity's parameters. As described in Chapter 10, activities are often used to describe the processing that occurs when a block is transitioning between states and what the block does while in a particular state.

8.14 Questions

1. What is the diagram kind of the activity diagram?
2. How are an action and its pins typically represented on an activity diagram?
3. What does action *a1* in Figure 8.3 require to start executing?
4. How are the parameters of activities shown on activity diagrams?
5. What is the difference in semantics between a streaming and nonstreaming parameter?
6. How are parameters with a lower-multiplicity bound of 0 identified on an activity diagram?
7. Draw an activity diagram for an activity “Pump Water,” which has a streaming input parameter “*w in*” typed by block “Water” and a streaming output parameter “*w out*,” also typed by “Water.”
8. How are the set of pins for a call behavior action determined?
9. What is an object flow used for and how is it represented?
10. How does the behavior of a join node differ from that of a merge node?
11. How does the behavior of a fork node differ from that of a decision node?
12. What are parameter sets used for and how are they represented, both in the definition and invocation of an activity?
13. Figure 8.10 only shows the object flows between the call behavior actions. What else does it need in order to perform as the method for the *get camera*

- request in Figure 8.24? Draw a revised version of Figure 8.10 with suitable additions.
14. What is the difference between a data store node and a central buffer node?
 15. What is the difference in behavior between a flow final and an activity final node?
 16. How is an initial node represented on an activity diagram, and what sort of flows can be connected to it?
 17. What special capability does a control operator have?
 18. An action “pump” invokes the activity “Pump Water” from Question 7, and can be enabled and disabled by the output of a control operator. What additional features does “pump” need in order to enable this?
 19. Another action “controller” calls a control operator called “Control Pump” with a single output parameter of type “Control Value.” Draw an activity diagram to show how the actions “pump” and “controller” need to be connected in order for “controller” to control the behavior of “pump.”
 20. Name three kinds of event that can be accepted by an accept event action.
 21. How can an interruptible region be exited?
 22. What does a flow rate of “25 per second” on an activity edge indicate about the flow of tokens along that edge?
 23. How would a modeler indicate that new tokens flowing into a full object node should replace tokens that already exist in the object node?
 24. If a call behavior action is placed in an activity partition representing a block, what does this say about the relationship between the block and the called behavior?
 25. Name the two different roles that an activity can play when owned by a block.
 26. Describe the four ways in which activities can be used as part of state machines.

Discussion Topic

Discuss the various ways that activities with continuous flows may be executed.

This page intentionally left blank

Modeling Message-Based Behavior with Interactions

9

This chapter discusses the use of sequence diagrams to model how parts of a block interact by exchanging messages.

9.1 Overview

In Chapter 8, behavior was modeled in terms of activity diagrams to represent a controlled sequence of actions that transform inputs to outputs. In this chapter, an alternative approach to representing behavior is introduced; it uses sequence diagrams to represent the **interaction** between structural elements of a block as a sequence of message exchanges. The interaction can be between the system and its environment or between the components of a system at any level of a system hierarchy. A message can represent the invocation of a service on a system component or the sending of a signal.

This representation of behavior is useful when modeling service-oriented concepts, where one part of a system requests services of another part. A service-oriented approach can be useful for representing discrete interactions between software components, where one software component requests a service of another and where the service is specified as a set of operations. However, the sequence diagram is in no way limited to modeling interactions between software components and has found broad application in modeling system-level behaviors as well. An interaction can be written as a specification of how parts of a system should interact, but it can also be used as a record of how the parts of a system did interact.

The structural elements of a block are represented by lifelines on a sequence diagram. The sequence diagram describes the interaction between these lifelines as an ordered series of different types of events that can correspond to the sending and receiving of messages, the creation and destruction of objects, or the start and end of behavior executions.

Many of the events described earlier are associated with the exchange of messages between instances represented by lifelines. There are many different types of messages, including both synchronous messages where the sender waits for a response, and asynchronous messages where the sender continues without waiting

for a response. A sending event marks when the message is sent by the sending instance, and a receiving event marks when the message is received by the receiving instance. On reception of a message, the receiving instance may start the execution of a behavior that implements the operation or signal reception referenced in the message. The receipt of a message may also trigger the creation or destruction of the instance represented by the receiving lifeline.

To model more complex event ordering than simple sequences, interactions can include specialized constructs called combined fragments. A combined fragment has an operator and a set of operands, which themselves may contain interaction fragments such as messages or more combined fragments, thus forming a tree. There are a number of operators that provide different options such as parallel, alternative, and iterative ordering of their operands.

Interactions themselves can also be composed to handle large scenarios or to allow reuse of common interaction patterns. An interaction may reference another interaction to abstract away the detail of some part of the interaction between multiple lifelines, or to reference an interaction between the parts of a particular lifeline.

An interaction executes in the context of an instance of its owning block, each lifeline in the interaction represents a single instance that is part of that instance. As behaviors execute on these instances, events occur as they send and receive requests corresponding to operation calls and signals, and also as they start and end their execution. The sequence of event occurrences for a given scenario of interest, in this case the lifetime of the interaction, is called a trace. A trace is valid if the event occurrences are consistent with the event ordering defined by the interaction.

9.2 The Sequence Diagram

A given **sequence diagram** represents an interaction. The frame label for a sequence diagram has the following form:

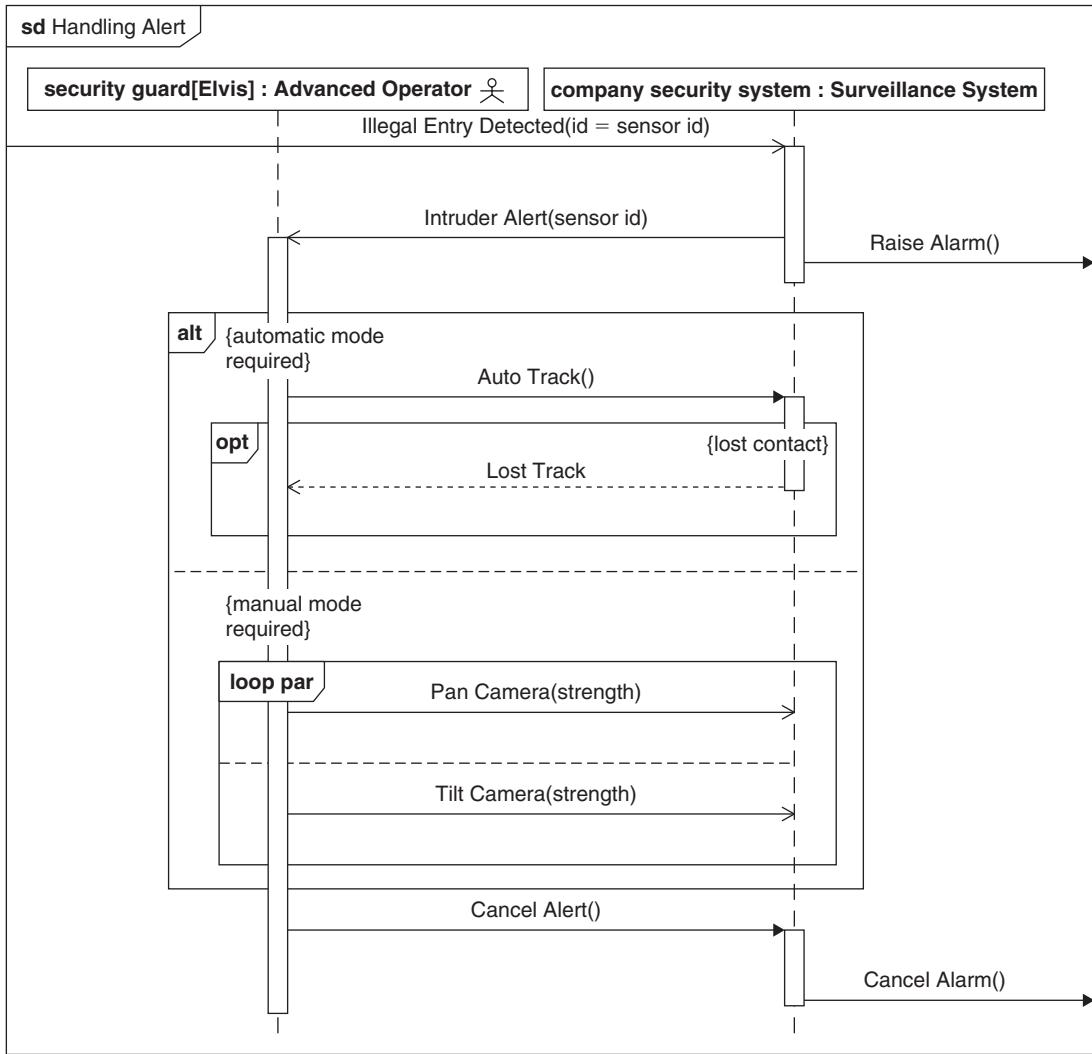
sd [Interaction] interaction name [diagram name]

The diagram kind for a sequence diagram is always **sd**. Sequence diagrams can only describe interactions, and therefore the model element type (Interaction) does not need to be shown in the diagram header. The *interaction name* is the name of the represented interaction, and the *diagram name* is user defined and may be used to describe the purpose of the diagram.

Figure 9.1 shows a sequence diagram with examples of many of the symbols. It shows an interaction between an operator and the surveillance system during the handling of an intruder alert. The notation for the sequence diagram is shown in detail in the Appendix, Tables A.15 through A.17.

9.3 The Context for Interactions

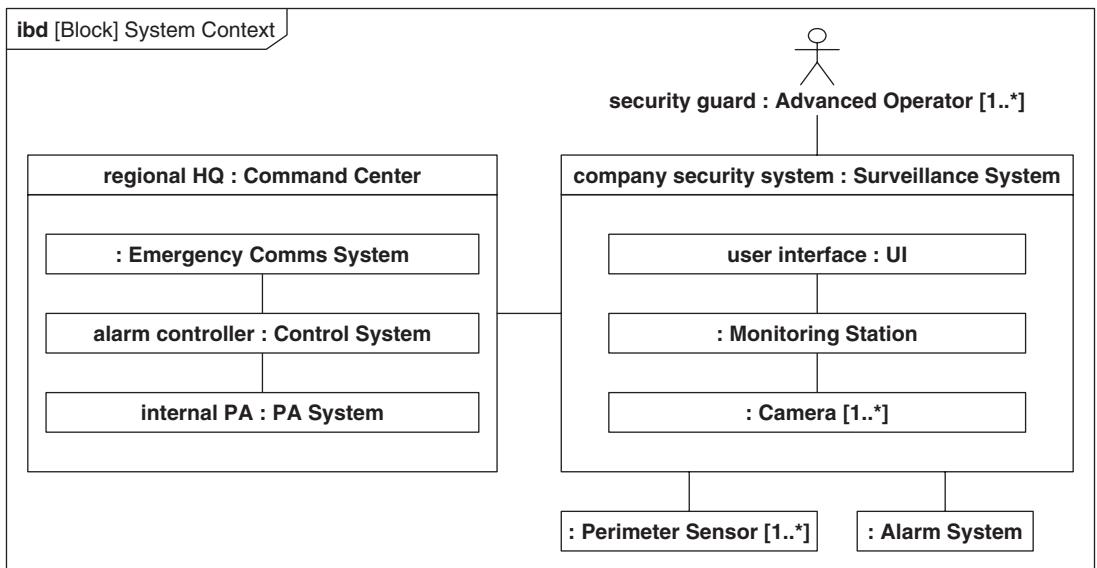
Interactions take place in the context of a block between elements of its internal structure. Figure 9.2 shows an internal block diagram of the *System Context* block

**FIGURE 9.1**

An example sequence diagram.

that contains all the significant participants in the interactions that are included in the figures in this chapter.

Figure 9.2 features an internal block diagram of a block called *System Context*, which is the context for a specific *Surveillance System* called *company security system*. In addition to the *company security system*, the context contains other parts including a *Regional HQ*, a set of *Perimeter Sensors*, an *Alarm System*, and a *security guard*, which may correspond to a number of physical actors. The diagram also shows the internal parts of the *regional HQ* and the *company security system* whose behavior is specified in the following interactions. The interaction

**FIGURE 9.2**

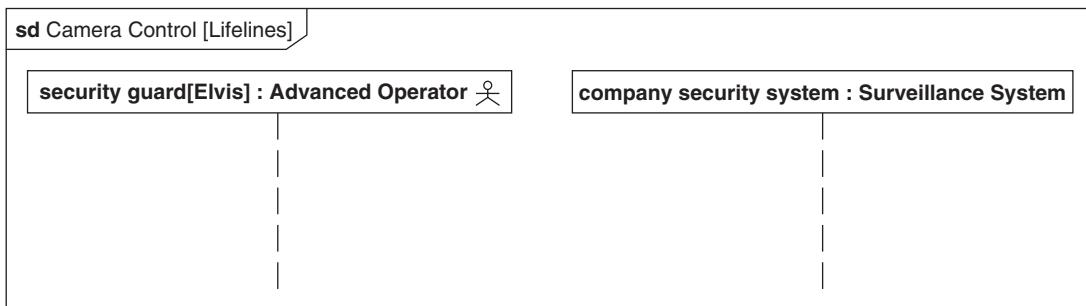
Internal block diagram of the interaction context.

lifelines can also represent reference properties, but this does not affect the notation or the semantics of the interaction. There are no lifelines that represent reference properties in the examples in this chapter.

9.4 Using Lifelines to Represent Participants in an Interaction

The principal structural feature of an interaction is the **lifeline**. A lifeline represents the relevant lifetime of a member of the interaction's owning block, which will be either a part property or a reference property, as described in Chapter 6. As explained there, a part can be typed by an actor, which enables actors to participate in interactions as well. However, since an actor cannot support operations, there are restrictions on its use. To avoid this restriction, an actor may be allocated to a block that is used to type the part. Lifelines can also represent standard ports, but because ports typically just relay messages they rarely contribute much to the understanding of an interaction.

When an interaction executes in an instance of its owning block, each lifeline denotes an instance of some part of the block (see Chapter 6 for a definition of block semantics). Thus, when the lifeline represents a member with multiplicity greater than 1, an additional **selector expression** can be used to explicitly identify one instance. Otherwise, the lifeline is taken to represent an arbitrarily selected instance. The selector expression can take many forms depending on how instances are identified in this part. For example, it may be an index into an ordered collection, or a specific value of some attribute of the part's block, or a more informal statement of identity.

**FIGURE 9.3**

An interaction with lifelines.

A lifeline is shown using a rectangle (the head) with a dashed line descending from its base (the tail). The rectangle contains the name and type (if applicable) of the represented member, separated by a colon. The selector expression, if present, is shown in square brackets after the name. The head may indicate the kind of model element it represents using a special shape or icon.

Figure 9.3 shows a simple sequence diagram with a diagram frame and two lifelines. One represents the *Surveillance System* under consideration, called *company security system*, and the other lifeline represents an *Advanced Operator*, called *security guard*. Because, the *security guard* from Figure 9.2 has an upper bound greater than 1, the lifeline also contains a **selector** called *Elvis* to specify exactly which instance is interacting. The *security guard* is shown with a small actor icon to indicate that it is a user of the *Surveillance System*.

9.4.1 Events and Occurrences

A lifeline is related to an ordered list of **events** that describes things that happen to the instance represented by the lifeline during the interaction. (Actually the lifeline features occurrence specifications that reference events, but to simplify the description, we refer to the conflated concept as an event.) During execution of the interaction, instances of events (called **occurrences**) are compared to the events expected on the lifeline.

Different types of events describe different types of occurrences. Three categories of events are relevant to interactions:

- The sending and receiving of messages
- The start and completion of execution of actions and behaviors
- Creation and destruction of instances

Constructs like messages and interaction operators, described later in this chapter, provide further order and structure to these occurrences. When an interaction is executed to validate an ordered set of occurrences in time, called a **trace** the order and structure of these events is used to determine whether the trace is valid.

9.5 Exchanging Messages between Lifelines

Messages can be exchanged between lifelines to achieve interactions. A message can be sent from a lifeline to itself to represent a message that is internal to a part.

A message often represents an invocation or request for service from the sending lifeline to the receiving lifeline, or the sending of a signal from the sending lifeline to the receiving lifeline. A message is shown on a sequence diagram as a line with different arrow heads and annotations depending on the type of message.

Messages are sent by behaviors that are executing on the lifeline, or more precisely, invocation actions, such as send signal or call operation actions, within those behaviors. (See Chapter 8 for more information on invocation actions.) Receipt of a message by a lifeline often triggers the execution of a behavior, but may simply be accepted by a currently executing behavior. Note that there may be a delay between a message being received and handled. The execution of behaviors is shown by solid bars drawn on top of the lifeline tails. When the execution is nested, the bars may also be nested.

Although typically messages are used to model information passed between computer systems and their users, they may also indicate the passage of material or energy. An interaction in a radar-tracking system might represent the detection of a target and the response to that detection. In a production system, the request for manufacture of a car and the subsequent delivery of that car to a dealer might be modeled as an interaction between the dealer and the manufacturer, as shown in Figure 9.4.

9.5.1 Synchronous and Asynchronous Messages

The two basic types of messages are asynchronous and synchronous. A sender of an asynchronous message continues to execute immediately after sending the message, whereas a sender of a synchronous message waits until it receives a reply from the receiver that it has completed its processing of the message before continuing execution.

Asynchronous messages correspond to either the sending of a signal or to an asynchronous invocation (or call) of an operation. A signal is a definition of messages passed asynchronously between objects. Signals are handled by receptions that are

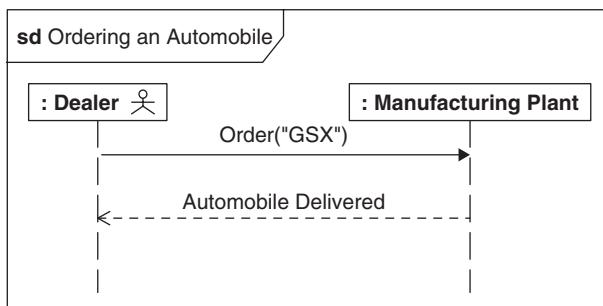


FIGURE 9.4

A simple example of message exchange.

part of the definition of a block or interface. A synchronous message corresponds to the synchronous invocation of an operation. In this case, the reply to the sender is indicated using a separate (optional) message from the receiver back to the sender. See Chapter 6 for a description of the behavioral features of blocks.

Call and send messages can include arguments that correspond to the input parameters of the associated operation or attributes of the associated send signal. Arguments can be literal values, such as numbers or strings; attributes of the part represented by the lifeline; or parameters of the currently executing behavior. A reply message can include arguments that correspond to output parameters or the return value of the operation called. When an operation returns a value, the features to which the output parameters and return value is assigned can be indicated. A feature can either be an attribute of the receiving lifeline or a local attribute or parameter of the receiver's current execution.

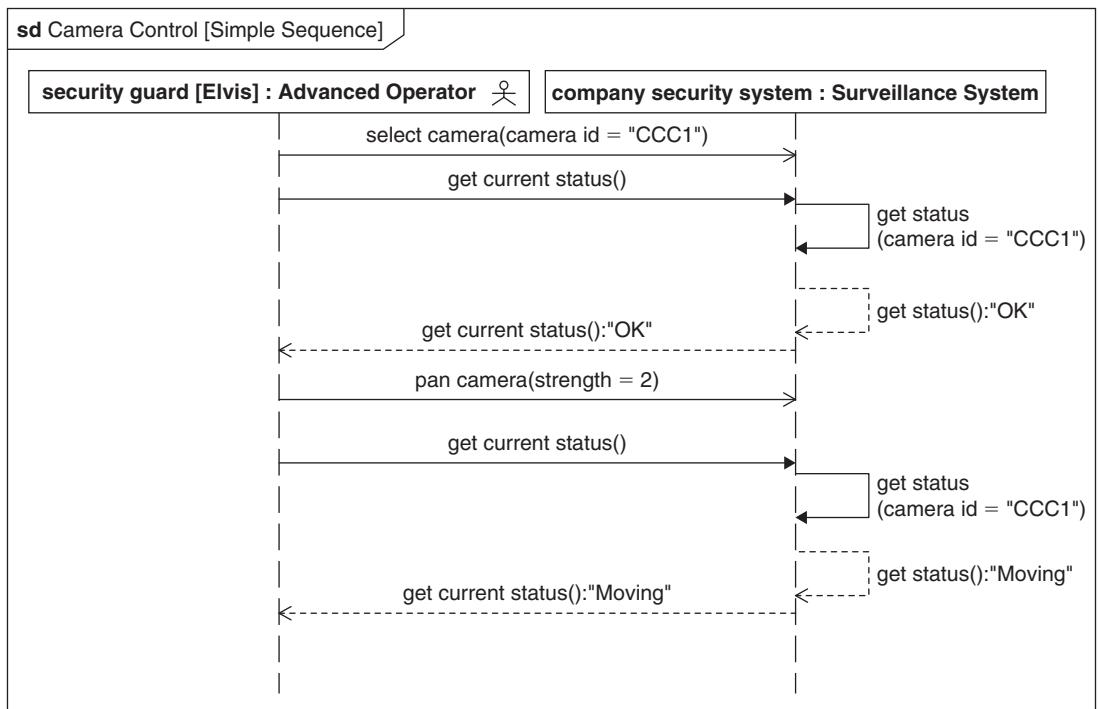
The actual sending of a message implies two occurrences: One is related to a **send message event** that happens to the instance corresponding to the sending lifeline; the other is related to a **receive message event** that happens to the instance corresponding to the receiving lifeline. As one might expect, the sending occurrence has to happen before the receiving occurrence.

Messages are represented by arrows between lifelines. The nonarrow (tail) end represents the occurrence corresponding to the sending of the message, and the arrow end represents the occurrence corresponding to the receipt of the message. The shape of the arrowhead and the line style of the arrow line indicate the nature of the message as follows:

- An open arrowhead means an **asynchronous message**. Input arguments associated with the message are shown in parentheses, as a comma-separated list, after the message name. The operation parameter or signal attribute name may be shown followed by an equal sign before an argument. If this notation is not used, then all the input arguments must be listed in the appropriate order.
- A closed arrowhead means a **synchronous message**. The notation for arguments is the same as for asynchronous messages.
- An open arrowhead on a dashed line shows a **reply message**. Output arguments associated with the message are shown in parentheses after the message name, and the return value, if any, is shown after the argument list. The feature to which the return value is assigned is shown before the message name, followed by an equal sign. As with input arguments, output arguments can be preceded by their corresponding parameter followed by an equal sign. In the rare case that both the parameter name and assigned feature are required, then the following syntax is used:

feature name = parameter name: argument

Figure 9.5 shows a sequence of messages exchanged between the two lifelines introduced in Figure 9.3. The *security guard* first selects camera "CCCI" to interact with. After the *company security system* returns control and stores the guard's selection, he gets that camera's current status, which is "OK." The *company security*

**FIGURE 9.5**

Synchronous and asynchronous messages exchanged between lifelines.

system obtains the status from the selected camera by issuing a subsidiary *get status* request. The *security guard* wishes to move the camera a little, so he or she gives a *pan camera* order (probably via a joystick), and without waiting for the camera to complete the move, asks for the status again, which this time is “*Moving*.”

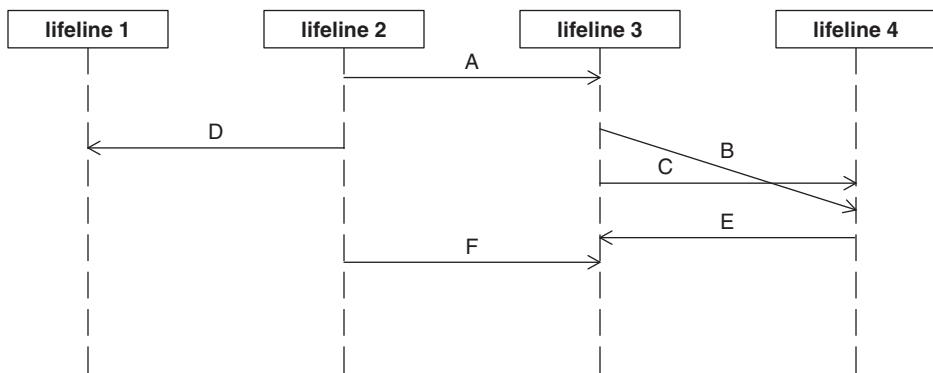
9.5.2 Lost and Found Messages

Normally message exchange is deemed complete; that is, it has both a sending and receiving event. However, it is also possible to describe lost messages, where there is no receiving event, and found messages, where there is no sending event. This capability is useful, for example, to model message traffic across an unreliable network and to model how message loss affects the interaction.

The notation for lost messages is an arrow with the tail on a lifeline and the head attached to a small black circle. The notation for found messages is the reverse—the tail of the arrow attached to a small black circle and the head attached to a lifeline. An example can be seen in this book’s Appendix.

9.5.3 Weak Sequencing

An interaction imposes the most basic form of order on the messages and occurrences that it contains, called **weak sequencing**. Weak sequencing means that

**FIGURE 9.6**

Explanation of weak sequencing.

the ordering of occurrences on a lifeline must be followed, but other than the constraint that message receive occurrences are ordered after message send occurrences, there is no ordering between occurrences on different lifelines.

The messages on the sequence diagram in Figure 9.6 impose an order on send and receive occurrences; for example, *A.send* happens before *A.receive* and *B.send* happens before *B.receive*. Lifelines also impose an order on occurrences, so *lifeline 3* states that *A.receive* happens before *B.send*. However, nothing is said about the ordering of *B.send* and *D.send*. Note also that it is not the messages that are sequenced but their send and receive occurrences. For example, *B.send* happens before *C.send* but *B.receive* happens after *C.receive*. This phenomenon is sometimes referred to as **message overtaking** and is dealt with in more detail in Section 9.6.

9.5.4 Executions

The arrival of a message at a lifeline may trigger the **execution** of a behavior in the receiver. In this case the receiving lifeline executes the behavior (called the method) for the operation or reception that the message represents. Alternatively, the message arrival may simply trigger an executing behavior, for example, a state machine or activity to execute some additional actions. The arguments contained in a call or send message are passed to the behavior that handles it. If and when a reply message is sent, the output arguments are provided to the execution that sent the corresponding synchronous call message.

Lifelines can send messages to themselves. If the message is synchronous, it may cause a new execution to be started, nested within the current execution.

Lifelines are hosts to executions, either of single actions or entire behaviors. The extent to which executions are modeled is down to the modeler. Typically execution-start events are coincident with a message-receipt event, but do not have to be in all cases (i.e., the execution can occur later due to message scheduling delays). When an execution is triggered by the receipt of a synchronous message, its end event may be coincident with the sending of a reply message.

Focus of control bars or activations are overlaid on lifelines and correspond to executions; they begin at the execution's start event and end at the execution's end event. When executions are nested, the focus of control bars are stacked from left to right. If an execution is triggered by the arrival of a message, the arrow is attached to the top of the activation. If an execution ends with the production of a reply message, then the tail of the reply arrow is attached to the bottom of the activation. An alternate notation for activations is a box symbol overlaid on the lifeline with the name of the behavior or action inside.

Figure 9.7 shows the same interaction as Figure 9.5 but with activations added. The relevant behaviors and actions on the *company security system* and *security guard* lifelines are now explicit. The *select camera* operation tells the *company security system* to store a currently selected camera. In a change from Figure 9.5, the action executed to store the camera id, *current camera = camera id*, is explicitly shown here using the box notation. The processing of *get current status* causes a new execution to start that is triggered by a *get status* message with the previously stored camera id as an argument. This new execution ends with a

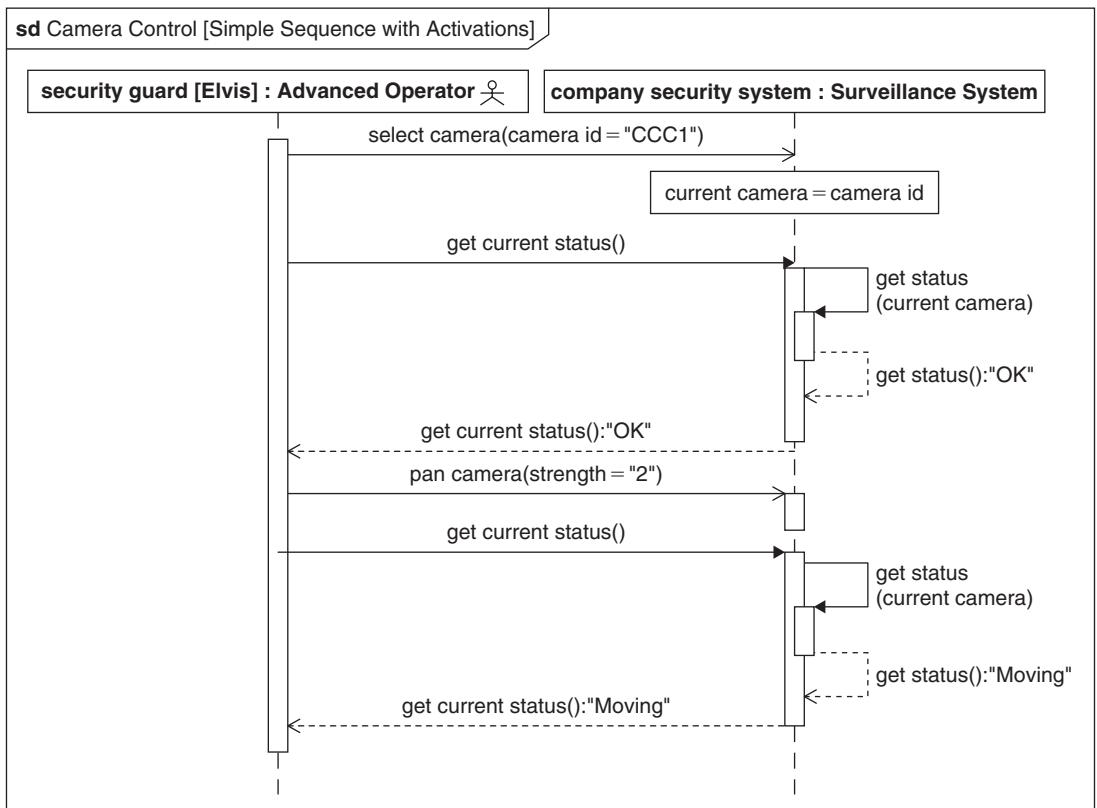


FIGURE 9.7

Lifelines with activations.

status reply of “OK.” After the *pan camera* command, another *get status* message triggers a new execution that returns the result “Moving.” The execution on the *security guard*’s lifeline continues throughout the interaction even while waiting for a response from the *company security system*. There have been suggestions that a “suspension” notation should be introduced to indicate when an execution is waiting but nothing has been standardized as yet.

9.5.5 Create and Destroy Messages

During an interaction, specialized message types, called **create messages** and **destroy messages**, can be used to represent the creation and destruction of instances, which are represented by lifelines. These messages generally apply to the allocation and return of memory to execute software instances. However, this can also be used to add or remove a physical part of a system from a scenario.

The creation of an instance is indicated by a **creation occurrence** and an instance’s destruction by a **destroy occurrence**. Where they occur, they are, respectively, the first and last occurrences on a lifeline.

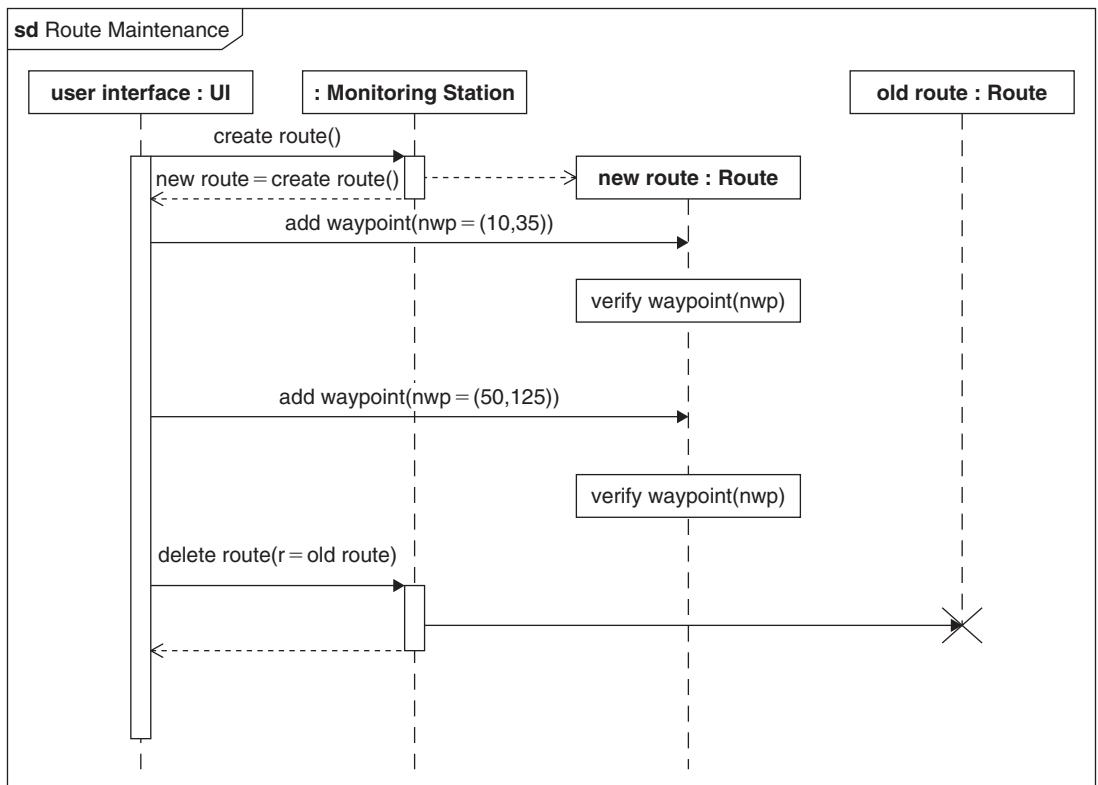
The notation for a create message is a dashed line with an open arrow, terminating on the header box of the lifeline being created; it is moved down in the sequence diagram to accommodate the notation. The dashed “tail” of the lifeline is drawn as normal. The creation message name and input arguments are displayed in the same way as those of a call message.

The notation for a destroy message is a solid line with a filled arrow. Where the arrow terminates on the lifeline of the receiver, the lifeline ends with a cross symbol.

The sequence diagram in Figure 9.8 shows how new routes are created and destroyed by a surveillance system. A *Route* is a set of pan-and-tilt angle pairs that a surveillance camera follows when in an automated surveillance mode. In this case the *user interface* component communicates to the *Monitoring Station* to perform the route maintenance operations. First, the *user-interface* calls the *create route* service offered by the *Monitoring Station*, which in turn creates a new route and returns a reference to the *user interface* via the *new route* attribute. The *user interface* then interacts with this new route in order to add waypoints; finally, when the route is complete (only some of the waypoints are shown here), it uses the *delete route* service to delete *old route*. Note that the execution of action *verify waypoint* is shown using box notation.

9.6 Representing Time on a Sequence Diagram

In a sequence diagram, time progresses vertically down the diagram and, as stated earlier, events on a lifeline are correspondingly ordered. In addition, the send event and receive event for a single message are also ordered in time. However, particularly in distributed systems, a message may be overtaken by a subsequent message sent from the same lifeline; that is, the first message may arrive after receipt of the second message. Sequence diagrams allow this kind of situation to

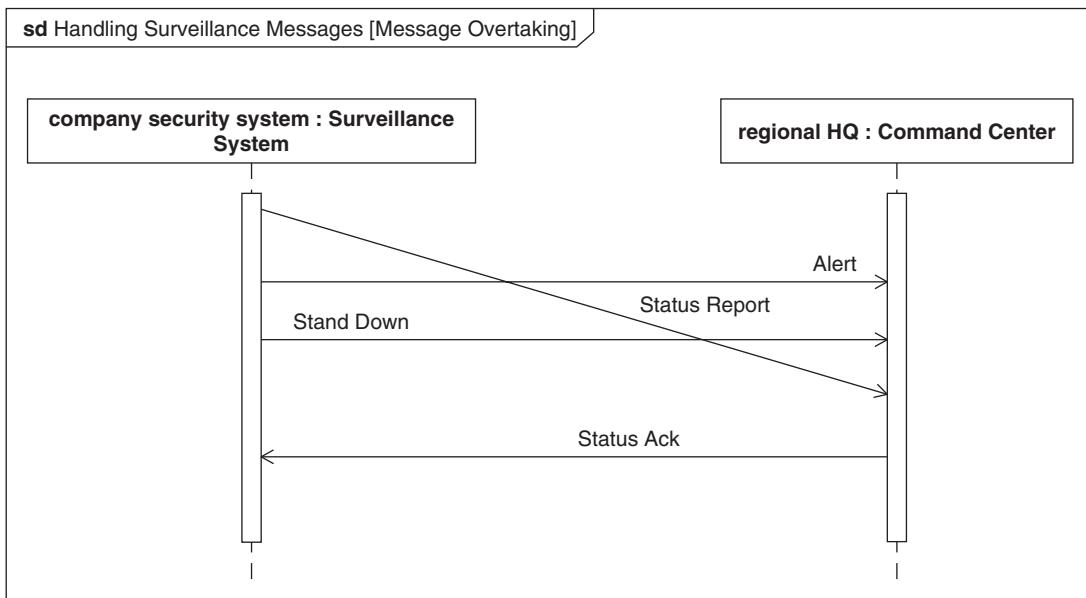
**FIGURE 9.8**

Create and destroy messages.

be drawn using a downwards-slanting arrow between two lifelines, as shown in Figure 9.9.

The sequence diagram in Figure 9.9 shows what happens when an *Alert* message overtakes a regular *Status Report* message. This may be because the *Status Report* message is queued waiting to be processed, or it may indicate a manual process for handling messages. Once the *Alert* message has been received by the *regional HQ*, it defers handling of the *Status Report* message until a *Stand Down* message has been received.

In addition to relative ordering in time, time can be represented explicitly on sequence diagrams. A modeler can use a **time observation** to note the time at some instant during the execution of the interaction, and a **duration observation** to note the time taken between two instants during the execution of the interaction. A **time constraint** and a **duration constraint** can use observations to express constraints involving the values of those observations. A time constraint identifies a constraint that applies to a single event on the sequence diagram. A duration constraint identifies two events, called start and end events, and expresses a constraint on the duration between them. A duration constraint often

**FIGURE 9.9**

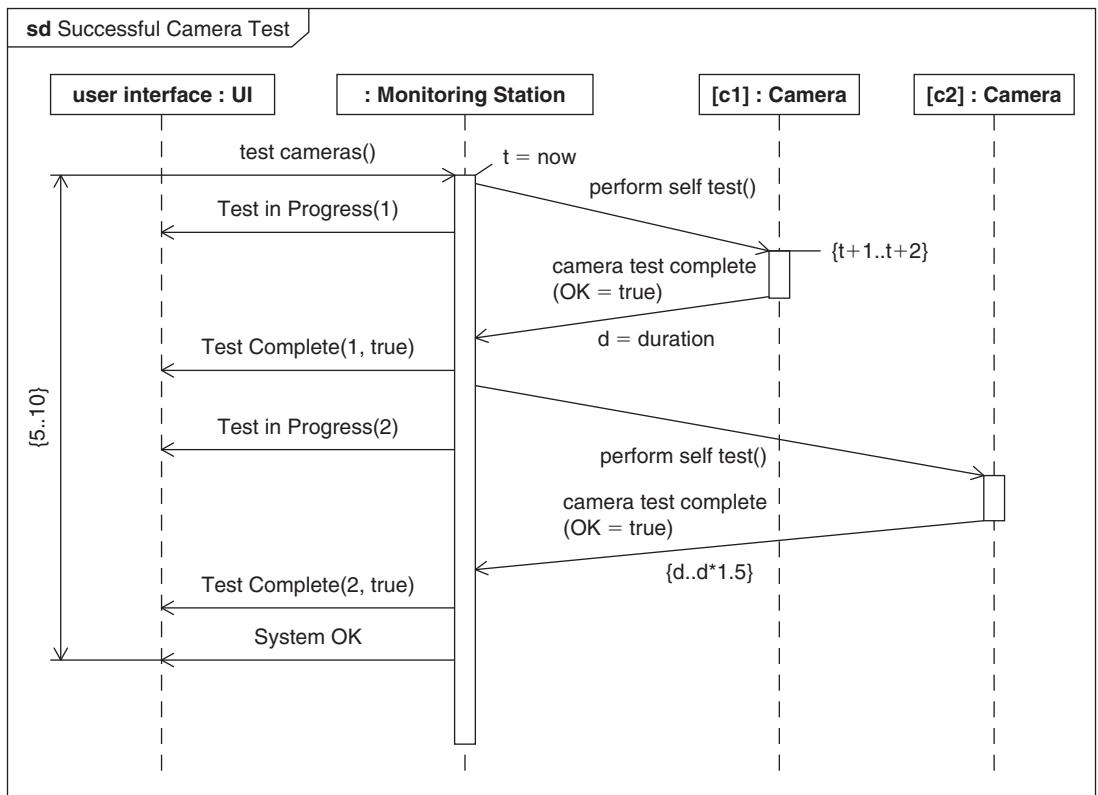
Message overtaking scenario.

applies to any element, such as a message, deemed to have duration or an execution, in which case the constraint applies between the events that bracket the element's duration.

SysML does not mandate a particular model of time. The expressions used in observations and time constraints may assume a single clock or may reference a more complex model of time with multiple clocks.

A time constraint is shown using a standard constraint expression in braces attached by a line to the constrained event. A duration constraint is shown by a double-headed arrow between the two constrained events with the constraint floating near it, also expressed in standard constraint notation (i.e., in braces). A duration constraint may also be shown as a standard constraint floating close to an element with duration, such as a message, or an interaction occurrence. Observations are shown in a similar way to constraints, but instead of an expression in braces, an observation has the name of the observation followed by an equal sign and then some expression indicating how the observation is taken. The actual language used to express observations and constraints, including default time units, and so on, must be stated as part of the observation or constraint.

Figure 9.10 shows a scenario where the *Monitoring Station* is asked by the *user interface* to test the system's cameras. The *Monitoring Station* in turn requests each camera to perform a self-test and awaits the result. While waiting for a response from each camera, the *controller* component internal to the *Monitoring Station* needs to provide a progress indication to the *user interface*, so it uses asynchronous messages to interleave communication. In this case the

**FIGURE 9.10**

Representing time on a sequence diagram.

communication between the *Monitoring Station* and the cameras is over a network, and the communication between the controller and the user interface is local. As a result of network delays, the *Monitoring Station* receives the response from the camera after the progress message is sent. Note that although sloping lines are used here to indicate the passage of time, there is no formal semantic implication in the slope; the only timing implications are expressed using the time and duration constraints and the ordering of events.

Also, a number of observations and constraints on this interaction are expressed, in a time unit of seconds. A time observation, t , is taken at the point when the first self-test message is sent using the expression $t = \text{now}$. A time constraint on the message receipt indicates that the time must be between 1 and 2 seconds after t . The duration between sending and receipt of the first self-test response message is observed via a duration observation d , and there is a constraint on the second response message to not exceed 1.5 times the first duration. The total time taken between the user interface requesting a test command and the completion of both camera self-tests should be between 5 and 10 seconds, as indicated by the duration constraint on the left of the diagram.

9.7 Describing Complex Scenarios Using Combined Fragments

The most basic form of an interaction is, as stated earlier, a weak sequence of occurrences—broadly speaking, read from top to bottom of the sequence diagram. However, more complex patterns of interaction are often needed and can be modeled using constructs called **combined fragments**. Different combined fragments specify different rules for the ordering of messages and their associated occurrences such as parallel and alternative traces.

A combined fragment consists of an **interaction operator** and its **operands**. The interaction operator defines the type of ordering logic, and its operands are all subject to that rule. Each operand has a **guard** containing a constraint expression that indicates the conditions under which it is valid. Each guard is bound to a single lifeline and can only reference attributes of that lifeline in its constraint. The operands may themselves contain combined fragments, and thus can be composed into a tree hierarchy. During execution of an interaction, all operands use weak sequencing semantics on their contents.

A combined fragment must specify which lifelines participate in the interaction defined by the operands. Only the events on the participating lifelines are valid when considering the traces of the fragment.

9.7.1 Basic Interaction Operators

The following subset of interaction operators is used more frequently:

- Seq—weak sequencing, as described in Section 9.5.3. Weak sequencing is the default form of sequencing for all operands, so is rarely used explicitly.
- Par—an operator where operands can occur in parallel, each following weak sequencing rules. There is no implied order between occurrences in different operands. This operator has an alternate shorthand notation, when applied to a single lifeline called a **coregion**, where instead of a frame the operands are bracketed by vertical square brackets.
- Alt/else—an operator where exactly one of its operands will be selected based on the value of its guard. The guard on each operand is evaluated before selection, and if the guard on one of the operands is valid, then that one is selected. If more than one operand has a valid guard then the selection is nondeterministic. An optional else fragment is valid only if none of the guards on the other operands are valid. A common situation is where the choice of operand is based on whether the next occurrence matches the first event in one of the operands. In this case there is no guard.
- Opt—a unary operator that is equivalent to an alt with only one operand. This implies that the operand is either executed or skipped depending on the validity of the guard.
- Loop—an operator where the trace represented by its operand repeats until its termination constraint is met. A loop may define lower and upper bounds

on the number of iterations as well as the guard expression. These bounds are documented in brackets after the loop keyword in the fragment label as: “(lower bound, upper bound),” where the upper bound may have the value “*” indicating an infinite upper bound.

A combined fragment is shown using a frame whose label indicates the type of operator and potentially other information depending on the type of operator.

Alt and par operators have multiple horizontal partitions, separated by dashed lines, that correspond to their operands. Other operators have just a single partition. Messages and possibly other combined fragments are nested within each operand. Where an operator has a single operand that is itself a combined fragment, their frames can be merged into one, and the frame label for the merged frame is used to indicate all the contents such as **loop par**.

The frame symbol for the combined fragment must not obscure the lifelines that participate in its interaction, so the tails of the participating lifelines are visible on top of the frame. The frame does obscure the lifelines that do not participate in the fragment’s interaction.

In Figure 9.11, lifelines 1 through 3 participate in the **opt** fragment, but only lifelines 1 and 4 participate in the **loop** fragment. So, to maintain the current layout, lifelines 2 and 3 are obscured by the **loop** frame to indicate that they do not participate.

Figure 9.12 shows what happens when an intruder is detected by the *company security system* and tracked. The interaction is started when some lifeline external to this interaction detects a potentially illegal entry into the monitored areas. This triggers the system to alert the user (*the security guard*) with the id of the sensor and raise the alarm. The *security guard* then locates the sensor and attempts to find and track the intruder and eventually (in this case) cancels the alert.

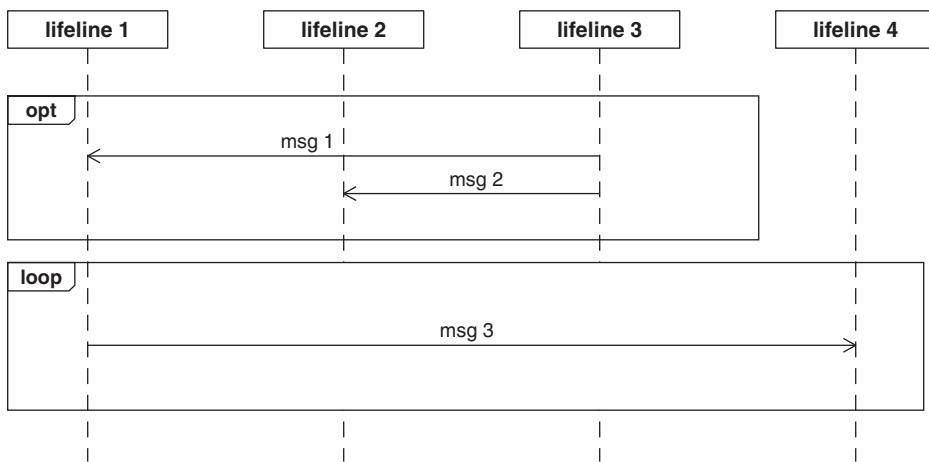
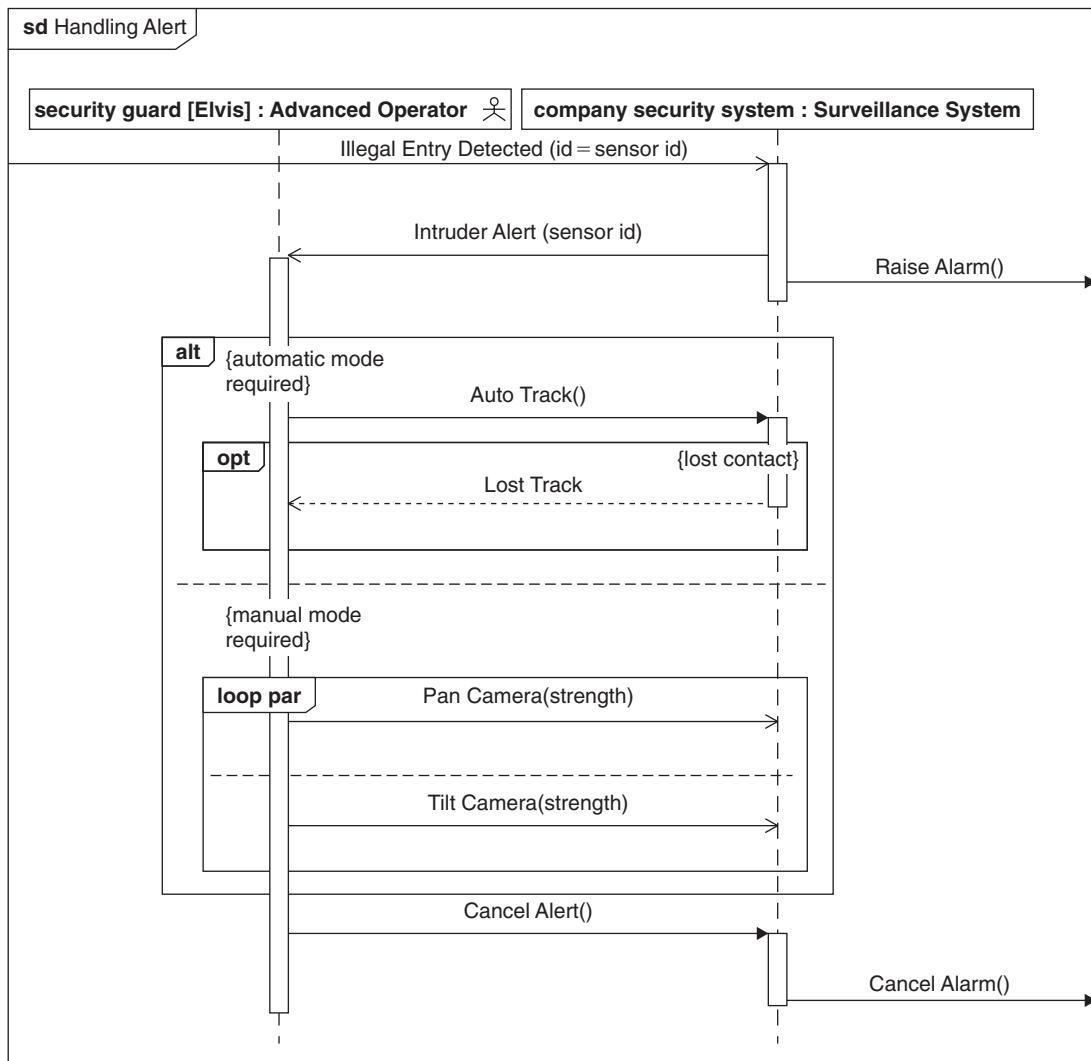


FIGURE 9.11

Example of overlapping and nonoverlapping lifelines.

**FIGURE 9.12**

Complex interaction described using interaction operators.

Within this sequence, the **alt** operator indicates that the *security guard* has a choice between using the system's auto-track feature and manually tracking the intruder. In the automatic case, the system attempts to acquire and track a target. Failure to acquire a target, or loss of an acquired target, is indicated by a *Lost Track* message. In the manual-tracking case, the *security guard* uses an input device to repeatedly pan and tilt the cameras, as indicated by the **loop par** fragment.

In all scenarios, the *security guard* is responsible for canceling the alert, which prompts the *company security system* to cancel the alarm. In this case the *Raise Alarm* and *Cancel Alarm* messages terminate at gates on the frame,

to interact with lifelines outside the current interaction (see Section 9.8 for a description of gates).

9.7.2 Additional Interaction Operators

The following are other interaction operators that are not as commonly used.

- Strict—like “seq” except that the occurrences represented by its operands are sequenced in order across all participating lifelines. The strict rule does not apply to the operands of any nested combined fragments.
- Break—an operator whose operand is executed rather than the remainder of the enclosing fragment. This is often used to represent the handling of exceptional scenarios.
- Critical—an operator where the sequence of operands must take place with no interleaving of other occurrences, at least within the participating lifelines of the fragment. This may be used when some higher-level **par** operator indicates that interleaving can occur, and this operator is used to constrain the interleaving.
- Neg—an operator where the traces described by its operand cannot occur.

There are cases in interaction modeling where covering all potential message occurrences is very onerous, such as a where there are a large number of occurrences related to messages that are not relevant to the scenario being described. For these cases, the following operators provide the ability to filter messages in their operand:

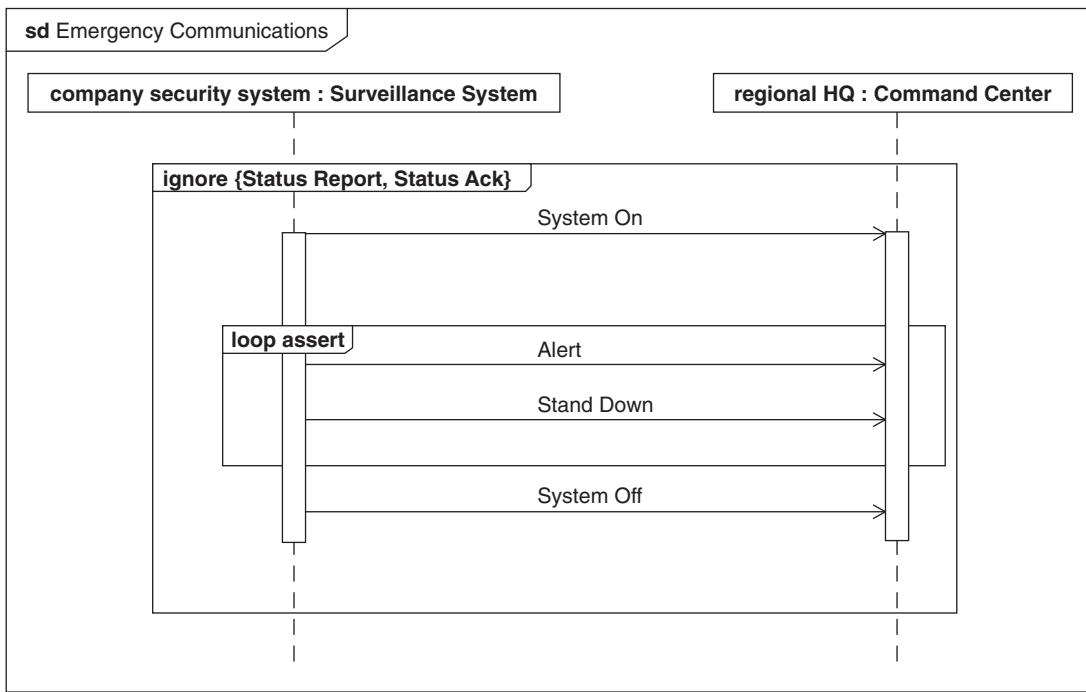
Consider—only consider messages for a specified set of operations and/or signals. All event occurrences corresponding to other messages are ignored; that is, they are not considered for validity. Only considered messages can appear in the operand.

Ignore—do not consider messages for a specified set of operations and/or signals. Event occurrences corresponding to ignore messages are not considered for validity. Ignored messages cannot appear in the operand.

Consider and ignore operators allow occurrences and messages that have been explicitly ignored (or not considered) to be interleaved with valid traces of their operand. The **assert** operator provides a mechanism to assert that only the occurrences in its operand are valid, even if according to an enclosing consider or ignore fragment, ignored (not considered) events could occur.

The messages to be ignored or considered are shown in braces following the keyword in the fragment label.

Figure 9.13 describes the sequence of messages exchanged when the *company security system* is communicating with the *regional HQ* in an emergency. There are always regular status updates and acknowledgments between any surveillance system and the *regional HQ*, but these are not of interest in this scenario and so are ignored, as indicated by the ignore fragment. Alerts are only

**FIGURE 9.13**

Message-filtering scenario.

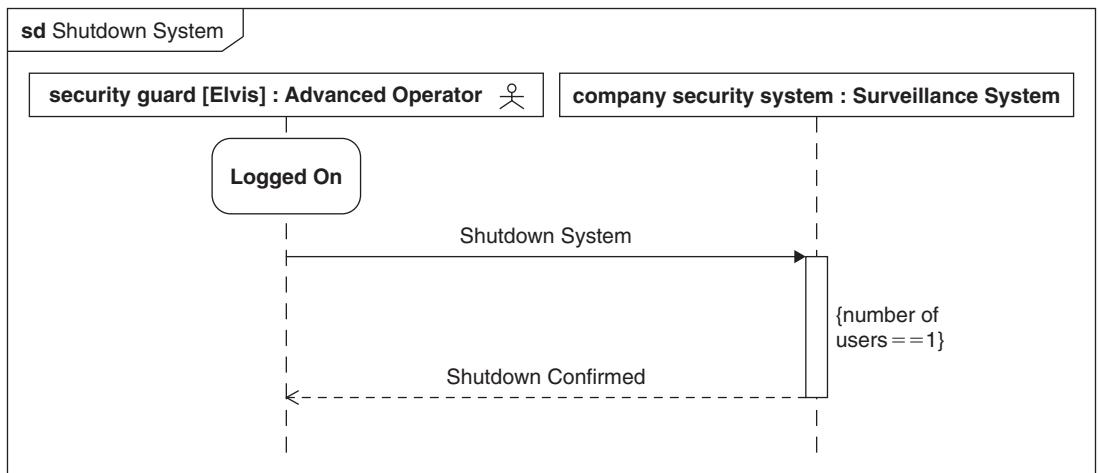
going to happen while the surveillance system is on, so the *regional HQ* can discount any alerts apparently received when the system is off (although they may wish to investigate why they happened). However, when a valid *Alert* message has been sent, there must be no other messages, including status messages, until a *Stand Down* message has been received. Given that status messages are being ignored, the only way to be explicit about their exclusion is to use an assert fragment, making it explicit that no other messages will be sent between *Alert* and *Stand Down*.

9.7.3 State Invariants

It is often useful to augment the message-oriented expression of valid traces by adding constraints on the required state of a lifeline at a given point in a sequence of event occurrences. This can be achieved using a **state invariant** on a lifeline. The invariant constraint can include the values of properties or parameters, or the state (of a state machine) that the lifeline is expected to be in.

The notation for state invariants is an expression in braces shown on top of the lifeline that is constrained. If the invariant specifies the state of a state machine, then it is shown as a state symbol on the lifeline.

Figure 9.14 shows a scenario for shutting down the system. The state invariant on the *security guard's* lifeline indicates that he or she has to be logged on for

**FIGURE 9.14**

State invariants.

the *Shutdown System* message to be valid. The state invariant on the *company security system* lifeline indicates, for a shutdown request to be valid, the number of users must be one; that is, there are no other users currently logged on. A valid trace ends with a *Shutdown Confirmed* message reply to the *security guard*.

9.8 Using Interaction References to Structure Complex Interactions

In most systems engineering projects, the size of systems and hence often the size of interactions becomes very large. There are also many patterns of interaction, for example, initialization and shutdown, which are repeated many times as parts of different scenarios.

To support large-scale uses of interactions, any interaction may reference (the official term is **interaction use**) one or more existing interactions described on other sequence diagrams. Interactions can be nested such that one interaction can use an interaction that in turn uses still others. This capability significantly enhances the scalability of interactions. It also facilitates reuse since an interaction can be used by more than one interaction. The using interaction identifies the participants in the used interaction. The interaction's definition must have lifelines that represent all the identified participants, but may include additional lifelines as well.

To allow messages to pass into and out of an interaction when it is being used by another, an interaction can have connection points, called **formal gates**, at its boundary. There is a gate for every message that enters or leaves the interaction at its boundary. When the interaction is used, the using interaction has **actual gates** that correspond one-to-one with the formal gates of the used interaction.

The messages arriving or leaving the actual gates must match those arriving or leaving at their corresponding gates in terms of direction, type, and cause (signal/operation).

In the definition of an interaction, messages can connect to the frame of the interaction. There is a formal gate at each connection point, although there is no symbol representing the gate itself. Gates can be named but the name is typically not shown. An example of such a definition appears in Figure 9.12.

Interaction uses are shown as frames with the keyword **ref**. The body of the frame contains the name of the used interaction. Messages that terminate/start at the boundary of the frame imply the presence of actual gates. Lifelines that participate in the nested interaction are obscured by the frame symbol. Note that this is opposite of how participants are represented on combined fragments, where participants are not obscured.

Figure 9.15 shows an interaction that uses four other interactions, as indicated by **ref**. The first-used interaction describes the *company security system* being set up by the *security guard*. During the guard's shift, one of two things are shown as potentially occurring. If things are quiet (normal status), the guard might perform some maintenance on the automated surveillance routes (the scenario in Figure 9.8), or the guard and the system might handle an alert (the scenario from Figure 9.12). These two alternatives may occur repeatedly as indicated by the **loop alt** fragment, until the guard shuts down the system. To use the *Handling Alert* interaction, this interaction needs to attach compatible messages to all its gates.

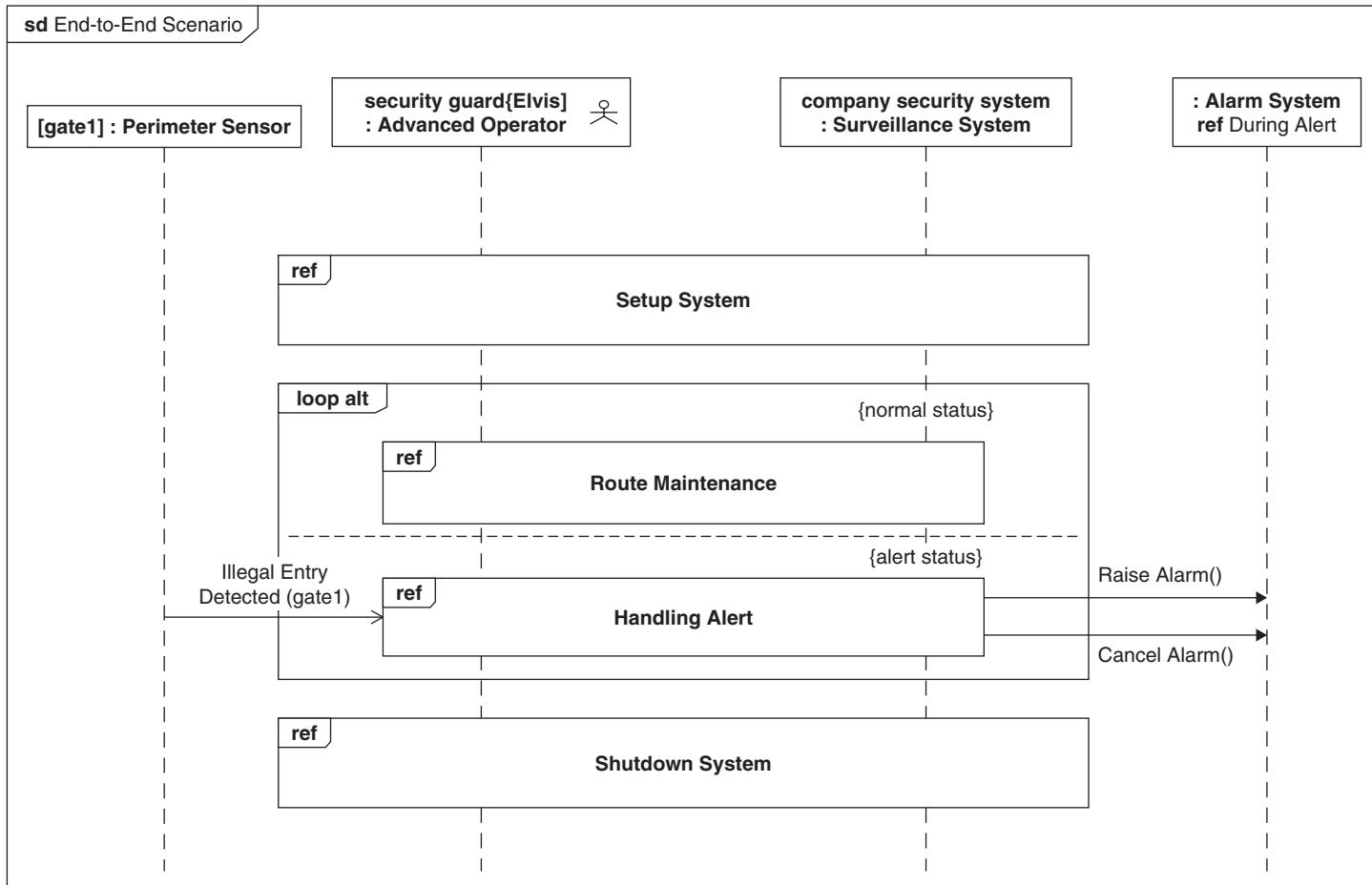
9.9 Decomposing Lifelines to Represent Internal Behavior

A lifeline can represent a part that corresponds to a usage of a block for the specified interaction. The block can correspond to a system or component at any level of a system hierarchy. Each lifeline may be further decomposed based on the constituent parts of the block.

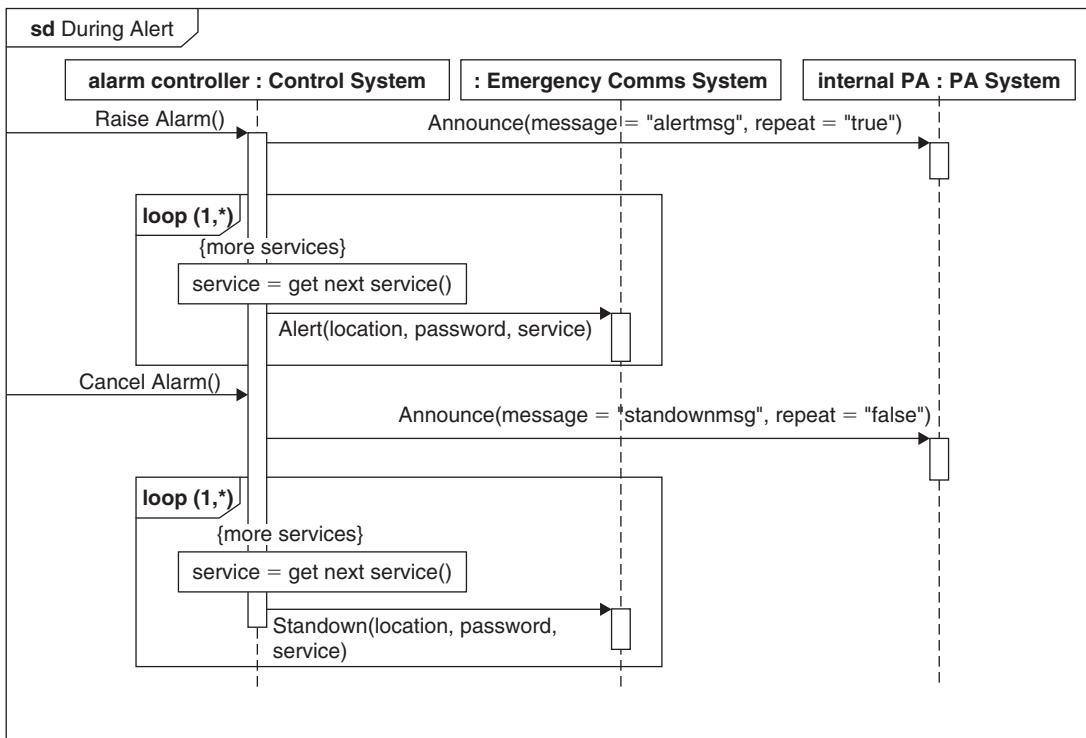
A sequence diagram includes the provision to decompose the lifeline and further elaborate the interaction among its parts. If, for example, the lifeline represents a system, an interaction can be specified between the system and its external environment. This is often referred to as a black-box interaction, where the internal behavior of the system is hidden and only external behavior is visible. The system lifeline can then be decomposed into its parts to specify a nested interaction that supports the black-box interaction.

The interaction among these parts is defined by a separate interaction that is used by the parent lifeline being decomposed. The used interaction includes gates that correspond to where it sends or receives its messages. The messages at the gates of this interaction must be compatible with the messages of the parent lifeline, and the message send and receive events must occur in the same order as on the parent lifeline. Only lifelines representing parts of the block that type the parent lifeline may appear in the used interaction.

The **lifeline decomposition** is shown by adding the name of the referenced interaction below the name of the lifeline, prefixed by the keyword **ref**. The same

**FIGURE 9.15**

Reference to another interaction.

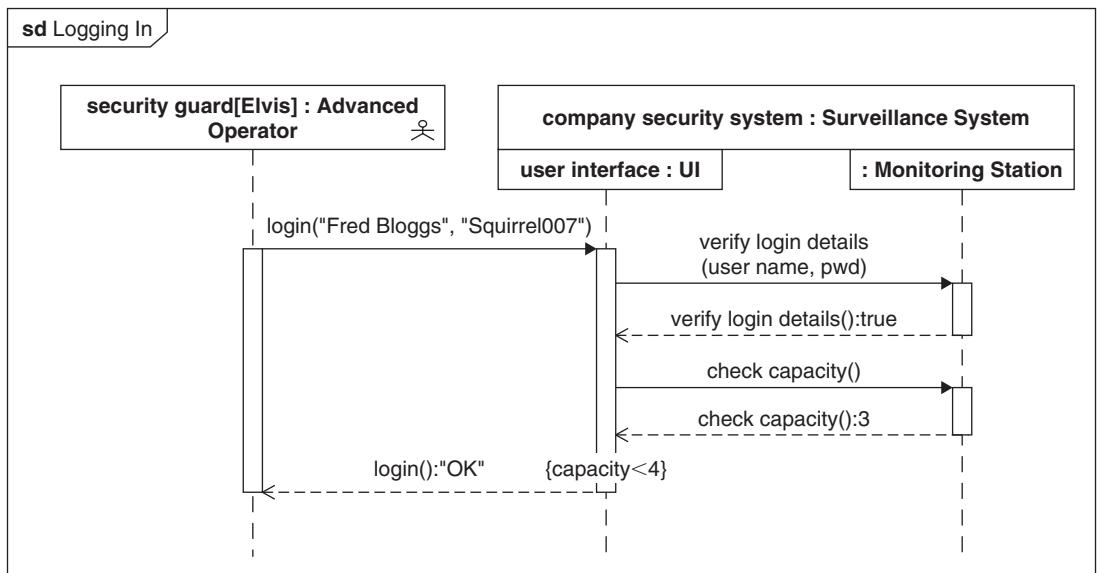
**FIGURE 9.16**

A decomposed lifeline.

name is used in the frame label of the referenced interaction. Gates are implied in the reference interaction by messages that start or terminate on its frame.

Figure 9.16 shows the decomposition of the black-box lifeline for the *Alarm System* from Figure 9.15. It shows how the *Alarm System* handles alerts. When the *alarm controller* receives a *Raise Alarm* message, it requests an announcement on the *internal PA*, and then alerts all the registered emergency services through the *Emergency Comms System*; it provides a *location* and a *password* to authenticate the alert. When the *Cancel Alarm* message is received, the *alarm controller* requests another announcement and then sends a request to the emergency services to stand down. At least one emergency service must be alerted, but the maximum number may depend on circumstances.

There is an alternative to using the reference sequence diagram for representing a nested interaction. This is accomplished by showing the lifeline with its nested parts on the same sequence diagram. This is depicted on the diagram by showing the black-box lifeline on top of the lifelines corresponding to the nested parts. The header boxes of the parts are attached to the underside of the parent lifeline's header box. The nested lifelines can be used to show interactions that occur within the parent lifeline, or to send and receive messages directly to and from other external lifelines.

**FIGURE 9.17**

Inline nesting of lifeline decomposition.

Figure 9.17 shows a white-box view of what happens when the *security guard* wishes to log in to the *company security system*. The two significant parts of the *company security system*—the *user interface* and the *Monitoring Station*—are shown underneath the lifeline of the *company security system*. In this scenario a *login* message is received by the *user interface* and requests the *Monitoring Station*—to verify it. The *user interface* then checks that the maximum number of logins has not been exceeded and returns control to the *security guard*.

9.10 Summary

Sequence diagrams describe interactions used to capture system scenarios as a set of message exchanges between lifelines. An interaction is specified as an ordered sequence of events that result from the execution of behaviors by parts of the owning block represented by the lifelines. The most significant source of events is the exchange of messages between lifelines that may trigger executions and the creation of new instances in the system. The following list highlights key aspects of interactions.

- Lifelines represent parts (or references to parts) of the block that owns the interaction. During execution, a lifeline may represent only one instance; so where the part has an upper bound greater than 1, an additional selector expression is required to specify exactly one of the instances represented by the part. Lifelines may run from top to bottom of the sequence diagram indicating that the parts they represent exist before and after the execution of

the interaction. They also may start and/or end within the sequence diagram, indicating the creation or destruction of instances during execution. Lifelines may be physically nested on a diagram to show a white-box view of the interactions within that lifeline. State invariants on the lifelines assert conditions that must hold at that point for the interaction to be valid.

- Messages are exchanged between lifelines and typically represent an invocation of an operation or a sending of a signal. Messages do not represent data flows, but the flow of data (or other items such as matter or energy) can be captured via arguments of the message. Messages are sent and received by behaviors executing on the lifelines and can be either asynchronous (sender continues executing) or synchronous (sender waits for a response).
- The default ordering of events imposed by an interaction is weak sequencing, where unrelated event occurrences are sequenced within but not across lifelines. A combined fragment is a means for specifying different ordering semantics. A combined fragment includes an operator and operands, where the operator identifies the ordering of its operands that may themselves be combined fragments. Commonly used operators include **par**, **alt**, and **loop**. Each operand may have a guard expression that must be satisfied in order for the operand to be executed.
- Interactions can use other interactions as part of their definition to enhance scalability, as denoted by the keyword **ref**. An interaction can use another interaction to describe the internal interactions of one of its lifelines; this enables a black-box specification style. An interaction can also use another to specify part of its total behavior, which may involve a number of its lifelines. This decomposition is either done to reduce the size of a sequence diagram, or to reuse some common interaction pattern. Interaction frames can feature connection points on their perimeter, called gates, to enable messages to pass across interaction boundaries.

9.11 Questions

1. What is the diagram kind for a sequence diagram, and which type of model element does it represent?
2. What is the context for an executing interaction?
3. Draw a sequence diagram with two lifelines: one representing a part with no name, typed by the actor “Customer,” and the other with the name “m,” typed by the block “Vending Machine.”
4. What is a selector expression used for?
5. Which kinds of event are relevant when specifying interactions?
6. List the different types of messages that can be exchanged between lifelines.
7. On the diagram from Question 3, add a message from the “Customer” lifeline to the “Vending Machine” lifeline representing the signal “Select Product” with the argument “C3.”

8. What does the term “message overtaking” mean?
9. How is an action or behavior execution represented on a sequence diagram?
10. What is an observation and how is it used?
11. In the diagram from Question 7, observe the current time (provided by the “clock” function) when the “Select Product” message is sent?
12. How is a combined fragment represented on a sequence diagram?
13. Name four common interaction operators.
14. In the diagram from Question 7, change “Select Product” from a signal to an operation on “Vending Machine” and show two different replies: If the machine has stock, then it replies with the return string “Stock Available”; otherwise, it replies with the string “Sold Out.”
15. Messages M1 and M2 from lifeline L2 can occur in any order on lifeline L1. Show two different ways that this can be expressed on a sequence diagram.
16. Are the lifelines that participate in a combined fragment shown in front of or behind the frame box for the combined fragment?
17. Which messages are valid inside an ignore fragment?
18. What does a state invariant specify?
19. What are gates used for?
20. Name two ways of showing the interaction between the children of a lifeline.
21. Are the lifelines that participate in an interaction occurrence shown in front of or behind the frame box for the interaction occurrence?

Discussion Topic

Sequence diagrams can be used to capture test specifications or test results. What differences would you expect to see between sequence diagrams used for these two purposes?

10

Modeling Event-Based Behavior with State Machines

This chapter describes how to model behavior in terms of the response of blocks to internal and external events, using state machines.

10.1 Overview

State machines typically are used in SysML to describe the state-dependent behavior of a block throughout its life cycle in terms of its states and the transitions between them. A state machine for a block may be started, for example, when it initiates power up, transitions through multiple states in response to different stimuli, and terminates when it completes power down. In each state, the block may perform different sets of actions. Thus, the state machine defines how the block's behavior changes as it transitions through different states. State machines in SysML can be used to describe a wide range of state-related behavior, from the behavior of a simple lamp switch, to the complex modes of an advanced aircraft.

Although a state machine is a behavior, and therefore can be called from an activity or referenced by an interaction, the semantics of these combinations are not completely clear, so they should be used with care.

State machines are normally owned by blocks and execute within the context of an instance of that block. (It is possible for a state machine to be owned by a package, but its usefulness is much restricted so that particular use will not be covered here.) The behavior of a state machine is specified by a set of regions, each of which defines a set of states. The states in any one region are exclusive; that is, when the region is active, exactly one of its substates is active. A region normally has an initial pseudostate, which is the place where the region starts when it first becomes active. When a state is entered, an (optional) entry behavior (e.g., an activity) is executed. Similarly on exit, an optional exit behavior is executed. While in a state, a state machine can execute a behavior called a do activity. It also normally has a final state that, when active, signifies that the region has completed. Regions and states are described in Section 10.3.

Change of state is effected by transitions that connect a source state to a target state. Transitions are defined by triggers, guards, and effects, where the trigger indicates an event that can cause a transition to the target state, the guard is evaluated in order to test whether the transition is valid, and the effect is a behavior executed once the transition is triggered. Triggers may be based on a variety of events such as the expiration of a timer, or the receipt of a signal by the state machine's owning object. Junction and choice pseudostates support the construction of compound transitions between states, with multiple guards and effects. Transitions are described in Section 10.4. Operation calls on the owning block are also valid trigger events for transitions; these are described in Section 10.5.

State machines in different blocks may interact with one another by either sending signals or invoking operations. For example, the state machine of one block can send a signal to another block as part of a transition effect or state behavior. The event corresponding to the receipt of this signal by the receiving block can trigger a state transition in its state machine. Similarly, a state machine in one block may call an operation on another block that causes an event that triggers a transition.

The rest of the chapter covers more advanced state machine concepts. Section 10.6 deals with state hierarchies that occur when a state contains its own regions. A state with just one region is the most common case and is called a composite state. A state with more than one region is called an orthogonal composite state. Finally, a kind of state called a submachine state, may reference another state machine. To model state hierarchies effectively, additional constructs are needed. Fork and join pseudostates are needed to specify transitions into and out of orthogonal composite states. Entry and exit point pseudostates can be used to add connection points for transitions on the boundary of a state or state machine.

State machines may also be used to define continuous behaviors, as described in Section 10.7, where a set of discrete states of a block, and changes in that state, are defined in terms of the values of other continuous variables such as heat and pressure.

State machines can be used in conjunction with other behaviors. A state machine can use another behavior (e.g., an activity) to specify what happens on state entry and exit, or when a transition fires. State machine states are also used within interactions (see Section 9.7.3) and activities (see Section 8.8.4) to constrain certain aspects of their behavior.

10.2 State Machine Diagram

State machine diagrams are sometimes referred to as **state charts** or state diagrams, but the actual name in SysML is the state machine diagram. The frame label has the following form:

stm [State-Machine] state machine name [diagram name]

The diagram kind for a state machine diagram is **stm**. The diagram frame always represents a state machine, and therefore the type of the model element is

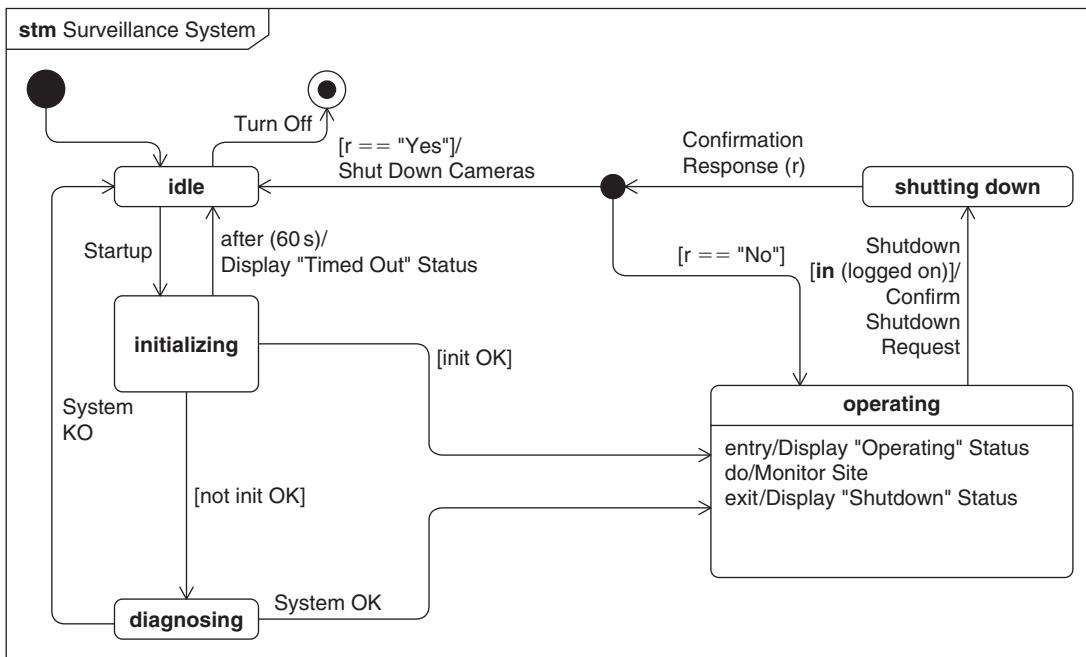


FIGURE 10.1

A state machine.

always *State Machine* and is often elided. The *state machine name* is the name of the represented state machine, and the *diagram name* is user defined and is intended to describe the purpose of the diagram. Figure 10.1 shows many of the basic notational elements for describing state machines.

Figure 10.1 describes a state machine for an ACME *Surveillance System*. It starts in the *idle* state; runs through a series of states during its life cycle; and, finally, ends up at *idle* again, from where it may receive a *Turn Off* signal that causes it to complete its state machine behavior. The notation for the state machine diagrams is shown in the Appendix, Tables A.18 through A.20.

10.3 Specifying States in a State Machine

A **state machine** is a potentially reusable definition of some state-dependent behavior. State machines typically execute in the context of a block, and events experienced by the block may cause state transitions.

10.3.1 Region

A state machine can contain one or more regions, which together describe the state-related behavior of the state machine. Each **region** is defined in terms of

states and **pseudostates**, collectively termed vertices, and transitions between those vertices. An active region has exactly one active state within it. The difference between a state and a pseudostate is that a region can never rest in pseudostate; it merely exists to help determine the next active state.

Where a state machine has multiple regions, they may be describing some concurrent behavior happening within the state machine's owning block. This may in turn be an abstraction of the behavior of different parts within the block, as discussed in Section 6.5.1. For example, one part of a factory may be storing incoming material, another turning raw material into finished products, and yet another sending out finished goods. If the parts are ever specified with behaviors of their own, the modeler has to be clear on the relationship between the state machine for the parent block and the behaviors of its parts. It may also be that the state machine needs to keep track of concurrent behavior in its environment such as a camera being panned and tilted at the same time.

States can also contain multiple regions, as described in Section 10.6.2, but this section confines itself to simple states with only a single region. Where a state machine or state contains a single region, it typically is not named, but where multiple regions are present, it often makes sense to name them.

The initialization and completion of a region are described using an initial pseudostate and final state, respectively. An **initial pseudostate** specifies the initial state of a region. The outgoing transition from an initial pseudostate may include an effect (see Section 10.4.1 for a detailed discussion of transition effects). Such effects are often used to set the initial values of value properties used by the state machine.

When the active state of a region is the **final state**, the region has completed and no more transitions take place within it. Hence, a final state can have no outgoing transitions.

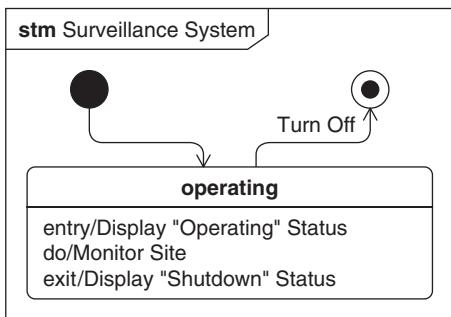
The **terminate pseudostate** is always associated with the state of an entire state machine. If a terminate pseudostate is reached, then the behavior of the state machine terminates. A terminate pseudostate has the same effect as reaching the final states of all the state machine's regions. The termination of the state machine does not imply the destruction of its owning object, but it does mean that the object will not respond to events via its state machine.

A single region is represented by the area inside the frame of the state machine diagram. The notation for the concepts introduced thus far is as follows:

- An initial pseudostate is shown as a filled circle.
- A final state is shown as a “bulls-eye”; that is, a filled circle surrounded by a larger hollow circle.
- A terminate pseudostate is shown as an “X”.

10.3.2 State

A **state** represents some significant condition in the life of a block, typically because it represents some change in how the block responds to events. This condition can be specified in terms of the values of selected properties of the block, but typically the condition is expressed in terms of an implicit state variable

**FIGURE 10.2**

A state machine containing a single state.

(or variables) corresponding to its regions. It is sometimes helpful to think of a state as corresponding to a switch position for the block, where the block can exhibit some specified behavior in each switch position. A state machine can define all valid switch positions (i.e., states) of the system. Switch positions can correspond to a truth table similar to how logic gates can be specified.

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 activity** that executes once the entry behavior has completed and continues to execute until it completes or the state is exited. Although any SysML behavior can be used, typically entry and exit behaviors and do activities are activities or opaque behaviors.

A state is represented by a round-cornered box containing its name. Entry and exit behaviors and do activities are described as text expressions preceded by the keywords “entry,” “exit,” or “do” and a forward slash. There is some flexibility in the content of the textual expression. The text expression typically is the name of the behavior, but where the behavior is an opaque behavior, the body of the opaque behavior can be used instead.

Figure 10.2 shows a simple state machine for the *Surveillance System*, with a single *operating* state in its single region. A transition from the region’s initial pseudostate goes to the *operating* state. On entry, the *Surveillance System* displays the fact that it is operational on all operator consoles, and on exit, it displays a shutdown status. While the *Surveillance System* is in the *operating* state, it performs, via a do activity, its standard function of *Monitor Site*; that is, monitoring the building where it is installed for any unauthorized entry. When in the *operating* state, a *Turn Off* signal triggers a transition to the final state, and then the region, and hence the state machine has completed.

10.4 Transitioning between States

A **transition** specifies how states change within a state machine. State machines always run to completion once a transition is triggered, which means that they

are not able to consume another trigger event until the state machine has completed the processing of the current event.

10.4.1 Transition Fundamentals

A transition may include one or more triggers, a guard, and an effect as described next.

Trigger

Triggers identify the possible stimuli that cause a transition to occur and are associated with events. The four main types of events are:

- **Signal events** indicate that a new asynchronous message has arrived. A signal event may be accompanied by a number of arguments that can be used in the transition effect as described later.
- **Time events** indicate either that a given time interval has passed since the current state was entered (relative), or that a given instant of time has been reached (absolute).
- **Change events** indicate that some condition has been satisfied (normally that some specific set of attribute values hold). Change events are discussed in more detail in Section 10.7.
- **Call events** indicate that an operation on the state machine's owning block has been requested. A call event may also be accompanied by a number of arguments. Call events are discussed in more detail in Section 10.5.

Once the entry behavior of a state has completed, transitions can be triggered by events irrespective of what is happening within the state. For example, a transition may be triggered while a do activity is executing, in which case the do activity is interrupted.

By default, events must be consumed as soon as they are presented to the state machine, even if they do not trigger transitions. However, events may be explicitly deferred while in a specific state for later handling. In that case, unless they actually do trigger a transition, they are not consumed as long as the state machine remains in that state. As soon as the state machine has entered a state where the event is not deferred, the event must be consumed immediately. It may be consumed to trigger a transition but, if not, then it is consumed anyway and has no effect.

Transitions can also be triggered by internally generated **completion events**. For a simple state with no internal states, a completion event is generated when the entry behavior and the do activity have completed.

Guard

The **transition guard** contains an expression that must be true for the transition to occur. The guard is specified using a constraint, introduced in Chapter 7, that includes a textual expression to represent the guard condition. When an event satisfies a trigger, the guard on the transition, if present, is evaluated. If the guard evaluates to true, the transition is triggered, and if the guard evaluates to false,

then the event is consumed with no effect. Guards can test the state of the state machine using the operators **in** (state x) and **not in** (state x).

Effect

The third part of the transition is the **transition effect**. The effect is a behavior, normally an activity or an opaque behavior, executed during the transition from one state to another. For a signal or call event, the arguments of the corresponding signal or operation call can be used directly within the transition effect, or the arguments can be assigned to attributes of the block owning the state machine. The transition effect can be an arbitrarily complex behavior that may include send signal actions or operation calls used to interact with other blocks.

If the transition is triggered, then first the exit behavior of the current (source) state is executed, then the transition effect is executed, and finally the entry behavior of the target state is executed.

A state machine can contain transitions, called internal transitions, that do not effect a change in state. An internal transition has the same source and destination and, if triggered, simply executes the transition effect. By contrast, an external transition with the same source and destination state—sometimes called a “transition-to-self”—triggers the execution of that state’s entry and exit behaviors as well as the transition effect. One frequently overlooked consequence of internal transitions is that, because the state is not exited and entered, timers for relative time events are not reset.

Transition Notation

A transition is shown as an arrow between two states, with the arrow pointing to the target state. Transitions to self are shown with both ends of the arrow attached to the same state. Internal transitions are not shown as graphical paths but are listed within the state symbol.

The definition of the transition’s behavior is shown in a formatted string on the transition with the list of triggers first, followed by a guard in square brackets, and finally the transition effect preceded by a forward slash. Section 10.4.3 describes an alternate graphical syntax for transitions.

The text for a trigger depends on the event, as follows:

- *Signal and call events*—the name of the signal or operation followed optionally by a list of attribute assignments in parentheses. Typically, call events are distinguished by including the parentheses even when there are no attribute assignments, although this is just a (useful) convention, not part of the standard notation.
- *Time events*—the term “after” or “at” followed by the time; “after” indicates that the time is relative to the moment when the state is entered; “at” indicates that the time is an absolute time.
- *Change events*—the term “when” followed by the condition that has to be met in parentheses. Like other constraint expressions, the condition is expressed in text with the expression language optionally in braces.

The effect expression may either be the name of the invoked behavior or may contain the text of an opaque behavior.

When an event is deferred in a state, the event is shown inside the state symbol for that state using the text for the trigger followed by a "/" and the keyword defer. See Figure 10.12 (page 260) for an example.

Transitions can also be named, in which case the name may appear alongside the transition instead of the transition expression. A name is sometimes a useful shorthand for a very long transition expression.

Figure 10.3 shows a more sophisticated state machine for the *Surveillance System* than in Figure 10.2, with all the principal states and the transitions between them. Compared to Figure 10.2, the initial pseudostate now indicates that the region starts at the *idle* state. The final state is also reached from the *idle* state, but it is still triggered by the receipt of a *Turn Off* signal. Having completed processing in the *initializing* state (refer to Figure 10.14 on page 262 to view inside the *initializing* state), a completion event for *initializing* will be generated. If the condition variable *init OK* is true, the system enters the *operating* state. Otherwise, the system enters the *diagnosing* state where an operator will look at the error logs and try to manually initialize the system. Just in case something happens and the test procedure does not complete, the system has a timeout after 60 seconds, which returns the system to the *idle* state.

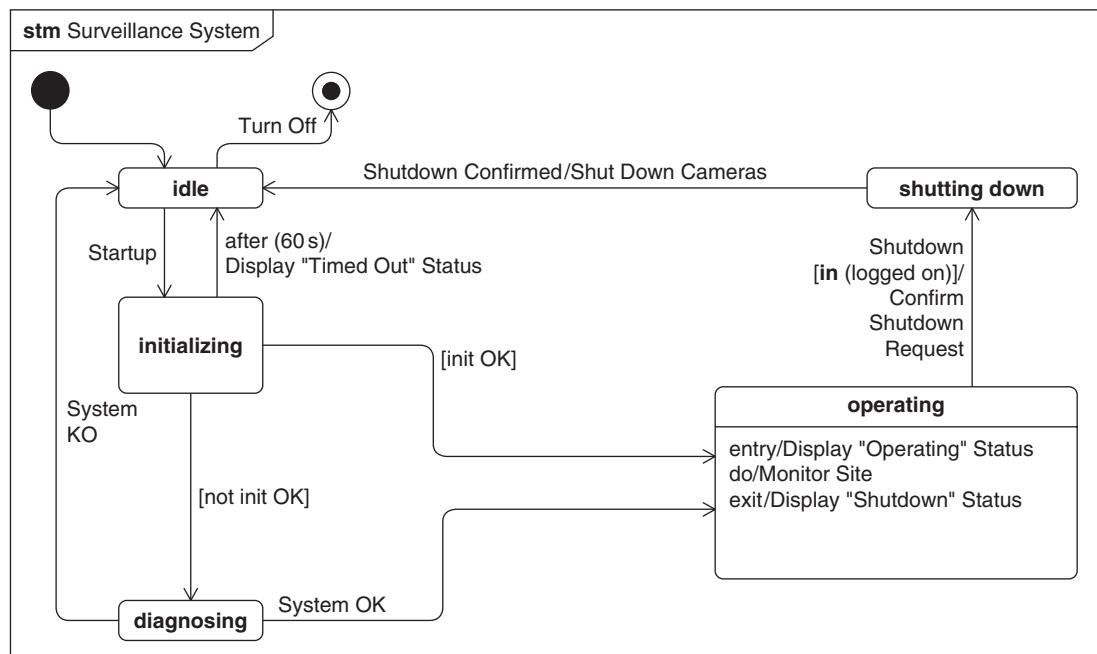


FIGURE 10.3

Transitions between states.

From the *diagnosing* state, the operator indicates success using the signal *System OK*, which allows the system to enter the *operating* state. The signal *System KO* indicates that the system is beyond operator repair and causes a transition back to *idle*. From the *operating* state, a *Shutdown* signal will cause a transition to the *shutting down* state, as long as the operating state is in substate *logged on* (refer to Figure 10.9 for a view inside the *operating* state). As part of shutting down, the system requests a confirmation and will only exit the *shutting down* state when it receives a *Shutdown Confirmed* signal, whereon it executes the *Shut Down Cameras* activity.

Unless the graphical notation for transitions is being used, transition effects, with the exception of opaque behaviors, are specified on separate diagrams appropriate to the type of behavior. Figure 10.4 shows the activity diagram for the *Shut Down Cameras* activity.

When invoked as a transition effect, *Shut Down Cameras* loops over all known cameras and sends each a *Shutdown* signal. Note that the activity does not include an accept event action; this would leave the invoking state machine in an ambiguous (mid-transition) state when waiting for new events to occur.

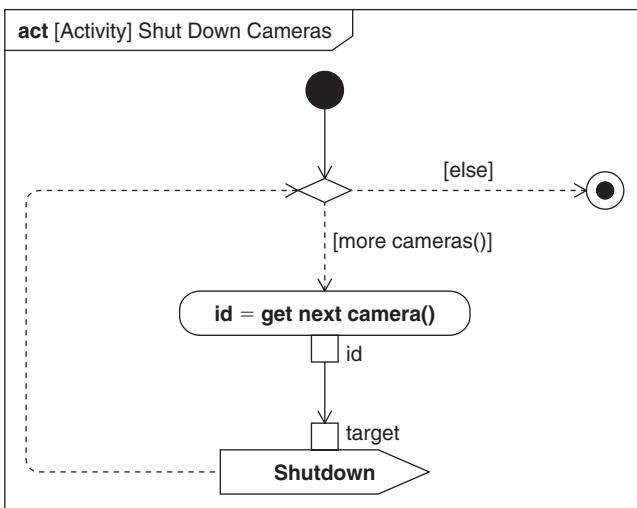


FIGURE 10.4

Defining a transition effect using an activity.

10.4.2 Routing Transitions Using Pseudostates

There are a variety of situations where a simple transition directly between two states is not sufficient to express the required semantics. SysML includes a number of pseudostates to provide these additional semantics. This section introduces junction and choice pseudostates, which support compound transitions between states.

A **junction pseudostate** is used to construct a compound transition path between states. The compound transition contrasts with a simple transition by allowing more than one alternative transition path between states to be specified, although only one path can be taken in response to any single event. Multiple transitions may either converge to or diverge from the junction pseudostate. When there are multiple outgoing transitions from a junction pseudostate, the selected transition will be one of those whose guard evaluates to true at the time the event was served up for processing. If more than one guard does evaluate to true, SysML does not define which one of the valid transitions is chosen for execution. If a particular compound transition path includes more than one junction between two states, all the guards along that path must evaluate to true before the compound transition is taken.

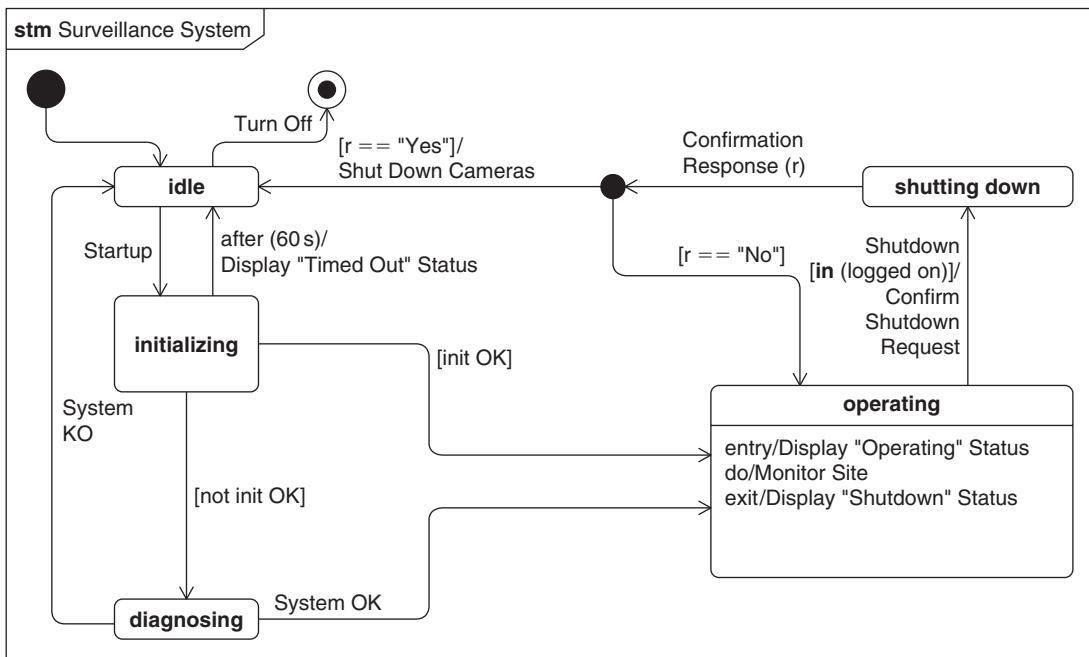
The **choice pseudostate** also has multiple incoming transitions and outgoing transitions and like the junction pseudostate is part of a compound transition between states. The behavior of the choice pseudostate is distinct from that of a junction pseudostate in that the guards on its outgoing transitions are not evaluated until the choice pseudostate has been reached. This allows effects executed on the prior transition to affect the outcome of the choice. When a choice pseudostate is reached in the execution of a state machine, there must always be at least one valid outgoing transition. If not, the state machine is invalid. A technique that is often used to ensure the validity of a choice pseudostate is to use a catch-all guard on at most one outgoing transition. This is specified using the keyword “else.” Whether a compound transition contains junction pseudostates, choice pseudostates, or both, any possible compound transition must contain only one trigger, normally on the first transition in the path.

The various routing pseudostates are represented as follows:

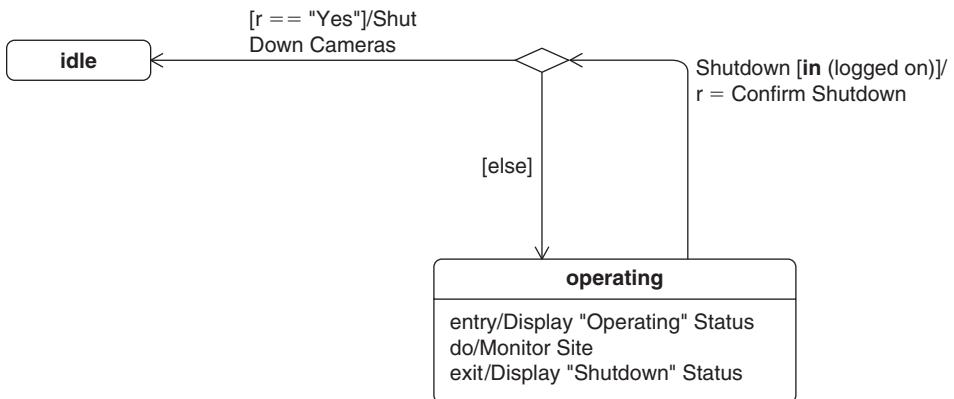
- A junction pseudostate is shown, like an initial pseudostate, as a filled circle.
- A choice pseudostate is shown as a diamond.

Figure 10.5 completes the state machine for the *Surveillance System* shown in Figure 10.3. The handling of shutdown has been improved to describe what happens if the operator does not actually want to shut down the system after all. The argument of the *Confirmation Response* signal, which takes values of “Yes” or “No,” is mapped to attribute *r*. The transition triggered by the *Confirmation Response* signal now ends at a junction, with two outgoing transitions with different guards. If *r* == “Yes,” then the system shutdown proceeds; if *r* == “No,” then the system returns to the operating state.

The transition from shutting down to idle/operating was able to be specified using a junction pseudostate in Figure 10.5 because the value of *r*, needed to determine the complete transition path, was available before the transition was triggered. However, Figure 10.6 shows another approach to system shutdown without a *shutting down* state. Here, the confirmation request is made as an effect of the transition out of the *operating* state, so the value of *r* is not known before the first leg of the compound transition has been taken. In this case a choice pseudostate is needed to allow the value of *r* returned from *Confirm Shutdown* to be used in the guard conditions on its exit transitions. As noted earlier, the modeler

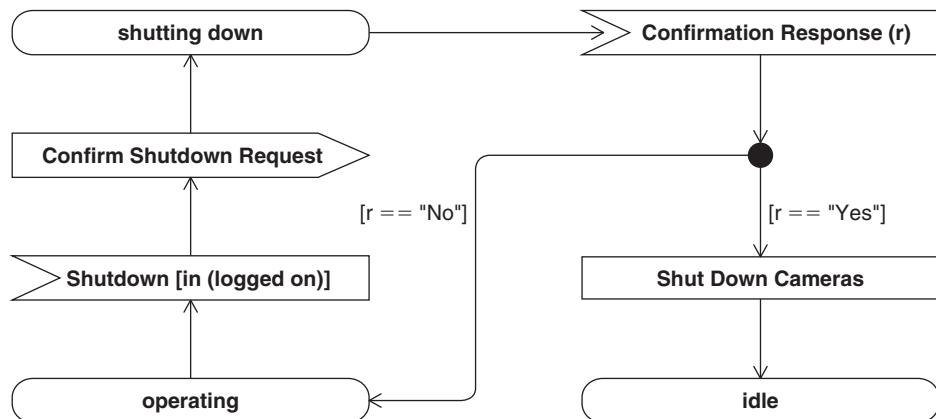
**FIGURE 10.5**

Routing transitions.

**FIGURE 10.6**

Specifying shutdown using a choice pseudostate.

must ensure that there is always at least one valid path from a choice pseudostate, so the guard on the transition has been changed to *[else]* in order to deal with any values other than “Yes.” Then, even if *Confirm Shutdown* unexpectedly returns a value other than “Yes” or “No,” the state machine will still operate.

**FIGURE 10.7**

Transition-oriented notation.

10.4.3 Showing Transitions Graphically

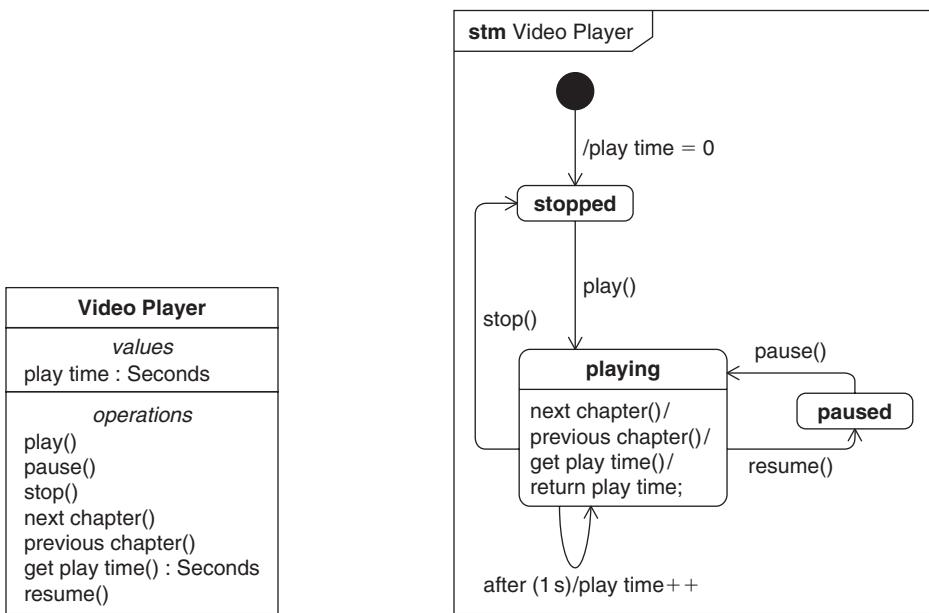
Some modelers prefer to show transitions graphically on state machine diagrams. SysML introduces a set of special symbols that allow a modeler to graphically depict triggers, signal send actions, and other actions; their graphical syntax is as follows:

- A rectangle with a triangular notch removed from one side represents all the transition's triggers, with descriptions of the triggering events and the transition guard inside the symbol.
- A rectangle with a triangle attached to one side represents the send signal action. The signal's name, together with any arguments being sent, are shown within the symbol. There may be many send signal actions in a single transition effect, each with their own symbol. Signals are very important when communicating between state machines, hence the separate treatment of this action.
- Any other action in the transition effect is represented by a rectangle containing text that describes the action to be taken. There may be many actions as part of a transition effect, each with their own symbol.

Figure 10.7 shows the use of transition notation to provide an equivalent definition of the transitions between *operating*, *idle*, and *shutdown*, originally shown on Figure 10.5.

10.5 State Machines and Operation Calls

State machines can respond to operation calls on their parent block via call events. A call event may either be handled in a synchronous fashion—that is,

**FIGURE 10.8**

A state machine driven by call events for operations on its owning block.

the caller is blocked while waiting for a response—or asynchronously, which results in similar behavior to the receipt of a signal. The state machine executes all actions triggered by the call event until it has reached another state, and then returns any outputs created by the actions to the caller. Note that the invocation of an operation or reception may also trigger the invocation of a method (see Section 6.5.5). If so, the method is invoked and completely executed before the transition enabled by the call event is triggered.

One of the components used by the surveillance system's operators is a video player that allows them to review recorded surveillance data. The *Video Player* block, shown in Figure 10.8, provides a set of operations in its interface to control playback. Although many of the operations do not return data, it makes sense for any client of *Video Camera* to wait until a request for these operations has been processed; hence, it makes sense for the interface to be defined as operations. The response of the block to requests from these operations is defined using the state machine shown in Figure 10.8, where call events related to the operations are used as triggers on transitions. Calls to the *play*, *stop*, *pause*, and *resume* operations cause call events that trigger transitions between the various states of *Video Player*. Calls to the operations, *next chapter*, *previous chapter*, and *get play time* cause call events that trigger transitions that are local to state *playing*. To simplify the example, Figure 10.8 does not show many of the transition effects, but it does show how a request on *get play time* gets its return argument.

10.6 State Hierarchies

Just as state machines can have regions, so can states; such states are called **composite** or **hierarchical states**. This capability allows state machines to scale to represent arbitrarily complex state-based behaviors. This section discusses composite states with single and multiple regions, and also the reuse of an existing state machine to describe the behavior of a state.

10.6.1 Composite State with a Single Region

Arguably the most common situation is a composite state that has a single region. A state nested within a region can only be active when the state enclosing the region is active. Thus, the switch position analogy described earlier can apply to nested states by requiring that the switch position corresponding to the enclosing state be enabled in order to enable any of its nested states.

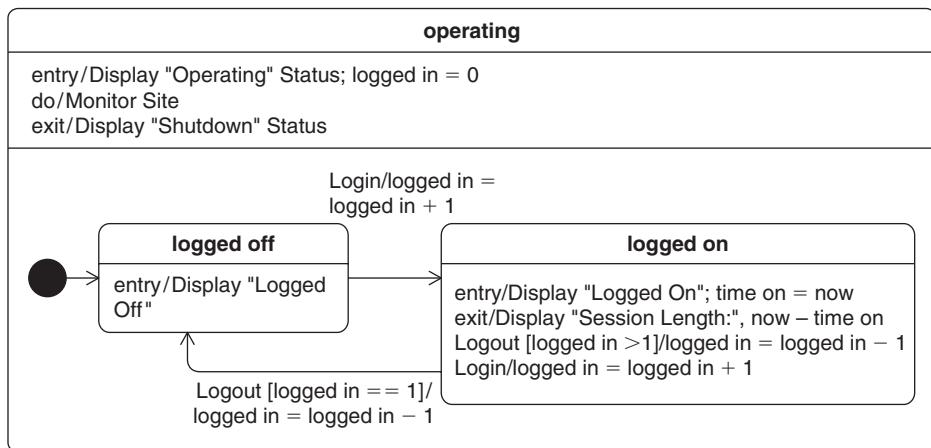
As stated earlier, a region typically will contain an initial pseudostate and a final state, a set of pseudostates and substates, which may themselves be composite states. If the region has a final state, then a completion event is generated when that state is reached.

When an initial pseudostate is missing from a region in a composite state, the initial state of that region is undefined, although extensions to SysML are free to add their own semantics. However, a composite state may be porous; that is, transitions may cross the state boundary, starting or ending on states within its regions (see Figure 10.10 later). In the case of a transition ending on a nested state, the entry behavior of the composite state, if any, is executed after the effect of the transition and before the execution of the entry behavior of the transition's end state. In the opposite case, the exit behavior of the composite state is executed after the exit behavior of the source state and before the transition effect. In the case of more deeply nested state hierarchies, the same rule can be applied recursively to all the composite states whose boundaries have been crossed.

Figure 10.9 shows the decomposition of the state *operating* into its substates. On entry to the *operating* state, two entry behaviors are executed: the entry behavior of *operating*, *Display "Operating" status; logged in = 0*, and then the entry behavior of *logged off*, *Display "Logged Off."* This is because on entry, as indicated by the initial pseudostate, the initial substate of *operating* is *logged off*.

When in state *logged off*, a *Login* signal will cause a transition to the *logged on* state and will increment the value of *logged in*. While in the *logged on* state, repeated *Login* and *Logout* signals will increment and decrement the value of *logged off*, often as internal transitions without a change of state. However, if a *Logout* signal is received when the value of *logged in* is 1, then the signal will trigger a transition back to *logged off*. The entry behavior for *logged on* records the time in the variable *time on*, and its exit behavior uses that to display the session length.

State *operating* does not have a final state, so a completion event is never generated. As can be seen in Figure 10.5, this state is exited when a *Shutdown* signal is presented.

**FIGURE 10.9**

States nested within a sequential state.

The do activity *Monitor Site* executes as long as the state machine for the *Surveillance System* is in the *operating* state, irrespective of what substate of *operating* is currently active.

10.6.2 Composite State with Multiple (Orthogonal) Regions

A composite state may have many regions, which may each contain substates. These regions are orthogonal to each other, so a composite state with more than one region is sometimes called an **orthogonal composite state**. When an orthogonal composite state is active, each region has its own active state that is independent of the others and any incoming event is independently analyzed within each region. A transition that ends on the composite state will trigger transitions from the initial pseudostate of each region, so there must be an initial pseudostate in each region for such a transition to be valid. Similarly, a completion event will occur when all the regions are in their final state.

In addition to transitions that start or end on the composite state, transitions from outside the composite state may start or end on the nested states of its regions. In this case one state in each region must be the start or end of one of a coordinated set of transitions. This coordination is performed by a fork pseudostate in the case of incoming transitions and a join pseudostate for outgoing transitions.

A **fork pseudostate** has a single incoming transition and as many outgoing transitions as there are orthogonal regions in the target state. Unlike junction and choice pseudostates, all outgoing transitions of a fork are part of the compound transition. When an incoming transition is taken to the fork pseudostate, all the outgoing transitions are taken. Because all outgoing transitions of the fork pseudostate have to be taken, they may not have triggers or guards, but may have effects.

The coordination of outgoing transitions from an orthogonal composite state is performed using a **join pseudostate** that has multiple incoming transitions and one outgoing transition. The rules on triggers and guards for join pseudostates are the opposite of those for fork pseudostates. Incoming transitions of the join pseudostate may not have triggers or a guard but may have an effect. The outgoing transition may have triggers, a guard, and an effect. When all the incoming transitions can be taken and the join's outgoing transition is valid, the compound transition can happen. Incoming transitions are taken first and then the outgoing transition. An example of this can be seen in Figure 10.10.

Note that a transition can never cross the boundary between two regions of the same composite state. Such a transition, if triggered, would leave one of the regions with no active state, which is not allowed.

When an event is associated with triggers in multiple orthogonal regions, the event may trigger a transition in each region, assuming the transition is valid based on the other usual criteria. A simple example of this scenario is shown later in Figure 10.11.

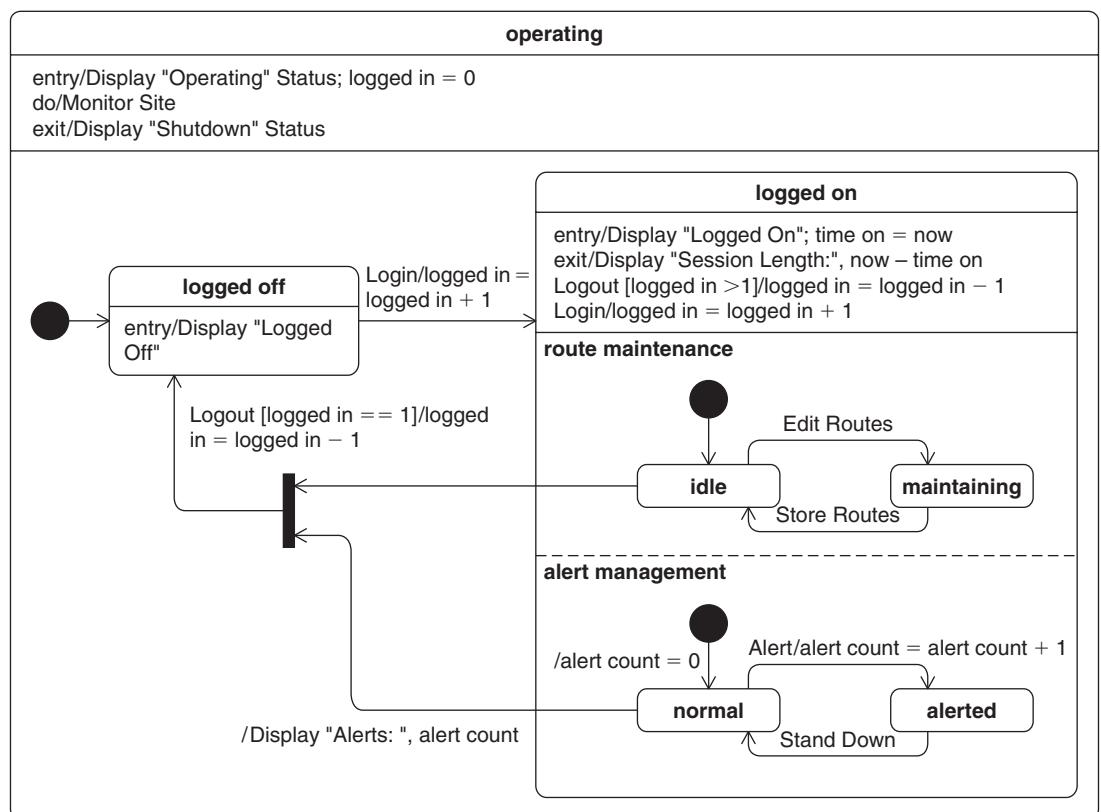


FIGURE 10.10

Entering and leaving a set of concurrent regions.

The presence of multiple regions within a composite state is indicated by multiple compartments within the state symbol, separated by dashed lines. The regions can optionally be named, in which case the name appears at the top of the corresponding compartment. All vertices within such a compartment are part of the same region. When an orthogonal composite state has no other compartments, it is preferable to use an alternative notation for a state, where the name of the state is placed in a tab attached to the outside of the state symbol. An example of this can be seen in Figure 10.11.

Fork and join pseudostates are described by a vertical or horizontal bar, with transition edges either starting or ending on the bar.

Figure 10.10 shows an elaboration of the *operating* state first shown in Figure 10.9. In this elaboration, the *logged on* state has two orthogonal regions. One region, called *alert management*, specifies states and transitions for *normal* and *alerted* modes of operation; the other region, called *route maintenance*, specifies states and transitions for updating the route (i.e., pan-and-tilt angles) for when the automatic surveillance feature of the system is engaged. As before, in state *logged off*, the receipt of a *Login* signal triggers transition to *logged on*. Based on the initial pseudostates in the two regions, the two initial substates of *logged on* are *idle* for region *route maintenance* and *normal* for *alert management*. The receipt of an *Alert* signal triggers the transition from *normal* to *alerted* in *alert management*. Similarly, the receipt of an *Edit Routes* signal triggers the transition from *idle* to *maintaining* in *route management*.

To prevent dereliction of duty, the last operator can only log off if the *logged on* state is in substates *idle* and *normal*. This constraint is specified using a join pseudostate whose outgoing transition is triggered by a *Logout* signal with a guard of *logged in == 1*. The two incoming transitions to the join pseudostate start on *idle* and *normal*, so even if there is a *Logout* signal and the number of logged on operators is one, the outgoing transition from the join pseudostate will be valid only if the two active substates of *logged on* are *idle* and *normal*. Because the transitions from *idle* and *normal* cross the boundary of state *logged in*, its exit behavior is executed before any effects on the transitions. The order of execution triggered by a valid *Logout* signal is thus:

- Exit behavior of *logged in*—Display “Session Length:”, now-time on
- Incoming transition effect to join—Display “Alerts:,” alert count
- Outgoing transition effect from join—“logged in = logged in – 1”
- Entry behavior of *logged off*—Display “Logged Off”

Having elaborated the *operating* state, it is apparent that the transitions *Logout* [*logged in > 1*] and *Login* are rightly internal transitions rather than transitions to self. Transitions to self always exit and reenter the state, which in this case would reset the substates of *route maintenance* and *alert management*; obviously, this is not desirable in the middle of an intruder alert!

10.6.3 Transition Firing Order in Nested State Hierarchies

It is possible that the same event may trigger transitions at several levels in a state hierarchy, and with the exception of concurrent regions, only one of the transitions

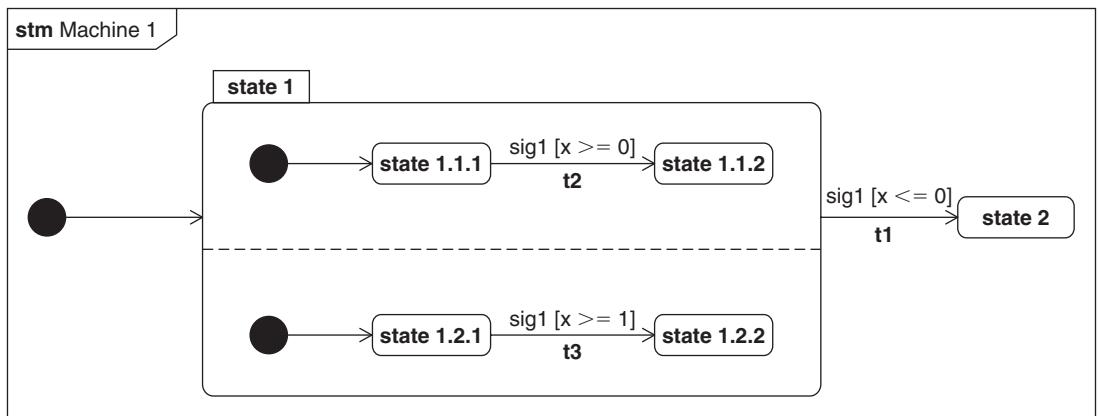
**FIGURE 10.11**

Illustration of transition firing order.

can be taken. Priority is given to the transition whose source state is innermost in the state hierarchy.

Consider the state machine, *Machine 1*, shown in Figure 10.11, in its initial state (i.e., in state 1.1.1 and 1.2.1). The signal *sig1* is associated to the triggers of three transitions, each with guards based on the value of variable *x*. Note that, in this case, the transitions have both a name and a transition expression, whereas a transition edge normally would show one or the other. This has been done to help explain the behavior of the state machine. The following list shows the transitions that will fire based on values of *x* from -1 to 1 :

- x equals -1 —transition *t1* will be triggered because it is the only transition with a valid guard
- x equals 0 —transition *t2* will be triggered because, although transition *t1* also has a valid guard, *state 1.1.1* is the innermost of the two source states
- x equals 1 —both transitions *t2* and *t3* will be triggered because both their guards are valid

The normal rules for execution of exit behaviors apply, so, for example, before the transition from *state 1* to *state 2* can be taken, any exit behavior of the active nested states of *state 1*, as well as the exit behavior of *state 1*, must be executed.

The example in Figure 10.11 is fairly straightforward. Assessing transition priority is more complex when compound transitions and transitions from within orthogonal composite states are used. However, the same rules apply.

10.6.4 Using the History Pseudostate to Return to a Previously Interrupted State

In some design scenarios, it is desirable to handle an exception event by interrupting the current state, responding to the event, and then returning back to the state that the system was in at the time of the interruption. This can be achieved

by a type of pseudostate called a **history pseudostate**. A history pseudostate represents the last active substate of its owning region, and a transition ending on a history pseudostate has the effect of returning the region to that state. An outgoing transition from a history pseudostate designates a default history pseudostate. This is used where the region has no previous history or its last active substate was a final state.

The two kinds of history pseudostate are deep and shallow. A **deep history pseudostate** records the states of all regions in the state hierarchy below and including the region that owns the deep history pseudostate. A **shallow history pseudostate** only records the top-level state of the region that owns it. As a result, the deep history pseudostate will enable a return to a nested state, while a shallow history pseudostate will enable a return to only the top-level state.

A history pseudostate is described using the letter "H" surrounded by a circle. The deep history pseudostate has a small asterisk in the top right corner of the circle.

The *Surveillance System* supports an emergency override mechanism, as shown in Figure 10.12. In a change from Figure 10.10, the reception of an *Override* signal, with a valid password, will always cause a transition from the *operating* state, even if there is an ongoing alert. However, once the emergency is over, a *Resume Operation* signal needs to restore the *operating* state completely to its previous state so that the system can continue with its interrupted activities. To achieve this, the transition triggered by the *Resume Operation* signal ends on a deep history pseudostate, which will restore the complete previous state including substates of *operating*. By comparison, if a shallow history pseudostate was used, and the previous substate of *operating* was *logged on*, then the initial rather than previously active substates of *logged on* would be used.

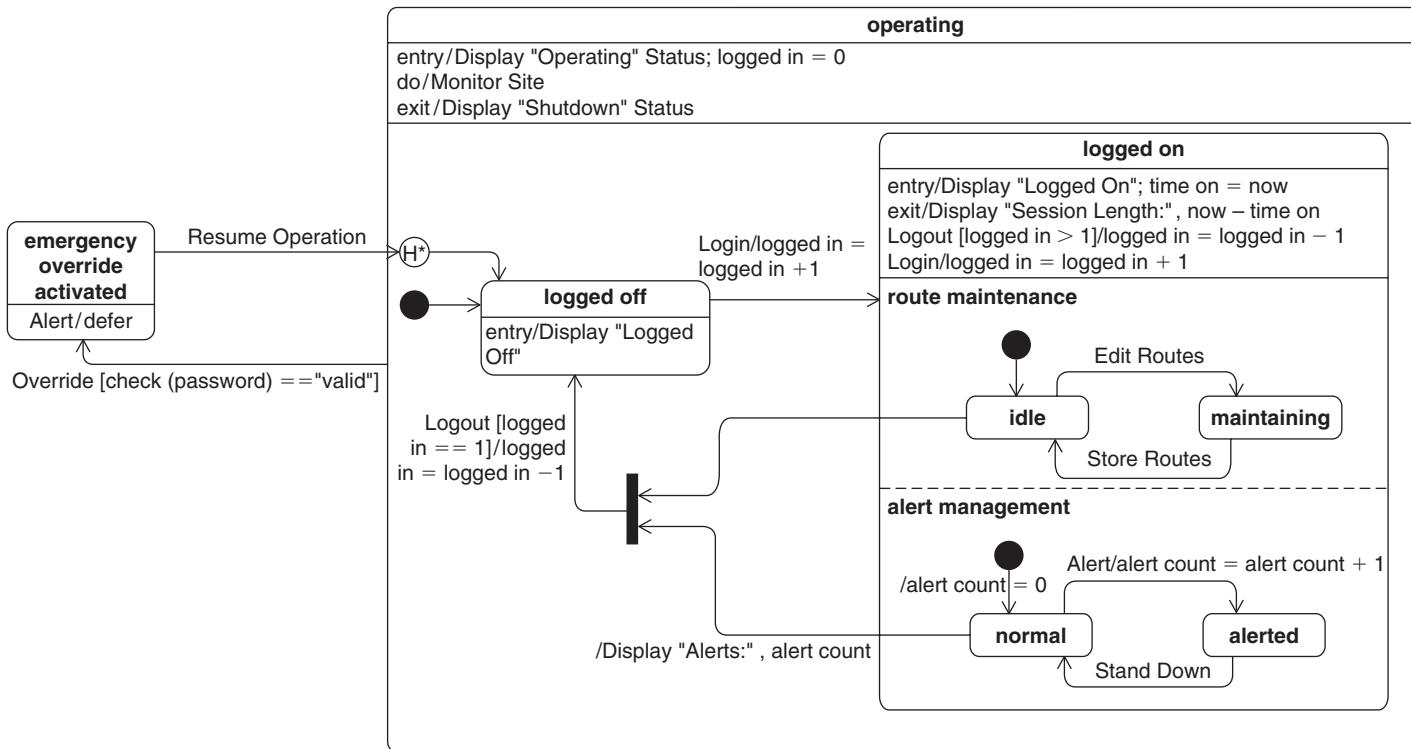
Alert events are deferred in the *emergency override activated* state so that they can be handled, if appropriate, in the resumed *operating* state.

10.6.5 Reusing State Machines

A state machine may be reused to specify the behavior of a kind of state called a **submachine state**. A transition ending on a submachine state will start its referenced state machine, and similarly, completion events trigger transitions whose source is the submachine state when the referenced state machine completes. However, modelers can also benefit from two additional types of pseudostates, called **entry-** and **exit-point pseudostates**, that allow the state machine to define additional entry and exit points that can be accessed from a submachine state.

Entry and Exit Points on State Machines

For a single-region state machine, entry- and exit-point pseudostates are similar to junctions; that is, they are part of a compound transition. Outgoing guards have to be evaluated before the compound transition is triggered, and only one outgoing transition will be taken. On state machines, entry-point pseudostates can only have outgoing transitions and exit-point pseudostates can only have incoming transitions.

**FIGURE 10.12**

Recovering from an interruption using a history pseudostate.

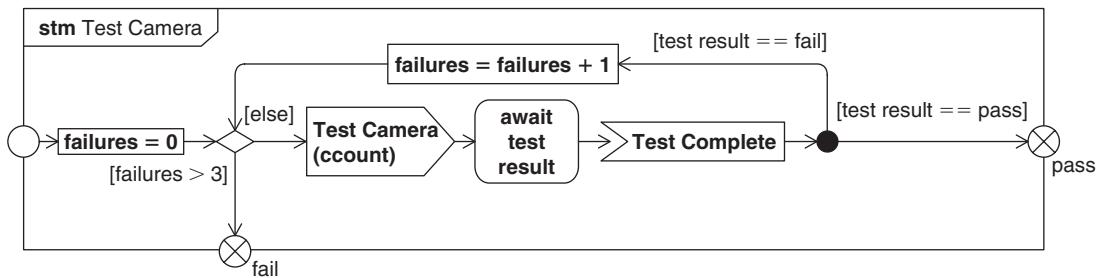


FIGURE 10.13

A state machine with entry and exit points.

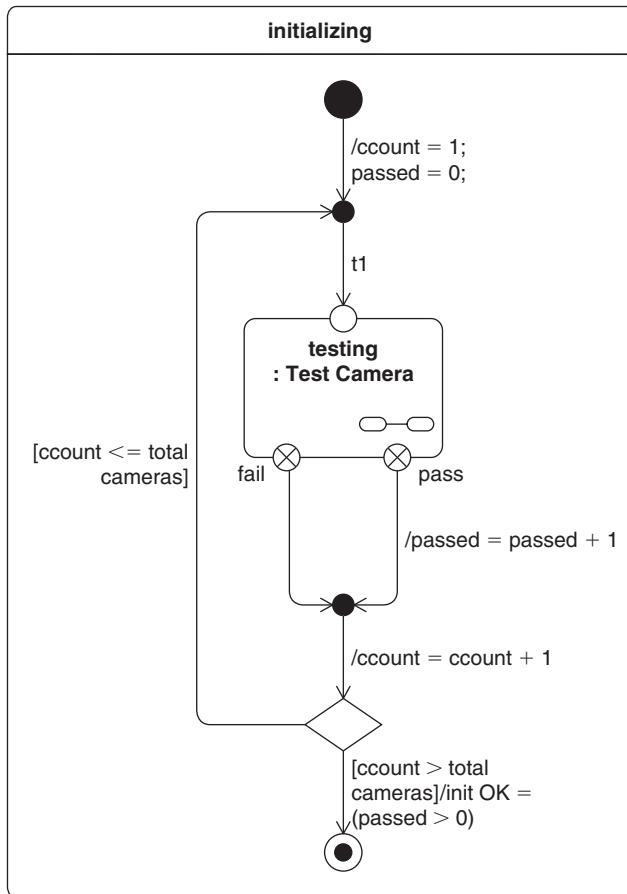
Entry- and exit-point pseudostates are described by small circles that overlap the boundary of a state machine or composite state. An entry-point symbol is hollow, whereas an exit-point symbol has a cross, rotated 45° from the vertical/horizontal axis.

Figure 10.13 shows a state machine for testing cameras, called *Test Camera*, that uses the graphical form for specifying transitions. From the entry-point pseudostate, the first transition simply sets the *failures* variable to 0 and ends on a choice pseudostate. On first entry, the state machine will always take the */else* transition, which will result in the sending of a *Test Camera* signal with the current camera number (*ccount*) as its argument. The state machine then stays in the *await test result* state until a *Test Complete* signal with argument *test result* has been received. The transition triggered by a *Test Complete* signal ends on a junction that either leads to the exit-point pseudostate *pass*, if the test passed, or back to the initial choice pseudostate, if the test failed, incrementing the *failures* variable on the way. If the camera has failed its self-test more than three times, then the transition with guard *[failures > 3]* will be taken to exit-point *fail*.

Submachine States

A submachine state contains a reference to another state machine that is executed as part of the execution of the submachine state's parent. The entry- and exit-point pseudostates of the referenced state machine are represented on the boundary of the submachine state by special vertices called **connection points**. Connection points can be the source or target of transitions connected to states outside the submachine state. A transition whose source or target is a connection point forms part of a compound transition that includes the transition to or from the corresponding entry- and exit-point pseudostate in the referenced state machine. An example of this can be seen in Figure 10.14. In any given use of a state machine by a submachine state, only a subset of its entry- and exit-point pseudostates may need to be externally connected.

A submachine state is represented by a state symbol showing the name of the state, along with the name of the referenced state machine, separated by a colon. A submachine state also includes an icon shown in the bottom right corner depicting a state machine. Connection points may be placed on the boundary of the

**FIGURE 10.14**

Invoking a substate machine.

submachine state symbol. These symbols are identical to the entry- and exit-point pseudostate symbols used in the referenced state machine. Note that only those connection points that need to be attached to transition edges need be shown on the diagram.

Figure 10.14 shows the *initializing* state of the *Surveillance System*. On entry, *ccount* (i.e., a property of the owning block that counts the number of cameras tested) and *passed* (i.e., a property that counts the number of cameras that passed their self-test) are initialized to 1 and 0, respectively. A junction pseudostate, which allows the algorithm to test as many cameras as required, follows. To test each camera, the *testing* state uses the *Test Camera* state machine. The transition leaving the *pass* exit-point pseudostate has an effect that adds one to the *passed* variable; the transition leaving its *fail* exit-point pseudostate does not. Both transitions end in a junction whose outgoing transition increments the count of cameras tested. This transition ends on a choice, with one outgoing transition looping

back to test another camera if $\text{ccount} \leq \text{total cameras}$ and the other reaching the final state of *initializing*. On the transition to the final state, the effect of the transition sets the *init OK* variable to true if at least one camera passed its self-test, and false otherwise.

As stated earlier, entry- and exit-point pseudostates form part of a compound transition, that in the case of submachine states, incorporates transitions (and their triggers, guards, and effects) from both containing and referenced state machines. Looking at both Figures 10.13 and 10.14, it can be seen that the compound transition from the initial pseudostate of state *initializing* will be as follows:

1. Initial pseudostate of the (single) region owned by state *initializing*
2. Transition labeled with effect $\text{ccount} = 1; \text{passed} = 0$
3. Transition named *t1*
4. Transition with effect $\text{failures} = 0$
5. Transition with guard */else* (at least this time)
6. (Graphical) transition with effect send *Test Camera* signal with argument *ccount*
7. State *await test result*

Entry- and Exit-Point Pseudostates on Composite States

Entry- and exit-point pseudostates can be used on the boundaries of composite states as well. Where the composite state has a single region, they behave like junctions. Where the composite state has multiple regions, they behave like forks in the case of entry-point pseudostates and joins in the case of exit-point pseudostates. For entry-point pseudostates, the effects of their outgoing transitions execute after the entry behavior of the composite state. For exit-point pseudostates, their incoming transitions execute before the composite state's exit behavior.

10.7 Contrasting Discrete versus Continuous States

The examples shown so far in this chapter have been based on discrete semantics, and specifically state machines where the triggering event is a specific stimulus (i.e., a signal, an operation call, or the expiration of a timer). SysML state machines can also be used to describe systems where transitions are driven by the values of either discrete or continuous properties. Such transitions are triggered by change events.

A trigger on a transition may be associated with a change event whose change expression states the conditions, typically in terms of the values of properties, which will cause the event to occur and hence trigger the transition. The change expression has a body containing the expression, and an indication of the language used, which allows a wide variety of possible expressions.

Figure 10.15 shows a very simple state machine, called *Lamp Switch*, for controlling a lamp with an unlatched button. It starts in state *off*, which has an entry

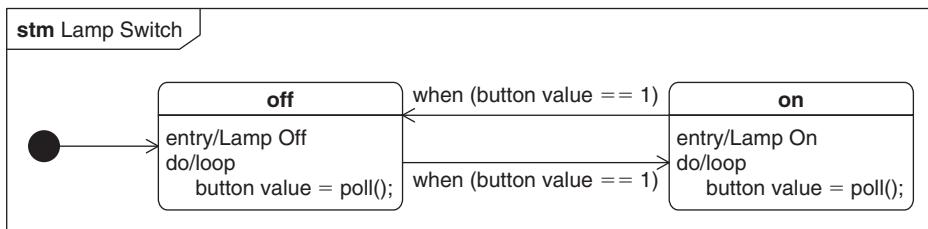


FIGURE 10.15

A discrete state machine driven by change events.

behavior that turns the lamp off and a do activity that repeatedly polls an input line and places the value of the input into the variable *button value*. A change event, *when (button value == 1)*, triggers a transition to state *on*, so as soon as the polled value changes to 1, the *off* state is exited and the do activity is terminated. On entry into the *on* state, the lamp is turned on and the state machine again repeatedly polls the input line. The transition out of state *on* to state *off* is again triggered by the change event *when (button value == 1)*. This type of solution is suitable for describing digital systems that execute continuously monitoring inputs and writing outputs.

The transitions between states in the *Lamp Switch* state machine are triggered by a change to the value of a discrete property, *button value*. This is in contrast to the continuous state representation of a system in terms of continuous state variables (expressed as value properties).

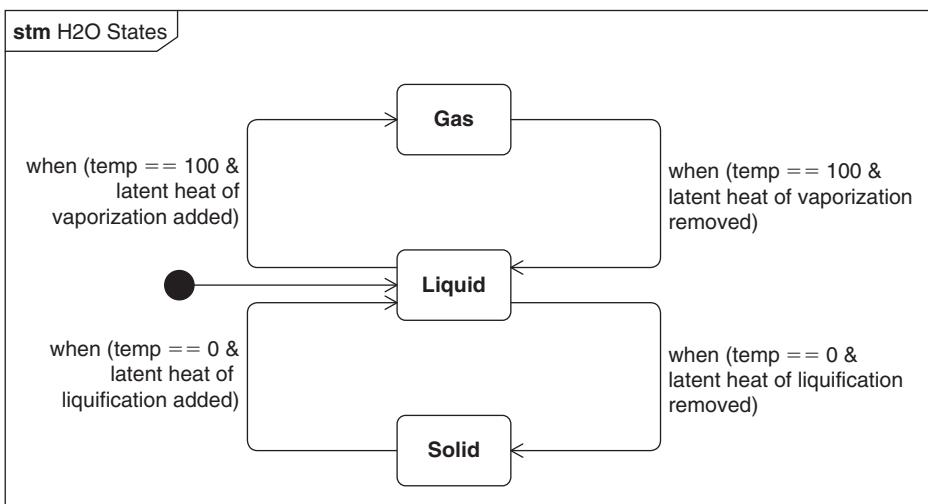
The state machine for H_2O , shown in Figure 10.16, defines the transitions between its *solid*, *liquid*, and *gas* states. These represent discrete states of H_2O , while the values of its properties, such as temperature and pressure, represent continuous state variables. Specific values for the variable *temp*, plus other conditions (e.g., the withdrawal or addition of energy), define the expressions for the change events on the transitions. So implicitly, the values of its state variables determine the discrete state of H_2O via the transitions between them. Similarly, the discrete state of other continuous systems can be defined in terms of values of selected continuous properties of the system.

10.8 Summary

State machines are used to describe behavior of a block in terms of its states and transitions. State machines can be composed hierarchically, like other SysML behavioral constructs, enabling arbitrarily complex representations of state-based behavior.

The significant state machine concepts covered in this chapter include the following.

- A state machine describes a potentially reusable definition of the state-dependent behavior of a block. Each state machine is described using a state machine diagram.

**FIGURE 10.16**

State machine for H₂O.

- Each state machine contains at least one region, which itself can contain a number of substates, pseudostates (called collectively vertices), and transitions between those vertices. During execution of a state machine, each of its regions has a single active state that determines the transitions that are currently viable in that region. A region can have an initial pseudostate and final state that correspond to its beginning and completion, respectively.
- A state is an abstraction of some significant condition in the life of a block, and specifies the effect of entering and leaving that condition, and what the block does while it is in that condition, using behaviors such as activities.
- Transitions describe valid state changes, and under what circumstances those changes will happen. A transition has one or more triggers, a guard, and an effect. A trigger is associated to an event, which may correspond either to the reception of a signal (signal event) or operation call (call event) by the owning block; the expiration of a timer (time event); or the satisfaction of a condition specified in terms of properties of the block and its environment (change event). A transition can also be triggered by a completion event that occurs when the currently active state has completed.
- A guard expresses any additional constraints that need to be satisfied if the transition is to be triggered. If a valid event occurs, the guard is evaluated, and if true, the transition is triggered; otherwise, the event is consumed. A transition can include a transition effect that is described by a behavior such as an activity. If the transition is triggered, the transition effect is executed.
- A state may specify that certain events can be deferred, in which case they are only consumed if they trigger a transition. Deferred events are consumed on transition to a state that does not further defer them.

- There are a number of circumstances where simple transitions between states are not sufficient to specify the required behavior. Junction and choice pseudostates allow several transitions to be combined into a compound transition. Although the compound transition can include only one transition with triggers, it can have multiple transitions with guards and effects. Junction and choice pseudostates can have multiple incoming transitions and outgoing transitions. They are used to construct complex transitions that have more than one transition path, each potentially with its own guard and effect. History pseudostates allow a state to be interrupted and then subsequently resume its previously active state or states.
- States may be composite with nested states in one or more regions. Just like state machines, during execution an active state will have one active substate per region. Composite states are porous; that is, transitions can cross their boundaries. Special pseudostates called fork and join pseudostates allow transitions to and from states in multiple regions at once. A given event may trigger transitions in multiple active regions.
- State machines may be reused via submachine states. Interactions with the reused state machine take place via transitions to and from the boundary of the corresponding submachine state, either directly or through entry- and exit-point pseudostates.
- Change events are driven by the values of variables of the state machine or properties of its owning block. In addition to discrete systems, change events can be used to describe the state of continuous systems, where transitions between the system's discrete states are triggered by changes in the values of other continuous properties.

10.9 Questions

1. What is the diagram kind for a state machine diagram?
2. Which types of model element may a state machine region contain?
3. What is the difference between a state and a pseudostate?
4. A state machine has two states, “S1” and “S2”; how do you show that the initial state for this machine is “S1”?
5. What is the difference between a final state and a terminate pseudostate?
6. A state has three behaviors associated with it; what are they called and when are they invoked?
7. What are the three components of a transition?
8. Under what circumstances does a completion event get generated for a state with a single region?
9. What is the difference in behavior between an internal transition and an external transition with the same source and target state?
10. What would the transition string for a transition look like if triggered by a signal event for signal “S1,” with guard “ $a > 1$ ” and an effect “ $a = a + 1$ ”?

11. Draw the same transition using the graphical notation for transitions.
12. Where and how is a deferred event represented?
13. What is the difference between a junction and a choice pseudostate?
14. If a state has several orthogonal regions, how are they displayed?
15. What is the difference between a shallow and deep history pseudostate?
16. How can a state machine be reused within another state machine?
17. How are entry- and exit-point pseudostates represented on a state machine?
18. Under what circumstances will a given change event occur?

Discussion Topic

Discuss the relative benefits of using orthogonal regions in a single state machine, or creating a composition hierarchy of blocks, each with their own state machine.

This page intentionally left blank

Modeling Functionality with Use Cases

11

This chapter describes how to model the high-level functionality of a system with use cases.

11.1 Overview

Use cases describe the functionality of a system in terms of how its users use that system to achieve their goals. The users and other interested participants of a system are described by actors, which may represent external systems or humans who use the system. Use cases have a textual and graphical description that may be further elaborated with detailed descriptions of their behavior, using activities, interactions, or state machines. The relationships between the system under consideration, its actors, and use cases are described on a use case diagram that shares many characteristics with a block definition diagram.

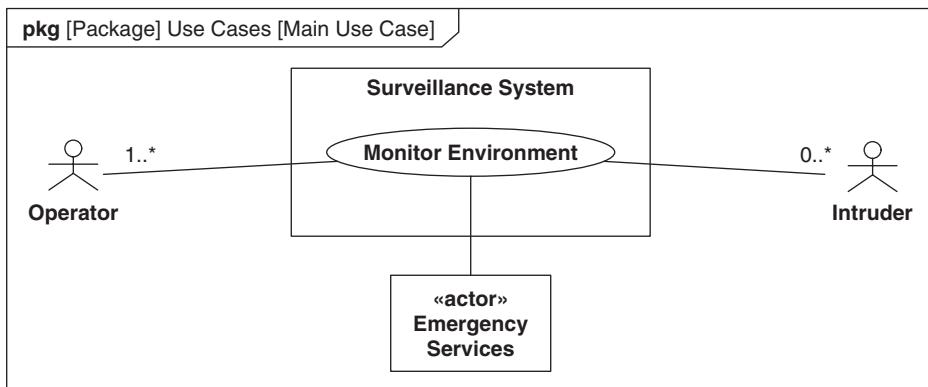
Different methodologies apply use cases in different ways [34]. For example, some methods require a use case description for each use case captured in text, which may include pre- and postconditions, and primary, alternative, and/or exceptional flows.

Use cases have been viewed as a mechanism to capture system requirements in terms of the uses of the system. SysML requirements can be used to more explicitly capture text requirements and establish relationships with use cases and other model elements (refer to Chapter 12 for a discussion on requirements). The steps in a use case description can also be captured as SysML requirements.

11.2 Use Case Diagram

On a **use case diagram**, the frame represents a package or block, and the content of the diagram describes a set of actors and use cases and the relationships between them. The diagram header for a use case diagram has the following form:

uc [model element type] model element name [diagram name]

**FIGURE 11.1**

Example use case diagram.

The diagram kind for a use case diagram is **uc**. The *model element type* for a use case diagram may be either a package or a block. *Model element name* is the name of the model element that contains the use cases and actors in the diagram, and the *diagram name*, as usual, is user specified and may be used to describe the purpose of the diagram.

Figure 11.1 shows an example of a use case diagram containing the key diagram elements, a system, a use case, and some actors. The diagram shows the main use case for the *Surveillance System* and the participants in that use case. The notation for use case diagrams is shown in the Appendix, Table A.21.

11.3 Using Actors to Represent the Users of a System

An **actor** is used to represent the role of a human, an organization, or any external system that participates in the use of some system being investigated. Actors may interact directly with the system or indirectly through other actors.

It should be noted that “actor” is a relative term because an actor who is external to one system may be internal to another. For example, assume individuals in an organization request services from an internal help desk department that provides IT support for the organization. The help desk is considered the system and the members of the organization who are requesting service are considered the actors. However, these same individuals may in turn be providing services to an external customer. In that context, the individuals who were previously considered actors relative to the help desk are considered part of the system relative to the “external” customer.

Actors can be classified using the standard generalization relationship. Actor classification has a similar meaning to the classification of other classifiable model elements. For example, a specialized actor participates in all the use cases that the more general actor participates in.

An actor is shown either as a stick figure with the actor’s name underneath, or as a rectangle containing the actor’s name below the keyword «actor». The choice

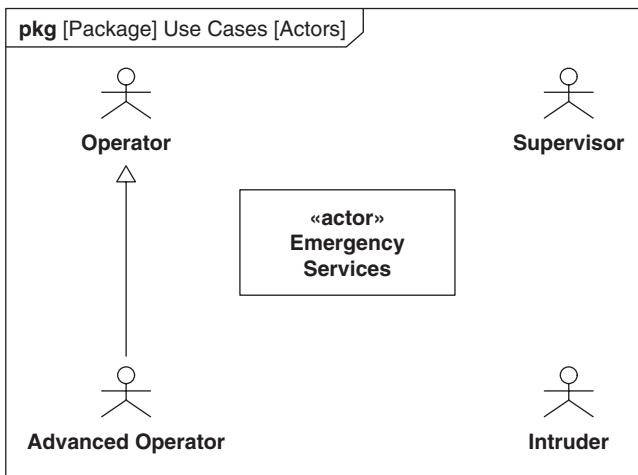


FIGURE 11.2

Representing actors and their interrelationships on a use case diagram.

of symbol is dependent on the tool and methodology being used. Actor classification is represented using the standard SysML generalization symbol—a line with a hollow triangle at the general end.

The *Use Cases* package for the *Surveillance System* contains descriptions of the system's users. Five actors are shown in Figure 11.2. The actors include an *Operator* who operates the system and a *Supervisor* who manages the system. There is also an *Advanced Operator* whose role is a specialized version of the *Operator* because that role has additional specialized skills. Note that an *Intruder* is also modeled as an actor. Although strictly speaking not a user, an intruder does interact with the system and is an important part of the external environment to consider. Also of interest are the *Emergency Services* to whom incidents may need to be reported. This actor is not modeled using an actor stick-figure symbol because it is an organization composed of people, systems, and other equipment.

11.3.1 Further Descriptions of Actors

Although not defined in SysML, there are many methodologies that suggest additional descriptive properties that can apply to actors as users of a system. These may include the following:

- The organization that the actor is a part of (e.g., procurement)
- Physical location
- Skill level required to use the system
- Clearance level required to access the system

11.4 Using Use Cases to Describe System Functionality

A **use case** describes the functionality that some system must provide in order to achieve some user goals. Typically, the use case description identifies the goal

or goals of the use case, a main pattern of use, and a number of variant uses. The system that provides functionality in support of use cases is called the **system under consideration** and often represents a system that is being developed. The system under consideration is sometimes referred to as the **subject** and is represented by a block. We will use the term system or subject interchangeably in reference to the system under consideration.

A use case typically covers many **scenarios** that are different paths the actors can take through the use case under different circumstances.

Actors are related to use cases by associations, with some restrictions. The association ends can have multiplicities, where the multiplicity at the actor end describes the number of actors involved in each use case. The multiplicity at the use case end describes the number of instances of the use case in which the actor or actors can be involved at any one time. Composite associations in either direction are not permitted; actors and use cases are always regarded as peers.

Neither actors nor use cases may own properties, so role names on associations do not represent reference properties as they might do on block definition diagrams. The role name on an actor end can be used, literally, to describe the role an actor plays in the associated use case whenever it is not obvious from the actor's name. The role name on the use case end can be used to describe how use case functionality is relevant to the associated actor.

A use case is shown as an oval with the use case name inside it. Associations between actors and use cases are shown using standard association notation. The default multiplicity of the association ends, if not shown, is “0..1.” Associations cannot have arrows in use case diagrams because neither actors nor use cases may own properties. The subject of a set of use cases can be shown as a rectangle enclosing the use cases, with the subject's name centered at the top.

Figure 11.3 shows the central use case of the *Surveillance System*, called *Monitor Environment*. The two main actors associated with *Monitor Environment* are the system's *Operator* and the *Intruder*. The multiplicities on

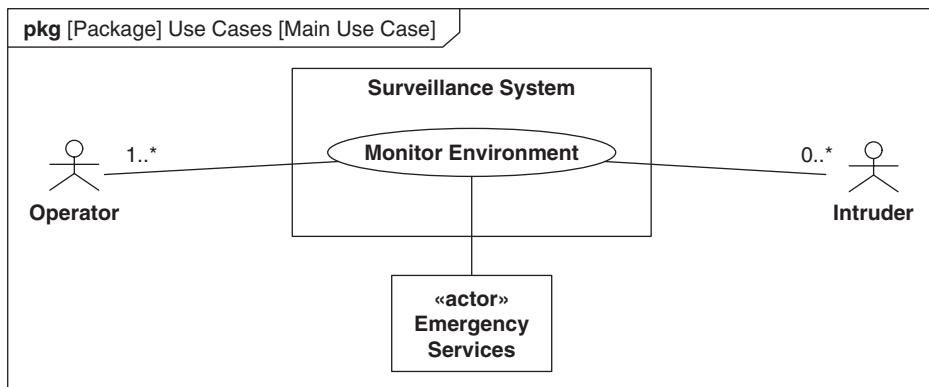


FIGURE 11.3

A use case and the actors that participate in it.

the associations indicate that there must be at least one *Operator* and potentially many *Intruders*. The *Emergency Services* are also associated with the *Monitor Environment* use case, although they may not be active participants unless an *Intruder* is detected and reported.

11.4.1 Use Case Relationships

Use cases can be related to one another by specialization, inclusion, and extension.

Inclusion

The **inclusion** relationship allows one use case, referred to as the **base use case**, to include the functionality of another use case, called the **included use case**, as part of its functionality when performed. The included use case is always performed when the base use case is performed. A behavior that realizes the base use case often references the behavior of the included use case.

It is implicit in the definition of inclusion that any participants of a base use case may participate in an included use case, so an actor associated with a base use case need not be explicitly associated to any included use case. For example, as shown in Figure 11.4, the *Operator* implicitly takes part in *Initialize System* and *Shutdown System* through their association with *Monitor Environment*.

Included use cases are not intended to represent a functional decomposition of the base use case, but rather are intended to describe common functionality that may be included by other use cases. In a functional decomposition, the lower-level functions represent a complete decomposition of the higher-level function and contain the actual functionality. By contrast, a base use case often describes a significant proportion of the overall functionality required. For example, in the case of *Monitor Environment* in Figure 11.4, the key monitoring function is described by the base use case, and additional functionality is described by the included use cases.

Extension

A use case can extend a base use case using the **extension** relationship. The **extending use case** is a fragment of functionality that is not considered part of the normal base use case functionality. It often describes some exceptional behavior in the interaction, such as error handling, between subject and actors that does not contribute directly to the goal of the base use case.

To support extensions, a (base) use case defines a set of **extension points** that represent places where it can be extended. An extension point can be referenced as part of the use case description. For example, if the use case had a textual description of a sequence of steps, the extension point could be used to indicate at which step in the sequence an extending use case would be valid. An extension has to reference an extension point to indicate where in the base use case it can occur. The conditions under which an extension is valid can be further described by a constraint that is evaluated when the extension point is reached to determine whether the extending use case occurs on this occasion. The presence

of an extension point does not imply that there will be an extension related to it, and the base use case is unaware of whether there is an extension.

Unlike an included use case, the base use case does not depend on an extending use case. However, an extending use case may be dependent on what is happening in its base use case; for example, it is likely to assume that some exceptional circumstance in the base use case has arisen. There is no implication that an actor associated with the base use case participates in the extending use case, and the extended use case in fact may have entirely different participants, as demonstrated by the use case *Handle Camera Fault* in Figure 11.4.

Classification

Use cases can be classified using the standard SysML generalization relationship. The meaning of classification is similar to that for other classifiable model elements. One implication, for example, is that the scenarios for the general use case are also scenarios of the specialized use case. It also means that the actors associated with a general use case do not participate in any scenarios solely described by a specialized use case. Classification of use cases is shown using the standard SysML generalization symbol.

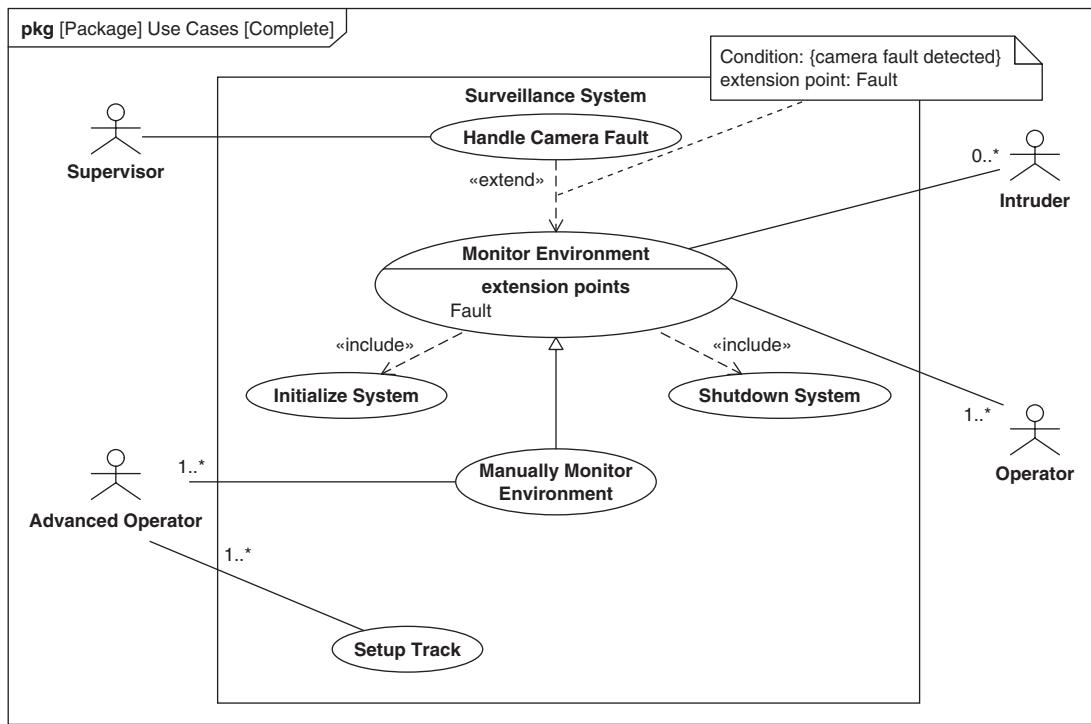
Inclusion and extension are shown using dashed lines with an open arrow at the included and extended ends, respectively. An inclusion line has the keyword «include» and an extension line has the keyword «extend». The direction of the arrows should be read as tail end includes or extends arrow end. Thus, a base use case includes an included use case, and an extending use case extends a base use case.

A use case may have an additional compartment under its name compartment that lists all its extension points. The extension line can have a call-out that names its extension point and shows the condition under which the extending use case occurs.

Figure 11.4 shows a use case diagram containing the complete set of use cases for the *Surveillance System*. As part of *Monitor Environment*, normal *Operators* are only allowed to oversee the automatic tracking of suspicious movements—that is, where the system controls the cameras. This allows the company to employ untrained people and avoid issues with health and safety legislation. *Advanced Operators* can participate in the *Manually Monitor Environment* use case, where they control the cameras manually using a joystick. *Advanced Operators* also have the option to set up surveillance tracks for the cameras to follow.

The complete specification for *Monitor Environment* also includes system initialization and shutdown as indicated by the include relationships between *Monitor Environment* and *Initialize System* and *Shutdown System*.

The *Fault* extension point represents a place in the *Monitor Environment* use case where camera fault might be handled. The *Handle Camera Fault* use case extends *Monitor Environment* at the *Fault* extension point. It is an exceptional task that will only be triggered when camera faults are detected,

**FIGURE 11.4**

A set of use cases for the *Surveillance System*.

as indicated by its associated condition, and may only be performed by the *Supervisor*.

11.4.2 Use Case Descriptions

A text-based **use case description** should be used to provide additional information to support the use case definition. This description contributes significantly to the use case's value. The description text can be captured in the model as a single or multiple comments. It is also possible to treat each step in a use case description as a SysML requirement. A typical use case description may include the following:

- **Preconditions**—the conditions that must hold for the use case to begin.
- **Postconditions**—the conditions that must hold once the use case has completed.
- **Primary flow**—the most frequent scenario or scenarios of the use case.
- **Alternate and/or exception flows**—the scenarios that are less frequent or off nominal. The exception flows may reference extension points and generally represent flows that are not directly in support of the goals of the primary flow.

- *Other information* may augment the basic use case description to further elaborate the interaction between the actors and the subject.

Here is an extract from the use case description for *Monitor Environment*:

Precondition

The *Surveillance System* is powered down.

Primary Flow

The *Operator* or *Operators* will use the *Surveillance System* to monitor the environment of the facility under surveillance. An *Operator* will initialize the system (see *Initialize System*) before operation and shut the system down (see *Shutdown System*). During normal operation, the system's cameras will automatically follow preset routes that have been set to optimize the likelihood of detection.

If an *Intruder* is detected, an alarm will be raised both internally and with a central monitoring station, whose responsibility it is to summon any required assistance. If fitted, an intelligent intruder tracking system, which will override the standard camera search paths, will be engaged at this point to track the suspected intruder. If not fitted then it is expected that *Operators* will keep visual track of the suspected intruder and pass this knowledge onto the *Emergency Services* if and when they arrive.

Alternate Flow

Immediately after system initialization but before normal operation begins, it is possible that a fault will arise in which case it can be handled (c.f. *Fault* extension point), but faults will not be handled thereafter.

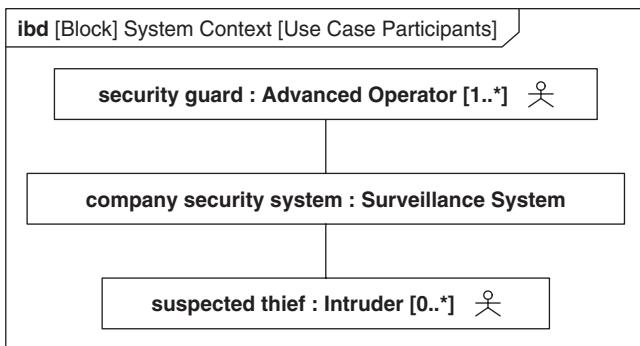
Postcondition

The *Surveillance System* is powered down.

11.5 Elaborating Use Cases with Behaviors

The textual definition for a use case, together with the use case models described previously, can describe the functionality of a system. However, if desired, a more detailed definition of the use case may be modeled with interactions, activities, or state machines, described in Chapters 8 through 10. Typically these additional definitions are added after the use case definition has been reviewed and agreed on and may represent the first step toward design. The choice of behavioral formalism is often a personal or project preference, but in general:

- Interactions are useful where a scenario is largely message-based.
- Activities are useful where the scenario includes considerable control logic, flow of inputs and outputs, and/or algorithms that transform data.
- State machines are useful when the interaction between the actors and the subject is asynchronous and not easily represented by an ordered sequence of events.

**FIGURE 11.5**

Context for use case scenarios.

11.5.1 Context Diagrams

When using interactions or activities, the lifelines and/or partitions represent participants in the interaction. It is useful to create an internal block diagram where the enclosing frame corresponds to the **system context**, and the subject and participating actors correspond to parts in the internal block diagram. To support this technique, actors can appear on a block definition diagram, and a part on an internal block diagram can be typed by the actor. Alternatively, the actors can be allocated to blocks using the allocation relationship described in Chapter 13, and then the parts representing actors can be typed by the block.

Figure 11.5 shows an internal block diagram that describes the internal structure of the block *System Context*, which represents the context for the *Surveillance System* and its associated use cases. The system under consideration, *Surveillance System*, is represented as part of the *System Context*, called *company security system*. Two of the actors, *Advanced Operator* and *Intruder*, who participate in the use cases are also represented as parts *security guard* and *suspected thief*, respectively.

11.5.2 Sequence Diagrams

A use case, in addition to being described in a use case description, can be elaborated by one or more interactions described by sequence diagrams. The different interactions may correspond to the (base) use case, any included use cases, and any extending use cases. The block that owns the interactions must have parts that correspond to the subject and participants, which can then be represented by lifelines in the interactions.

As stated earlier, an included use case must always occur as part of its base use case. As a result, the use of an interaction describing an included scenario will typically be a mandatory part of the interaction representing a base scenario. This is typically indicated by the use of mandatory operators, such as seq, strict, or loop.

Strictly speaking, an interaction representing a base use case should be specified without reference to extending use cases, simply noting the extension points.

However, a popular approach is to reference extending use cases as optional constructs in the interaction representing the base scenario. In this approach, an interaction corresponding to an extending use case is typically contained in an operand of a conditional operator, such as break, opt, or alt. The operand should be guarded using the constraint on the extension, if one is specified.

The block *System Context*, whose internal block diagram was shown in Figure 11.5, owns a number of interactions. The interaction for the primary scenario of the *Manually Monitor Environment* use case, *Handling Alert*, is shown in Figure 11.6. In Figure 11.4, the *Manually Monitor Environment* use case included the *Initialize System* use case and the *Shutdown System* use case. The *Handling Alert*

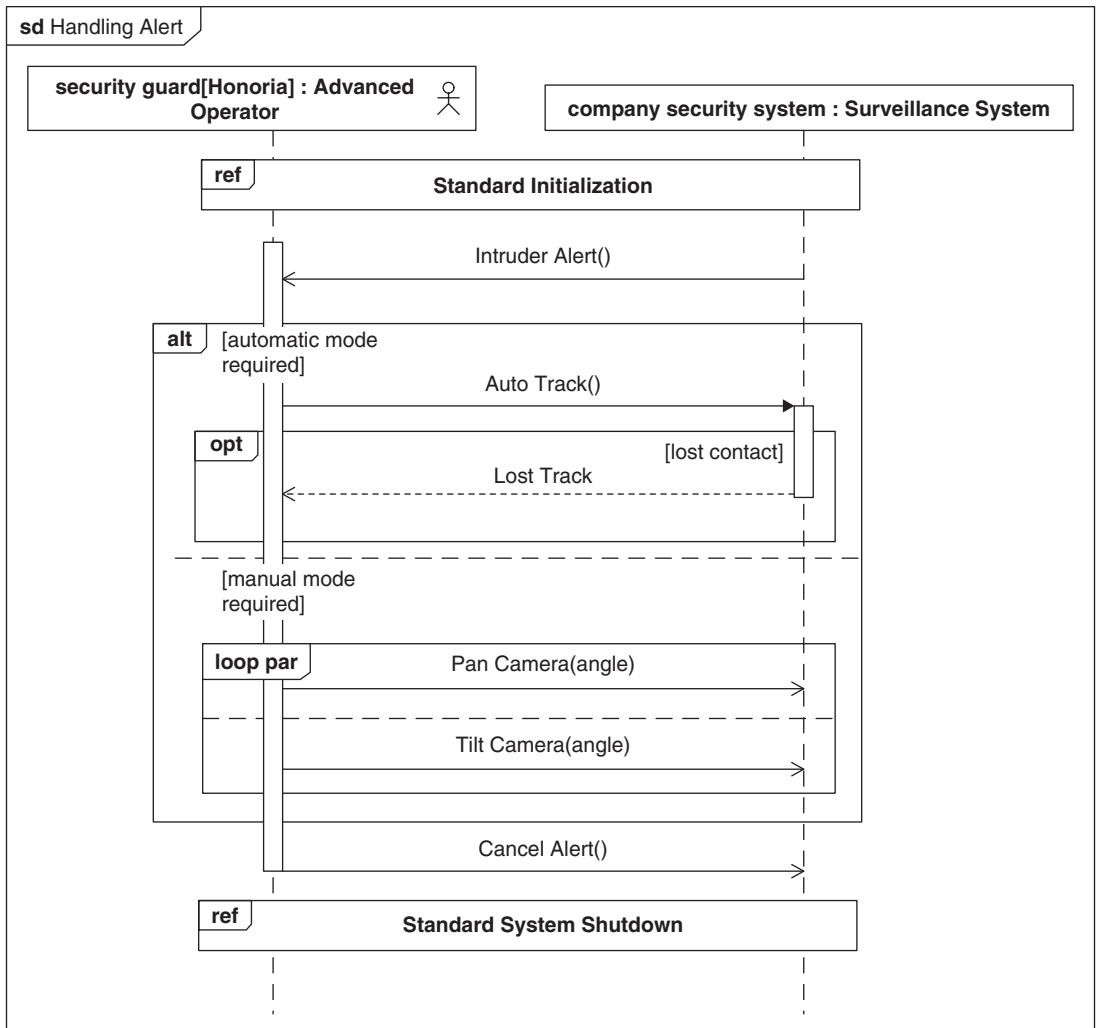


FIGURE 11.6

Scenario for a use case represented by a sequence diagram.

interaction includes corresponding uses of the interaction *Standard Initialization* that is a scenario for the *Initialize System* use case, and the interaction *Standard Shutdown* that is a scenario for the *Shutdown System* use case.

In between these two interactions, the scenario describes how the security guard, *Honoria*, deals with an intruder alert. Because she is an *Advanced Operator*, she can manually control the cameras if she wishes, or she can elect to allow the system to automatically track the suspected intruder.

11.5.3 Activity Diagrams

As mentioned previously, a use case scenario can also be represented by an activity diagram, where the participants are represented as activity partitions. As with interactions, an activity can elaborate a base use case, included use cases, and extending use cases.

Figure 11.7 shows an alternate description of how manual tracking of suspected intruders is handled. Two activity partitions, representing the *security*

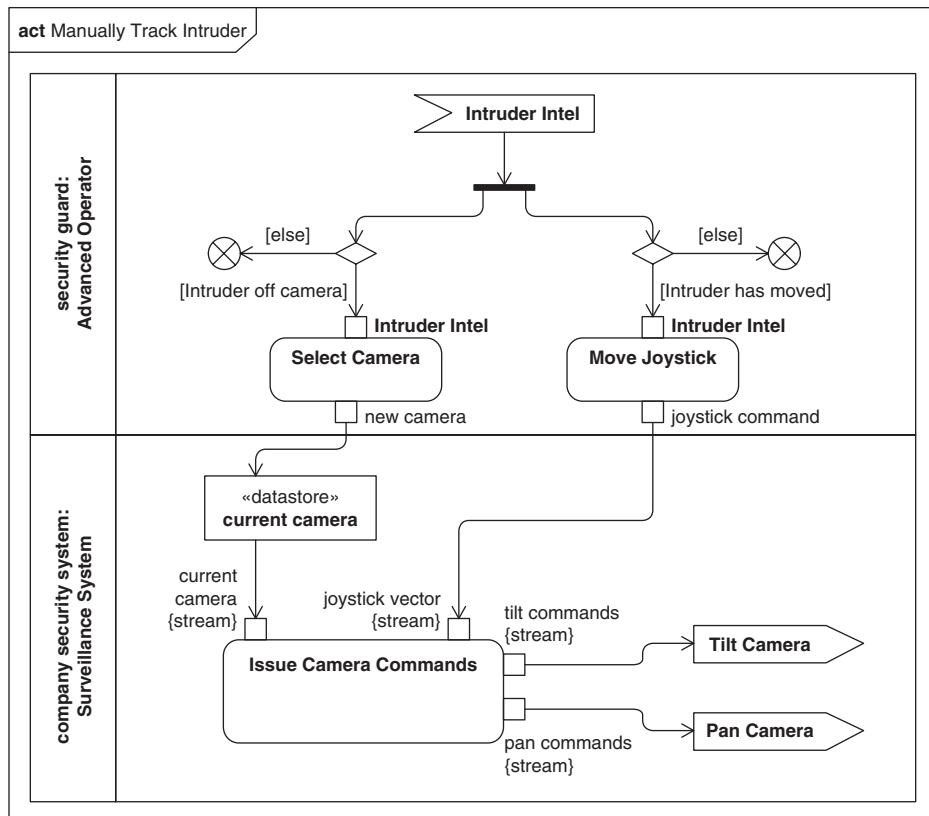


FIGURE 11.7

Using an activity to describe a scenario.

guard and the *company security system*, are used to indicate which use case participant takes responsibility for which actions.

New intruder intelligence (from what source we are not told) is analyzed. The control flow initiated by the reception of the intelligence is forked to address two concerns. If the intruder has moved, then a *Move Joystick* action is performed to follow the intruder. If the intruder appears to have moved out of range of the current camera, then a *Select Camera* action is performed to select a more appropriate camera. In both cases, a flow final node is used to handle situations where no action is required. Meanwhile, this stream of inputs is turned into *Pan Camera* and *Tilt Camera* messages to the appropriate camera by the *Issue Camera Commands* action.

11.5.4 State Machine Diagrams

State machines can also be used to represent scenarios, although some methods encourage the use of a single state machine to represent all possible scenarios of the use case, including exception cases. Note that when using a state machine, there are no constructs, such as interaction lifelines or activity partitions, to explicitly identify the parties responsible for taking actions. However, a state machine may have a number of regions that can informally be related to the participants involved in the use case.

Figure 11.8 shows part of a state machine describing the *Manually Monitor Environment* use case. It shows three states, *operator idle*, *intruder present*, and

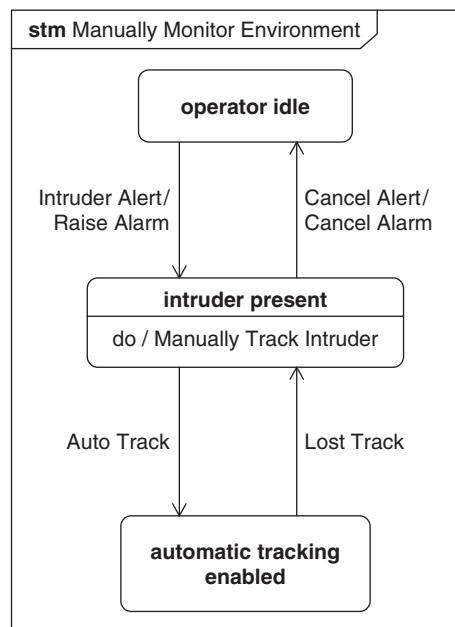


FIGURE 11.8

Using a state machine to describe the *Manually Monitor Environment* use case.

automatic tracking enabled. When in the *operator idle* state, an *Intruder Alert* event causes the *Raise Alarm* message to be sent, and a transition taken to the *intruder present* state. Once in the *intruder present* state, the intruder can be manually tracked, but an *Auto Track* event will trigger a transition to *automatic tracking enabled* and prohibit manual tracking until a *Lost Track* event happens.

This description shares many of the signals with Figure 11.6, but it focuses on states rather than messages.

11.6 Summary

Use cases are used to capture the functionality of a system needed to achieve user goals. The use case is often used as a means to describe the required functionality for a system and can augment SysML requirements to further refine the definition of text-based functional requirements. The way in which use cases are used is highly methodology dependent. The following are the key use case concepts introduced in this chapter.

- The system under consideration (also known as the subject) provides the functionality required by actors, expressed by use cases.
- Use cases describe a particular use of a system to achieve a desired user goal. Use case relationships for inclusion, extension, and specialization are useful for factoring out common functionality into use cases that can be reused by other use cases. The included use case is always performed as part of the base use case. A use case that extends the base use case is usually performed by exception, and generally is not in direct support of the goals of the base use case.
- Actors describe a role played by an entity external to the system and may represent humans, organizations, or external systems. Generalization relationships may be used to represent the relationships between different categories of users. Associations relate actors to the use cases in which they participate.
- The functionality described by a use case is often elaborated in more detail using interactions, activities, and state machines. Although these behaviors' different formalisms do make them more suitable in some situations than others, the choice of which to use is often based on personal or project preference.

11.7 Questions

1. What is the diagram kind for a use case diagram?
2. Which types of model elements can a use case diagram represent?
3. What does an actor represent?
4. How are actors represented on a use case diagram?
5. If one actor specializes another, what does that imply?
6. What does a use case represent?

7. What is another term for the system under consideration?
8. How does a scenario differ from a use case?
9. How is an inclusion relationship represented?
10. Apart from a base and extending use case which two other pieces of information might an extension relationship have?
11. If a use case specializes another, what does that imply about its scenarios?
12. How may use case participants and the system under consideration be represented on an internal block diagram?
13. How are use case participants and the system under consideration represented in interactions?
14. How are use case participants and the system under consideration represented in activities?

Discussion Topics

Apart from those listed discuss two additional descriptive properties that would be useful for describing actors.

Apart from those listed discuss two additional descriptive properties that would be useful for describing use cases.

12

Modeling Text-Based Requirements and Their Relationship to Design

This chapter describes how text-based requirements are captured in the model and related to other model elements. This chapter also describes the diagrammatic representations and special notations used to represent requirements and other cross-cutting relationships, such as allocations, in a SysML model.

12.1 Overview

A **requirement** specifies a capability or condition that must (or should) be satisfied, a function that a system must perform, or a performance condition a system must achieve.

Requirements come from many sources. Sometimes requirements are provided directly by the person or organization paying for the system, such as a customer who hires a contractor to build his or her house. Other times, requirements are generated by the organization that is developing the system, such as an automobile manufacturer that must determine the consumer preferences for its product. The source of requirements often reflects multiple stakeholders. In the case of the automobile manufacturer, the requirements will include government regulations for emissions control and safety as well as the direct preferences of the consumer.

Regardless of the source, it is common practice to group similar requirements into a **specification**. The individual requirements should be expressed in clear and unambiguous terms, sufficient for the developing organization to implement a system that meets stakeholder needs. However, the classic systems engineering challenge is to ensure that requirements are consistent (not contradictory) and feasible, have been validated to adequately reflect real stakeholder needs, and have been verified to ensure that they are satisfied by the system design.

Requirements management tools are also widely used to manage both requirements and the relationships between them. Requirements are typically maintained in some kind of digital repository. SysML has introduced a requirements modeling capability to provide a bridge between the text-based requirements that may be

maintained in a requirements management tool and the system model, using the requirements and configuration management processes define how to keep the requirements in sync with the model. This capability is intended to significantly improve requirements management throughout the life cycle of a system by enabling rigorous traceability between text-based requirements and model elements that represent the system design, implementation, and test cases.

The individual text requirements may be imported from a requirements management tool or text specification, or created directly in the system modeling tool. The specifications are typically organized into a hierarchical package structure that corresponds to a specification tree. Each specification contains multiple requirements, such as a systems specification that contains the requirements for the system, or the component specifications that contain the requirements for each component. The requirements contained in each specification are modeled in a containment hierarchy partitioning them into a tree structure that corresponds to how the specification is organized.

The individual or aggregate requirements within the containment hierarchy can then be linked to other requirements in other specifications and to model elements that represent the system design, implementation, or test cases. The derivation, satisfaction, verification, refinement, trace, and copy relationships supports a robust capability for relating requirements to one another and to other model elements. In addition to capturing the requirements and their relationships, a capability is provided to capture the rationale, or basis for a particular decision and for linking it directly to the relationship or other model element.

SysML provides multiple ways for capturing requirements and their relationships, in both graphical and tabular notations. A **requirement diagram** can be used to represent many of these relationships. In addition, compact graphical notations are available to depict the requirements relationships on any other SysML diagrams. The browser view of the requirements that is generally provided by the tool implementer also provides an important mechanism for visualizing requirements and their relationships.

Use cases have been used to support requirements analysis in many of the model-based approaches using UML and SysML. Different development methods may choose to leverage use cases in conjunction with SysML requirements. Use cases are typically effective for capturing the functional requirements, but are not as well suited for capturing a wide array of other requirements, such as physical requirements (e.g., weight, size, vibration); availability requirements; or other so-called nonfunctional requirements. The incorporation of text-based requirements into SysML effectively accommodates the broadest possible range of requirements that a systems engineer will face.

Use cases, like any other model element, can be related to requirements using the various relationships (e.g., the refine relationship). In addition, use cases are often accompanied by a use case description (see the example in Chapter 11). The steps in the use case description can be captured as individual text requirements, and then related to other model elements, to provide more granular traceability between the use cases and the model. Many other techniques with varying degrees of formalism can be used in addition to the examples cited here.

12.2 Requirement Diagram

Requirements captured in SysML can be depicted on a requirement diagram as well as other diagrams. The requirement diagram is generally used to graphically depict hierarchies of requirements or to depict an individual requirement and its relationship to other model elements. The requirement diagram header is depicted as follows:

req [package or requirement] Model Element Name [diagram name]

The diagram frame for a requirement diagram designates a model element type that can be a package or a requirement. The *model element name* is the name of the package or requirement containing the requirements, and the *diagram name* is user defined and often used to describe the purpose of the diagram. Figure 12.1 shows a generic example of a requirement diagram that contains some of the most common symbols.

This example highlights a number of different requirements relationships and alternative notations. For example, *Camera* satisfies the requirement called *Sensor Decision*. It also includes three different representations of *containment*, *deriveReqt*, *Verify*, and *satisfy* relationships (shown as direct relationships, in compartment notation, and in callout notation). In practice, only one of these representations would be used. Each of the symbols depicted on this diagram are discussed later in this chapter. Tables A.22 through A.24 in the Appendix contain a complete description of the SysML notation for requirements.

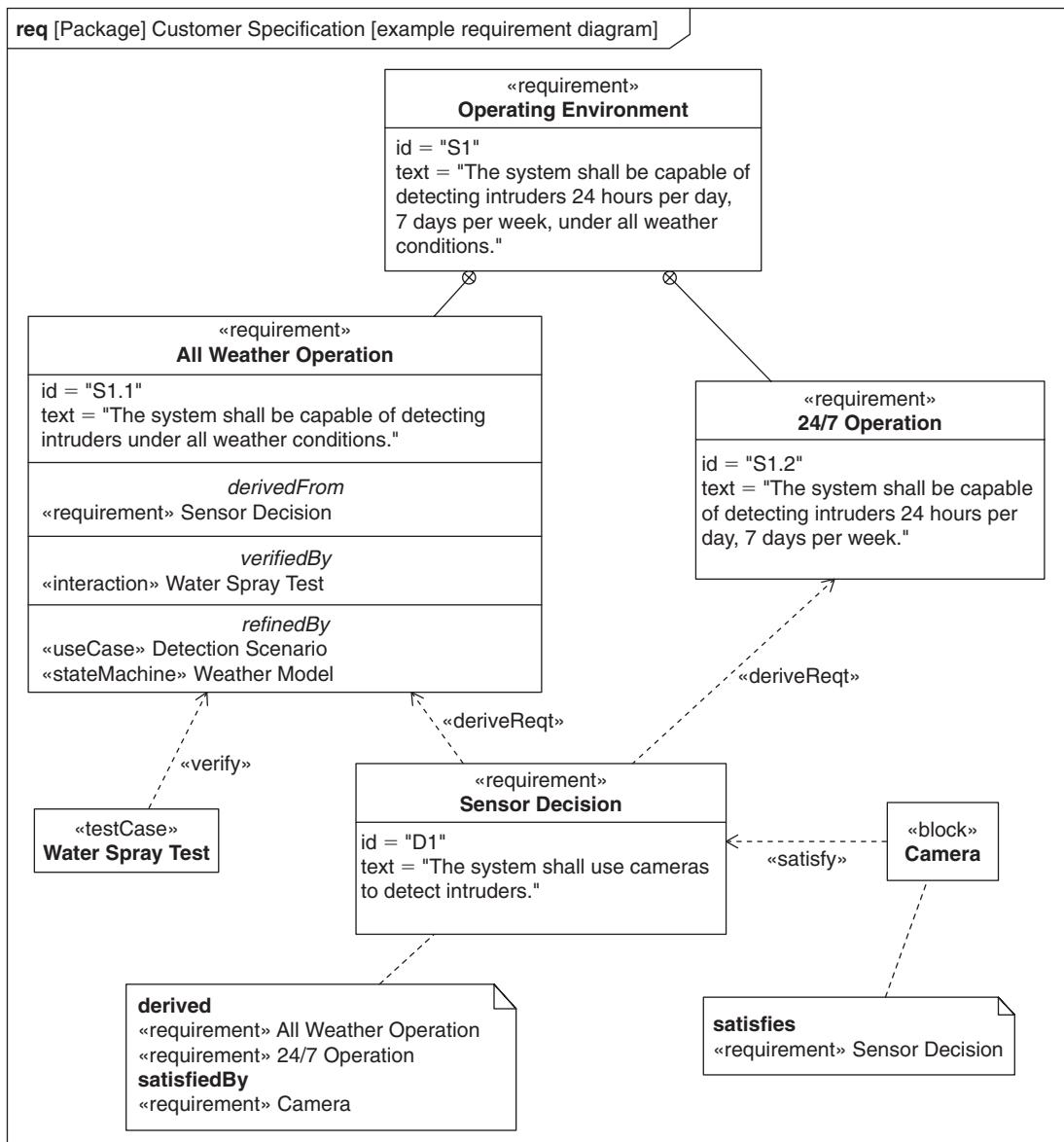
The requirements construct can be directly shown on block definition diagrams, package diagrams, and use case diagrams. The relationships between requirements and other model elements can be represented on other diagrams (e.g., block definition diagrams, internal block diagrams, and others) using compartment and callout notations; see Sections 12.5.2 and 12.5.3 for examples. Alternative ways to view requirements are discussed in Section 12.7 (tabular views) and Section 12.9.1 (browser view).

12.3 Representing a Text Requirement in the Model

A **requirement** that is captured in text represented in SysML using the «requirement» model element. Once captured, it can be related to other requirements and to other model elements through a specific set of relationships. Each requirement includes predefined properties for a unique identifier, and for a text string.

Figure 12.2 is an example of a text-based requirement called *Operating Environment* as represented in SysML. It is distinguished by the keyword «requirement» and will always contain, as a minimum, properties for *id* and *text*. This same information can be displayed in a tabular format that is described later in this chapter.

Requirements can be customized by adding additional properties such as verification status, criticality, risk, and requirements category. The verification status property, for example, may include values such as not verified, verified by inspection,

**FIGURE 12.1**

Generic example of a requirement diagram.

verified by analysis, verified by demonstration, and verified by test. A risk or criticality property may include the values high, medium, and low. A requirements category property may include values such as functional, performance, or physical.

An alternative method for creating requirements categories is to define additional subclasses of the requirement stereotype (see Chapter 14 for discussion on

«requirement»	
Operating Environment	
id = "S1"	
text = "The system shall be capable of detecting intruders 24 hours per day, 7 days per week, under all weather conditions."	

FIGURE 12.2

Example of a requirement as depicted in SysML.

subclassing stereotypes). The stereotype enables the modeler to add constraints that restrict the types of model elements that may be assigned to satisfy the requirement. For example, a functional requirement may be constrained so that it can only be satisfied by a behavioral model element such as an activity, state machine, or interaction. Annex C of the SysML specification [1] includes some nonnormative requirement subclasses, which are also shown in Table 12.1.

As shown in the table, each category is represented as a stereotype of the generic SysML «requirement». Table 12.1 also includes a brief description of the category. Additional stereotype properties or constraints can be added as deemed appropriate for the application.

Other examples of requirements categories may include operational requirements, specialized requirements for reliability and maintainability, store requirements, activation and deactivation requirements, and a high-level category for stakeholder needs. Some guidance for applying a requirements profile follows. (General guidance on defining a profile is included in Chapter 14.)

- The categories should be adapted for the specific application or organization and reflected in the table. This includes agreement on the categories and their associated descriptions, stereotype properties, and constraints. Additional categories can be added by further stereotyping the categories shown in Table 12.1, or adding additional categories at the peer level of these categories.
- Apply the more specialized requirement stereotype (functional, interface, performance, physical, design constraint) as applicable and ensure consistency with the description, stereotype properties, and constraints of these requirements.
- A specific text requirement can include the application of more than one requirement category, in which case each stereotype should be shown in guillemets (« ») or in a comma-separated list.

12.4 Types of Requirements Relationships

SysML includes specific relationships to relate requirements to other requirements as well as to other model elements. These include relationships for defining a requirements hierarchy, deriving requirements, satisfying requirements, verifying requirements, refining requirements, and copying requirements.

Table 12.1 Optional Requirements Stereotypes from SysML 1.0 Annex C.2

Stereotype	Base Class	Properties	Constraints	Description
«extendedRequirement»	«requirement»	source: String risk: RiskKind verifyMethod: VerifyMethodKind	N/A	A mix-in stereotype that contains generally useful attributes for requirements.
«functionalRequirement»	«extendedrequirement»	N/A	Satisfied by an operation or behavior	Requirement that specifies an operation or behavior that a system, or part of a system, must perform.
«interfaceRequirement»	«extendedrequirement»	N/A	Satisfied by a port, connector, item flow, and/or constraint property	Requirement that specifies the ports for connecting systems and system parts and that optionally may include the item flows across the connector and/or interface constraints.
«performanceRequirement»	«extendedrequirement»	N/A	Satisfied by a value property.	Requirement that quantitatively measures the extent to which a system, or a system part, satisfies a required capability or condition.
«physicalRequirement»	«extendedrequirement»	N/A	Satisfied by a structural element.	Requirement that specifies physical characteristics and/or physical constraints of the system, or a system part.
«designConstraint»	«extendedrequirement»	N/A	Satisfied by a block or a part.	Requirement that specifies a constraint on the implementation of the system or system part, such as “the system must use a commercial off-the-shelf component”.

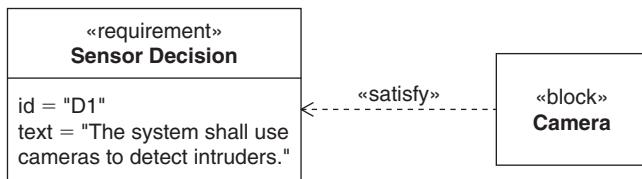
Table 12.2 Requirement Relationships and Compartment Notation

Relationship Name	Keyword Depicted on Relation	Requirement (arrow) End Callout/Compartment	Client (no arrow) End Callout/Compartment
Satisfy	«satisfy»	Satisfied by «model element»	Satisfies «requirement»
Verify	«verify»	Verified by «model element»	Verifies «requirement»
Refine	«refine»	Refined by «model element»	Refines «requirement»
Derive Requirement	«deriveReqt»	Derived «requirement»	Derived from «requirement»
Copy	«copy»	(None)	Master «requirement»
Trace	«trace»	Traced «model element»	Traced from «requirement»
Containment (Requirement decomposition)	(Crosshair icon)	(No callout)	(No callout)

Table 12.2 summarizes the specific relationships, which are discussed later in this chapter. The *containment*, *derive*, and *copy* relationships can only relate one requirement to another. The *satisfy*, *verify*, *refine*, and *trace* relationships can relate requirements to other model elements.

12.5 Representing Cross-Cutting Relationships in SysML Diagrams

Requirements can be related to model elements, even if they may appear in different hierarchies or on different diagrams. These relationships can be shown directly if the related model elements happen to appear on the same diagram. If the model elements do not appear on the same diagram, they can still be shown by using the compartment or callout notation. Direct notation may be used, for example, to show a derive requirement relationship between requirements on a requirement diagram. The compartment or callout notation is used when requirements do not appear on other kinds of diagrams, or when other model elements do not appear on a requirement diagram. An example is a block definition diagram showing a block that satisfies one or more requirements. In addition to these graphical representations, SysML provides for a flexible tabular notation for representing requirements and their relationships. Note that the allocation relationship, described in Chapter 13, is represented using the same notational approaches as those used here to represent the relationship between requirements and other model elements.

**FIGURE 12.3**

Example of direct notation depicting a satisfy relationship.

12.5.1 Depicting Requirements Relationships Directly

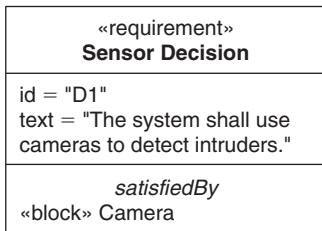
When the requirement and the model element it relates to are shown on the same diagram, this relation may be depicted directly. **Direct notation** depicts this relationship as a dashed arrow with the name of the relationship displayed as a keyword (e.g., «satisfy», «verify», «refine», «derive requirement», «copy», and «trace»).

Figure 12.3 provides an example of a «satisfy» relationship between a *Camera* and a requirement, *Sensor Decision*, where the camera is part of the design that is asserted to satisfy the requirement. Note that the arrowhead points to the requirement.

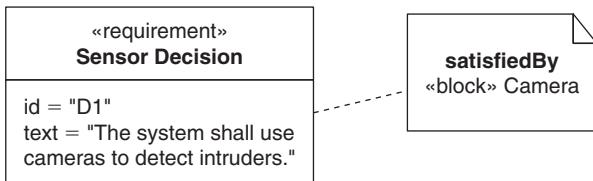
It is important to recognize the significance of the arrow direction. Since most requirement relationships in SysML are based on the UML dependency relationship, the arrow points from the dependent model element (called the client) to the independent model element (called the supplier). The interpretation of this relationship is that the camera design is dependent on the requirement, meaning that if the requirement changes, the design must change. Similarly, a derived requirement will be dependent on the requirement that it is derived from. In SysML, the arrowhead direction is opposite of what has typically been used for requirements flow-down or requirements allocation, in which the requirement points to the design element intended to satisfy it, or the higher-level requirement points to the lower-level requirement.

12.5.2 Depicting Requirements Relationships Using Compartment Notation

Compartment notation is an alternative method for displaying a requirement relationship between a requirement and another model element, such as a block, part, or another requirement, that supports compartments. This is a compact notation that can be used instead of displaying a direct relationship. It also can be used for diagrams that preclude display of a requirement directly, such as an internal block diagram. In Figure 12.4, compartment notation is used to show the same satisfy relationship to the requirement from Figure 12.3. This should be interpreted as “the requirement is satisfied by the *Camera*.” The compartment notation explicitly displays the relationship and direction (*satisfiedBy*), the model element type (*«block»*), and the model element name (*Camera*).

**FIGURE 12.4**

Example of compartment notation depicting a satisfy relationship.

**FIGURE 12.5**

Example of callout notation depicting a satisfy relationship.

12.5.3 Depicting Requirements Relationships Using Callout Notation

Callout notation is an alternative notation for depicting requirements relationships. It is the least restrictive notation in that it can be used to represent a relationship between any requirement and any other model element on any diagram type. This includes relationships between requirements and model elements, such as pins, ports, and connectors, that do not support compartments and therefore cannot use compartment notation.

A **callout** is depicted as a comment symbol that is graphically connected to a model element. The callout symbol represents the model element at the other end of the relationship. The callout notation depicted in Figure 12.5 shows the same information as the compartment notation in Figure 12.4, and it should be interpreted as “the requirement is satisfied by the *Camera*.”

12.6 Depicting Rationale for Requirements Relationships

A **rationale** is a SysML model element that can be associated with either a requirement or a relationship between requirements. As the name implies, the rationale is intended to capture the reason for a particular design decision. Although rationale is described here for requirements, it is a model element that can be applied throughout the model to capture the reason for any type of decision. A **problem** is like a rationale, but is used instead to specifically identify a particular problem that needs to be solved.

As shown in Figure 12.6, the rationale is expressed using a comment notation with the keyword «rationale». The problem is likewise identified with the

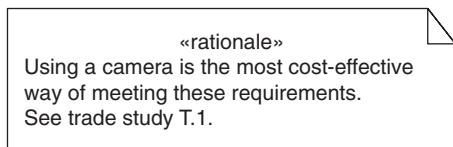


FIGURE 12.6

Example of rationale as depicted on any SysML diagram.

«problem» keyword. The text in the comment can either provide the rationale directly or reference an external document (e.g., a trade study or analysis report) or another model part such as a parametric diagram. The reference may include a hyperlink, although this is not explicit in the language. In this particular example, there is a reference to a trade study, *T.1*. The context of this particular rationale is shown in Figure 12.14 later in this chapter.

A rationale or problem can be attached to any requirements relationship or to the requirement. For example, a rationale or problem can be attached to a satisfy relationship, and refer to an analysis report or trade study that provides the supporting rationale for why the particular design satisfies the requirement. Similarly, the rationale can be used with other relationships such as the derive relationship. It also provides an alternative mechanism to the verify relationship by attaching a rationale to a satisfy relationship that references a test case.

12.7 Depicting Requirements and Their Relationships in Tables

The requirement diagram has a distinct disadvantage when viewing large numbers of requirements. Large amounts of real estate are needed to depict and relate all the requirements needed to specify a system of even moderate complexity. The traditional method of viewing requirements in textual documents is a more compact representation than viewing them in a diagram. Modern requirements management tools typically maintain requirements in a database, and the results of queries to the database can be displayed clearly and succinctly in tables or matrices. SysML embraces the concept of displaying results of model queries in tables, but the specifics of generating tables is left to the tool implementer.

Figure 12.7 provides an example of a simple **requirements table**. In this example, the table lists the requirements in the *System Specification* package. Depending on its capability, a tool may also apply query and filter criteria to generate requirements reports from a query of the model. This report can represent a view of the model, as described in Chapter 5. In addition, the tool may support editing requirements and their properties directly in the table view.

12.7.1 Depicting Requirement Relationships in Tables

A relationship path can be formed by selecting one or more requirements (or other model elements), and navigating the relationships from the selected requirement.

table [Package] System Specification [Decomposition of top-level requirements]		
id	name	text
S1	Operating Environment	The system shall be capable of detecting intruders 24 hours per day...
S1.1	Weather Operation	The system shall be capable of detecting intruders under all weather...
S1.2	24/7 Operation	The system shall detect intruders 24 hours per day, 7 days per week
S2	Availability	The system shall exhibit an operational availability (Ao) of 0.999...

FIGURE 12.7

Example of requirements table.

table [Requirement] Camera Decision [Requirements Tree]					
id	name	relation	id	name	Rationale
D1	Sensor Decision	derivedFrom	S1.1	24/7 Operation	Using a camera is the most cost-effective way of meeting these requirements. See trade study T1.
		derivedFrom	S1.2	Weather Operation	Using a camera is the most cost-effective way of meeting these requirements. See trade study T1.

FIGURE 12.8

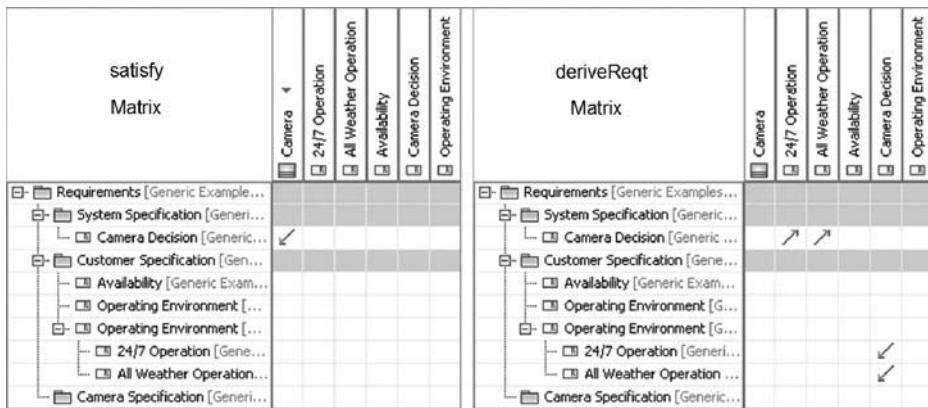
Example of table following the derivedFrom relationship.

This can be represented in **tabular form**, as shown in Figure 12.8. In this example, *D1* is the selected requirement, and the path includes a derivedFrom relationship and the rationale associated with the relationship.

The relationship paths can be arbitrarily deep; that is, navigates a single kind of relationship from one model element to the next, or navigates different types of relationships from one model element to the next. This can be particularly useful when analyzing the impact of requirements changes across the model. This query mechanism could also be used as a method to construct a view, as described in Chapter 5. Depending on tool capability, it may also be possible to edit requirement relationships and properties directly in the tabular view.

12.7.2 Depicting Requirement Relationships as Matrices

The tabular notation can also be used to represent multiple complex interrelationships between requirements and other model elements in the form of matrices. Figure 12.9 shows the result of a query in tabular (**matrix**) form; it depicts the satisfy and derive relationships. In this example, the requirements are shown in the left column, and the model elements that have a derive or satisfy relationship are shown in the other columns. Filtering criteria can be applied to limit the size of the matrix. In this example, the requirements properties have been excluded, and only the derive and satisfy relationships have been included. These relationships are discussed later in this chapter. Again, this is an example of a mechanism that a tool vendor might use to construct a view of the model.

**FIGURE 12.9**

Example of tabular view of requirements as matrices tracing satisfy and derive requirement relationships, respectively.

12.8 Modeling Requirement Hierarchies in Packages

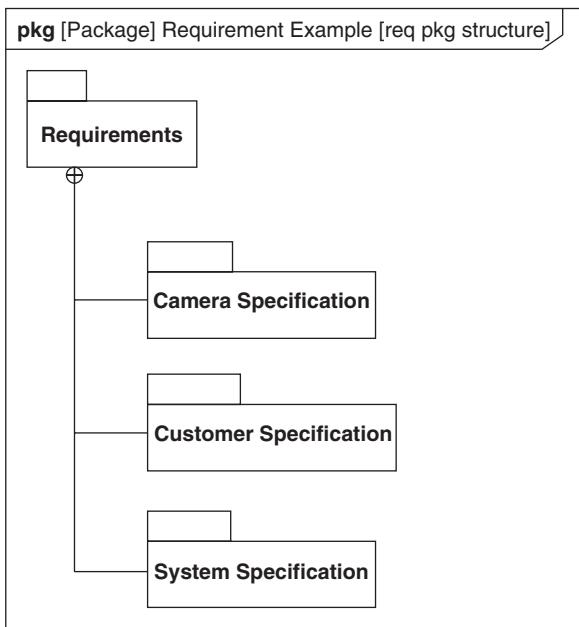
Requirements can be organized into a package structure. A typical structure may include a top-level package for all requirements relative to this model. Each package within this package structure may correspond to a different specification, such as the system specification, subsystem specifications, and component specifications. Each specification package contains the text-based requirements for that specification. This package structure corresponds to a typical specification tree that is a useful artifact for describing the scope of requirements for a project.

An example of a requirements package structure, or **specification tree**, is shown in the package diagram in Figure 12.10. The containment relationship with the crosshairs symbol at the owning end is used to indicate that the *Customer Specification* package, the *System Specification*, and the *Camera Specification* are contained in the *Requirements* package.

Organizing requirements into packages corresponding to various specifications provides familiarity and consistency with document-based approaches and facilitates configuration management of individual specifications at the package level. Also, a specification report can be generated directly from the contents of the appropriate package.

12.9 Modeling a Requirements Containment Hierarchy

The **containment** relationship is used to represent how a complex requirement can be partitioned into a set of simpler requirements without adding meaning or other implications. A containment relationship can be viewed as a logical anding (conjunction) of the contained requirements with the container requirement. The partitioning of complex requirements into simpler requirements is essential

**FIGURE 12.10**

Example of a package structure for organizing requirements.

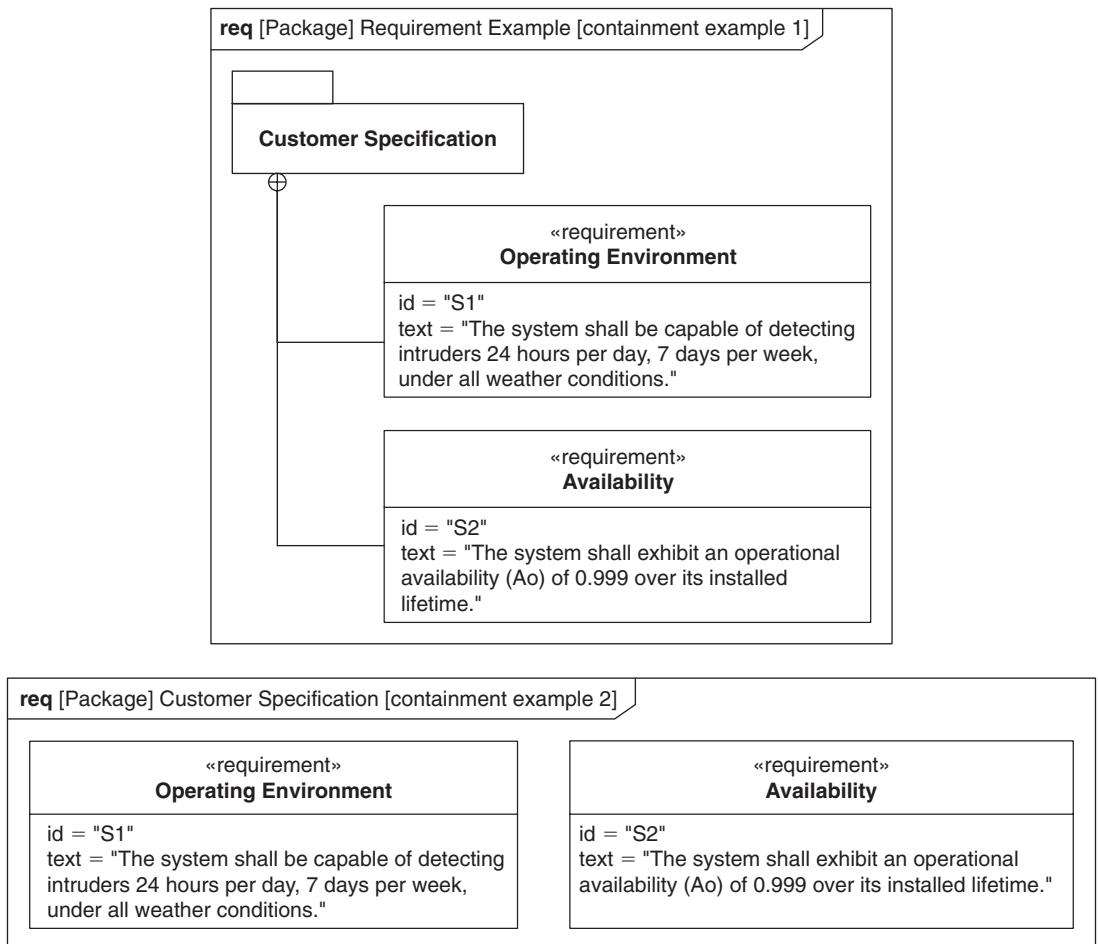
to establish full traceability and show how individual requirements are the basis for further derivation, and how they are satisfied and verified.

Figure 12.11 shows a requirement diagram with a simple containment hierarchy. The *Customer Specification* package from Figure 12.10 represents a top-level specification that serves as a container for all other customer-generated requirements. In this example, the *Customer Specification* package contains two other requirements, as depicted by the crosshairs symbol. Note that instead of using a package, a specification may be modeled as a «requirement» that contains a hierarchy of other requirements. A typical specification may contain from hundreds to thousands of individual requirements, but they generally can be organized into a hierarchy that corresponds to the organization of a specification document.

Figure 12.12 shows how containment hierarchies can be used to create multiple levels of **nested requirements**. In this example, the *Operating Environment* requirement contains two additional requirements for *All Weather Operation* and *24/7 Operation*.

12.9.1 The Browser View of a Containment Hierarchy

A typical modeling tool will include a **browser view** of the model that includes the requirements hierarchy. In Figure 12.13, the specification packages corresponding to the package diagram in Figure 12.10 are shown along with the requirements corresponding to the containment hierarchy in Figure 12.12. This representation is a compact way to view the requirements containment hierarchy.

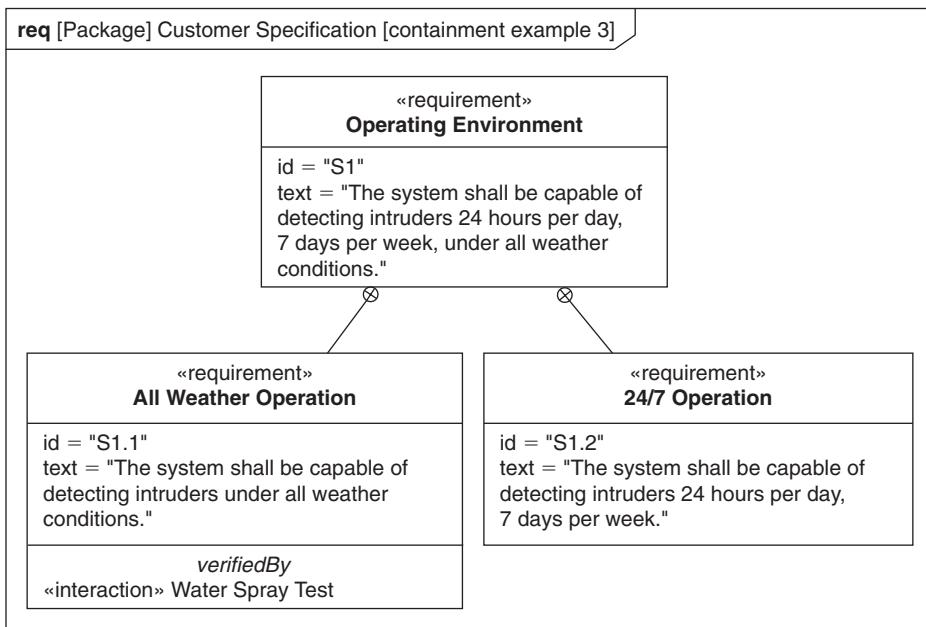
**FIGURE 12.11**

Two equivalent examples of requirements contained in a package.

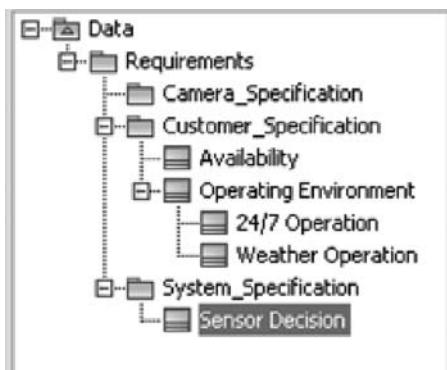
12.10 Modeling Requirement Derivation

Deriving requirements from source, customer, or other high-level requirements is fundamentally different from the containment relationship described in the previous section. A **derive relationship** between a derived requirement and a source requirement is based on an analysis. The analysis can be associated with the derived relationship by a «rationale».

An example of the derive relationship is represented in the requirement diagram in Figure 12.14. The relationship is shown with a dashed line with the keyword «deriveReqt» with the arrowhead pointing to the source requirement. Note that the «rationale» has been associated with the derivation relationship and includes a reference to trade study documentation.

**FIGURE 12.12**

Example of requirements containment hierarchy.

**FIGURE 12.13**

Example of requirements containment in a tool browser/explorer.

The requirements traceability matrix, included in traditional specification documents, often shows relationships between requirements in one specification to requirements in other higher- or lower-level specifications. This relationship is semantically equivalent to a set of SysML derive relationships. A derive relationship often shows relationships between requirements at different levels of the specification hierarchy. It is also used to represent a relationship between requirements at the peer level of the hierarchy, but at different levels of abstraction. For example, the analysis of hardware or software requirements, which are originally specified by

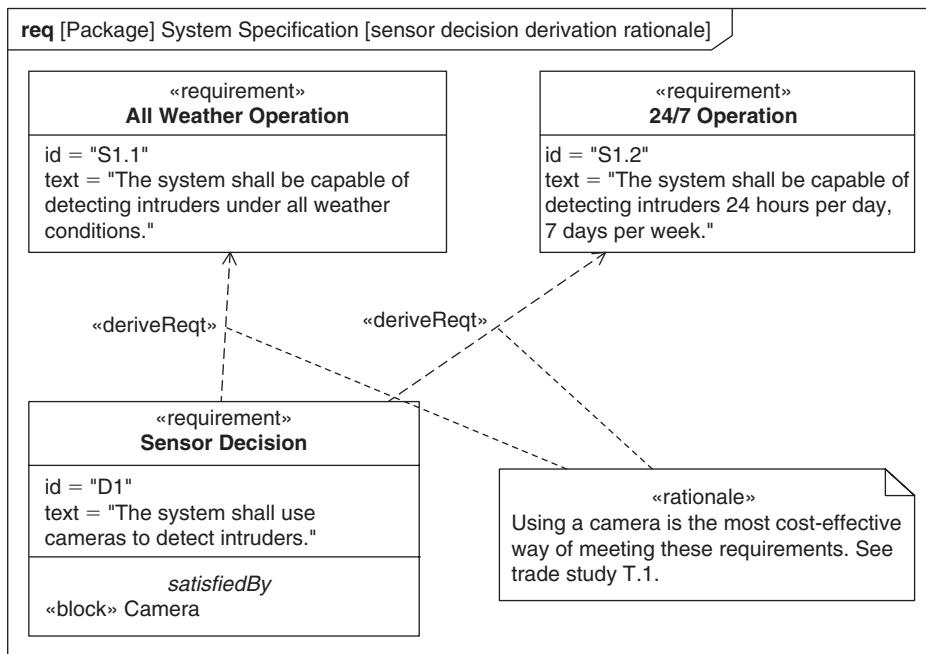


FIGURE 12.14

Example of «deriveReqt» relationship, with rationale attached.

the systems engineering team, may result in more detailed requirements that reflect additional implementation considerations or constraints. The more detailed requirements may be related to the original requirements through a derive relationship.

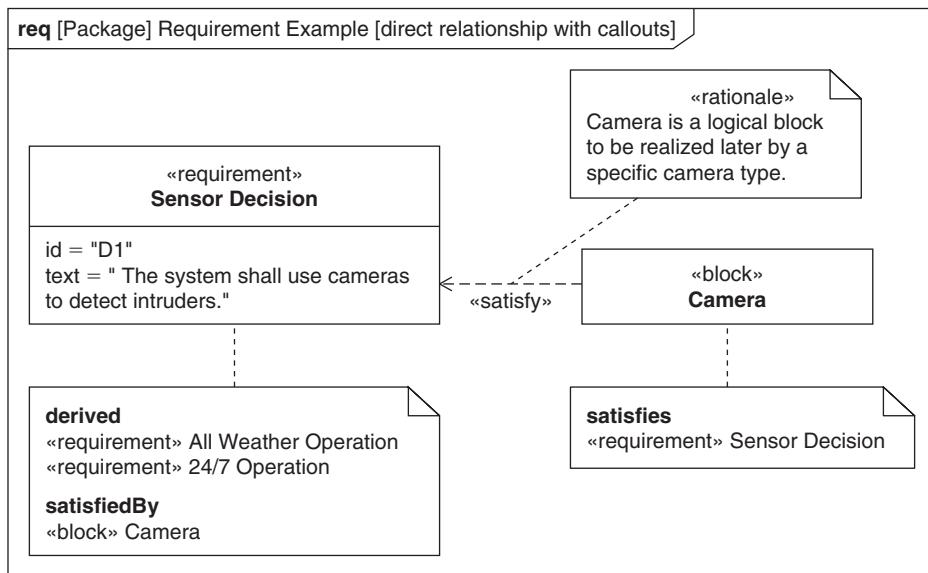
12.11 Asserting That a Requirement Is Satisfied

The **satisfy relationship** is used to assert that a model element corresponding to the design or implementation satisfies a particular requirement. The actual proof that the assertion is correct is accomplished by the verify relationship described in the next section. Figure 12.15 provides examples of the satisfy relationship.

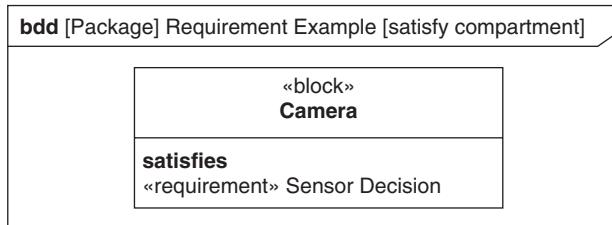
The satisfy relationship is shown with a dashed line with the keyword «satisfy» with the arrowhead pointing to the requirement to assert that the *Camera* satisfies the requirement. An alternative callout notation is also shown to represent this relationship. The «rationale» is associated with the satisfy relationship to indicate why this design is asserted to satisfy the requirement. In Figure 12.16, the same satisfy relationship from Figure 12.15 is shown on the block definition diagram using the compartment notation.

12.12 Verifying That a Requirement Is Satisfied

The **verify relationship** is a relationship between a requirement and a test case that is used to verify that the requirement is satisfied. As stated in the previous

**FIGURE 12.15**

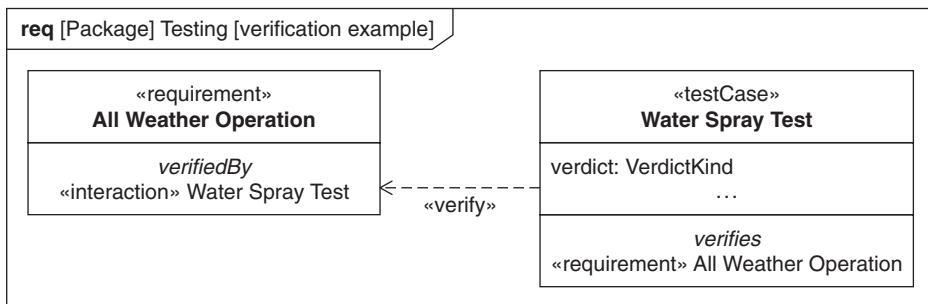
Example of requirement satisfy relationship and associated callout notation.

**FIGURE 12.16**

Example of satisfy relationship using compartment notation.

In this section, the satisfy relationship is an assertion that the model elements representing the design or implementation satisfy the requirement, but the verify relationship is used to prove the assertion is true (or false). A **test case** can represent any method for performing the verification, including the standard verification methods of inspection, analysis, demonstration, and testing. Additional stereotypes can be defined by the user if required to represent the different verification methods. The test case can reference a documented verification procedure, or it can represent a model of the verification method, such as an interaction (sequence diagram). The results of performing the test case is called the verdict, which can include a value of pass or fail or a specific value.

Figure 12.17 provides an example of the use of the verify relationship. The verify relationship is shown with a dashed line with the keyword «verify» with the arrowhead pointing from the *Water Spray Test* test case to the *All Weather*

**FIGURE 12.17**

Example of verify relationship.

Operation requirement that is being verified. An alternative compartment notation is also shown to represent this relationship.

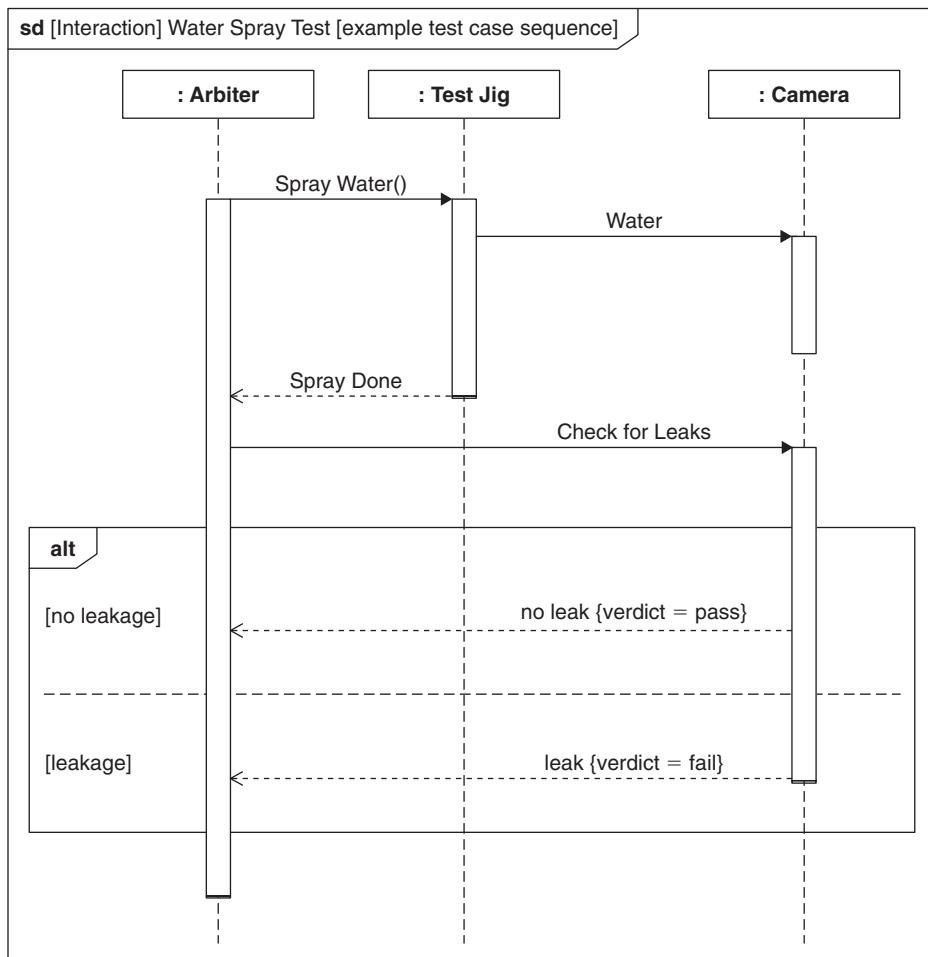
A test case keyword can be applied to other behaviors, including a sequence diagram, activity diagram, and state machine diagram, to specify the test case method. An example of applying the test case keyword to a sequence diagram is shown in Figure 12.18. In this case the test case shows an emulator representing the verification system, providing a stimulus to the system under test, and the interaction represents the expected system in response. The expected response can be compared with the actual response from running the test to assess whether the system actually satisfies the requirement.

A test case that is modeled as a behavior, in general, can represent a measurement of almost any characteristic, including structural characteristics. For example, the test case could represent a behavior that measures system weight. In this sense, a test case is a general-purpose mechanism for verifying requirements. In addition, other model elements can be used to verify a requirement. An example may include using a constraint block as an analysis method to verify a requirement.

The test case in SysML is defined consistent with the UML Testing Profile [35] to facilitate its integration. The test profile provides additional semantics for representing many other aspects of a test environment. The integration with the testing profile is covered briefly in Chapter 17 as part of the discussion on integrated system development environments.

12.13 Reducing Requirements Ambiguity Using the Refine Relationship

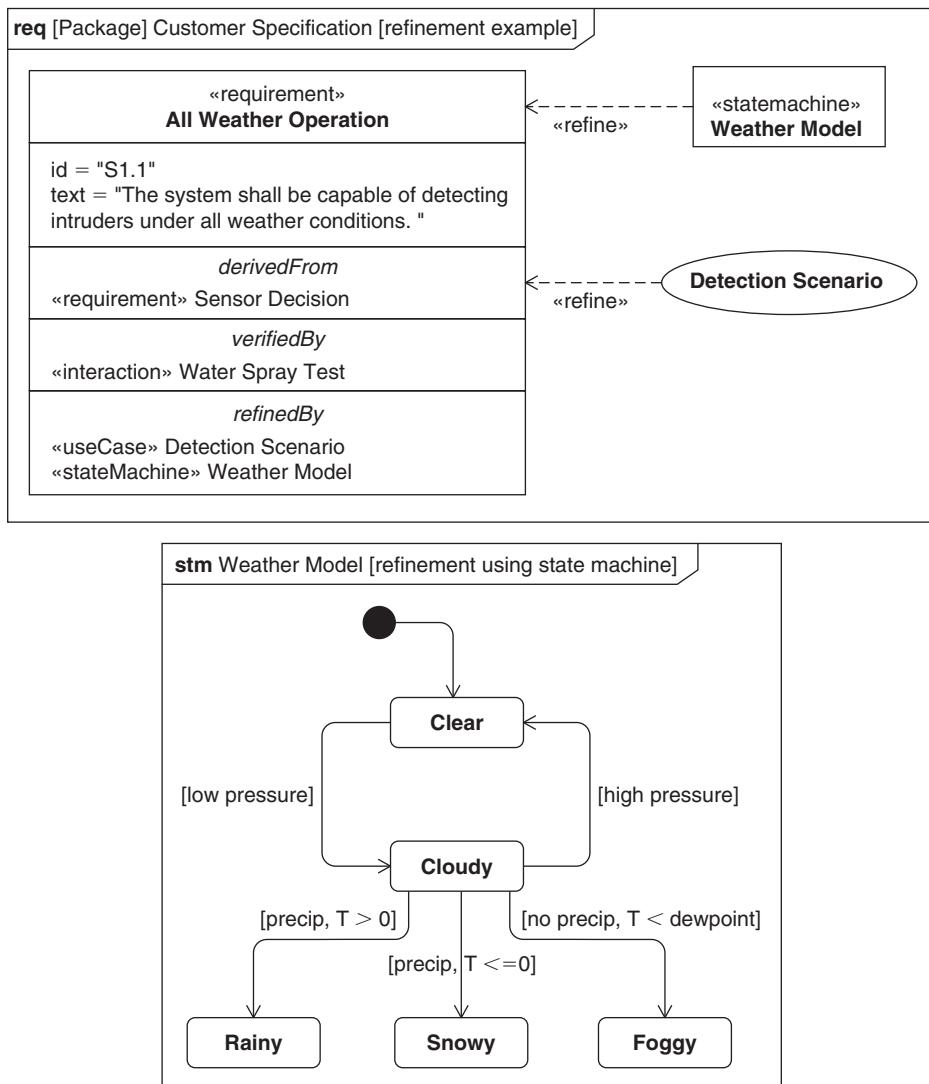
The **refine** relationship provides a capability to reduce ambiguity in a requirement by relating a SysML requirement to another model element that clarifies the requirement. This relationship is typically used to refine a text-based requirement with a model, but it can also be used to refine a model with a text-based requirement. For example, a text-based functional requirement may be refined with

**FIGURE 12.18**

Example of a test case interaction, depicted as a sequence diagram.

a more precise representation, such as a use case and its realizing activity diagram. Alternatively, the model element or elements may include a fairly abstract representation of required system interfaces that can be refined by an interface's text specification that includes a detailed description of an interface protocol or a physical layout of an interface envelope.

Refinement of requirements should clarify only the requirement meaning or context. It is distinguished from a derive relationship in that a refine relationship can exist between a requirement and any other model element, whereas a derive relationship is only between requirements. In addition, a derive relationship is not limited to a reexpression or clarification, but rather imposes additional constraints based on analysis.

**FIGURE 12.19**

Example of refine relationship applied to requirement.

An example of the refine relationship is provided in Figure 12.19; it shows how the *All Weather Operation* requirement is refined by a state machine that models weather conditions and transitions. The refine relationship is shown with a dashed line with the keyword «refine» with the arrowhead pointing from the element that represents the more precise representation to the element being refined. An alternative callout and compartment notation is also shown to represent this relationship. Note that the *Weather Model* state machine only partially refines the requirement. The *Detection Scenario* use case might address, for example, specific detection expectations in each weather condition.

12.14 Using the General-Purpose Trace Relationship

A **trace relationship** provides a general-purpose relationship between a requirement and any other model element. The trace semantics do not include any constraints and therefore are quite weak. However, the trace relationship can be useful for relating requirements to source documentation or for establishing a relationship between specifications in a specification tree.

As shown in Figure 12.20, the trace relationship is used to relate a particular requirement to a *Market Survey* that was conducted as part of the needs analyses. The trace relationship is shown with a dashed line with the keyword «trace» with the arrowhead pointing to the source document. The survey is represented as a user-defined model element with the keyword «document».

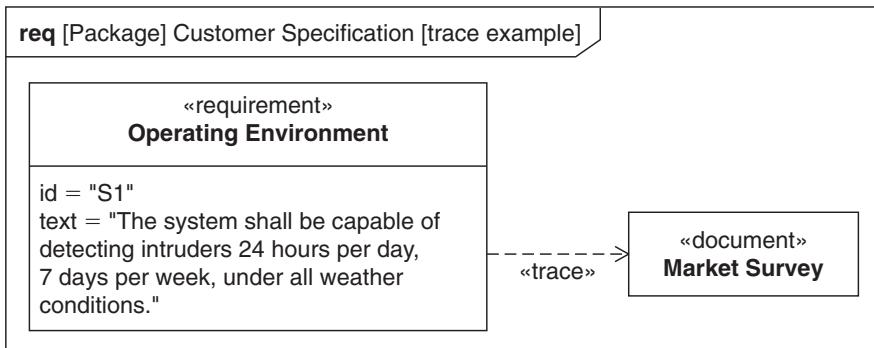


FIGURE 12.20

Example of trace relationship linking a requirement to an element representing an external document.

12.14.1 Reusing Requirements with the Copy Relationship

Requirements in SysML are not like blocks in that they cannot have subclasses, or be generalized or specialized. As in specification documents, SysML requirements are simply textual imperatives, and they cannot have multiple usages, but they can be copied. The **copy** relationship relates a copy of a requirement to its original to support reuse of requirements. A requirement exists in one namespace or containment hierarchy and has specific meaning in its containing context. To support reuse of the requirement, the copied requirement is a requirement whose text property is a read-only copy of the text property of the source requirement, but with a different id.

An example of a copy relationship is shown in Figure 12.21. The copy relationship is shown with a dashed line with the keyword «copy» with the arrowhead pointing from the copied requirement to the source requirement. In this example, the source requirement being copied is a requirement from a technical standard that is reused in many different requirements specifications.

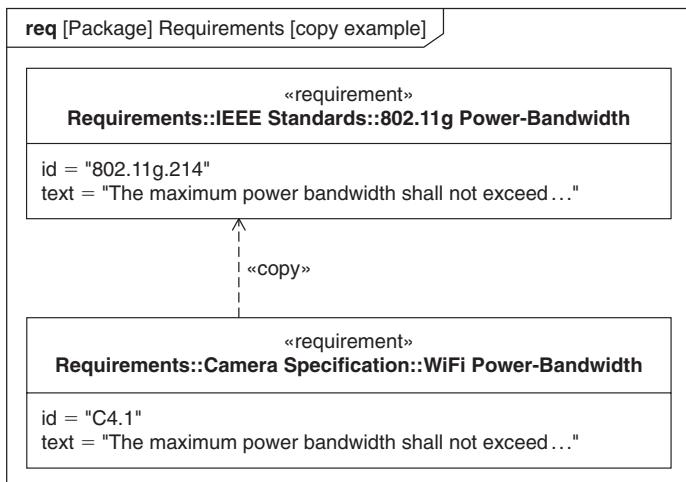


FIGURE 12.21

Example of a requirement copy relationship.

12.15 Summary

SysML can be used to model text-based requirements and relate them to other requirements and to other model elements. The following are some of the key requirements modeling concepts.

- The SysML requirements modeling capability serves as a bridge between traditional text-based requirements and the modeling environment. The requirements can be imported from a requirements management tool, or text specification, or created directly in the modeling tool.
- Each specification is generally captured in a package. The package structure can correspond to a traditional specification tree. Each specification in turn includes a containment hierarchy of the requirements contained within the specification. The browser view in most tools can be used to view the requirements containment hierarchy.
- The individual or aggregate requirements can then be related to other requirements in other specifications as well as model elements that represent the design, implementation, or test cases. The requirements relationships include derive, satisfy, verify, refine, trace, and copy. These relationships provide a robust capability for managing requirements and supporting requirements validation and verification so that the design satisfies the requirements.
- There are multiple notational representations to enable requirements to be related to other model elements on other diagrams; they include direct notation, compartment notation, and callout notation. The requirement diagram is generally used to represent a containment hierarchy or to represent the relationships for a particular requirement. Tabular notations are also used to efficiently report requirements and their relationships.

12.16 Questions

1. What is the abbreviation for a requirement diagram that appears in the diagram header?
2. Which kind of model element can the frame of a requirement diagram represent?
3. Which standard properties are expressed in a SysML requirement?
4. Can you add additional properties and constraints to a requirement?
5. What type of requirement relationships can only exist between requirements?
6. How do you read Figure 12.3?
7. How do you express the requirement relationship in Question 6 using callout notation?
8. How do you express the requirement relationship in Question 6 using compartment notation?
9. How do you represent a «deriveReqt» relationship between Reqt A and Reqt B in a matrix?
10. How do you represent the rationale for the derived requirement in Figure 12.14 that the derivation is based on the xyz analysis?
11. What is a satisfy relationship used for? (Select from answers a-c.)
 - a. to ensure a requirement is met
 - b. to assert a requirement is met
 - c. to more clearly express a requirement
12. What are the elements found on either end of a verify relationship?
13. What is used as a basis for a derived relationship? (Select from answers a-c.)
 - a. analysis
 - b. design
 - c. test case
14. How would you decompose the requirement A into two requirements A.1 and A.2 using the containment relationship?
15. Which relationship would you use to relate a requirement to a document? (Select from answers a-d.)
 - a. deriveReqt
 - b. satisfy
 - c. verify
 - d. trace
16. Why are requirements included in SysML? (This can be a discussion topic rather than a question.)

Discussion Topics

What are different uses of a requirement diagram?

When would you use a requirement diagram versus a table?

How can requirements and use cases be used together?

This page intentionally left blank

13

Modeling Cross-Cutting Relationships with Allocations

This chapter describes how allocation relationships are used to map from one model element to other model elements to support behavioral, structural, and other forms of allocation.

13.1 Overview

Beginning early in systems development, the modeler may need to associate various elements in the system model in abstract, preliminary, and sometimes tentative ways. It may be inappropriate to impose detailed constraints on the solution too early in the development of a system's architecture. **Allocation** is a mechanism to relate model elements that is typically a prelude to more rigorous relationships that are established through follow-on model refinement. Additional user-defined constraints can augment the allocation relationship to add the necessary rigor as the design progresses. For example, an allocation of functions (e.g., activities) to components may be done early in the design. As the design progresses, additional constraints are defined to ensure that the activity inputs, outputs, and controls are explicitly allocated to component interfaces. With appropriate user-defined constraints, allocation can be used to help enforce specific system development methods to ensure the model's integrity.

Allocation may be appropriate when modeling a system of systems (SoS), knowing that detailed system development and model refinement may be conducted by different teams, and perhaps even different companies. It also provides a mechanism for dealing with legacy system elements that have not been developed using rigorous modeling techniques. In both cases, allocation can be used to formalize constraints, expectations, or assumptions about that particular system element within the context of a broader system model.

The allocation relationship is used to support many forms of allocation including allocation of behavior, structure, and properties. A typical example of behavioral allocation is the allocation of activities to blocks (traditionally called functional allocation), where each system component is assigned responsibility for implementing a particular activity. An important distinction is made between

allocation of definition and allocation of usage. For functional allocation, allocating activities to blocks is an allocation of definition, and allocating actions to parts is an allocation of usage.

SysML includes several notational options to provide flexibility for representing allocations of model elements across the system model. The options include both graphical and tabular representations, similar to those used for relating requirements. Figure 13.1 shows some of the graphical representations of allocation on an activity diagram, on an internal block diagram, and on a block definition diagram. A complete description of the SysML notation for allocations can be found in the Appendix, Table A.23.

13.2 Allocation Relationship

An **allocation relationship** may be established between any two named model elements. Every SysML allocation relationship has one “from” end and one or more “to” or arrow ends. Model element *A* is said to be “allocated to” model element *B*, when the “from” end of the allocation relationship relates to *A* and the “to” (arrow) end relates to *B*. Additional constraints may be placed on allocations in special cases; for example, functional allocation may be constrained to occur only between blocks and activities. Section 13.4 discusses various types of allocation.

13.3 Allocation Notation

There are several types of notation to represent allocation of one model element to another. The notations used to represent allocation relationships are similar to the graphical and tabular notations used to represent requirements relationships, as described in Section 12.5 in Chapter 12. Graphical notations include the direct notation, compartment notation, and callout notation.

When the model elements at both ends of the allocation relationship can be shown on the same diagram, the allocation relationship can be depicted directly, as indicated in Figure 13.2, using the keyword «allocate» on the relationship. Here, the *Adjust Focus Motor* activity is allocated to the *Focus Optimizer*, and the arrowhead represents the “allocatedTo” end of the relationship. Although functional allocation is depicted in this example, this representation is equally valid for other types of allocations.

As with requirements relationships, it is often the case that the model elements at either end of the allocation relationship are on different diagrams. For these cases, compartment notation and callout notation can be used to identify the model element at the other end of the allocation relationship as described later.

The compartment notation identifies the element at the opposite end of the allocation relationship in a compartment of the model element, as shown in Figure 13.3. However, this can only be used when the model element can include compartments such as blocks and parts. It cannot be used for model elements that do not have compartments such as connectors.

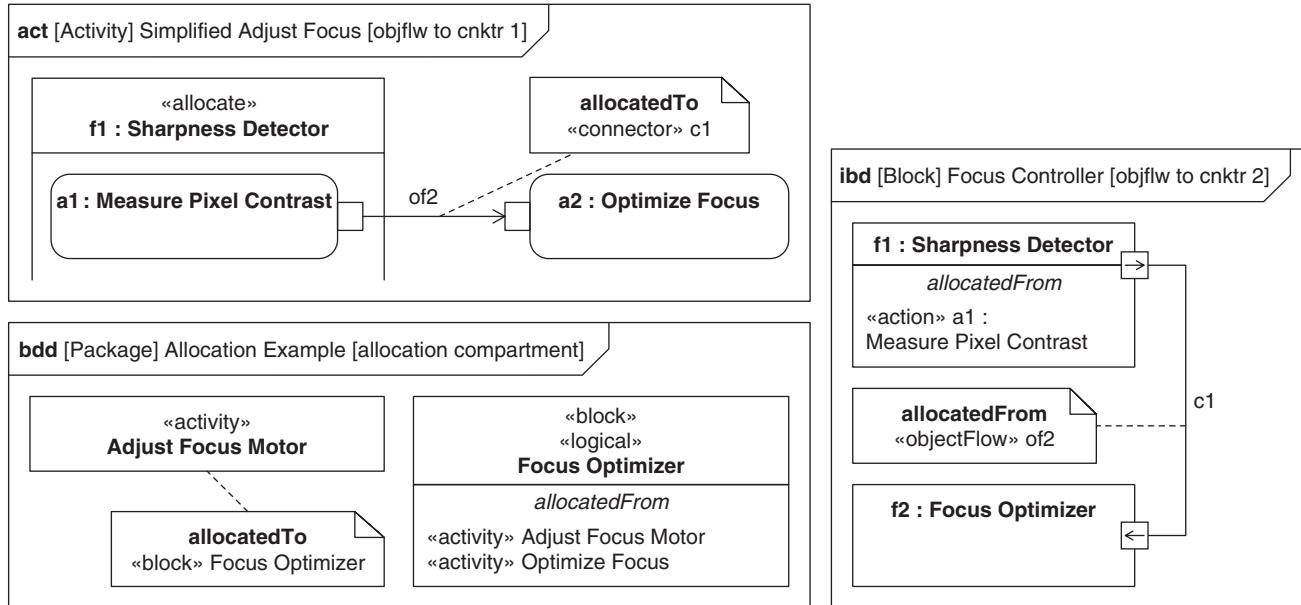
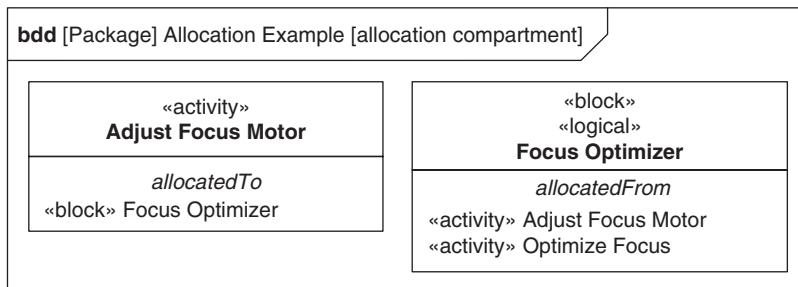


FIGURE 13.1

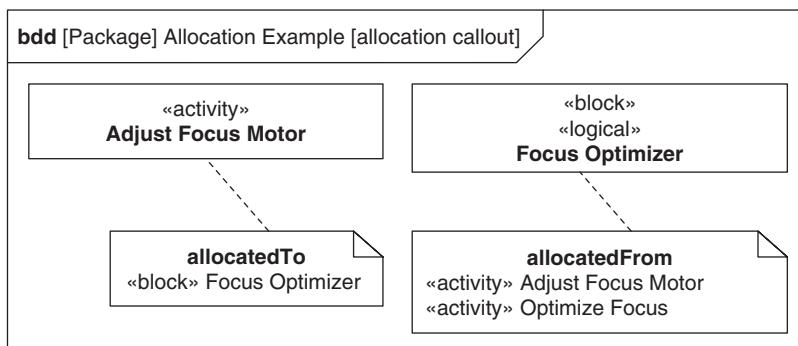
Examples of allocation on activity, block definition, and internal block diagrams.

**FIGURE 13.2**

Example directly depicting an allocation relationship, when both model elements appear on the same diagram.

**FIGURE 13.3**

Example depicting an allocation relationship in compartment notation.

**FIGURE 13.4**

Example depicting an allocation relationship in callout notation.

The callout notation shown in Figure 13.4 can be used to represent the opposite end of the allocation relationship for any model element whether it has compartments or not. Callout notation is represented as a note symbol that specifies the type and name of the model element at the other end of the allocation relationship. It also identifies which end of the allocation relationship applies to the model element as indicated by the *allocatedTo* or *allocatedFrom*. The callout notation is read by starting with the name of the model element that the callout notation attaches to, then reading the *allocatedTo* or *allocatedFrom*, and then reading the model element name in the callout symbol. For example, the allocation

	Focus Controller	Focus Optimizer	Sharpness Detector	Video Quality Che...
Behavior [Generic Examples]:...				
└ Adjust Focus(current : Im...				↗
└ Adjust Focus Motor(delta ...				↗
└ Measure Pixel Contrast(cu...				↗
└ Optimize Focus(contrast : ...				↗

FIGURE 13.5

Example depicting allocation relationships in tabular matrix form.

relationship in Figure 13.4 is read: “The activity *Adjust Focus Motor* is allocated to the block *Focus Optimizer*.“

A tabular or matrix notation is also used to depict multiple allocation relationships as shown in Figure 13.5. In this example, activities are in the left column and blocks are displayed in the top row. This format is not specifically prescribed by the SysML specification and will vary from tool to tool. The arrows in the matrix indicate the direction of the allocation relationships, consistent with those shown in Figures 13.3 and 13.4.

This matrix form of representing allocations is particularly useful when a concise, compact representation is needed, and it is used often in this chapter to illustrate allocation concepts. Allocations to or from some model elements, such as item flows, cannot be unambiguously depicted on any diagram, and thus can only be shown in a matrix or tabular form.

13.4 Types of Allocation

The following section describes different types of allocations including allocation of requirements, behavior, flow, structure, and properties.

13.4.1 Allocation of Requirements

The term **requirement allocation** represents a mechanism for mapping source requirements to other derived requirements, or mapping requirements to other model elements that satisfy the requirement. (See Chapter 12 for more information on these kinds of relationships.) SysML does not use the «allocate» relationship to represent this form of allocation, but instead uses specific requirements relationships that are described in Chapter 12.

13.4.2 Allocation of Behavior or Function

The term **behavioral allocation** generally refers to a technique for segregating behavior from structure. A common systems engineering practice is to separate models of structure (sometimes referred to as “models of form”) from models of behavior (sometimes referred to as “models of function”) so that designs can be optimized by considering several different structures that provide the desired emergent behavior and properties. This approach provides the required degrees of freedom—in particular, how to decompose structure, how to decompose behavior, and how to relate the two—to optimize designs based on trade studies among alternatives. The implication is that an explicit set of relationships must be maintained between behavior and structure for each alternative.

The behavior of a block can be represented in different ways. On a block definition diagram, the operations of a block explicitly define the responsibility the block has for providing the associated behavior (see Section 6.5 in Chapter 6 for more on specifying operations for blocks). In a sequence diagram, a message sent to a lifeline invokes the operation on the receiving lifeline to provide the behavior (see Chapter 9 for more on interactions). In activity diagrams, the placement of an action in an activity partition implicitly defines that the part represented by the partition provides the associated behavior (see Chapter 8 for more on activities).

In this chapter, the term behavioral allocation refers to the general concept of allocating elements of behavioral models (activities, actions, states, object flow, control flow, transitions, messages, etc.) to elements of structural models (blocks, properties, parts, ports, connectors, etc.). The term **functional allocation** is a subset of behavioral allocation, and it refers specifically to the allocation of activities (also known as functions) or actions to blocks or parts.

13.4.3 Allocation of Flow

Flow represents the transfer of energy, mass, and/or information from one model element to another. Flows are typically depicted as object flows between action nodes on activity diagrams, as described in Chapter 8, and as item flows between ports or parts on an internal block diagram, as described in Chapter 6. **Flow allocation** is often used to allocate flows between activity diagrams and internal block diagrams.

13.4.4 Allocation of Structure

Structural allocation refers to allocating elements of one structural model to elements of another structural model. A typical example is a **logical–physical allocation**, where a logical block hierarchy is often built and maintained at an abstract level, and in turn is mapped to another physical block hierarchy at a more concrete level. **Software–hardware allocation** is another example of structural allocation. In SysML, allocation is often used to allocate abstract software elements to hardware elements. UML uses the concept of deployment to

specify a more detailed level of allocation that requires software artifacts to be deployed to platforms or processing nodes. The transition from a SysML allocation to a UML deployment may be accomplished through model refinement and more detailed modeling and design of the software.

13.4.5 Allocation of Properties

Allocation can also be used to allocate performance or physical properties to various elements in the system model. This often supports the budgeting of system performance or physical property values to property values of the system components. A typical example is a weight budget in which system weight is allocated to the weights of the system's components. Once again, the initial allocation can be specified in more detail as part of model refinement using parametric constraints, as discussed in Chapter 7.

13.4.6 Summary of Relationships Associated with the Term “Allocation”

Table 13.1 is a partial list of some uses of the term “allocation” for systems modeling, along with proposed SysML relationships to meet usages’ purpose.

Table 13.1 Various Uses of “Allocation” and How to Represent in SysML

Kind of Allocation	Reference	Relationship	From	To
Requirement allocation	Section 12.11	Satisfy	requirement	named element
	Section 12.10	DeriveReqt	requirement	requirement
	Section 12.13	Refine	named element	requirement
Functional allocation	Section 13.6	Allocate	activity action	block part
Structural allocation (e.g., logical to physical, hardware to software)	Section 13.9	Allocate	block port	block port
	Section 13.10		item flow	item flow
	Section 13.9		connector	parts and connectors
Flow allocation	Section 13.7	Allocate	object flow object flow object flow	connector item flow item property
Property decomposition/ allocation	Section 7.7	Binding connector	value property	parameter

13.5 Planning for Reuse: Specifying Definition and Usage in Allocation

The terms definition and usage were discussed in Chapter 6. The term definition refers to a model element that is a classifier or type such as a block. The element is defined by specifying its features such as the properties and operations of a block. The term usage identifies a defined element in a particular context. For example, a part is a usage of a block in the context of a composite block, and the part is defined by the block that types it. The parts connection with other parts on an internal block diagram, and its interaction with other parts on an activity or a sequence diagram, describes how the part is used. Leveraging the concepts of definition and use is a significant strength of SysML, but it also requires careful consideration to maintain consistency across different potential usages.

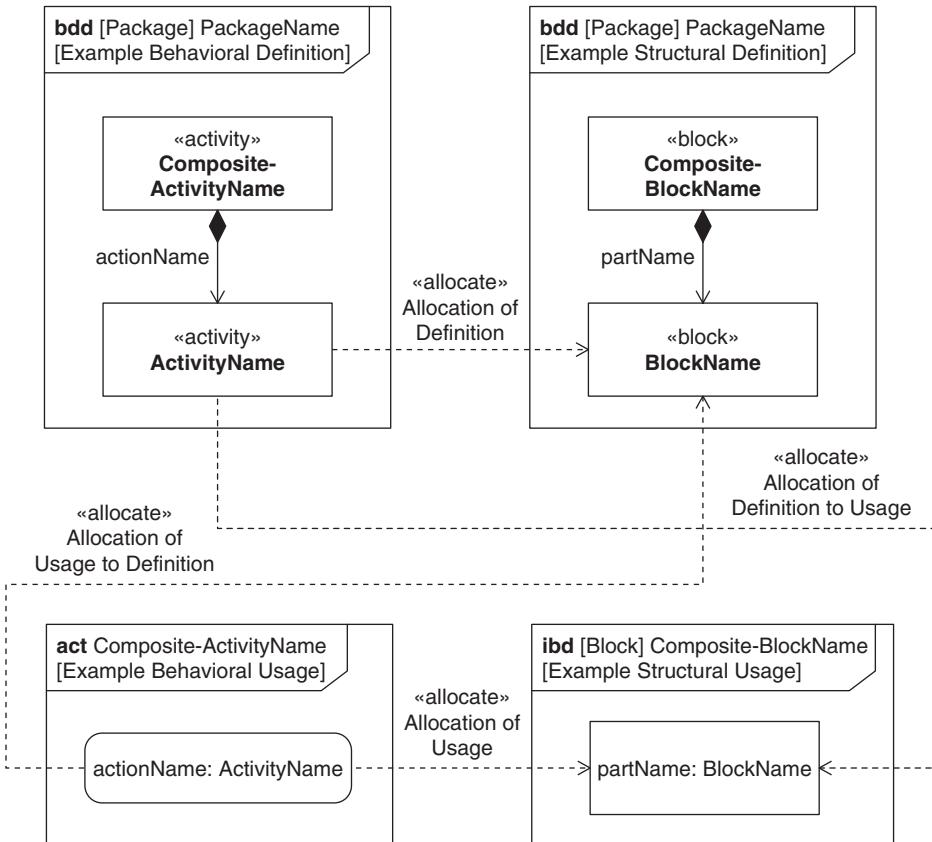
The concept of definition and usage is not restricted to structure. Chapter 10 discusses a similar depiction of activity composition on a block definition diagram and depicting that same composition as actions on an activity diagram. Similarly, constraint blocks are defined on a block definition diagram, and their usage is represented on a parametric diagram. Table 13.2 shows the different kinds of diagrams, the model elements that represent usages on the diagrams, and the model elements that can be used to type or define them.

The model element's definition is generally shown on a block definition diagram. However, the usage name can refer to the definition by its type—for example, *action name : Activity Name*. A common convention is that usage names are all lowercase and definition names start with leading uppercase.

Allocation can be used to relate elements of definition (blocks, activities, etc.) and elements of usage (actions, parts, etc.) in various combinations to provide considerable flexibility in how allocations are employed. As shown in Figure 13.6, activities and actions are allocated to both blocks and parts. While this figure

Table 13.2 Contextualized Elements Representing Usages and Their Definition

Diagram Kind	Model Element/Usage	Model Element/Definition
Activity diagram	action object node/action pin activity edge (object flow, control flow)	activity block (optional) (none)
Internal block diagram	Part connector item flow item property value property	block (optional) association (optional) (none) block (optional) value type
Parametric diagram	Constraint property	constraint block (optional)

**FIGURE 13.6**

Allocation of definition and usage. Functional allocation is shown here, but structural allocation is similar. Flow allocation will be discussed separately.

explicitly depicts functional allocation, the concept applies equally to structural allocation (block to block, part to part, etc.).

13.5.1 Allocating Usage

As shown in Figure 13.6, **allocation of usage** applies when both the “from” and “to” ends of the allocation relationship relate to usage elements (parts, actions, connectors, etc.). When allocating usage, nothing is inferred about the corresponding allocation of definition (blocks, activities, etc.). Only the specific usage is affected by the allocation. For example, if an action is allocated to a part on an internal block diagram, the allocation is only specific to that part, not to any other similar parts, even if they are typed by the same block.

Allocation of usage does not impact anything at the definition level, thus it does not impact other uses of similar parts. If there are a large number of similar parts with similar allocated characteristics or functions, it may be more appropriate

to allocate and provide these characteristics at the definition level to each of its parts as described next.

13.5.2 Allocating Definition

Allocation of definition applies when both “from” and “to” (arrow) ends of the allocation relationship relate to elements of definition (blocks, activities, associations, etc.). When allocating definition, every usage of the defining element retains the allocation. For example, if a block were used to define several parts, an allocation to the block would apply to all its parts (i.e., usages of the block).

13.5.3 Allocating Asymmetrically

Asymmetric allocation is when one end of the allocation relationship relates to an element of definition, and the other end relates to an element of usage. Asymmetric allocation is used by exception; that is, it is not generally recommended since it can introduce notational ambiguity. Allocation of usage or allocation of definition are the preferred allocation approaches.

13.5.4 Guidelines for Allocating Definition and Usage

The significance of using allocation of usage and allocation of definition relationships is discussed in Table 13.3. By examining these two approaches to allocation with respect to functional allocation, flow allocation, and structural allocation, the following conclusions can be drawn:

- Allocation of usage is localized to the fewest model elements and has no inferred allocations. It can be directly represented on diagrams of usage (e.g., internal block diagram or activity diagram), which establishes the context for the allocation.

Table 13.3 Allocation Guidelines Table

Allocation of Usage	Allocation of Definition
<p>Example: part to part, action to part, connector to connector, property to property</p> <p>Applicability: when the allocation is not intended to be reused</p> <p>Discussion</p> <ul style="list-style-type: none"> – Most localized with least implication on other diagrams and elements – Only way to allocate flows and connectors that have no definition – Possible redundancy or inconsistency as parts/actions used in multiple places 	<p>Example: block to block or activity to block</p> <p>Applicability: when the allocation is intended to apply to all usages</p> <p>Discussion</p> <ul style="list-style-type: none"> – Allocation inferred to all usages – Can result in overallocation (more activities allocated to a part than really necessary) – Not directly represented on an activity diagram with allocate activity partition (see Section 13.6.3)

It is appropriate to start with allocation of usage and consider allocation of definition after each of the uses has been examined.

- Allocation of definition is a more complete form of allocation because it applies (is inferred) to every usage. Allocation of definition follows from allocation of usage, as it typically requires blocks or activities to be specialized to the point, where the allocation of definition is unique, and overallocation (more allocations than really desired) is avoided. If a part requires a unique allocation, using allocation of definition requires the additional step of creating an additional block to define the part uniquely, and then allocating to (or from) that specialized block instead of to the part. This extra attention to refine the definition should facilitate future reuse of definition hierarchies.

13.6 Allocating Behavior to Structure Using Functional Allocation

Functional allocation is used to allocate functions to system components. Figure 13.7 defines a suitably complex behavioral hierarchy and a structural hierarchy to be used for the following functional allocation examples.

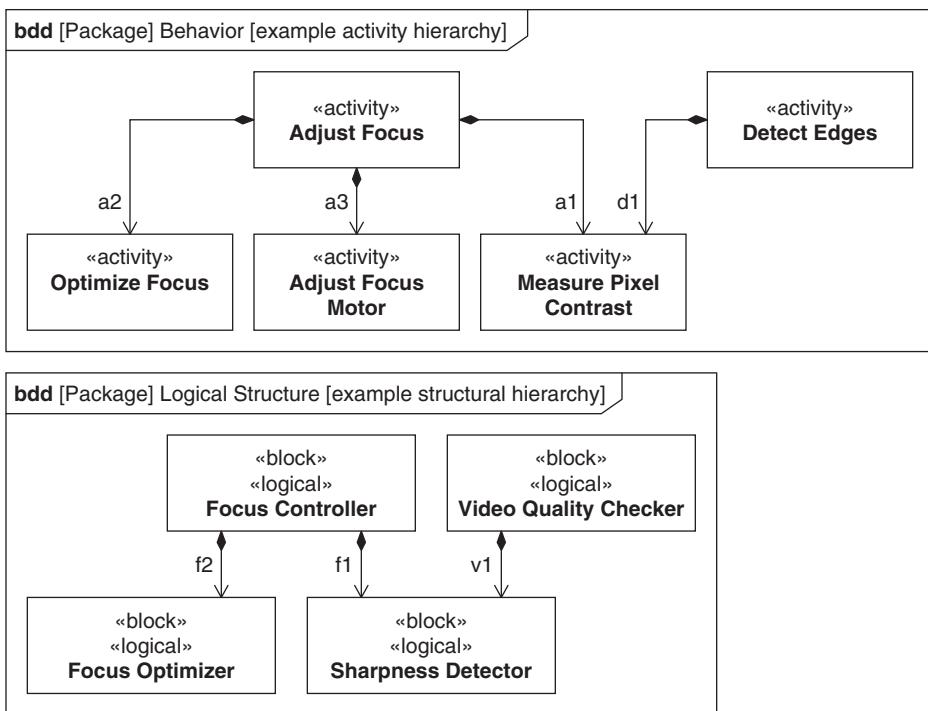


FIGURE 13.7

Example behavioral and structural hierarchy definition.

Note that in this example, *Measure Pixel Contrast* is used by more than one activity, and *Sharpness Detector* is used by more than one block. See Section 8.10 in Chapter 8 for modeling activity hierarchies on block definition diagrams and Section 6.2 in Chapter 6 for modeling composition hierarchies on block definition diagrams.

This example of the autofocus portion of a surveillance camera will be used throughout the remainder of this chapter. Assume that the surveillance camera will use a passive autofocus system that uses pixel-to-pixel contrast as a way of determining how well the optics are focused, and then it generates a signal to adjust the focus motor accordingly. The *Adjust Focus* activity, then, can be composed of actions defined by three other activities: *a1 : Measure Pixel Contrast*, *a2 : Optimize Focus*, and *a3 : Adjust Focus Motor*. An activity diagram describing the behavior of *Adjust Focus* is presented in Section 13.6.1. Consider, hypothetically, that a separate activity to detect edges of objects in the video frame may also want to use the *Measure Pixel Contrast* activity.

A logical structure for the auto-focus portion of the camera is also provided. The *Focus Controller* block is composed of parts *f1 : Sharpness Detector* and *f2 : Focus Optimizer*. Assume, hypothetically, that the block *Sharpness Detector* may also define a part used by some other logical block whose purpose is to check video quality.

13.6.1 Modeling Functional Allocation of Usage

As discussed in an earlier section, functional allocation of usage (e.g., action to part) should be used over functional allocation of definition (e.g., activity to block) when each action is allocated to parts that are typed by different blocks. Allocation of usage should also be considered if the action uses different inputs/outputs (i.e., pins) that may result in different interfaces on the associated block.

Figure 13.8 depicts functional allocation of usage. This example shows the use of the callout notation for representing allocations from the actions on the activity diagram, and the use of the compartment notation for representing allocation to the parts on the internal block diagram. Note that action *a1 : Measure Pixel Contrast* on the activity diagram is allocated to part *f1 : Sharpness Detector*, but that none of the other actions are allocated. This is because their defining activities are allocated in Section 13.6.2, so it is not appropriate to also allocate the usage. Also, notice that object flow *of2* is allocated to connector *c1*. This kind of flow allocation can only be allocation of usage; it is described in more detail in Section 13.7.2.

On the internal block diagrams, the allocation callouts are the reciprocal of the allocation callouts on the activity diagram. An allocation matrix is also provided as a concise alternative representation of the allocation relationships in the other diagrams.

13.6.2 Modeling Functional Allocation of Definition

Allocation of definition is used when each action is allocated to a part that is typed by the same block and can be depicted on block definition diagrams. The allocation must be to or from activities or blocks.

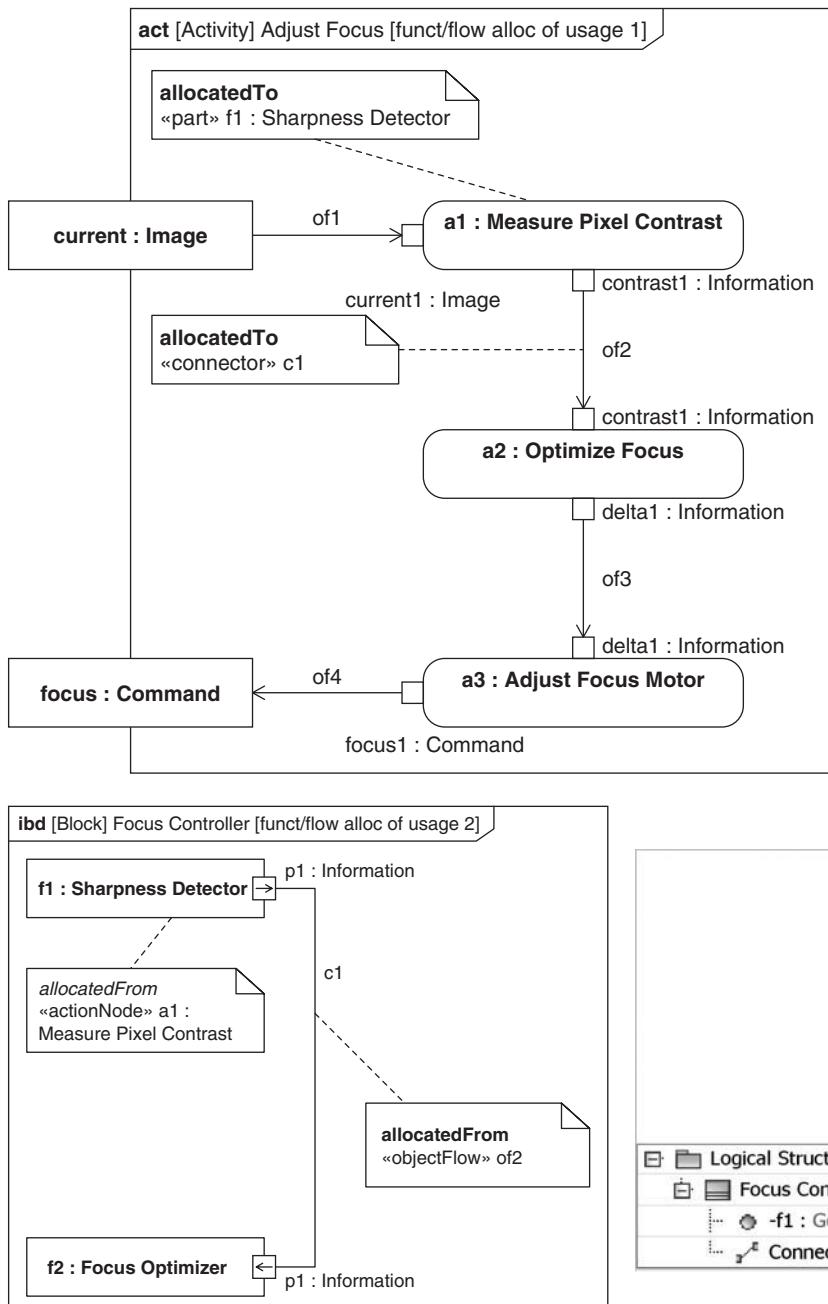
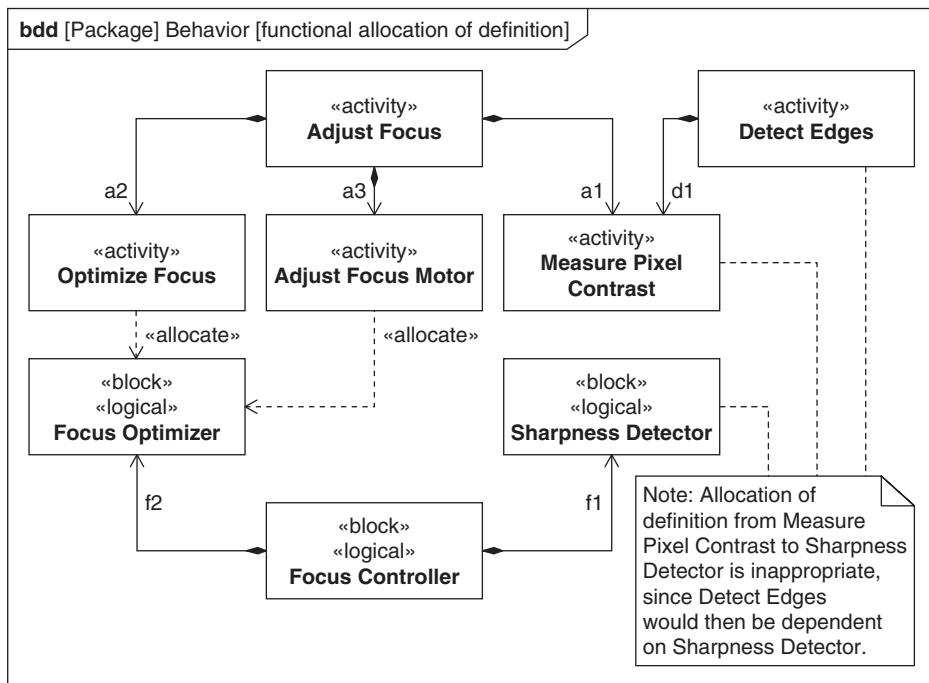


FIGURE 13.8

Example of functional allocation of usage.

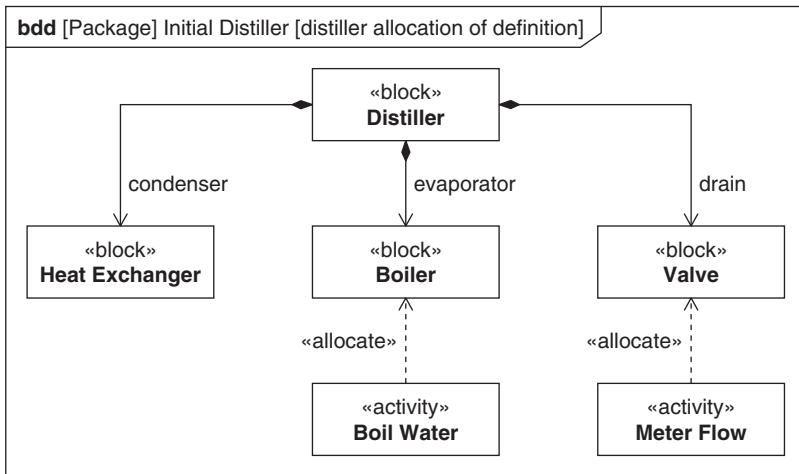
**FIGURE 13.9**

Example of functional allocation of definition.

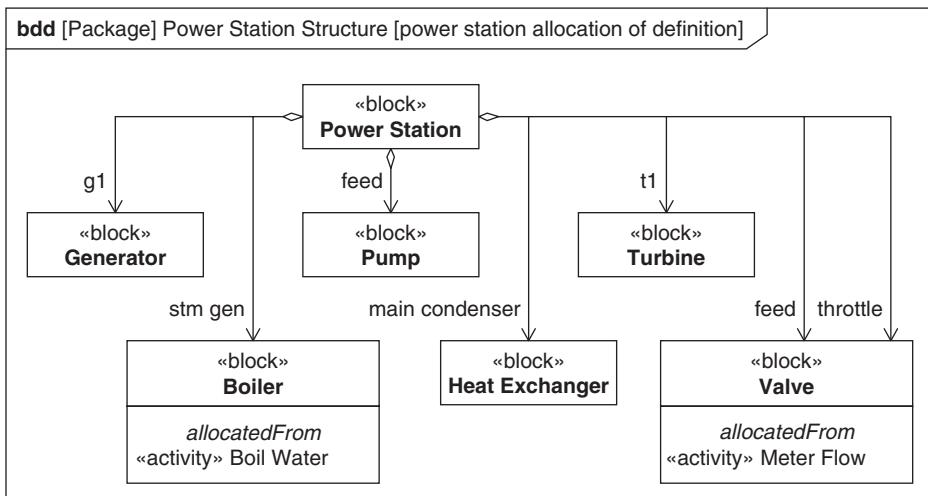
Figure 13.9 shows an example of functional allocation of definition using the allocation relationship, along with the alternative callout and compartment notation. Note that the activities *Optimize Focus* and *Adjust Focus Motor* are allocated to the block *Focus Optimizer*. The use of *Focus Optimizer* in the block *Focus Controller*, and everywhere else it is used, has an inferred allocation of these two activities. This allocation can later be realized by creating two operations for *Focus Optimizer* that would call *Optimize Focus* and *Adjust Focus Motor* as their methods. These new operations would then be available to every instance typed by *Focus Optimizer*.

Note that the activity *Measure Pixel Contrast* is not allocated to the block *Sharpness Detector*, even though from previous discussions there is a conceptual relationship between them. In this particular example, *Measure Pixel Contrast* is also used by the activity *Detect Edges*, which is a processing technique not associated with picture sharpness. *Measure Pixel Contrast* should not have any inferred allocation to *Sharpness Detector* when it is used in *Detect Edges*, thus allocation of definition is inappropriate. Allocation of usage is the correct technique in this case.

Figure 13.10 is a block definition diagram of a system similar to the water distiller example in Chapter 15. Note that the *Meter Flow* activity has been allocated to the block *Valve*, which infers that the *Meter Flow* activity applies to

**FIGURE 13.10**

Functional allocation of definition from distiller example.

**FIGURE 13.11**

Implications of functional allocation of definition as seen in the power station example.

each usage of the *Valve* block. This is appropriate because every valve performs an activity to meter fluid flow.

Note also that the activity *Boil Water* has been allocated to the block *Boiler*. This infers that all the usages of the *Boiler* can perform the activity *Boil Water*.

Figure 13.11 is a block definition diagram representing a *Power Station*, and it uses many of the blocks previously defined for the *Distiller*. The allocation of definition to the *Boiler* and *Valve* referred to in Figure 13.10 is still valid. The part *stm gen : Boiler* has an inferred allocation from the *Boil Water* activity, and

both the *feed* and *throttle* usages of *Valve* include an inferred allocation from the *Meter Flow* activity.

13.6.3 Notational Simplicity: Modeling Functional Allocation Using Allocate Activity Partitions (Swimlanes)

Allocate activity partitions are a special type of activity partitions that are distinguished by the keyword «allocate». The presence of an allocate activity partition on an activity diagram implies an allocate relationship between any action node within the partition and the part represented by the partition (which appears as the name of the partition), as depicted in Figure 13.12. Note that allocate activity partitions can only explicitly depict allocation of usage because the activities (definition) are not directly represented on activity diagrams. If allocation of definition is desired, the activity must be allocated to the block that can be directly depicted on a block definition diagram or by using compartment or callout notation.

Functional allocation using allocate activity partitions (a.k.a. swimlanes) is depicted in Figure 13.13. This is a subset of the example previously shown in Figure 13.8, where action node *a1* (a usage of activity *Measure Pixel Contrast*) has been allocated to part *f1* (a usage of block *Sharpness Detector*). This allocation is depicted graphically by the allocate activity partition on the activity diagram.

Allocate activity partitions are distinguished from other activity partitions by the keyword «allocate». If a standard activity partition is used without that keyword, the allocation implications for the actions are different: If an action is a call behavior action, then the activity called by the action must be a behavior of

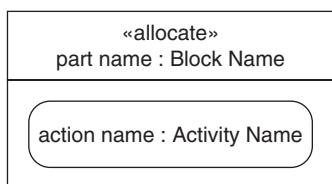


FIGURE 13.12

Allocate activity partition.

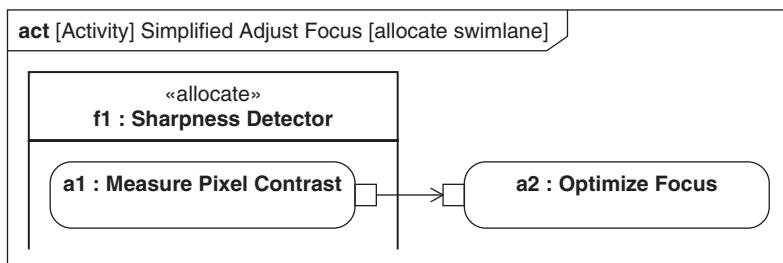


FIGURE 13.13

Simple example of functional allocation using an allocate activity partition (swimlanes).

the block that represents the partition. In particular, a block's operation can be defined in a way that calls this activity as its method, as described in Section 6.5 in Chapter 6. This does not employ the SysML allocate relationship, but instead tightly couples the behavior definition to the structural definition.

13.7 Connecting Functional Flow with Structural Flow Using Functional Flow Allocation

Flow between activities can either be control or object flow as described in Chapter 8. The following sections address allocating object flow as represented on activity diagrams. Allocation of control flow may be depicted in a similar way as allocation of object flow. Flow allocation is typically an allocation of usage because items that flow between model elements are usually specified in the context of their usage.

13.7.1 Options for Functionally Allocating Flow

Item flows are used to depict flow between parts on internal block diagrams, as described in Section 6.4.2 in Chapter 6. Item flows can have an associated item property. The item flow represents the direction of flow, and the item property is the usage of the item that flows. Item properties can be defined (i.e., typed) by blocks just like parts can be typed by blocks.

Chapter 8 discusses the equivalent depiction of object flows (solid arrows on activity diagrams) in either action pin notation (small squares on the edges of action nodes) or object node notation (larger rectangles between action nodes). Due to constraints in the underlying UML metamodel, object nodes on activity diagrams cannot be directly allocated either to item flows or their respective item properties. To avoid ambiguity of the allocation relationship, it is recommended that action pin notation be used when performing behavioral flow allocation.

The following sections discuss allocating an object flow to a connector, allocating an object flow to an item flow, and allocating item properties between diagrams. Other kinds of flow allocation can be used as well, such as allocating an action pin to an item flow or an activity parameter node to a port. These additional allocations are an advanced topic that is a function of the specific design method used and are not discussed here.

13.7.2 Allocating an Object Flow to a Connector

Figure 13.14 extends the example shown in Figure 13.13 and is also a subset of the example shown in Figure 13.8. The object flow *of2* is allocated to the connector *c1*. This is a convenient preliminary form of allocation to use before item flows have been defined, or if item flows are not modeled. It can be ambiguous, however, if more than one item flow or item property is associated with the connector.

Control flows can also be allocated to connectors, but the semantics and physical implications of allocating control flows are also highly dependent on the

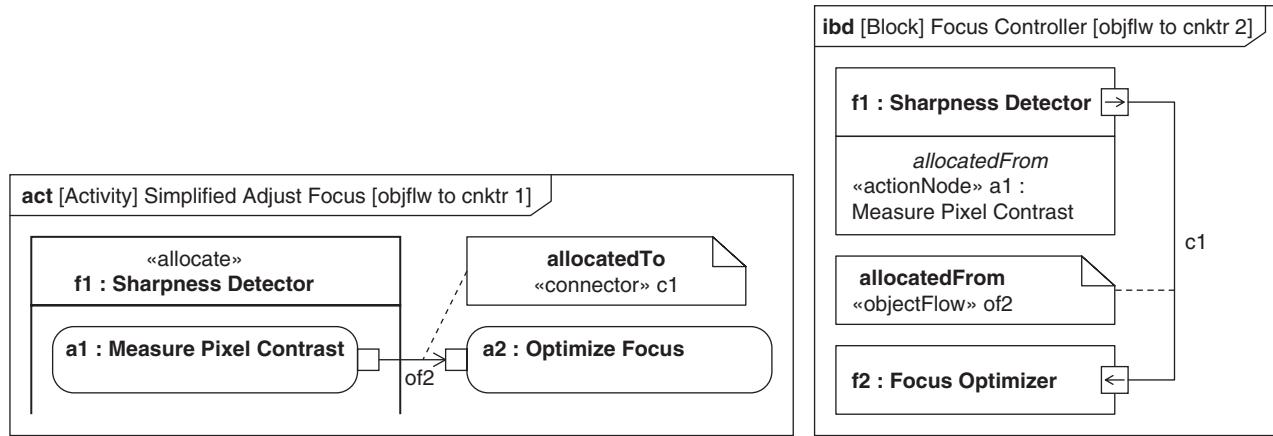


FIGURE 13.14

Object flow to connector allocation.

design method. Additional model refinement may be required before unambiguous control flow allocation can be achieved.

13.7.3 Allocating Object Flow to Item Flow

Figure 13.15 provides an alternative method of flow allocation from Figure 13.14. In this case object flow *of2* has been allocated to the item flow *if1*. While this can be easily depicted on the activity diagram using callout notation, it cannot be unambiguously depicted on an internal block diagram since the name of the item flow may not be visible. An allocation matrix is provided to explicitly show the allocation relationships. This is a more specific form of allocation than object flow to connector, and it will remain unambiguous even if more than one item flow is associated with the connector. In general, activity edges that represent control flow or object flow can be allocated to item flows.

Allocating an object flow or control flow to an item flow does not affect the behavior represented on the activity diagram. If the modeling tool animates or executes the activity diagram, it is the object flow that will be part of that execution semantic, not the item flow.

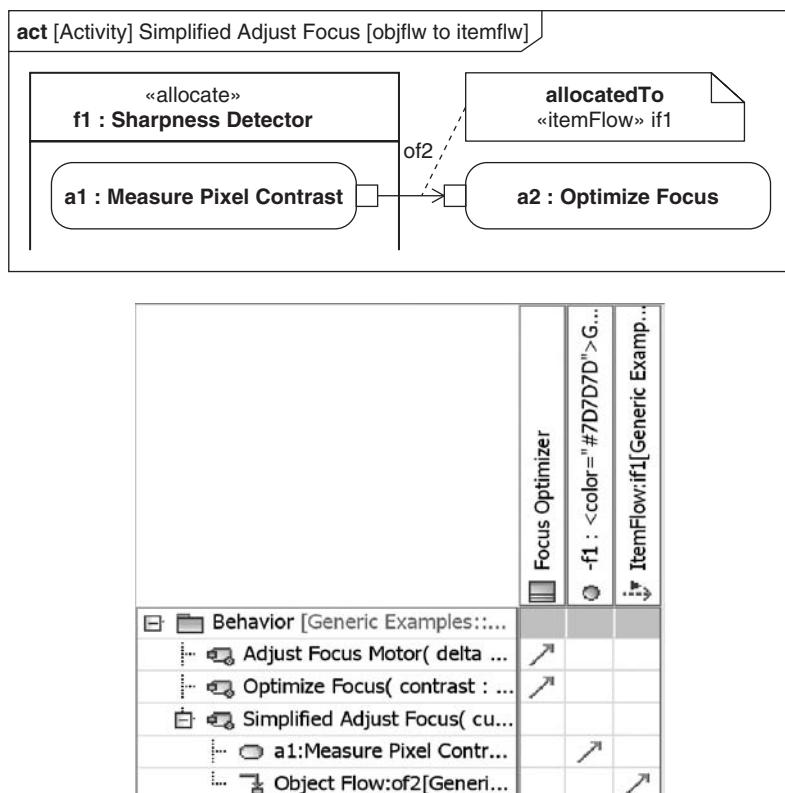


FIGURE 13.15

Object flow to item flow allocation.

When allocating object flows to item flows, it is important to ensure consistent typing. Action pins may be typed by blocks, but item flows are not typed directly. The item properties that are related to the item flows are typed by blocks. The built-in constraints on object flows ensure that the action pins on each end of the object flow are typed by the same (or at least consistent) blocks. When allocating the object flow to an item flow, the type of the action pins associated with the object flow should be consistent with the type of the item properties associated with the item flow. This is an example of what might be expected from a model checker provided by the tool to reduce the likelihood of error as well as the workload of the modeler.

Rather than allocate the object flow to the item flow, it may be appropriate to allocate the object flow to the item property associated with the item flow. Figure 13.16 shows the results of this kind of allocation; it is used in the water distiller example in Chapter 15 because it ties the object flows in the functional model to specific properties of the water flowing through the system. The values of these properties are used for subsequent engineering analysis.

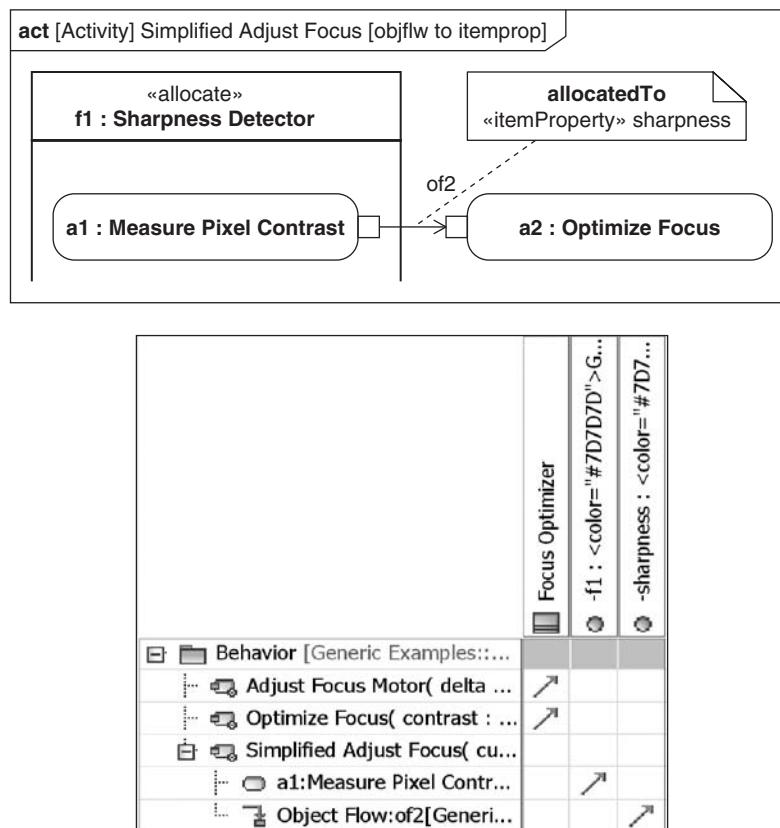


FIGURE 13.16

Object flow to item property allocation.

13.8 Modeling Allocation between Independent Structural Hierarchies

There are times to consider more than one model of structure (e.g., logical-physical). For example, it is a common practice to group capabilities, functions, or operations into an abstract, or **logical** structure, while maintaining a separate implementation-specific **physical** structure. Chapter 16 includes an example of developing a logical architecture.

A particular method for logical architecture development must include a way to relate elements of logical structure with elements of physical structure. SysML allocation provides an abstract vehicle to perform and analyze this mapping. Implementation of the physical structure may require further model development to realize the logical structure, such as inheritance of owned behaviors or flow specifications, but this development should wait until the logical-to-physical allocation is stable and consistent across the system model.

The physical structure may itself be divided into software structures and hardware structures. UML software modelers typically use deployment relationships to map software structures on to hardware structures. SysML allocation provides a more abstract mechanism for this kind of mapping, which does not have to consider host-target environment, compiler, or other more detailed implementation considerations. These considerations may be deferred until after preliminary hardware and software allocation has been performed and analyzed.

13.8.1 Modeling Structural Allocation of Usage

An example of a structural allocation of usage is shown in Figure 13.17 using a block definition diagram. The diagram shows both ends of the structural allocation of the blocks' internal structure. The structure compartment of a block on a block definition diagram corresponds to what is depicted on the internal block diagram of that block.

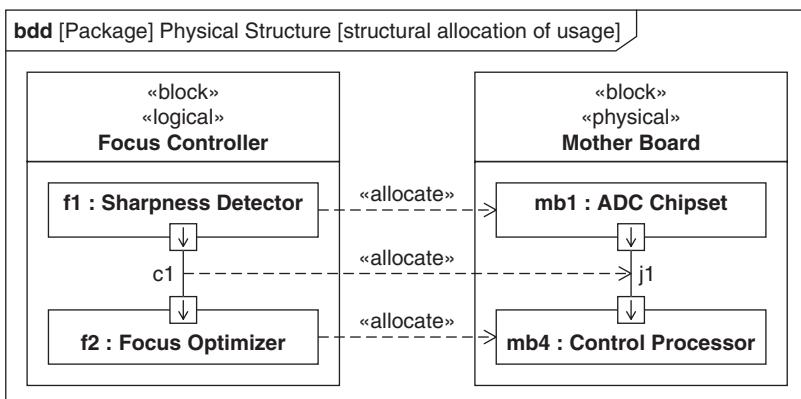


FIGURE 13.17

Structural allocation of usage example.

Allocation between parts in different structure compartments, as shown, can only depict allocation of usage. Likewise, allocation shown between connectors on internal block diagrams or structure compartments can only represent allocation of usage.

13.8.2 Allocating a Logical Connector to a Physical Structure

A connector depicted in an abstract, or logical structure, may need to be allocated to multiple elements in an implementation or physical structure. A connector binds parts or ports together, and it does not necessarily represent an interfacing part, such as a wiring harness or a network.

The example in Figure 13.18 depicts the allocation of a connector in a logical structure, where implementation details are not considered, to a physical part (*ea5 : PWB Backplane*) and associated connectors at the appropriate ends of the cable. The use of allocation is an appropriate way to show the refinement of the

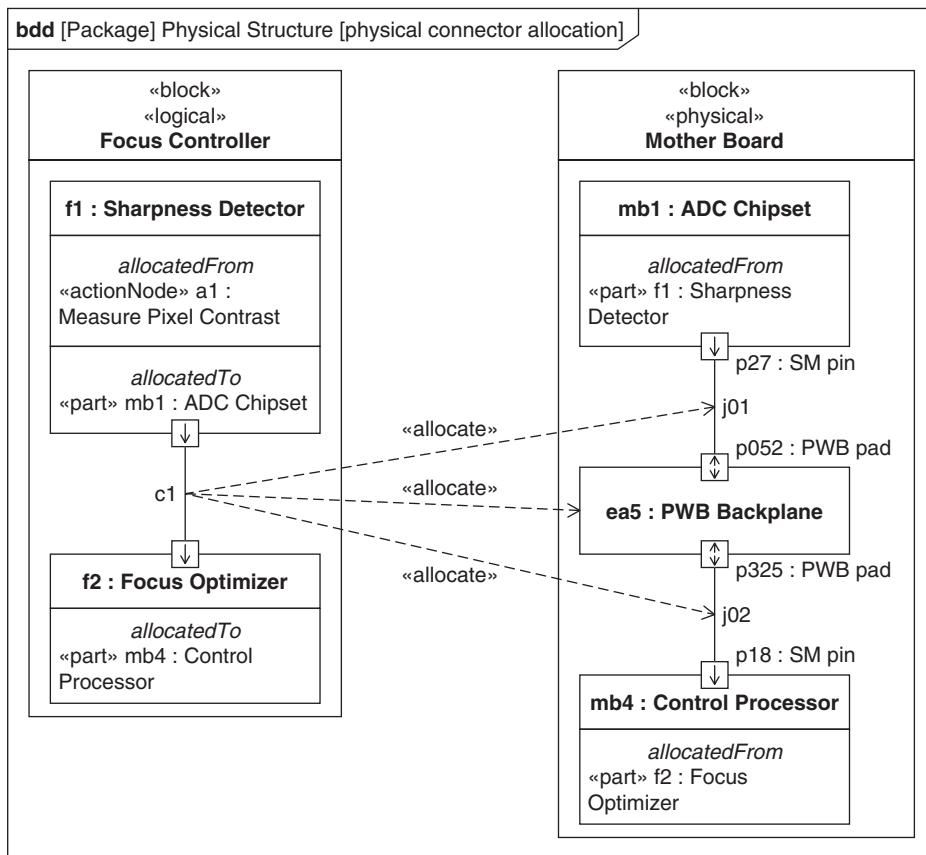


FIGURE 13.18

Refining a connector using allocation.

logical connector, without requiring undue extension of the logical architecture into implementation details. Any item flow on the logical connector should be allocated to multiple item flows in the physical structure, corresponding to flow entering and exiting the cable.

13.8.3 Modeling Structural Allocation of Definition

Figure 13.19 shows structural allocation of definition for the autofocus portion of the surveillance camera. This is different from the allocation represented previously in Figure 13.17, which depicted allocation of usage. If a structural allocation is meant to apply to all its usages, then allocation of definition is appropriate. In this example, wherever the block *Vector Processor* is used, it will include the inferred allocation from *Image Processor*, even if it is not used in an *Mother Board*.

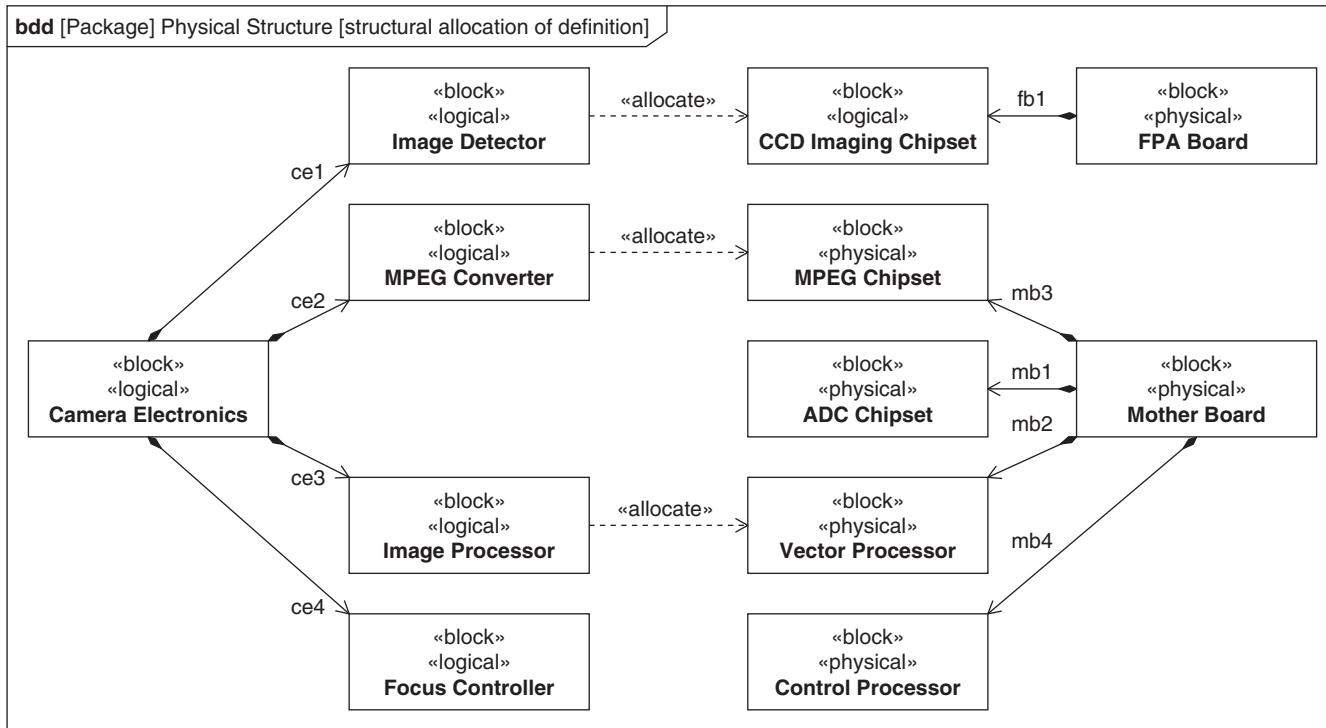
13.9 Modeling Structural Flow Allocation

An item flow can be a common item to both an abstract (e.g., logical) internal block diagram and a concrete (e.g., physical) internal block diagram. This enables a common structural data model to be maintained between logical and physical hierarchies.

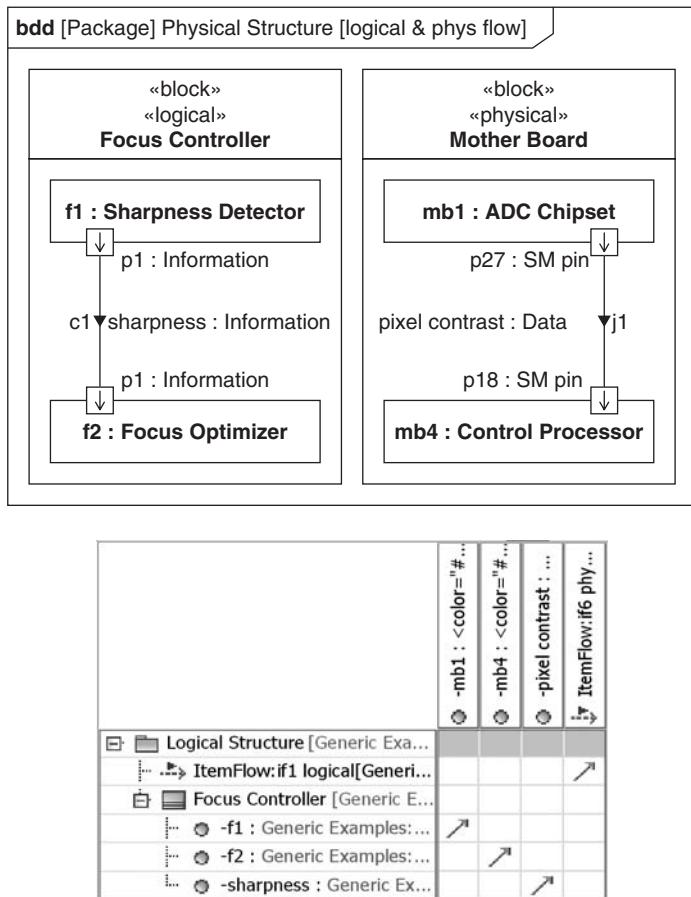
There may be good reasons, however, to establish separate abstract (e.g., logical) and concrete (e.g., physical) data models. For example, a standard logical data model may be required, but the data-level implementation may need to be optimized. In the case in which an item flow depicted at an abstract level needs to be allocated to structures at a more concrete level, it may be necessary to decompose the abstract item flow so that it may be uniquely allocated. If a block is used to represent the item that flows at the abstract level, it should be decomposed into a set of blocks that can be used to represent the items that flow at the more concrete level. The abstract item flow can then be allocated to the more concrete item flows that use the appropriate blocks to type item properties.

Figure 13.20 shows how an item flow at an abstract level can be allocated to an item flow at a more concrete level. The name of the item flow in *Focus Controller* is *if1*, but this name doesn't appear next to the item flow symbol. This is because *if1* has an item property, *sharpness*, that is typed by a block, *Information*, and this name supersedes the name of the item flow when represented on the internal block diagram. Likewise, the name of item flow *if6* in *Electronics Assembly* has been superseded by item property *pixel contrast* and its type called *Data*. Because of this naming convention, the allocation of *if1* to *if6* is not directly shown on the diagram and must be represented in an allocation matrix.

It is possible to allocate from an item property on one diagram directly to an item property on another diagram, such as from *sharpness : Information* to *pixel contrast : Data*. Allocation between item properties cannot be directly represented on any diagram because it would look like allocation between item flows. Allocation between item properties is best represented on an allocation matrix.

**FIGURE 13.19**

Depicting structural allocation of definition.

**FIGURE 13.20**

Example of structural flow allocation.

In most cases of allocation between item properties, the defining type (conveyed classifier) will be the same for both item properties. In the Figure 13.20 example, note that the logical data model is independent of the physical data model, and thus the types (conveyed classifiers) of each item property are different.

It is certainly possible to allocate the logical conveyed classifier to the physical. This would be allocation of definition, and thus should be done paying special attention to avoiding unwanted inferred allocation.

13.10 Evaluating Allocation across a User Model

An assessment of the integrity and completeness of the allocation relationships is largely dependent on the system's stage of development. Since allocation is used

as an abstract prelude to more concrete relationships, the quality of allocation at a given point in time can only be assessed with respect to the system development method or strategy being employed.

13.10.1 Establishing Balance and Consistency

The quality of the model can be assessed in terms of the completeness and consistency of the allocation relationships and the overall balance of the allocation as described next.

Completeness and consistency are evaluated using rules or constraints. In functional allocation, for example, allocation of a package of activities is said to be **complete** when each activity has an allocation relationship to a block elsewhere in the model. It may not be judged to be **consistent**, for example, until the action nodes defined by the activities are depicted in a valid activity diagram; the inferred allocation to parts are depicted on a valid internal block diagram; and any object flows on the activity diagram are allocated to appropriate connectors on the internal block diagram. Consistency can also involve checking for circular allocations, redundant allocations, and what the modeler may define as inappropriate allocations (e.g., allocating an activity to another activity). Again, automated model checking is expected to assist with this.

Evaluating **balance** is more subjective and likely to require experience and judgment on the part of the modeler. One aspect of balance may involve assessing the level of detail represented by the model element at each end of the allocation relationship. It may be inappropriate for the element at the “from” end to be more detailed than the element at the “to” end if one is allocating from a more abstract model part to a more concrete model part. A similar aspect of balance might involve examining portions of the model that are rich in allocation, and determining whether the level of detail is too high, or whether the allocation-poor portions of the model need further refinement. When evaluating functional allocation, for example, if a large number of activities are allocated to a single block and other blocks have few or no activities allocated, the modeler may ask: (1) Have the activities of the system been completely modeled? or (2) Has the structural design incorporated too much functionality into a single block? The answers to these questions will help determine the direction for the future modeling effort. For question 1, it might be fleshing out the activity model in other areas; for question 2, it might be decomposing the overallocated block into lower-level blocks.

13.11 Taking Allocation to the Next Step

Allocation is a means, not an end point. Once allocation across the model is balanced and complete, each allocation may be refined by a more formal relationship that preserves and elaborates the constraints from the “from” end to the “to” end of the allocation. In this way, allocation is used to direct the system design

activity through the model without prematurely deciding how the relationship between model elements will be refined. Of course, this is very dependent on the modeling method.

SysML allocations allow the modeler to keep model refinement options open. For example, functional allocations can be refined by: (1) Designating activities allocated to a block as methods called by operations of the block; this, of course, requires the additional step of creating the operations. (2) Designating the activities as owned behaviors of the block. Each approach has merits for downstream development; deferring the decision allows the modeler to work at a consistent level of abstraction, and not to get prematurely drawn into modeling details or methodological trade-offs.

Even after the model is refined, it is appropriate to retain the allocation relationships, possibly capturing supporting «rationale» in the model to provide a history of how the model was developed. This can be very important information when considering reuse of the model on a different program or product.

13.12 Summary

The allocation relationship provides significant flexibility for relating model elements to one another beginning early in the development process. Key concepts for modeling allocations include the following.

- How allocation can be used to support system modeling is discussed in this chapter, including examples. Also included is a brief discussion of how to assess allocations in terms of their completeness, consistency, balance, and flexibility in directing further model development efforts.
- A system model can be developed without using allocation. Use of allocation, however, enables certain implementation decisions to be deferred by specifying the model at higher levels of abstraction and then using allocations as a basis for further model refinement.
- There are many different types of allocation, including allocation of behavior, structure, and properties. Allocation supports traditional systems engineering concepts, such as allocating behavior to structure by allocating activities to blocks. Also supported is allocation of logical elements to physical elements, including logical connectors to physical interfaces, software to hardware, object flows to item flows, and many others.
- A key distinction must be made between the allocation of definition and the allocation of usage. In the former, defined elements (e.g., activities) are allocated to other defined elements (e.g., blocks). For allocation of definition, all usages of the activity are allocated to all usages of the block. For allocation of usage, only specific usages are allocated without impacting other usages, such as the case when an action is allocated to a part.
- An allocate activity partition provides an explicit graphical mechanism to allocate responsibility of an action to a part.

- There are multiple graphical and tabular representations for representing allocations similar to those used for representing requirements relationships. Graphical representations include direct notation, compartment notation, and callout notation. Tabular representations often include a compact form for representing multiple allocation relationships.

13.13 Questions

1. List four ways that allocations can be represented on SysML diagrams.
2. Which kinds of model elements can participate in an allocation relationship in SysML?
3. Is the allocate relationship appropriate to use when allocating requirements?
4. List and describe three uses of allocation in SysML that rely on the allocate relationship.
5. For each of the following kinds of diagrams, indicate whether they are diagrams of usage or diagrams of definition:
 - a. activity diagram
 - b. block definition diagram
 - c. internal block diagram
 - d. parametric diagram
6. For each of the following allocation relationships, indicate whether they are allocation of definition or allocation of usage:
 - a. action node (on activity diagram) to part (on internal block diagram)
 - b. activity to block
 - c. object flow to connector
 - d. activity parameter to flow specification
7. What is the significance of choosing an allocation of definition instead of an allocation of usage?
8. Should an object flow ever be allocated to a block? Explain your answer.
9. Should an activity ever be allocated to a part? A connector to a block? Explain your answers.
10. The following questions apply to Figure 13.20:
 - a Are the item flow names shown on the block definition diagram? Explain.
 - b Why is there no direct allocation shown on the block definition diagram?

Discussion Topics

What is the purpose of allocation? What role does it play in system development? How can good or poor allocation impact the overall quality of the system design?

Describe an appropriate next step after completing functional allocation. Which mechanisms are available to implement functionality in blocks?

Customizing SysML for Specific Domains

14

This chapter describes how to customize SysML using profiles and model libraries. These types of customization support the wide range of domains that systems modeling can be applied to. A number of advanced metamodeling concepts that are typically of interest to language designers and others who may be responsible for customizing the language to meet domain-specific needs are also addressed. Some of the metamodeling concepts were introduced in Chapter 4.

14.1 Overview

SysML is a general-purpose systems modeling language that is intended to support a wide range of domain-specific applications such as the modeling of automotive or aerospace systems. SysML has been designed to enable extensions that support these specialized domains. An example may be a customization of SysML for the automotive domain that includes specific automotive concepts and representations of standard domain elements such as engines, chassis, and brakes.

To accomplish this, SysML includes its own extension mechanisms, called stereotypes, that are grouped into special kinds of packages called profiles. Stereotypes extend existing SysML language concepts with additional properties and constraints. SysML also supports model libraries—collections of reusable model elements commonly used in a particular domain. Profiles and model libraries are themselves contained in models, but they typically are authored by language designers rather than the general system modeler. The term “user model” refers to a model authored by a system modeler to describe a system or systems.

Model libraries provide constructs that can be used to describe real-world instances represented by a model, be they blocks specifying reusable components or value types defining valid units and dimensions for block properties. Profiles, on the other hand, provide constructs that extend the modeling language itself; for example, stating that there is such a thing as a value type with units and dimensions in the first place.

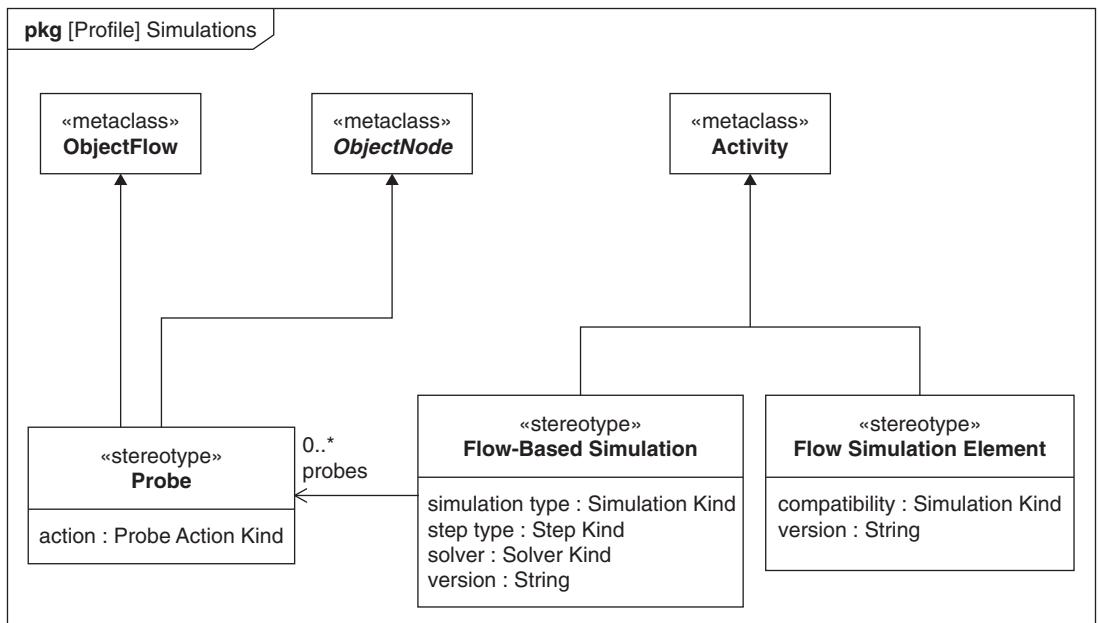


FIGURE 14.1

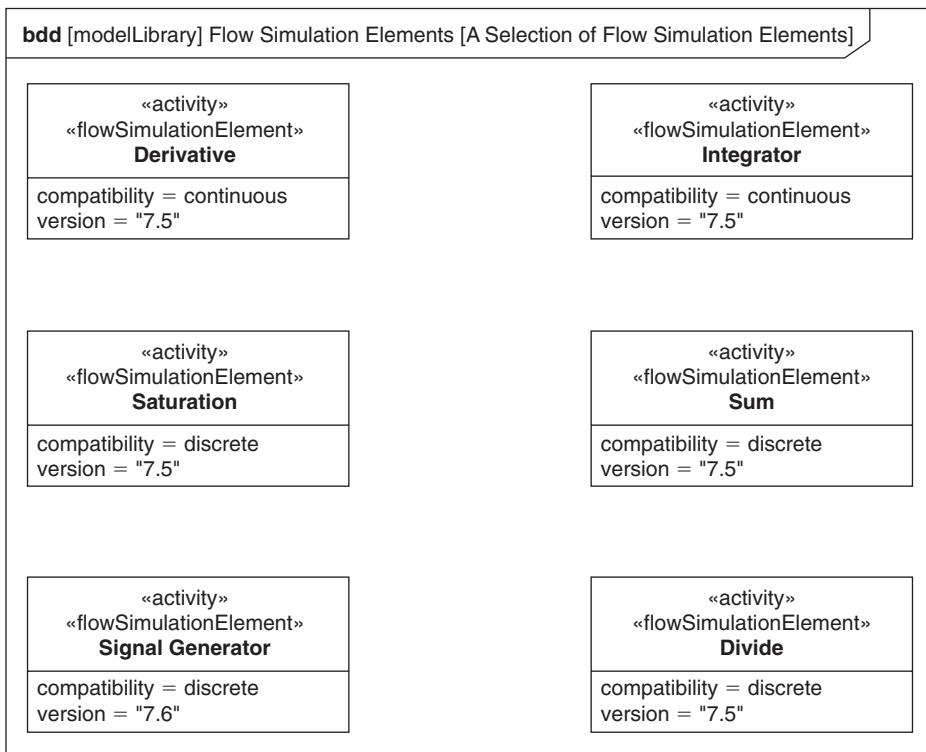
Example of a profile defined on a package diagram.

Profiles and model libraries are represented on package diagrams, as described in Chapter 5, with additional notations described in this chapter. Figure 14.1 shows a package diagram with much of the notation used for defining stereotypes.

The diagram in Figure 14.1 shows the definitions of three stereotypes and their properties to support simulations. *Flow-Based Simulation* and *Flow Simulation Element* both extend the SysML *Activity* metaclass and add information about the type of simulation and how it executes. *Probe* extends both the *ObjectFlow* and *ObjectNode* metaclasses—part of the activity specification—and is used to tell the simulation system which data to monitor.

Table A.2 in the Appendix shows the additional notation needed to represent the extensions to the package diagram for model libraries and profiles.

Figure 14.2 shows a model library of elements that are themselves extended using the stereotypes shown in Figure 14.1. The model elements in the *Flow Simulation Elements* model library are intended for use in building flow-based simulations. They are activities (i.e., model elements whose type is the metaclass *Activity* shown in Figure 14.1) with the stereotype *Flow Simulation Element* applied. Note that when stereotypes are applied, the keyword for a stereotype by convention has a different typographic style than the style of the stereotype's name. This convention is described later in this chapter. These activities can be invoked from actions owned by a flow-based simulation. The values for the stereotype's properties allow the simulation tool to determine their validity based on the type of simulation required.

**FIGURE 14.2**

Example of the application of stereotypes to model elements.

Table A.24 in the Appendix shows the additional notation needed on SysML diagrams to represent model elements that have been extended by stereotypes.

14.1.1 A Brief Review of Metamodeling Concepts

Although the topic of metamodeling is discussed in Chapter 4, the main concepts are reprized here for convenience. A modeling language has three parts:

- *Abstract syntax* describes the concepts in the language, the relationships between the concepts, and a set of rules about how the concepts can be put together. The abstract syntax for a modeling language is described using a **metamodel**. SysML is based on OMG standards for both modeling and metamodeling. The OMG defines a metamodeling mechanism, called the Meta Object Facility (MOF) [20], that is used to define metamodels such as UML and SysML.
- *Notation* describes how the concepts in the language are visualized. In the case of SysML, the notation is described in notation tables that map language concepts to graphical symbols on diagrams.

- *Semantics* describe the meaning of the concepts by mapping them to concepts in the domain of the language—for example, systems engineering. Sometimes the semantics are defined using formal techniques, such as mathematics, but in SysML the semantics are described in English text. Efforts are under way to provide a more formal definition of SysML semantics and many tools implicitly define the semantics by building simulators.

The individual concepts in the metamodel are described by **metaclasses** that are related to each other using generalizations and associations in a similar fashion to the way blocks can be related to one another on a block definition diagram. Each metaclass has a description and a set of properties that characterize the concept it represents, and also a set of constraints that impose rules on those properties' values.

The package diagram in Figure 14.3 shows a small fragment of *UML4SysML*—the metamodel on which SysML is based. It shows one of the fundamental concepts of UML, called *Class*, and some of its most important relationships. *Class* specializes *Classifier* through which it gains the capability of forming classification hierarchies. The figure also shows associations to *Property* and *Operation*, which between them define most of the important features of a *Class*.

A model of a system contains model elements that are instances of the metaclasses in the metamodel for the language. These instances have values and references to other instances based on the properties and relationships defined in

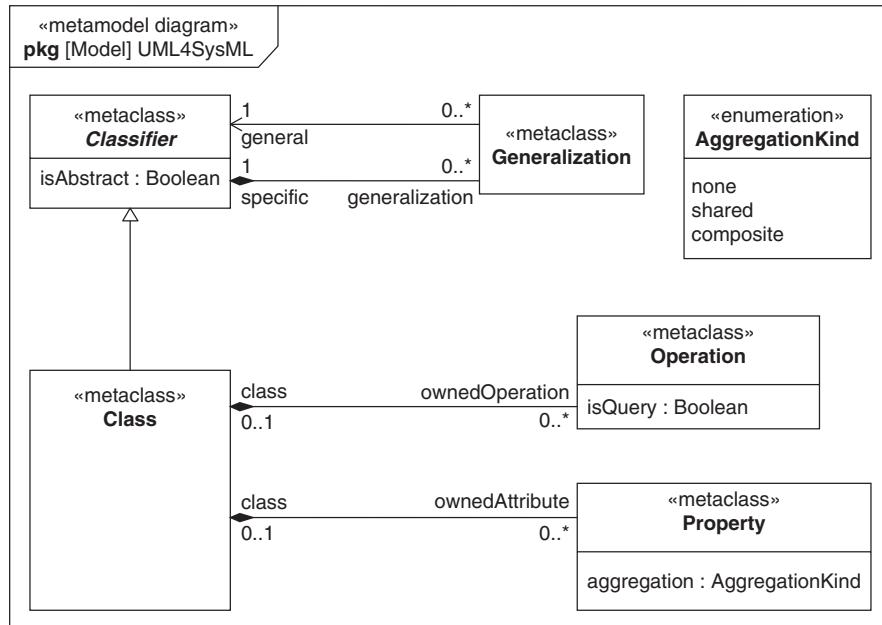


FIGURE 14.3

Fragment of *UML4SysML*, the underlying metamodel for SysML.

the metamodel. Some of these model elements just capture details of the model's internal structure, such as how the model elements are organized into packages (the equivalent of folders in Windows). However, the majority of the elements in a model describe entities in the world of the system.

The two SysML concepts presented in this chapter, model libraries and profiles, are used to add new capabilities to the modeling language. Model libraries contain normal model elements, described by metaclasses in the metamodel. Profiles extend a metamodel, called a reference metamodel, with additional concepts that have their own properties, rules, and relationships; thus, they allow the language defined by the metamodel (in this case SysML) to be augmented with concepts for domains not covered directly by SysML.

Modeling tools are normally specially engineered to support a specific metamodel and will only understand models that use that metamodel. Extending the language by adding to the metamodel is something typically done by a tool vendor. The benefit of a profile is that many UML tools are engineered to support not just the core metamodel but also any user-defined profiles. This means that a profile for a specific domain can be loaded into a UML tool and the tool will understand how to edit, display, load, and store elements of that profile without the need for a tool extension. So, a modeler can make use of a set of modeling elements for a specialized modeling domain without needing to change the modeling tool.

As discussed in Chapter 4, SysML is based very closely on a subset of the concepts in UML, so it is defined as a UML profile. This allows UML tools to support SysML simply by loading the SysML profile, although many UML tool vendors have extended their UML tools to make the SysML profile more usable.

The rest of this chapter discusses model libraries and profiles in detail. Section 14.2 describes model libraries and their use in defining reusable components. Sections 14.3 and 14.4 cover the definition of stereotypes and the use of profiles to describe a set of stereotypes and supporting definitions. Sections 14.5 and 14.6 focus on the use of profiles and model libraries to build domain-specific user models.

14.2 Defining Model Libraries to Provide Reusable Constructs

A **model library** is a special type of package that is intended to contain a set of reusable model elements for a given domain. Model libraries are not used to extend the language concepts of SysML, although model elements in the library may have stereotypes applied if they support a specialized domain, as shown in Figure 14.2. Model libraries can contain very specialized elements similar to parts catalogs containing the specifications of off-the-shelf components, or they can contain elements with wider applicability, such as the *SI Definitions* model library provided in the SysML specification.

Any packageable model element (see Chapter 5), such as blocks, value types, activities, and constraint blocks, can be included in a model library. Elements in

a model library may be contained directly in that library, or they may have been defined in other models or packages and imported. In the latter case, the model library acts as a mechanism to gather elements from disparate sources into a convenient unit for reuse.

The contents of a model library may be shown on a package diagram or block definition diagram using the standard symbols for those diagrams. When a model library is shown on a package diagram, it is designated by a package symbol with the keyword «modelLibrary» appearing before the name of the model library in the name compartment or tab of the package. See Figure 14.9 in Section 14.5 for an example of the former notation. When a model library corresponds to the frame of a diagram, the type *modelLibrary* is shown in square brackets in the diagram header as the model element type.

The model library in Figure 14.4 defines a set of blocks to represent some very basic physical concepts intended to be specialized by domain-specific blocks. *Physical Thing* describes things with *mass* and *density* and provides a constraint, via the constraint block *Mass Equation*, that defines the mass of a physical entity in terms of the mass of its *components*. The block *Moving Thing* specializes *Physical Thing* with properties of motion (e.g., *acceleration* and *velocity*). It also has a property, *force*, that allows force to be applied to get a *Moving Thing* moving, or to stop it. Instead of a set of equations, the properties of *Moving Thing* are calculated using a simulation, as shown later in Figure 14.11.

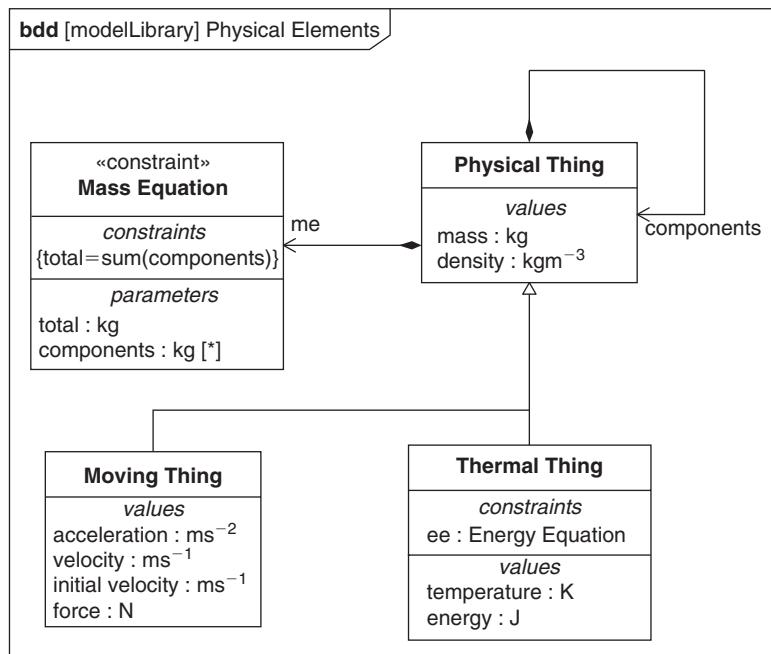


FIGURE 14.4

A model library defining some basic physical concepts.

14.3 Defining Stereotypes to Extend Existing SysML Concepts

Whereas the elements of model libraries use existing language concepts to describe reusable constructs, **stereotypes** add new language concepts, typically in support of a specific systems engineering domain. Stereotypes are grouped together in special packages called profiles. SysML itself is defined as a profile of UML and uses stereotypes to define systems engineering concepts such as block and requirement. Just as models can contain instances of metaclasses, they can also contain instances of stereotypes, although instances of stereotypes have special rules along with different conventions for how they are displayed.

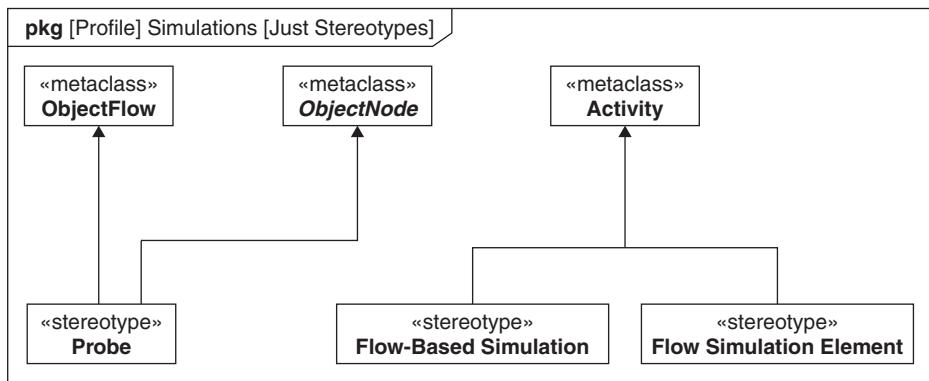
A stereotype is based on one or more metaclasses in a reference metamodel. In the case of SysML this is a subset of UML called UML4SysML. (See Chapter 4 for a description of metamodeling and in particular UML4SysML.) The relationship between the metaclass and the stereotype is called an **extension**, which is a kind of association that is conceptually closer to a generalization. The choice of the base metaclass or metaclasses for a stereotype depends on the kind of concepts that need to be described. A language designer will look for a metaclass with some of the characteristics needed to represent the new concept and then add others and, if necessary, remove characteristics that are not required.

Metamodels, including UML4SysML, contain abstract metaclasses that cannot be instantiated directly in the user model, but exist to provide a set of common characteristics that are specialized by concrete metaclasses instantiated in the user model. This is a powerful reuse mechanism that is widely used by metamodelers. A stereotype may extend an abstract metaclass, in which case it is equivalent to the stereotype extending all the concrete specializations of that metaclass.

Profiles are specified using an extension to package diagrams that allows them to show stereotypes, metaclasses, and their interrelationships. A metaclass is represented by a rectangle with the keyword «metaclass» centered at the top, followed by the name of the metaclass. A stereotype is represented by a rectangle with the keyword «stereotype» centered at the top, followed by the name of the stereotype. An extension relationship is depicted as a line with a filled triangle at the metaclass end.

Figure 14.5 shows a set of stereotypes that describe new concepts for representing flow-based simulation artifacts. The stereotype *Flow-Based Simulation* allows modelers to define simulations of system flow. *Flow-Based Simulation* extends *Activity* because activities already have a flow-based semantic and so have many of the right characteristics. The stereotype *Flow Simulation Element* is used to model a specialized form of activity that can be added to a flow-based simulation.

A very useful capability of simulations is to monitor the values of certain elements as the simulation runs. The *Probe* stereotype allows the modeler to designate that certain elements of the simulation should be monitored. *Probe* extends both *ObjectFlow* and *ObjectNode* because these are both constructs through which values (as tokens) flow. *Probe* extends *ObjectNode*, which is an abstract metaclass as indicated by the use of italic font for its name. This means that all the concrete

**FIGURE 14.5**

A package diagram containing stereotypes that support flow-based simulations.

subclasses of *ObjectNode* (e.g., *DataStoreNode* and *ActivityParameterNode* among many others) are implicitly extended as well. Note that this is an example of how extension and generalization differ. *Probe* is not a specialization of both *ObjectFlow* and *ObjectNode*; rather an instance of *Probe* may extend an instance of *ObjectFlow*, or an instance of *ObjectNode* (or concrete subclass thereof), but not both.

A stereotype can be defined by specializing an existing stereotype, or stereotypes, using the generalization mechanism described in Chapter 6. In this case the new stereotype inherits all the characteristics of the stereotypes it specializes, including extensions. The new stereotype can then add more characteristics, including new extensions, which are relevant to the new concept. Stereotypes may be abstract, which means they cannot be used directly in a user model, but can be specialized and their characteristics inherited. Stereotype specialization is shown using the standard generalization notation—a line with a hollow triangle at the general end.

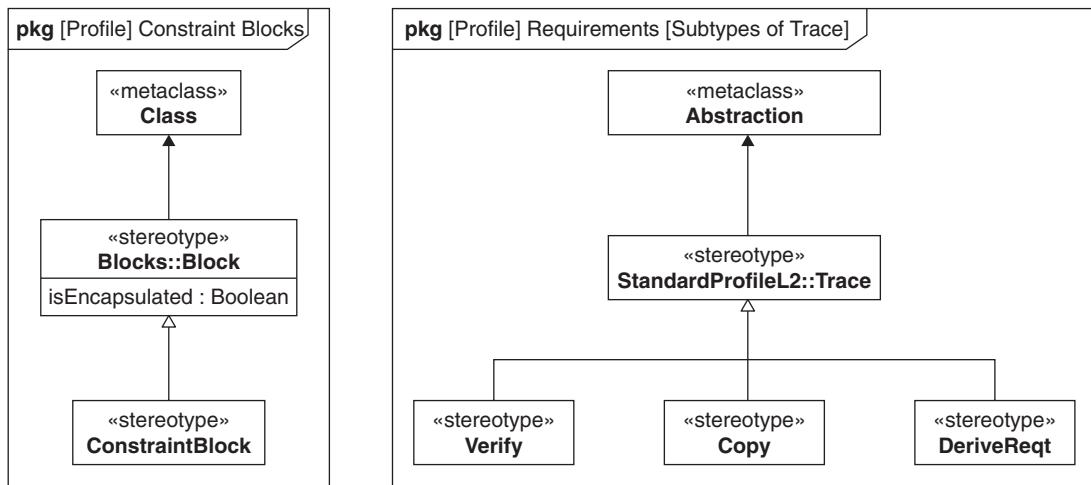
Figure 14.6 shows an example from SysML. *Block* extends the UML metaclass *Class* and *ConstraintBlock* specializes *Block*. It inherits the property *isEncapsulated*, which indicates whether a connector can cross its boundary, from *Block*. Here is a snippet of the description for *ConstraintBlock* in the SysML specification:

“A constraint block is a block that packages the statement of a constraint so it may be applied in a reusable way to constrain properties of other blocks.”

SysML also borrows a stereotype from *StandardProfileL2* of UML, called *Trace*, and specializes it to represent relationships in the *Requirements* profile.

14.3.1 Adding Properties and Constraints to Stereotypes

Sometimes stereotypes are defined to add a concept that is significant in terms of some domain but does not have any additional characteristics. For more sophisticated definitions of a new concept, the stereotype mechanism includes the ability to add both properties and constraints to the stereotype definition. Stereotypes

**FIGURE 14.6**

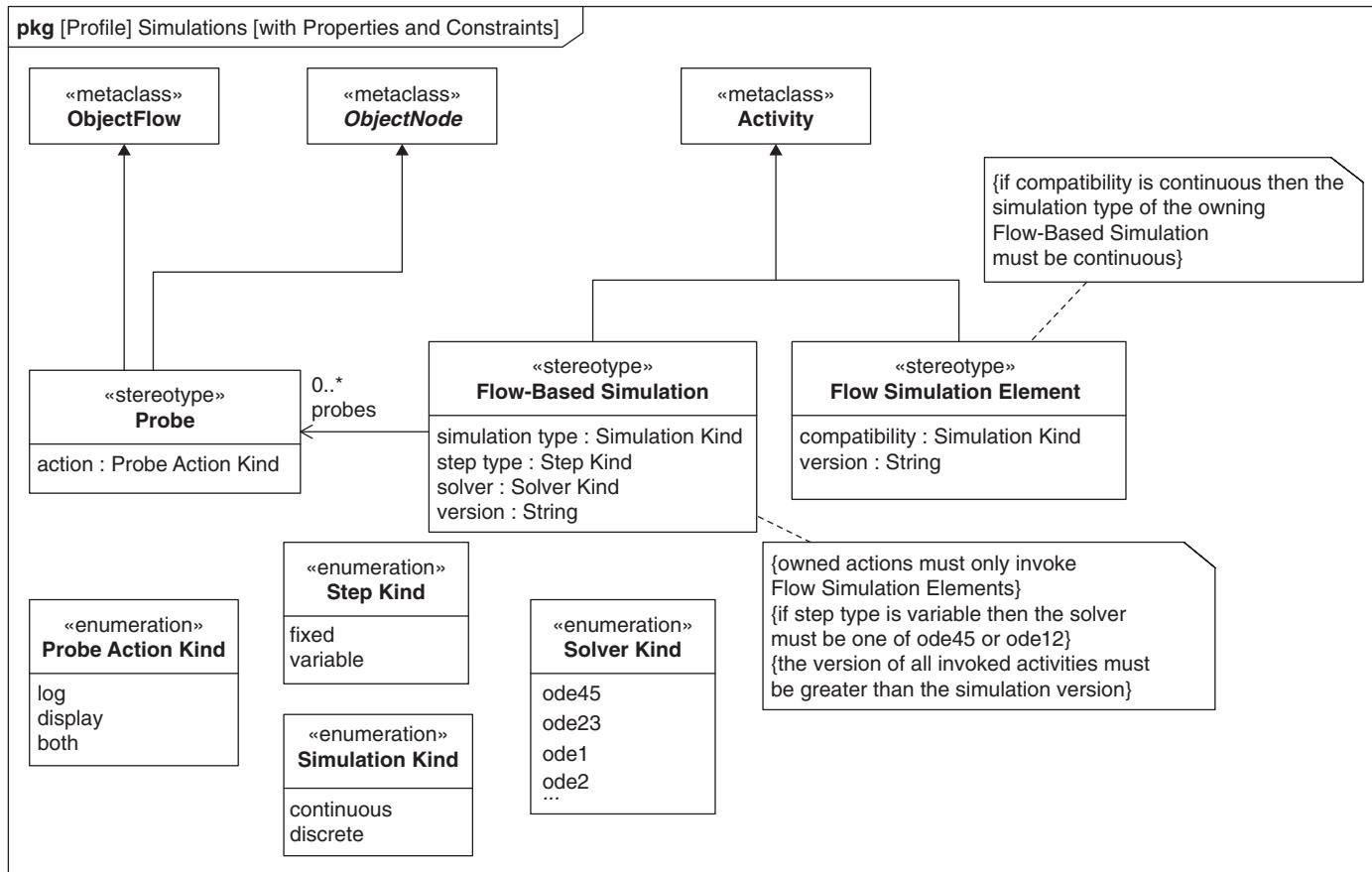
Specialization example from SysML.

that specialize other stereotypes will inherit the properties and constraints of their general stereotype. Stereotype properties represent information about the stereotype and have a type just like any other property. SysML defines a set of basic types—String, Integer, Boolean, and Real—but profiles can add their own types, or use types defined in model libraries. Constraints can be added to the stereotype to specify rules about valid use of new properties or to restrict the capabilities of an existing concept by further constraining the extended metaclasses' properties. Constraints are specified using a textual expression in a specified language. The language OCL is often used for expressing constraints in profiles.

A stereotype may also define properties that are typed by either stereotypes or metaclasses. This allows instances of the stereotype to contain references in the user model to instances of other stereotypes and metaclasses. These properties can be defined in the metamodel using associations or simply as attributes of the stereotype definition. Metaclasses in the reference metamodel cannot be modified; so, any association between a stereotype and metaclass can only define properties on the stereotype, not on the metaclass.

Stereotype properties and constraints are shown in a similar way to the properties and constraints of blocks. Properties and constraints are shown in compartments below the name compartment. Constraints can also be shown in notes attached to the constrained stereotype. In addition to properties and constraints, a stereotype definition may contain an image that can optionally be displayed when the stereotype is applied to a model element.

Figure 14.7 shows the properties and constraints of the stereotypes first shown in Figure 14.5, and also some enumerations that are needed to define some of those properties. The definition of *Flow-Based Simulation* includes three properties that govern the type of simulation performed. *Simulation type* is typed by an enumeration, *Simulation Kind*, that has two values, *discrete* and

**FIGURE 14.7**

Providing additional detail for the flow-based simulation stereotypes.

continuous, stating whether a continuous or discrete solution is required. *Step type* says whether the simulation steps are fixed in size or can vary. *Solver* defines the type of solver to be used. The definition of *Flow Simulation Element* includes a property called *compatibility*, which says what types of simulation it is compatible with. A value of *continuous* means that this element can be used only in continuous simulations; a value of *discrete* means it can be used in both.

These stereotypes also define constraints that affect activities with the various stereotypes applied. A constraint on *Flow Simulation Element* states that an element whose *compatibility* property has the value *continuous* can be used only if the *simulation type* of their owning activity has the value *continuous*. Another constraint states that a *Flow Simulation Element* may be invoked only by an action contained in a *Flow-Based Simulation*. A constraint on *Flow-Based Simulation* states that a variable step solver (*ode45* or *ode23*) must be used if the value for *step type* is *variable*.

Probe has a property *action* that indicates the action to take place for values on the monitored element. Its type, *Probe Action Kind*, has three values: *display* means display values in a simulation window; *log* means log these values to a log file; *both* means do both. A *Flow-Based Simulation* has a property *probes* that references all the probes defined within it, as indicated by the association between *Flow-Based Simulation* and *Probe*.

As stated earlier, for practical reasons of tool implementation, stereotypes are not metaclasses, but rather define additional elements that are created along with instances of metaclasses. However, some stereotypes act more like metaclasses and others act more like ancillary constructs. The two cases can be understood intuitively by considering whether the modeler will think in terms of creating an instance of the stereotype in the user model rather than an instance of the metaclass, or whether he or she will think more in terms of adding an instance of the stereotype to an existing metaclass instance.

For example, a modeler probably intends to create a *Flow-Based Simulation* (see Figure 14.7) rather than create an *Activity*, and then apply the *Flow-Based Simulation* stereotype to it. Quite apart from the previously stated intuitive understanding of the situation, a *Flow-Based Simulation* has constraints placed on it that an arbitrarily selected activity is unlikely to satisfy. On the other hand, the stereotype *Audited Item* in Figure 14.13 is an example of the other intuitive use of stereotypes as providers of ancillary information. *Audited Item* adds auditing information to a model element and is only needed once auditing of the element has begun. It is therefore natural in this scenario to imagine creating an instance of *Classifier* (like a block) and only applying *Audited Item* at some later date.

In a user model, a stereotype can be applied to any model element that has the same metaclass that the stereotype extends. Typically, it is the modeler who dictates whether a stereotype is used or not, but occasionally the profile designer may wish to enforce that every model element of a particular metaclass must have a specific stereotype applied. The extension is then said to be required. Required extensions can be useful when the use of the model depends on all model elements of a certain metaclass having some special characteristics. If the stereotype is required, then the property keyword {required} is shown near the stereotype end of the extension. Figure 14.13 in Section 14.6.1 shows an example

of a required extension that adds configuration data, perhaps in conjunction with some configuration management tool, to all model elements of metaclasses that are deemed worthy of configuration control.

14.4 Extending the SysML Language Using Profiles

A **profile** is a kind of package used as the container for a set of stereotypes and supporting definitions. Typically a profile will contain a set of stereotypes that represent a cohesive set of concepts for a given modeling domain. More complex profiles often contain subprofiles that further subdivide the overall domain into subsets of related domain concepts.

Profiles typically serve one of two potential uses: Either the profile defines a set of concepts that support a new domain, or it defines a set of concepts that add new information to a model in a domain that is already supported. It is often useful to bear this distinction in mind when creating a profile.

The former use is sometimes called a domain-specific language and offers a new set of language concepts that a modeler might use when building a new model in that domain. The *Simulations* profile shown in Figure 14.8 is an example of this use. A modeler will set out to build a simulation using language concepts in the *Simulations* profile and will think in terms of those concepts. In this type of use, the stereotypes in the profile will predominantly resemble metaclasses, as described in the previous section.

The latter use is a set of additional data that can be stored about existing model elements. A process or configuration management profile, such as the *Quality Assurance* profile shown later in Figure 14.13, is a good example of this use. Stereotypes from the *Quality Assurance* profile will be added to existing model elements, when quality-assurance information about them is required, and removed if and when the information is no longer relevant.

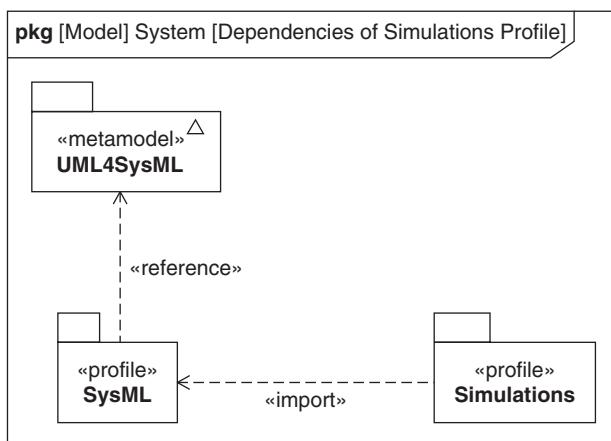


FIGURE 14.8

Defining the inputs required to specify the *Simulations* profile.

14.4.1 Specifying a Profile's Reference Metamodel and Other Inputs

Section 14.3 described how stereotypes are defined by either extending a metaclass or subclassing a stereotype. For a stereotype to extend a metaclass, the profile that contains the stereotype must include a reference to the metaclass, or a reference to the metamodel that contains the metaclass, using a special type of import relationship (see Chapter 5 for a discussion on the import relationship) called a **reference relationship**. To specialize a stereotype contained in another profile, the profile must import the stereotype, or import the profile that contains the stereotype. When a profile is importing an existing profile, references of the imported profile are the basis for its reference metamodel, although it may reference additional metaclasses as well.

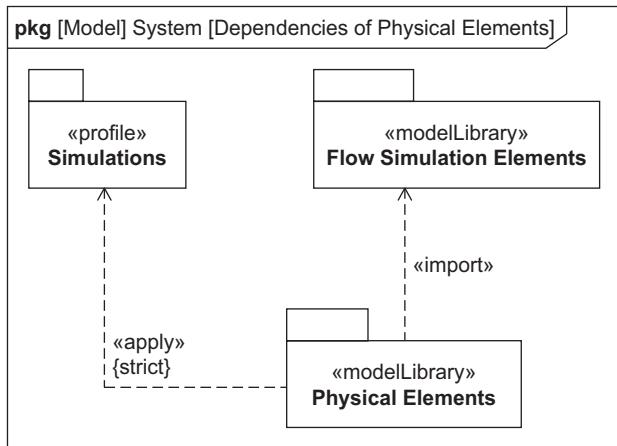
The notation for the reference relationship is a dashed arrow, annotated with the keyword «reference», with its head pointing at the referenced metaclass or metamodel. The import relationship is also shown as a dashed arrow with its head pointing toward the imported stereotype or profile, but it is annotated with the keyword «import».

In Figure 14.8, the *SysML* profile references the *UML4SysML* metamodel (the subset of UML used by SysML) to extend its metaclasses. The metamodel keyword is used, and the triangle indicates that this is a model. The *Simulations* profile imports the *SysML* profile and hence its reference metamodel is also *UML4SysML*. Stereotypes inside the *Simulations* profile can now extend metaclasses in *UML4SysML* (e.g., *Activity*) and subclass SysML stereotypes (e.g., *Block*).

14.5 Applying Profiles to User Models in Order to Use Stereotypes

The two previous sections in this chapter have described how to define a profile and the stereotypes contained within the profile. For modelers to use constructs from the profile in their model, they need to *apply* the profile to their model, or to a subpackage of their model. Once the profile has been applied, the stereotypes and other model elements in the profile, and the metaclasses from its reference metamodel, may be used anywhere within the containment hierarchy of the model or package.

A profile is applied to a model or package using a **profile application** relationship. The modeler can choose whether to apply the profile *strictly* by using the **strict property** of the profile application relationship. A strict application implies that only metaclasses from the profile's reference metamodel can be used within the model or package applying the profile. If the strict property is not set on the profile application, there is no restriction on which metaclasses can be used. A modeler can add or remove a profile application relationship at any time. However, when a profile application is removed, any instances of stereotypes from the profile are also removed from the model; so, any such removal should be undertaken with care and a backup copy of the model should be made.

**FIGURE 14.9**

Applying the *Simulations* profile to a model and importing elements to support flow-based simulations.

Whenever possible, it is recommended that the reference model for a profile be constructed in such a way that the profile can be applied strictly (i.e., that it has all the constructs required to support the profile domain). If users need to use metaclasses other than those referenced by the profile, it is likely that the impact of using them in combination with profile concepts may not have been fully considered. The SysML profile has been defined to be applied strictly, but this restriction can be removed to use additional software-related concepts from the UML metamodel if supported by a well-thought out systems and software development methodology.

The notation for applying a profile to a user model or subpackage is a dashed arrow, labeled with the keyword «apply», whose head points toward the profile that is applied.

Figure 14.9 shows a package diagram that contains the *Physical Elements* model library. *Physical Elements* applies the *Simulations* profile so that elements within it can have simulation extensions applied. Note that the *Simulations* profile is applied strictly, which means that only metaclasses from its reference metamodel (*UML4SysML* via its import of *SysML* shown on Figure 14.8) can be used in the *Physical Elements* model library. *Physical Elements* also imports a model library called *Flow Simulation Element* so that it can use the simulation elements it contains.

14.6 Applying Stereotypes when Building a Model

Once a user model has a profile applied to it, the stereotypes from the profile may be applied to model elements within that model. How stereotypes are used

depends on whether the intended purpose of the profile is a domain-specific language, or as a source of ancillary data and rules to support a particular aspect of the model. Although there is nothing in the specification of a profile to differentiate the two cases, often tool vendors will add custom support tailored to the intended use when building the profile.

For a given stereotype, its extension relationships define the model elements that it can validly extend, subject to the model element satisfying any additional constraints that the stereotype specifies. A model element may have any number of valid stereotypes applied to it, in which case it must satisfy the constraints of each stereotype.

Although the intention of the SysML graphical notation for stereotypes, and many tool vendor implementations of profiles, is to hide these details and to provide a visualization that matches the modeler's expectation, the mechanics of how stereotypes are applied is worthy of some explanation. When a stereotype is applied to a model element (i.e., a metaclass instance), an instance of the stereotype is created and is related to the model element. Once an instance of the stereotype exists, the modeler can then add values, which are stored in the instance, for the stereotype's properties. An instance of a stereotype cannot exist without a related metaclass instance to extend, and in consequence, when a model element is deleted, all its related stereotype instances are also deleted.

Subject to these basic rules, how the modeler actually applies stereotypes is often governed by a modeling tool based on the intended use of the stereotype. For example, the tool may create an instance of the stereotype and an instance of the base metaclass at the same time, or it may allow the modeler to create a model element first and then add and potentially remove the stereotype as separate actions.

Information from a stereotype is shown as part of, or attached in a callout to, the symbol of the model element to which it is applied. A stereotyped model element is shown with the name of the stereotype in guillemets (e.g., «*Stereotype-Name*»), followed by the name of the model element. Whereas the stereotype name may be capitalized, and may contain spaces in its definition, the convention is for the stereotype name to be shown as a single word using camel case (first letter lowercase, second and subsequent words in the original name have their first letter capitalized) when applied to a model element in a user model.

If a model element is represented by a node symbol (i.e., rectangle), the stereotype name is shown in the name compartment of the symbol. If the model element is represented by an path symbol (e.g., a line), the stereotype name is shown in a label next to the line and near the name of the element. Stereotype keywords can also be shown for elements in compartments when they are shown before the element name.

If a model element has more than one stereotype applied, then each stereotype name is, by default, shown on a separate line in a name compartment. If no stereotype properties are shown, multiple stereotype names can appear in a comma-separated list within one set of guillemets. See Figure 14.13 in Section 14.6.1 for an example of the application of multiple stereotypes. Whenever stereotypes are applied to a model element whose symbol normally has a keyword, its standard keyword is displayed before/above the stereotype keywords. The

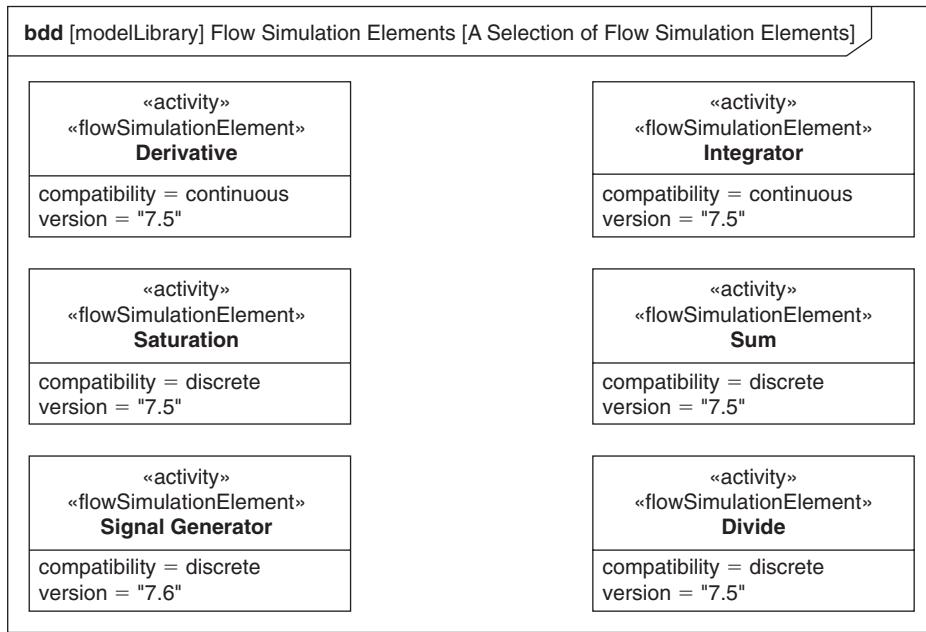


FIGURE 14.10

Defining a library of flow-based simulation elements using stereotypes to add simulation details.

properties for a stereotype may be displayed in braces after the stereotype label, or if the symbol supports compartments, in a separate compartment.

A stereotyped model element may also be shown with a special image that is part of the stereotype definition. For node symbols, that image may appear in the top right corner of the symbol, in which case it is often shown instead of the stereotype keyword. Alternatively, the image may replace the entire symbol.

Figure 14.10 shows some of the elements in the *Flow Simulation Element*'s model library. They all have the *flowSimulationElement* stereotype applied so that their *version* and *compatibility* properties can be specified. In this case *Derivative* and *Integrator* are only compatible with continuous simulations; the rest are compatible with discrete and continuous simulations. They all have version "7.5" except the *Signal Generator*, which has version "7.6." Note that because the underlying model elements are all activities, the keyword «activity» is shown, as described in Section 8.10. These elements can be used in the construction of flow-based simulations.

The activity diagram in Figure 14.11 shows a simulation model of the motion of the *Moving Thing* block, first shown in Figure 14.4. The activity *Motion Simulation* is the classifier behavior of *Moving Thing*, so the model shows what happens to it over its lifetime. The simulation calculates the values of acceleration, velocity, and distance over time. The algorithm first calculates the acceleration from

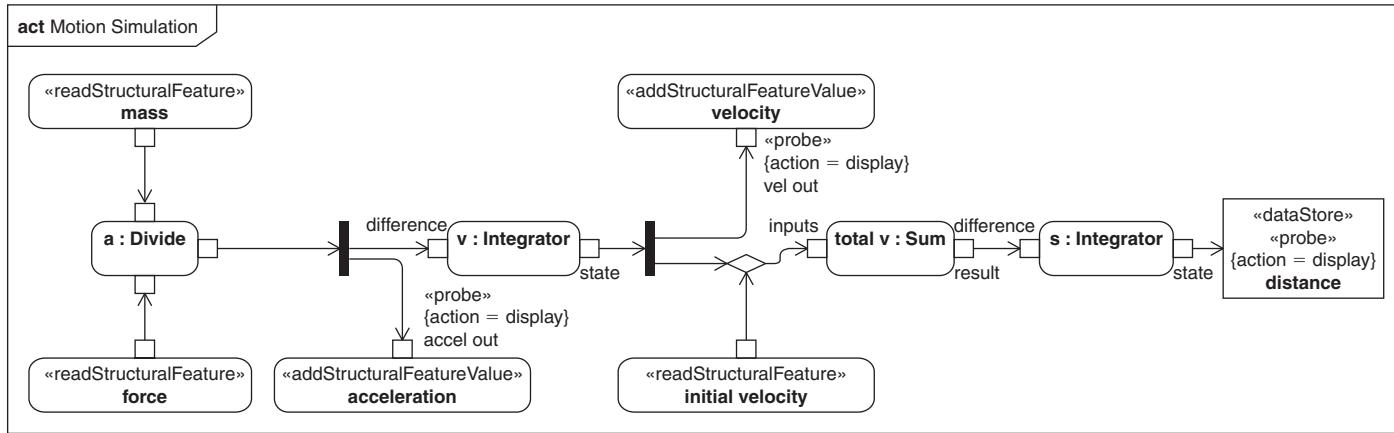


FIGURE 14.11

Using flow-based simulation stereotypes and library elements in the definition of a simulation.

the *mass* of the object (inherited from *Physical Thing*) and the *force* applied; then it integrates the acceleration to get the velocity. Finally, it integrates the sum of the velocity due to acceleration and the *initial velocity* to get the distance traveled, which is stored in data store *distance*. The current values of acceleration and velocity from the simulation are used to update the relevant properties of *Moving Thing*.

Three probes are used over time to display the values of acceleration, velocity, and distance. The first two values are obtained via probes on object flows, and the third by a probe on a data store.

Figure 14.12 shows *Motion Simulation* as an activity hierarchy. This view is useful because it shows the properties of the simulation elements. *Motion Simulation* and its children in the activity hierarchy satisfy all the constraints imposed by the stereotypes *Flow-Based Simulation* and *Flow Simulation Element*, as defined in Figure 14.7:

- All the invoked activities of *Motion Simulation* are stereotyped by *Flow Simulation Element*.
- All the invoked activities have version numbers at least as high as *Motion Simulation* itself.
- The *ode45* solver is appropriate for a variable step continuous simulation.

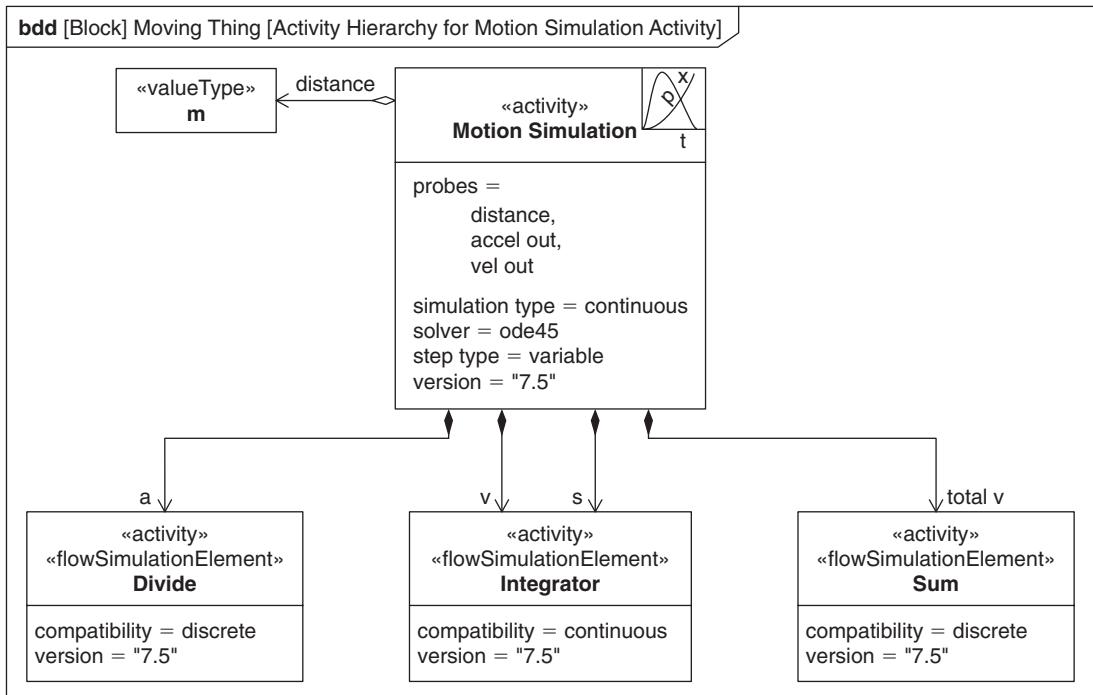


FIGURE 14.12

Block definition diagram showing the activity hierarchy for *Motion Simulation*.

- *Motion Simulation* is a continuous simulation, so both discrete and continuous *Flow Simulation Elements* are allowed.

Instead of showing the keyword «*flowBasedSimulation*» for *Motion Simulation* to illustrate the use of stereotype images, this figure shows the stereotype's image in the top right corner of the symbol.

14.6.1 What Happens When Model Elements with Applied Stereotypes Are Specialized?

A potential area of confusion when using stereotypes is the effect of subclassing a classifier—a model element that can be classified (i.e., have subclasses)—that has a stereotype applied to it in the user model. Application of a stereotype to a model element does not imply that the stereotype is applied to subclasses of the model element. Whenever such an outcome is desired, its stereotype definition should include a specific constraint to ensure this. Even when a constraint forces subclasses to have the same stereotype as their superclasses, they do not inherit values for stereotype properties. When this is desired, the stereotype should include an additional constraint that every subclass has the stereotype applied and also inherits the values of the stereotype's properties. Figures 14.13 and 14.14 show an example where neither applied stereotypes nor the values of their properties are inherited.

Figure 14.13 shows two stereotypes from the profile *Quality Assurance*. The stereotype *Audited Item*, which extends the metaclass *Classifier* and can be applied to blocks among other model elements, is used when a classifier has been audited for quality—typically, when it reaches a certain level of maturity. It has properties to capture the *audit date*, the *auditor*, and the *quality level* that may take values from *high* to *low*. The stereotype *Configured Item* contains properties that must be applied to every classifier, hence the presence of the *{required}* property.

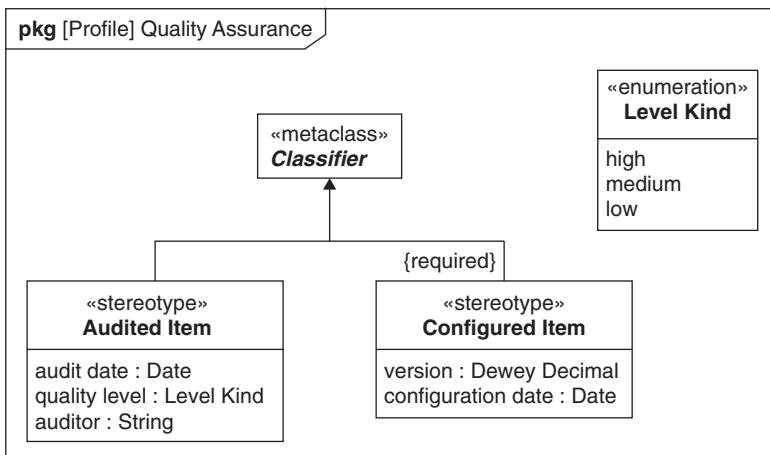
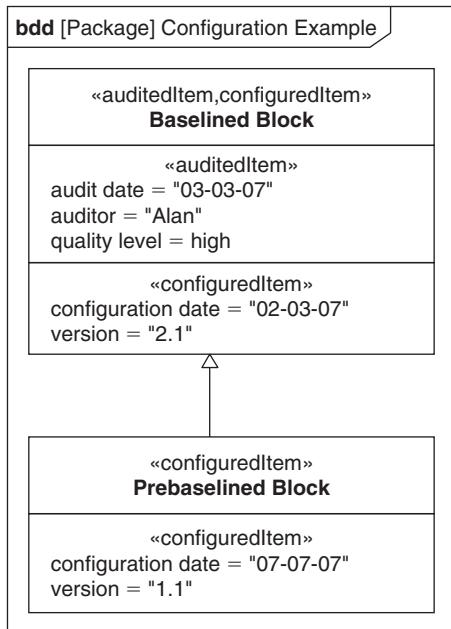


FIGURE 14.13

Definitions of two stereotypes used as part of quality assurance on a model.

**FIGURE 14.14**

Application of quality-assurance stereotypes to two blocks, one of which specializes the other.

Figure 14.14 shows the *Audited Item* and *Configured Item* stereotypes in use. In this case the block *Baselined Block* has been audited and so has values for *audit date*, *auditor*, and *quality level*. Its subclass *Prebaselined Block* is still in early design, so it has not yet been audited. It clearly does not make sense to assume, just because *Baselined Block* has the *Audited Item* stereotype applied to it, that *Prebaselined Block* will also have it.

Even when a stereotype, such as *Configured Item*, is required and therefore applied to all blocks, it clearly is not the case that the configuration properties of a block (e.g., *Baselined Block*) will be inherited by a subclass like *Prebaselined Block*. The information stored in the properties of *Configured Item* is specific to the model element to which it is applied.

Note that *Baselined Block* has two stereotypes applied to it, demonstrating the notations that are used where multiple stereotypes are applied. The keywords representing the two applied stereotypes both appear separated by a comma inside a single set of guillemets. The properties of the two stereotypes appear in separate compartments, labeled using the keyword of their owning stereotype.

14.7 Summary

SysML is a general-purpose systems modeling language that includes built-in mechanisms, called model libraries and profiles, to further customize the language.

When used properly, model libraries and profiles can be used to support domain-specific modeling for many different domains. The following are some of the key concepts for domain-specific modeling.

- A modeling language is defined using a metamodel and contains a number of distinct language concepts, represented by metaclasses. Metaclasses have a set of properties and constraints on them. Metaclasses can also be associated with each other, thus allowing the language concepts to be related to one another. The underlying metamodel for SysML is called UML4SysML and is based on UML—an existing modeling language. UML4SysML contains the subset of UML concepts that are needed for systems modeling. SysML defines a graphical notation, based on UML, to represent the concepts in the metamodel.
- User models contain model elements, which are instances of metaclasses contained in the metamodel. These model elements have values for the properties of their metaclasses and can be related according to the associations defined between their metaclasses.
- A model library is a special type of package that contains model elements intended for reuse in multiple models. They can vary from very specific, such as representing a set of electronic components, to general, such as a definition of a common set of units and dimensions for representing quantities.
- A profile adds new concepts to a language (in this case SysML) by means of stereotypes. A profile extends a reference metamodel, which for SysML profiles is always its reference metamodel—UML4SysML. SysML itself is defined as a set of profiles that extend UML4SysML, but it also makes the profile mechanism available to SysML modelers so that they may further extend the language. A profile can import an existing profile in order to reuse the stereotypes it contains.
- Stereotypes extend one or more metaclasses in the reference metamodel. A stereotype can contain properties and also constraints that may constrain both the values of its own properties and the property values of its base metaclasses.
- To use a profile, a modeler must apply it to his or her model or some subpackage of the model using a profile application relationship. A profile may be applied strictly, which means that model elements that apply to the profile may only be instances of metaclasses in the profile's reference metamodel.
- When a profile has been applied, stereotypes from that profile may be applied to appropriate model elements within it. Once a stereotype has been applied, modelers may provide values based on the stereotype's properties, and the constraints of the stereotype are applied to the model element. SysML includes a graphical notation that describes how a stereotyped model element appears in a diagram.

14.8 Questions

1. Which type of diagram is used to define model libraries and profiles?
2. List the three parts of a modeling language like UML.
3. What are metaclasses used for?
4. What is the relationship between metaclasses and model elements?
5. What is a model library used for?
6. What is the relationship between a stereotype and its base metaclass called and how is it represented on a diagram?
7. Which rule applies to an association between a stereotype and a metaclass and why?
8. Which model elements can a profile contain?
9. What is the reference relationship used for?
10. What must modelers do before they can apply stereotypes to elements in their models?
11. On a diagram, how can a modeler tell that a stereotype has been applied to a model element?
12. How can the applied stereotype and stereotype property values for a graphical path (line) symbol be shown?
13. How can the applied stereotype and stereotype property values for a block symbol be shown?
14. When a block subclasses a block with a stereotype applied to it, which of the following describes the effect?
 - a. The subclass automatically inherits the stereotypes applied to its superclass.
 - b. The subclass automatically inherits the stereotypes applied to its superclass and also inherits the values of any stereotype properties.
 - c. The subclass cannot inherit either applied stereotypes or the values of stereotype properties.
 - d. The subclass can inherit applied stereotypes and the values of stereotype properties but the stereotype has to be explicitly specified to allow that.

Discussion Topics

When adding new concepts to a language, when does it make sense to use a profile and when to use a model library?

What is the difference in meaning and use between a property of a stereotype and the property of a block?

PART

Modeling Examples

III

This page intentionally left blank

15

Water Distiller Example Using Functional Analysis

This chapter contains an example that describes the application of SysML to the design of a water distiller system using a traditional functional analysis method. This method is familiar and intuitive to many practicing systems engineers. The example was originally developed by the International Council on Systems Engineering (INCOSE) to support the initial evaluation of the SysML specification. Chapter 16 demonstrates the application of SysML to a more complex problem using the object-oriented systems engineering method. The following approach is used to address the problem:

- Stating the problem
- Defining the model-based systems engineering approach
- Organizing the model
- Establishing requirements
- Modeling behavior
- Modeling structure
- Analyzing performance
- Modifying the original design

15.1 Stating the Problem

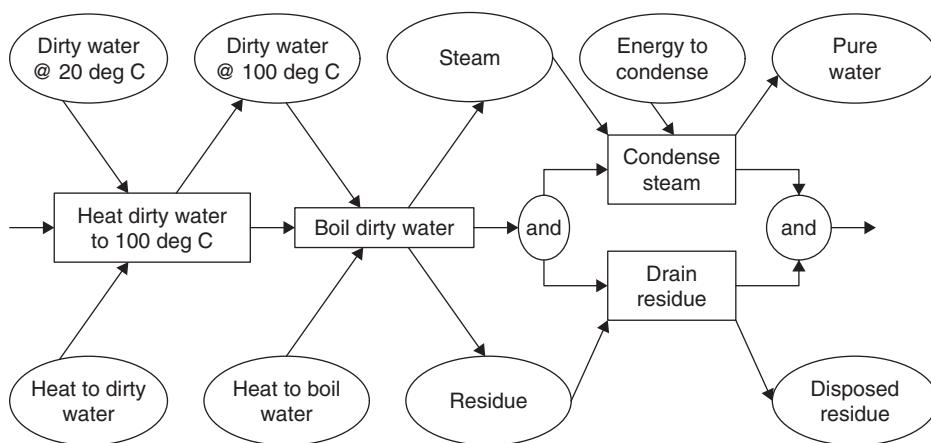
Eliciting and analyzing stakeholder requirements is a critical initial step in the systems engineering process. When this is done, stakeholder requirements are often revealed to contain ambiguity and contradictions. The ability to model the specification of a system provides a mechanism to better understand the requirements, reduce the ambiguity, and validate the requirements with the stakeholders to ensure the right problem is being solved.

In this example, the initial stakeholder requirements were provided as follows:

Describe a system for purifying dirty water

Heat dirty water and condense steam are performed by Counter Flow Heat Exchanger

Boil dirty water is performed by a Boiler

**FIGURE 15.1**

Informal, Non-SysML behavior diagram provided with distiller problem statement.

Drain residue is performed by a Drain

Water has the following properties: vol = 1 liter, density = 1 gm/cm³, temp = 20 °C, specific heat = 1 cal/gm °C, heat of vaporization = 540 cal/gm

These statements are elaborated in the diagram in Figure 15.1, which was provided by the customer to help communicate the requirements.

Although the diagram lacks formalism, it does emphasize certain features that appear to be important to the stakeholder specifying this problem. The diagram identifies the primary functions that the system is expected to perform—heating water, boiling water, condensing steam, and draining residue—along with the expected flows between system functions and expected sequence for the functional flow.

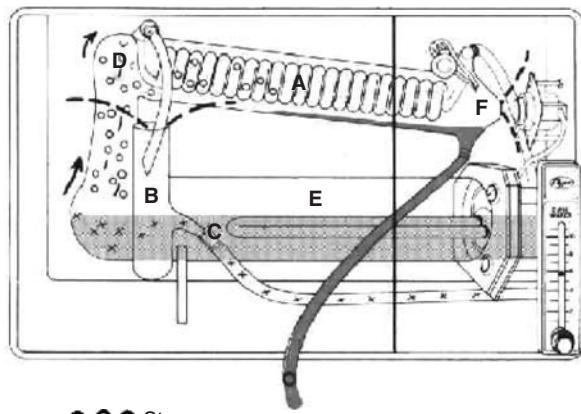
The system modeler must interpret and analyze the stakeholder requirements to develop a precise and complete specification and to remove potential ambiguities. For example, the diagram seems to imply that the distillation is a discrete process that produces purified water in batches, versus a distillation process where water continuously flows into the distiller and produces a continuous stream of purified water.

Figure 15.2 shows an example of a batch distiller that includes a boiler and a condenser. The boiler is filled with water. A heat source is used to heat the water in the boiler, steam is then generated, and the distilled water is collected from the condenser. The process stops when there is no more water in the boiler; purifying more requires refilling the boiler with water.

Figure 15.3 shows an example of a continuous distiller that can have water flow through it continually. It includes a boiler with an internal heating element and a counterflow heat exchanger that has cool liquid flowing in the coils and steam condensing around them. These terms are consistent with the original problem statement, indicating that a continuous distiller may be the preferred approach to address the total set of requirements, but this must be validated to ensure that customer needs are being satisfied.

**FIGURE 15.2**

Representation of a batch distiller.



○ ○ ○ Steam
✗ ✗ ✗ Salts, nitrates, and metals
— — — Chemical contaminant venting

A = condensing coil	D = evaporate
B = drain	E = heater
C = dirty water	F = pure condensate

FIGURE 15.3

Representation of a continuous distiller.

15.2 Defining the Model-Based Systems Engineering Approach

The model-based systems engineering approach taken to address this problem is outlined next. Note that while the steps are shown as a sequence, they are often performed in parallel and iteratively.

- Organize the model and identify reuse libraries
- Capture requirements and assumptions

- Model behavior
 - In similar form to problem statement
 - Elaborate as necessary
- Model structure
 - Capture implied inputs and outputs, and things flowing through the system
 - Identify structural components and their interconnections
 - Allocate behavior onto components and behavioral flow onto interconnections
- Capture and evaluate parametric constraints
 - Derive and represent the heat balance equation
 - Perform the analysis to assess feasibility of the solution
- Modify design as required to meet constraints

Organize the model The initial step in any model-based approach is to establish a model organization. In particular, the organization should include a package structure based on the concepts presented in Chapter 5. Model organization also includes the identification and incorporation of existing libraries of components or other elements that may be leveraged to support the model development.

Capture requirements and assumptions The next step is to capture the requirements and assumptions. This lays the necessary foundation for further development.

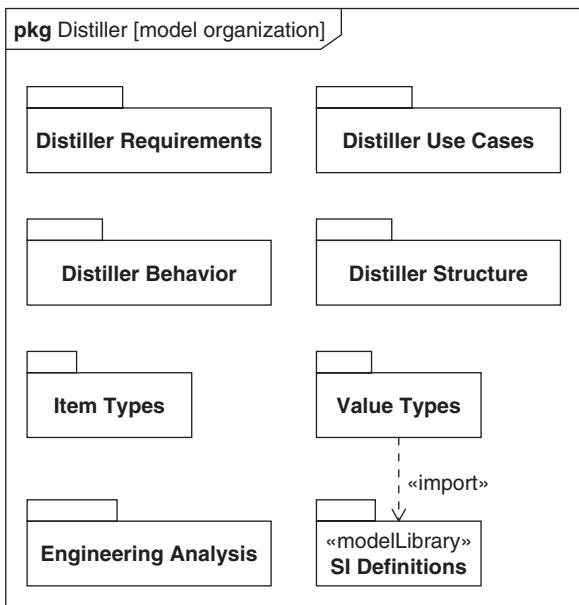
Model behavior Modeling behavior and structure can be done concurrently. In this example, behavior modeling is discussed first. Since the customer provided an initial behavior diagram, behavioral modeling focuses first on formalizing the behavior that was provided and reconciling it with the other requirements.

Model structure Modeling structure is discussed next. The customer has indicated a partial identification of system components. As with the behavior model, the structural model is formalized and reconciled, both with requirements and behavior. Part of this reconciliation involves assessment of how the behavior model is supported by the structure model.

Capture and evaluate parametric constraints and modify design as required This step is used to model the distiller performance in the form of a heat balance equation. The customer provided some of the information necessary to perform the analysis, so it is used to determine whether this system will perform as expected. If not, the design is modified as necessary to develop a feasible solution that addresses the requirements.

15.3 Organizing the Model

A critical step prior to initiating significant effort in specifying model elements and developing diagrams is to establish the initial organization of the model. This is done by defining the model's overall package structure. The organization

**FIGURE 15.4**

Package diagram documenting the organization of the distiller model.

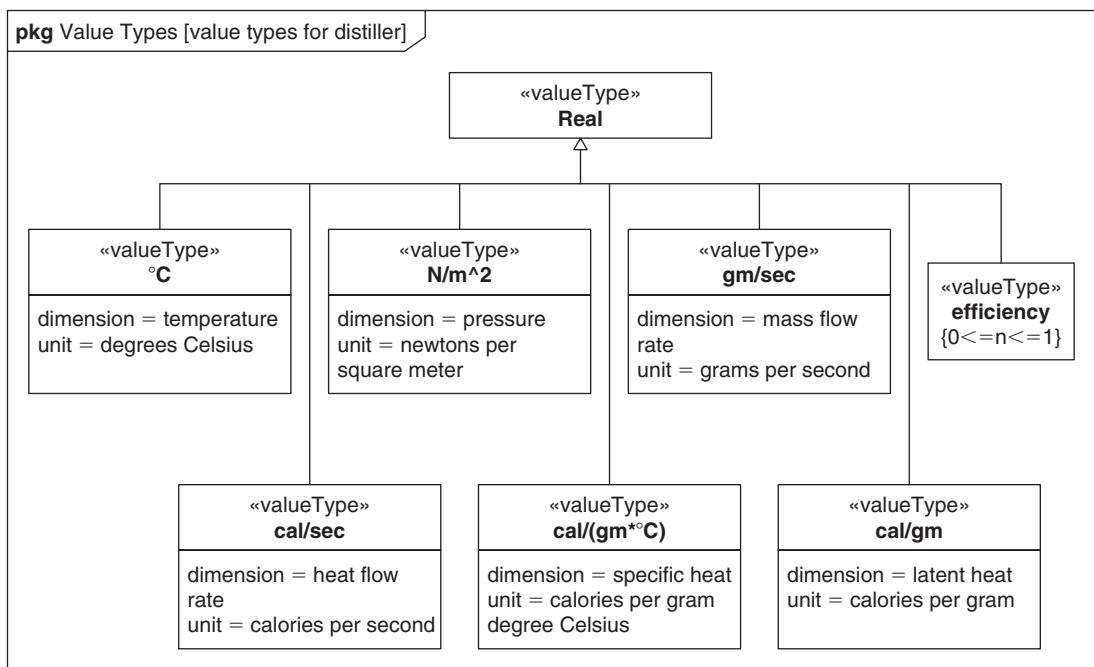
should also consider what model libraries may be leveraged for the development. Chapter 5 provided a number of approaches that can be used to organize the model. Caution must be exercised when organizing the model to avoid prematurely constraining or biasing the design.

The package diagram in Figure 15.4 describes the organization for this model. The packages are primarily organized based on the types of artifacts developed using the selected process, including requirements, use cases, and structural and behavioral models. The *Engineering Analysis* package includes the constraint blocks and parametric models used to analyze the performance.

Note that the *Value Types* package imports from the *SI Definitions* package—a reusable library package available to multiple models. The *Value Types* package uses the imported definitions of units and dimensions to create specific value types as indicated in Figure 15.5. The value types are then applied to value properties with consistent units throughout the model.

A package for *Item Types* is included to separately capture the types of things that flow in the system. Segregating item types into its own package allows the modeler to concentrate on defining the things that flow and leverage reuse libraries that may exist independent of where they flow or how they are used. This segregation is similar to establishing a reusable library of components, and has proved effective in the past. For this example, water and heat flow through the system. Putting the item types in a separate package allows the modeler to consolidate all the relevant information about water, heat, and the other *Item Types* used in this model.

The browser structure of the modeling tool typically provides a view of these packages in a folderlike structure that is populated as the model is developed.

**FIGURE 15.5**

Block definition diagram documenting the *valueTypes* used in analyzing distiller performance.

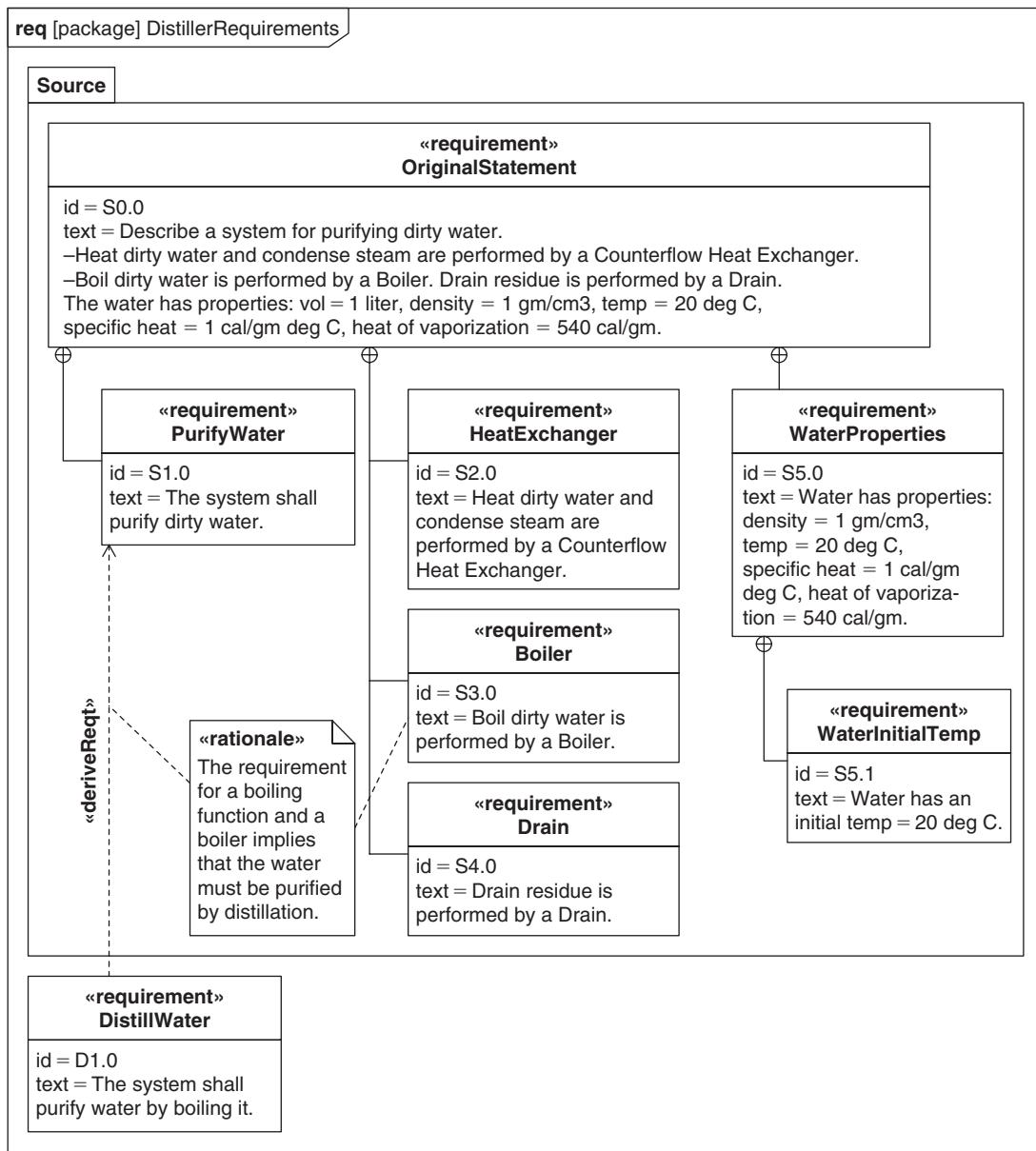
It may be convenient to revise the organization of the model over time as the model is refined and updated. For example, after an initial design has been established, packages may be established for each component that is subject to further design and analysis.

15.4 Establishing Requirements

The system requirements for this example were provided by the stakeholder in the form of the problem statement described in Figure 15.1. The problem statement is captured in the requirement diagram in Figure 15.6. The diagram's header indicates that the frame represents a package called *Distiller Requirements*.

The original requirements statement is designated with requirement id *S0.0*. This diagram shows the containment relationship: how requirement *S0.0* is decomposed into individual atomic requirements *S1* through *S5*. Requirement decomposition indicates that nothing was added or subtracted from the statements in the source requirement *S0.0*, but that the compound statement was replaced by a set of atomic statements, each of which can be individually analyzed and verified.

It often becomes necessary to derive more explicit requirements from an existing set of requirements. For example, requirement *S1.0* states that “*The system shall purify dirty water.*” It does not say that the system shall boil water, but that it

**FIGURE 15.6**

Requirement diagram with top-level distiller requirements from the problem statement.

shall purify water. To explicitly understand that the system needs to purify water by boiling it, we have to derive a new requirement, as follows: requirement S2.0 states that the system must “*Heat dirty water....*,” and requirement S3.0 states that the system must “*Boil dirty water....*” It is merely necessary to show the derivation

relationships between purify, heat, and boil. This provides the rationale for the derivation of requirement *D1.0*.

Requirement *S2.0* states that “*Heat dirty water and condense steam are performed by a Counterflow Heat Exchanger.*” Requirements can be derived from *S2.0* that state “*The system shall contain a Counterflow Heat Exchanger*” and “*The system shall heat dirty water*” and “*The system shall condense steam*.” It should also be noted that these statements were provided as requirements, yet they impose a design solution. It is generally undesirable to have requirements that overconstrain the solution space; however, this may not be within the control of the system designer. It is the responsibility of the systems engineer to establish a dialog with the customer and sort the true requirements from assumptions about the solution.

This process for capturing requirements has resulted in a more granular set of requirements statements and rationale than the source requirement that was provided by the customer. The source requirement has been broken up into requirements related to heat exchangers, boilers, and drains. The source statement about water properties has also been broken out separately and then further broken down into initial water temperature, density, specific heat, and heat of vaporization. The added granularity of the requirements statements will now enable each of them to be uniquely traced to the parameters used in the distiller performance analysis.

It may seem surprising to see properties of water tracked as requirements that show up on a requirement diagram. The point of doing this is to have complete traceability from the customer’s problem statement through the design. If the properties of water were somehow assessed to be incorrect, documenting them in this way would provide a rationale for why they were changed, or at least provide a dialog with the customer to assist in the requirements validation process.

In this example, all source requirements, both original and decomposed, start with *S*. All derived requirements start with *D*. It is expected that the project establishes a convention for requirements numbering, and that tools can assist in enforcing the convention. The requirements numbers may correspond to specific paragraph numbers in a specification.

Figure 15.7 includes the requirements in a tabular format, which is an allowable notation in SysML. This is a traditional way to view the requirements. The table in the figure is a report out from the model and contains some of the same information shown in the requirement diagram; it provides requirement id, name, and text. In this case the table relies on the numbering to indicate the hierarchy or containment relationships, but this could have been shown by level of indenture or on another mechanism.

Figure 15.8 also shows a tabular format for requirements, but it includes the relationships between requirements. In addition to id and name, the table captures the derive relationship, which shows how one requirement is derived from another, along with the rationale for the relationship. When multiple relationships are to follow, this generates a requirements tree. This information is also shown graphically on the requirement diagram, which is a useful way to enter the relationships; however, it is often more compact to view the information in tabular format. Tools are expected to provide the tabular format for requirements and other types of modeling information, as described in Chapter 4.

table [Package] DistillerRequirements [Decomposition of OriginalStatement]		
id	name	text
S0.0	OriginalStatement	Describe a system for purifying dirty water. ...
S1.0	PurifyWater	The system shall purify dirty water.
S2.0	HeatExchanger	Heat dirty water and condense steam are performed by a ...
S3.0	Boiler	Boil dirty water is performed by a Boiler.
S4.0	Drain	Drain residue is performed by a Drain.
S5.0	WaterProperties	Water has properties: density = 1 gm/cm ³ , temp = 20 deg C, ...
S5.1	WaterInitialTemp	Water has an initial temp of 20 deg C

FIGURE 15.7

Requirements decomposition table.

table [Package] DistillerRequirements [Requirements Trace from S1.0]					
id	name	relation	id	name	Rationale
S1.0	PurifyWater	deriveReqt	D1.0	DistillWater	The requirement for a boiling function and a boiler implies that the water must be purified by distillation.

FIGURE 15.8

Requirement trace in tabular format.

Note that Appendix C of the OMG SysML specification lists nonnormative requirement types that may be used. Users may want to leverage these types and/or create user-defined extensions using the profile mechanism described in Chapter 14.

15.5 Modeling Behavior

This section describes techniques used for modeling distiller behavior and flow and introduces behavioral allocation.

15.5.1 Simple Behavior

The first step in analyzing the system behavior is to recast the customer's original behavior diagram as an activity diagram in SysML. This initial version uses the Enhanced Functional Flow Block Diagram (EFFBD) format, which is a nonnormative profile included the SysML 1.0 specification [1].

The diagram in Figure 15.9 characterizes the behavior of the *Distill Water* activity. The enclosing frame designates an enclosing activity called *Distill Water* as shown in the diagram header. As described in Chapter 8, round-cornered boxes represent actions (usages) that are typed by activities (definitions). The dashed lines

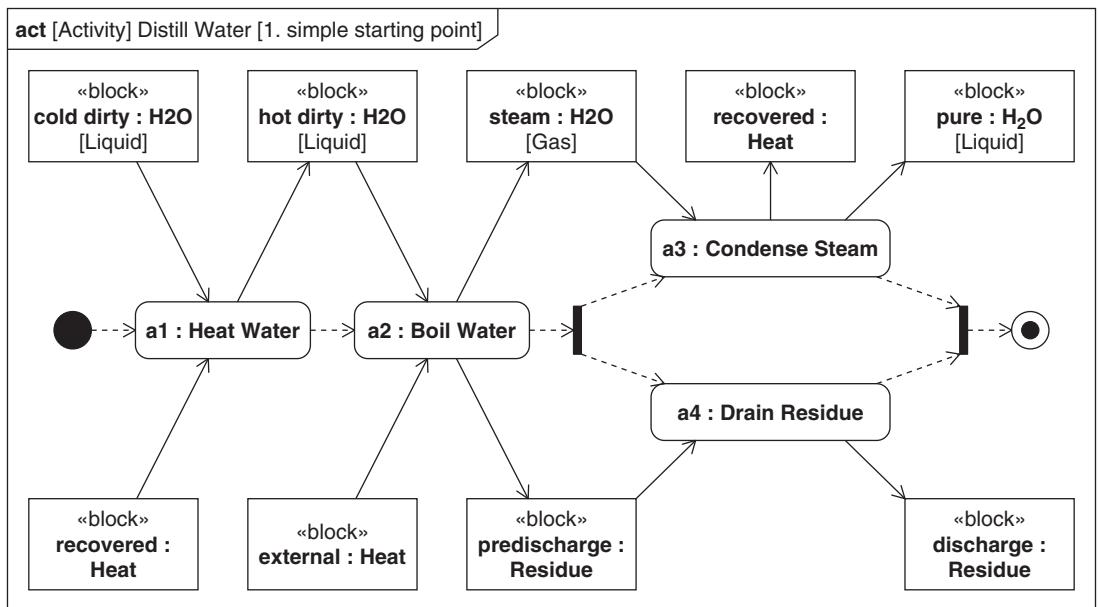


FIGURE 15.9

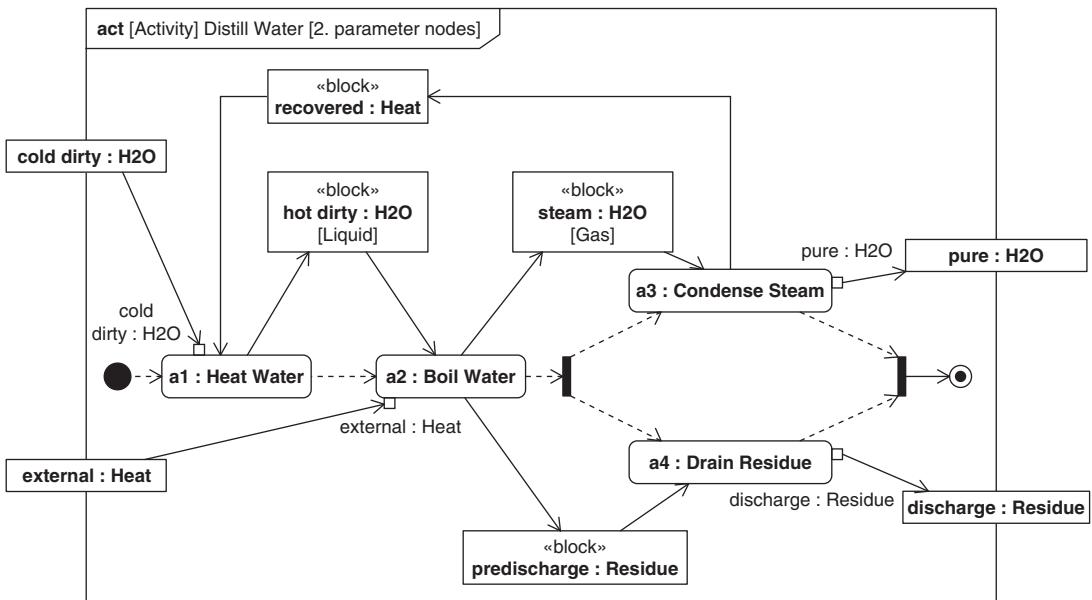
Original problem statement diagram captured as a SysML activity diagram.

are control flows that define the sequence of actions; dashed lines are optional but help to more clearly distinguish control flow from object flow. The things that flow are represented as object nodes, in this case drawn as square-cornered boxes connected to actions via object flows. The actions and object nodes include their role names (usages) and types (definitions) using the role name : Type Name notation.

Note that the object node *recovered : Heat* flows into action *a1 : Heat Water*. This begs the question: Where does that heat come from? On the customer's original behavior diagram in Figure 15.1, *Energy to condense* was shown flowing into the activity *Condense steam*. This energy is actually heat that must be removed in order for the steam to condense. Hence, in recasting this diagram into SysML in Figure 15.9, the object node *recovered : Heat* is shown flowing out of *a3 : Condense Steam*. This is consistent with the operation of a counterflow heat exchanger. Although this diagram accurately represents all the elements on the customer's original diagram, it still does not adequately describe the desired distiller behavior and needs further refinement.

Note too that this model considers heat to be provided from a source external to the distiller. For both the batch and continuous distillers shown in Figures 15.2 and 15.3, one could consider the heat source to be internal to the distiller system, in one case consisting of an oil lamp, and in the other, an electric heating coil. Here, the heat source is initially modeled as an external component, but later in the example it is introduced as an electrical heater that is part of the boiler.

Figure 15.10 shows a refined version of the activity diagram. Note that action pins are necessary when object nodes send or receive object flows from activity

**FIGURE 15.10**

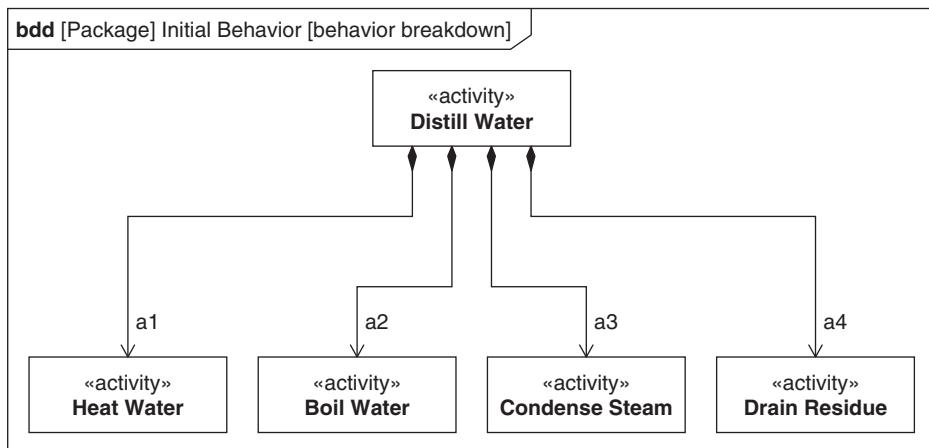
Elaborating the original diagram with activity parameter nodes and a feedback loop.

parameter nodes (on the diagram frame). The use of action pins will be important when considering flow allocation. In this version, the *recovered : Heat* from *a3 : Condense Steam* is connected directly to *a1 : Heat Water*, thus eliminating redundancy in the previous diagram. It is also appropriate at this point to recharacterize those object nodes that flow outside the distiller as activity parameter nodes, thus graphically moving them to the boundary of the *Distill Water* activity diagram. This more clearly defines the activity inputs and outputs and enables *Distill Water* to fit into a larger behavioral context.

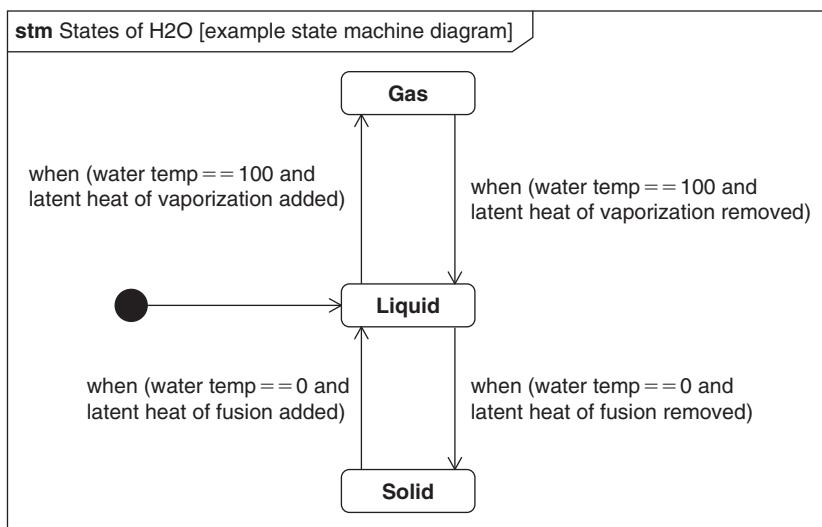
Note the operational implications of the control flow on the model. When the distiller stops heating water, it initiates the action *Boil Water*. When it stops boiling water, it initiates the parallel actions—*Condense Steam* and *Drain Residue*. When these actions are complete, the *Distill Water* activity is complete. This is consistent with the definition of a batch distiller shown in Figure 15.2.

The decomposition of the *Distill Water* activity is represented in a block definition diagram in Figure 15.11. This functional decomposition concept is discussed in Section 8.14 in Chapter 8. Note that the composition relationships use role names on the part end of the associations that are consistent with action names shown on the activity diagram in Figure 15.10. The actions represent the usage of these activities.

The control flows and object flows from the activity diagram are not shown on a block definition diagram. The blocks used to type the object nodes can be shown

**FIGURE 15.11**

Distill Water functional hierarchy.

**FIGURE 15.12**

Representing states of H₂O.

on the block definition diagram to provide the definition of the items flowing into the activity diagram. In this case the items include water, residue, and heat, but are not shown.

It should also be noted that water changes state between gas (i.e., steam) and liquid as it proceeds through the *Distill Water* process. Figure 15.12 shows a state machine diagram for the change in state of water that includes solid, liquid, and gas states. The transitions are shown, along with the guard condition. Latent heat of vaporization must be added to transition from liquid to gas. The same latent

heat of vaporization must be removed when transitioning from gas to liquid. The state machine provides useful information for analyzing the distiller system's heat balance since removing the latent heat turns out to be a driving factor.

15.5.2 Parallel Flow

Up to this point, activity diagrams have represented a highly sequential flow that adequately represents a simple batch distillation process. However, as pointed out in the requirements analysis earlier, customer requirements require further validation to determine whether this is truly what is desired. As a result, the activity diagram will be modified to represent continuous and parallel behavior that is consistent with a continuous distiller.

Figure 15.13 is an initial modification of the activity diagram that represents a parallel-versus-sequential flow of actions triggered by inputs and outputs. Note that the inputs and outputs to the *Distill Water* activity (activity parameter nodes) are the same. The control flow has been reoriented so that all the activities are enabled simultaneously when a token is placed on the initial node.

15.5.3 Continuous Flow

In Figure 15.14, the object flows have been stereotyped as continuous to more accurately represent the continuous distiller process. As described in Chapter 8, continuous flows mean that the delta time between inputs approaches zero, as is the case with many physical flows such as water and heat. In addition, the continuous flows are streaming, which means that the action can consume inputs and produce outputs while it is processing. The implications of an action with continuous and streaming inputs and outputs are that it does not automatically terminate when it produces an output. As a result, another mechanism is required to indicate when the actions complete their execution.

In Figure 15.15, an interruptible region, indicated by the dashed line, encloses the actions. All the actions in this interruptible region terminate when the region is exited, which occurs when the *shutdown* signal is received by the accept event action. This behavior model now represents the steady-state behavior for a continuous distiller. It does not address how the distiller is started up or shut down, but it is adequate to proceed with the initial design. The details of startup and shutdown are addressed later. The next step is to formalize how this functionality is implemented in terms of distiller structure (e.g., the blocks in the block definition diagram and parts in the internal block diagram).

15.5.4 Allocated Flow

The initial allocation of behavior to structure can be specified through the use of activity partitions (i.e., swimlanes). In Figure 15.16, the initial allocation of actions is specified by the use of partitions to represent the parts *condenser : Heat Exchanger*, *evaporator : Boiler*, and *drain : Valve*. The use of the keyword

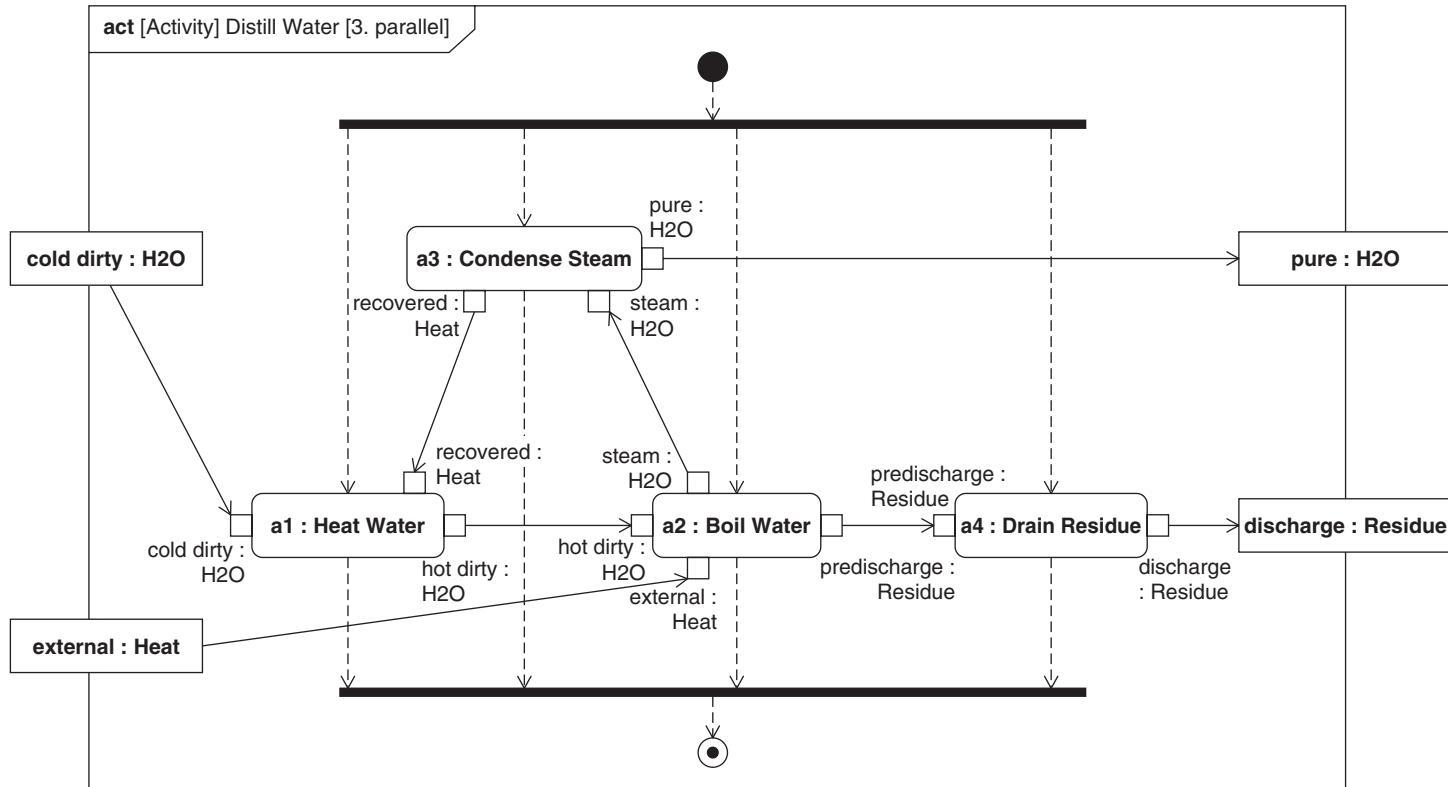


FIGURE 15.13

Depicting parallel flow in the distiller.

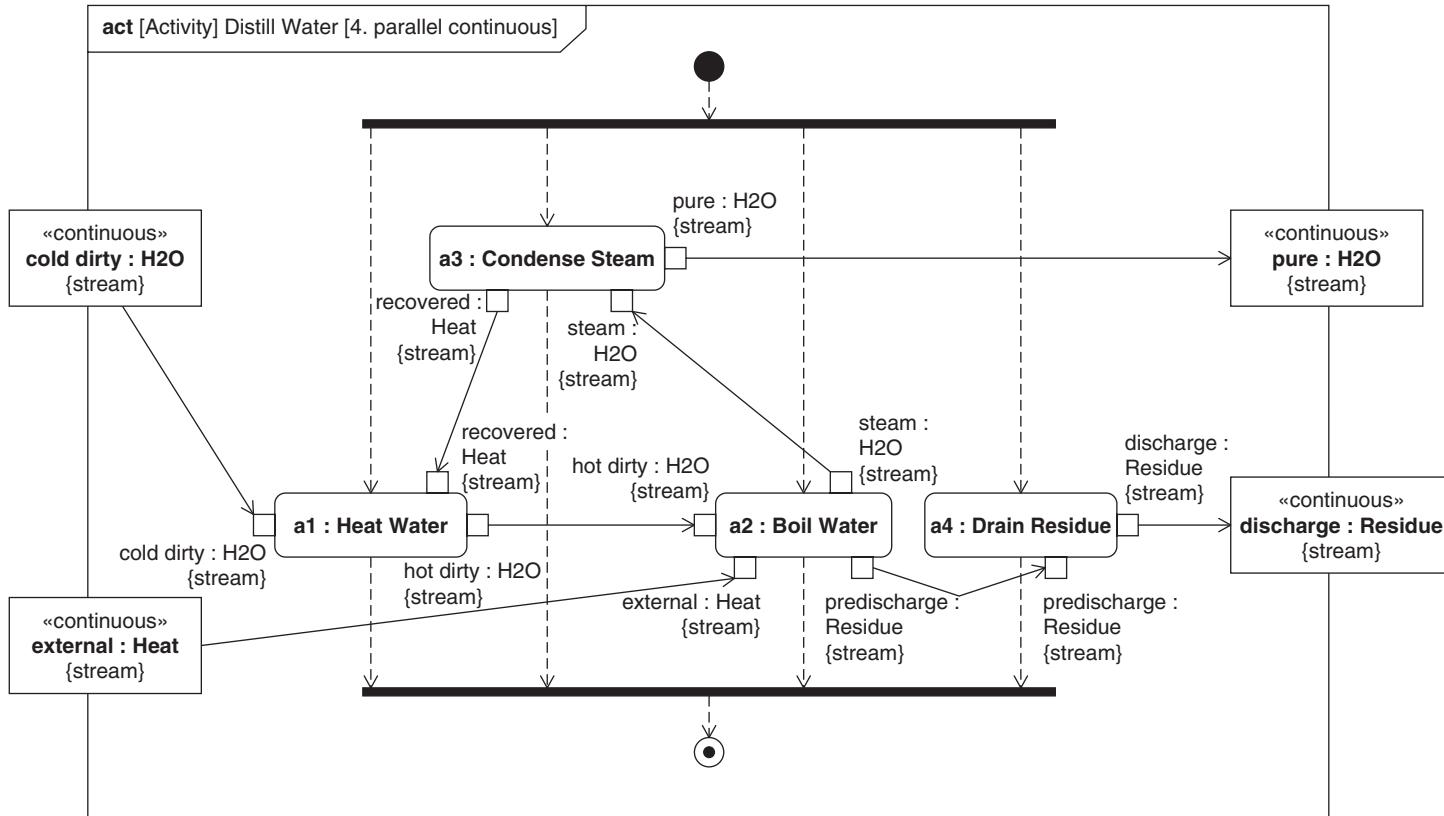
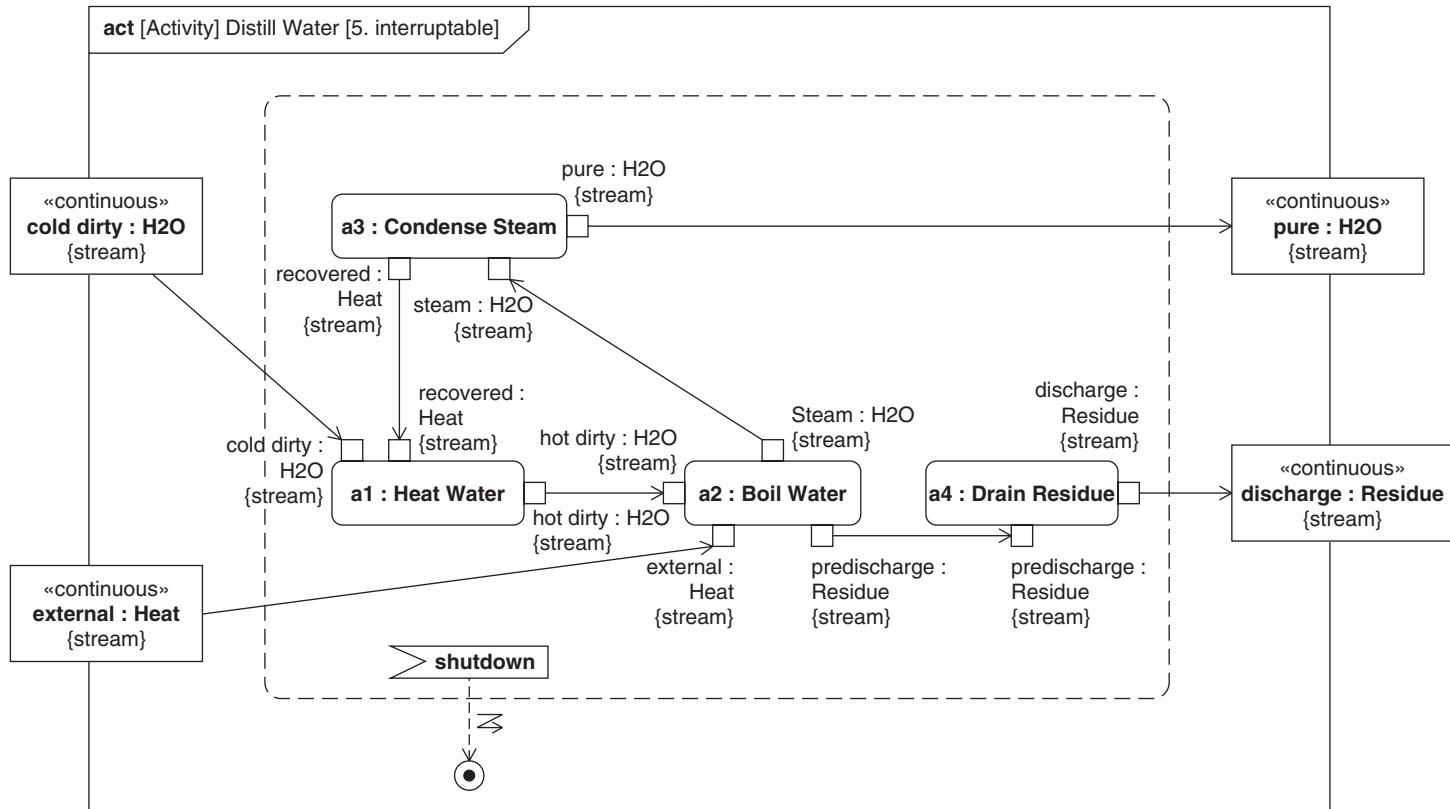


FIGURE 15.14

Depicting continuous flow in the distiller.

**FIGURE 15.15**

Using an interruptible region to simplify control flow.

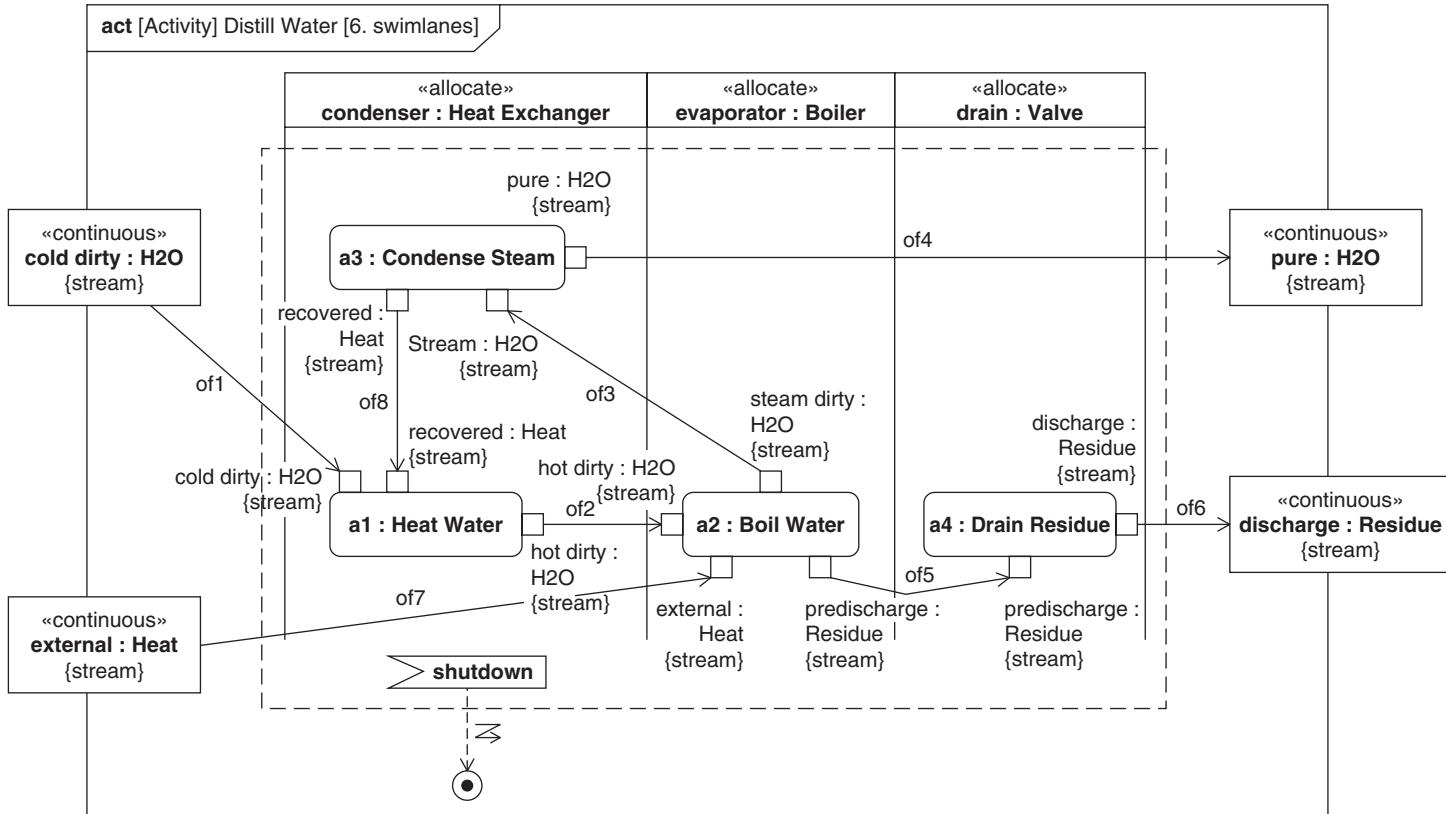


FIGURE 15.16

Using allocate activity partitions (swimlanes) for functional allocation.

«allocate» in the partition means that the partition is an allocate activity partition that has an explicit allocation relationship to the part that represents the partition, as described in Chapter 8. This in turn specifies that the part is responsible for performing the actions within the partition.

As an example, the part *evaporator* is a usage of the block *Boiler*, and the action *a2 : Boil Water* is allocated to *evaporator : Boiler*. Note that we have defined role names for each part, and typed each part by a block. For example, the role *drain* is a part of type *Valve*. This distinction is important because other valves with the same definition may have different roles, as will be evident later in the example. The specification of the parts and blocks are described next as part of the distiller structure.

15.6 Modeling Structure

This section describes the use of blocks, parts, and ports for the modeling of a distiller's structure, and it completes the example of behavioral allocation.

15.6.1 Defining Distiller's Blocks in the Block Definition Diagram

Figure 15.17 is a block definition diagram for the distiller system. This diagram frame designates the *Initial Distiller Structure* package. Use of packages for organizing models was discussed in Chapter 5. Each block on the diagram is contained within this package unless it includes a qualified name that indicates that it is contained in a different package.

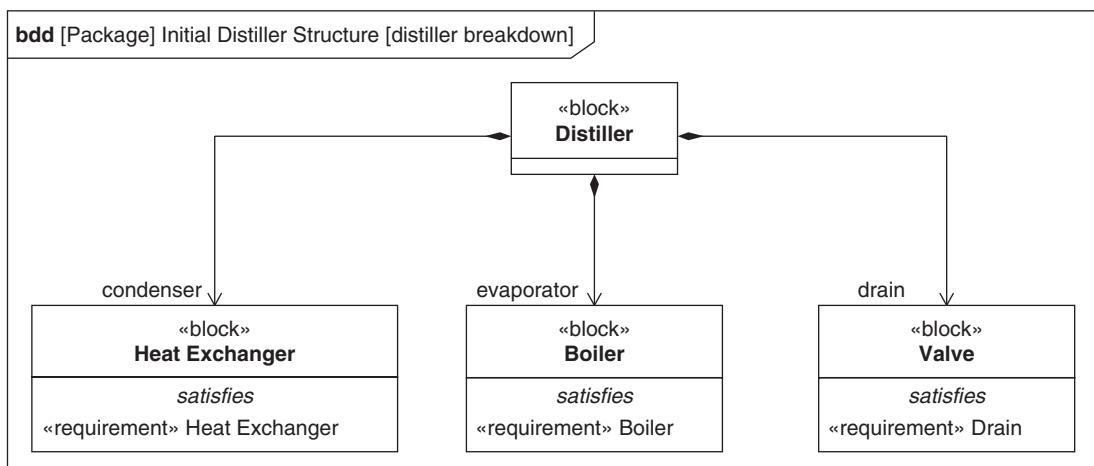


FIGURE 15.17

Initial structural hierarchy of the distiller.

The user-defined diagram name for this block definition diagram is *distiller breakdown*, to differentiate it from any other block definition diagram that designates the same *Distiller Structure* package for its diagram frame. This diagram shows the block named *Distiller*, which is composed of a block named *Heat Exchanger*, a block named *Boiler*, and a block named *Valve*. The composition relationship shows that the *Distiller* is composed of one *Heat Exchanger* that fulfills the role *condenser*, one *Boiler* that fulfills the role *evaporator*, and one *Valve* that fulfills the role *drain*.

The requirements described in Section 15.4 included requirements that were to be satisfied by various *Distiller* components. In particular, the requirement id *S2.0 Heat Exchanger*, id *S3.0 Boiler*, and id *S4.0 Drain* clearly state the customer's requirements for these elements of the distiller system. Since the blocks *Heat Exchanger*, *Boiler*, and *Drain* are intended to satisfy these requirements, a satisfy relationship can be established between the blocks and the requirement as shown using the requirements compartment notation (refer to Chapter 12 for details on the requirements compartment notation). As an example, the compartment notation can be read as “*Boiler* satisfies the «requirement» *Boiler*.”

15.6.2 Defining the Ports on the Blocks

An internal block diagram can be developed based on the block definition diagram to show how parts are connected to one another. However, before doing this, the blocks on the block definition diagram are further elaborated by identifying the ports on the blocks and their definitions so that the ports can be connected in the internal block diagram.

The ports are identified on the blocks on the block definition diagram in Figure 15.18. In this case all the ports are defined as unidirectional atomic flow ports, meaning that only one type of item flows through the port, and in only one direction. The flow port labels describe the flows from the perspective of the block rather than the perspective of the *Distiller*. For example, the *Valve* has flow ports for *in : Fluid* and *out : Fluid*, which generally apply to all uses of a two-port valve. The *Heat Exchanger* has a cold loop (*c in* and *c out*) and a hot loop (*b in* and *b out*); both features are common to all counterflow heat exchangers. Thinking through the port configurations and labels facilitates the interface definitions that are further specified on the internal block diagram.

In Figure 15.18, the constraints compartment in the *Heat Exchanger* specifies a set of constraints on the temperature of items flowing through each port. This further defines the *Heat Exchanger*. Alternatively, these constraints could have been applied to local usage by using property-specific types, as described in Chapter 6, in which case the constraints would not apply to the definition but to the use. The constraints of temperature and pressure can be validated against the results of the performance analysis to ensure the heat exchanger is fit for use in this application. In addition, an engineer can assess whether a particular off-the-shelf heat exchanger, which meets these constraints, can be procured.

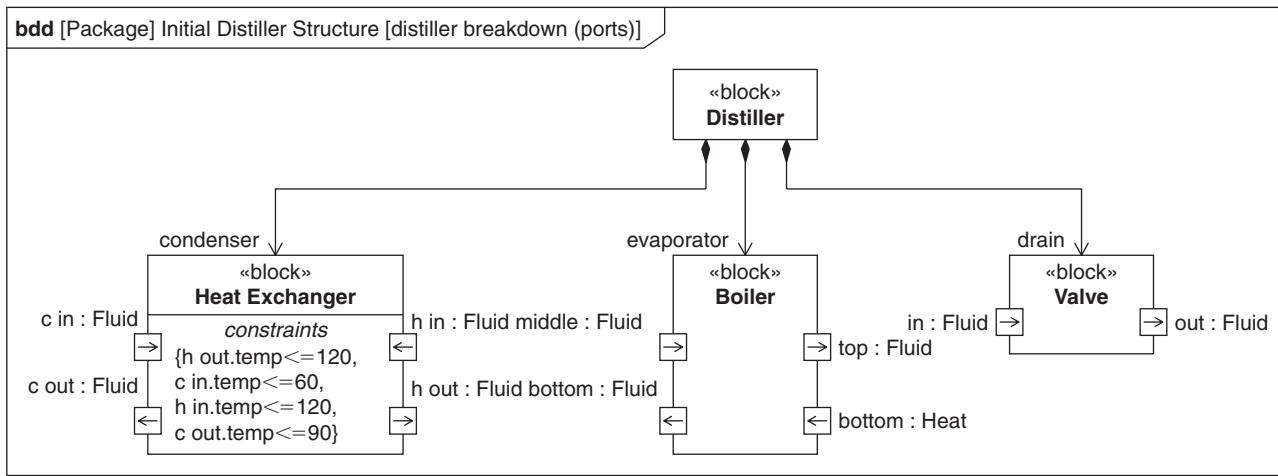


FIGURE 15.18

Distiller hierarchy with flow ports defined.

The next step is to show usage of these Blocks in the context of the distiller system on an internal block diagram, including the connections and flows between them.

15.6.3 Creating the Internal Block Diagram with Parts, Connectors, and Item Flows

Figure 15.19 is an internal block diagram for the *Distiller* system. The diagram header identifies the enclosing block as the *Distiller*. The user-defined diagram name is *1. distiller block diagram (initial)*. The parts represent how the blocks are used in the *Distiller* context and have the same role names as were shown on the block definition diagram. The flow ports are consistent with their definition on the block definition diagram.

The additional information on the internal block diagram that was not on the block definition diagram is the representation of the connectors between the parts and the item flows on the connectors. The connectors connect the ports and reflect the distiller's internal structure. The item flows represent what items flow across the connector and in and out of the ports.

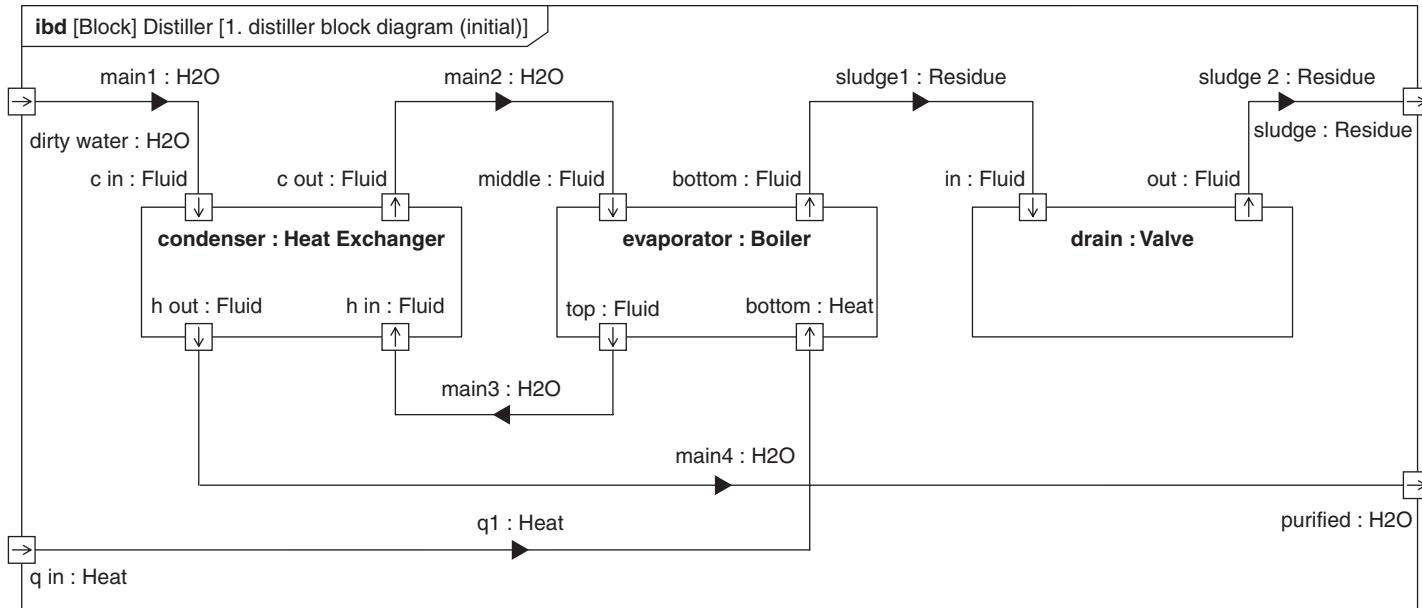
As discussed in Chapter 6, item flows have associated properties (item properties). The item flow defines the direction of a flow on the connector, and the item property represents the thing that is flowing in the context of the enclosing block (i.e., the *Distiller*). The item property is typed by a block in this case.

In this example, a naming convention for item properties has been used to identify the items flowing through the system. The main H₂O flow has been designated starting with *main*: *main1* is the flow of H₂O into the system and into the cold loop of the heat exchanger; *main2* is the flow of H₂O out of the cold loop of the heat exchanger and into the boiler; *main3* is the flow of H₂O (steam) out of the boiler and into the hot loop of the heat exchanger; and *main4* is the flow of H₂O (condensate, or pure water) out of the heat exchanger and out of the system. The flow of sludge has been similarly designated: *sludge1* out of the boiler and into the drain valve, and *sludge2* out of the drain valve and out of the system. The only additional flow is *q1*, which represents heat flowing into the system and into the boiler.

At this point, the distiller system's structure has been expressed in definition on the block definition diagram and in usage on the internal block diagram, along with the physical flows. It is now appropriate to further elaborate the allocation of behavior to structure that was initially specified in the Figure 15.16 activity diagram with swimlanes.

15.6.4 Allocation of Actions and Flows

Figure 15.20 is an internal block diagram, identical to what was previously created, but enhanced to include an allocation compartment on each part. The information in the allocation compartments is consistent with the allocation relationship from the activity diagram in Figure 15.16. As previously discussed, placing an action in an allocate activity partition on the activity diagram resulted in

**FIGURE 15.19**

Distiller internal structure with item flows.

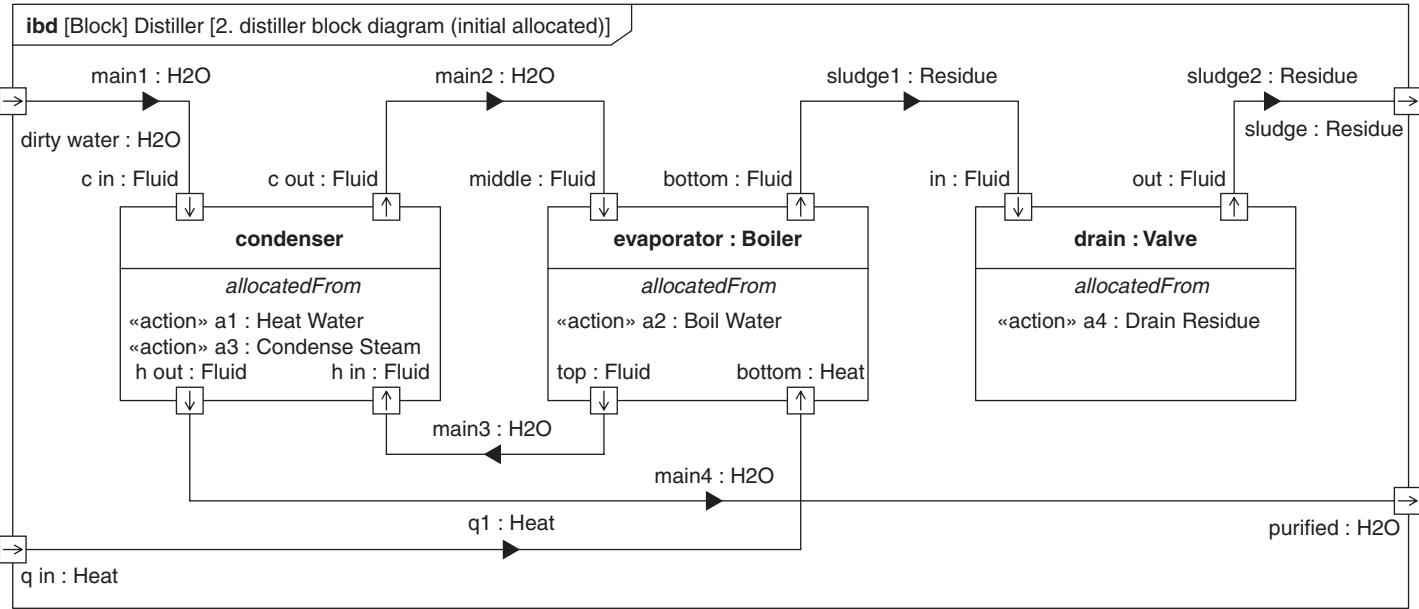


FIGURE 15.20

Distiller internal structure showing allocation of actions.

an allocate relationship between the action and the part represented by the partition. These allocation relationships are explicitly depicted in the allocation compartments; *allocatedFrom* indicates the direction of the relationship—namely, *from* the elements specified in the compartment *to* the part.

In addition to allocating actions to parts, it is also appropriate to reconcile the flow in the behavior model with the flow in the structural model. In this particular example, it was decided to allocate the object flows from the activity model to the item properties of the item flows in the structural model. This is in anticipation that the heat balance analysis will focus on these item properties. As shown in Figure 15.21, each object flow on the activity diagram is allocated to specific item properties. To avoid ambiguity, both object flows and item properties are uniquely named. For example, Figure 15.21 shows that object flow *of1* has been allocated to item property *main1*, *of2* to *main2*, *of3* to *main3*, and so on, with each allocation being uniquely identifiable.

As discussed in Section 13.8 in Chapter 13, allocation of object flow to item flow/item property cannot be unambiguously represented on internal block diagrams. If a callout were used to show the object flow allocated to a black triangle on an internal block diagram, it is not clear whether this is meant to represent allocation to the item flow, the item property, or the type of the item property. To avoid this ambiguity, the example uses a matrix to depict both functional and flow allocation, as shown in Figure 15.22. The arrows in the matrix represent the direction of the allocation relationship.

15.7 Analyzing Performance

Now that system behavior has been allocated to system structure, the implications on system performance are considered next.

15.7.1 Item Flow Heat Balance Analysis

The key aspect of distiller performance is the appropriate balance of water and heat flow through the system. To evaluate the flow balance, the analysis focuses on the physical flow of water and heat, as expressed by item flows on the internal block diagram. An alternative approach may have been to analyze the object flows on the activity diagram, but the object flows have been uniquely allocated to the item flows, so the approach is equivalent. The feasibility of the design can be assessed by analyzing the mass flow rate of the H₂O through the system, and analyzing the heat flow required to heat the H₂O and the associated phase changes. This analysis is simplified by the fact that the entire system is isobaric; that is, the pressure throughout the system is assumed to be atmospheric.

Figure 15.23 is a parametric diagram of the *Distiller* block, representing simple mathematical relationships between the physical flows. In this simple example, it was decided to apply the constraints directly to the *Distiller* block rather than creating a separate analysis context block, as described in Chapter 7. The

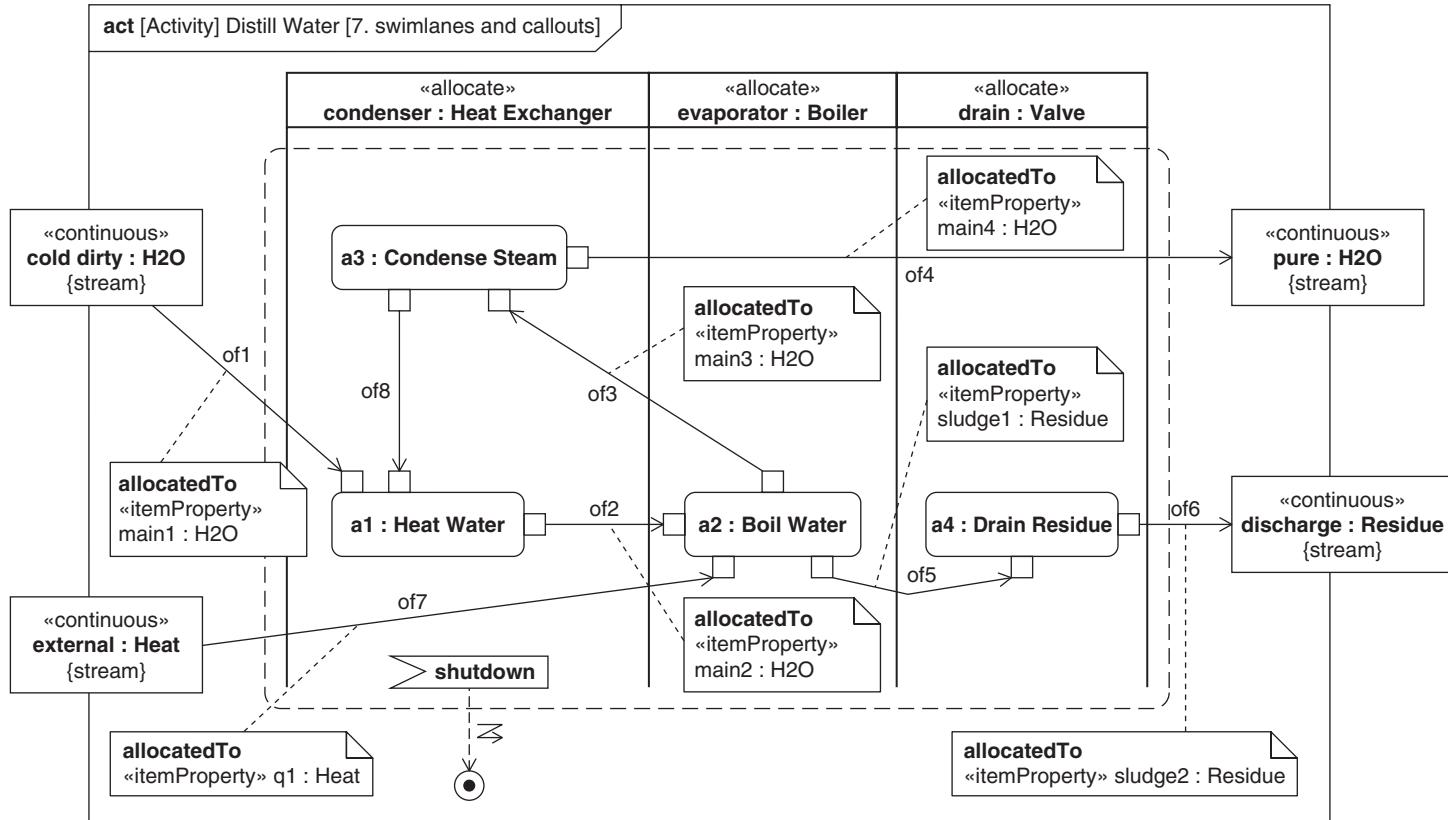
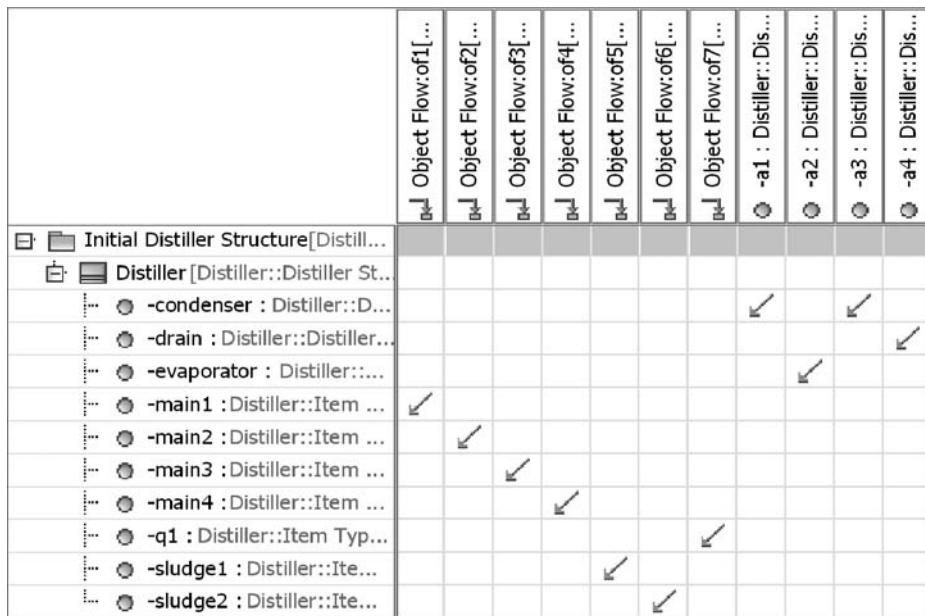


FIGURE 15.21

Initiating functional flow allocation (pin names have been elided).

**FIGURE 15.22**

Functional and flow allocation from behavior to structure.

six square boxes around the outside of the diagram (*main1 : H₂O*, *main2 : H₂O*, and so on) represent the item properties previously identified on the internal block diagram. Each item property can have associated value properties unique to its usage, such as temperature and mass flow rate. Specific heat and latent heat are common, invariant properties of water that also need to be considered in the analysis. The three round-cornered boxes in the center of the diagram represent constraint properties of the *Distiller*; each has a corresponding constraint expressed as a mathematical formula. This constraint is identified by curly brackets ({}), and can either be displayed on the constraint property directly or shown in a separate constraint callout.

Based on the topology of the distiller, the mass flow rate of the water input (*main1*) has to be equal to the mass flow rate of the water output of the heat exchanger (*main2*) because there is nowhere else for it to go. This equivalence is depicted in Figure 15.23 by directly binding the *mass flow rate* value property of the *main1 : H₂O* item property and the *mass flow rate* value property of the *main2 : H₂O* item property. Likewise, the mass flow rate of the steam output from the boiler (*main3*) must equal the mass flow rate of the water output from the *Distiller* (*main4*), and the same kind of binding is used.

The system needs to heat water and condense steam at the same time as specified in the activity diagram. The single-phase heat transfer equation, which is applied when heating liquid water, relates mass flow rate, change in temperature, and specific heat to heat flow (*q rate*). Note that the constraint *s1 : Single Phase*

par Distiller Isobaric Heat Balance [composition of equations]

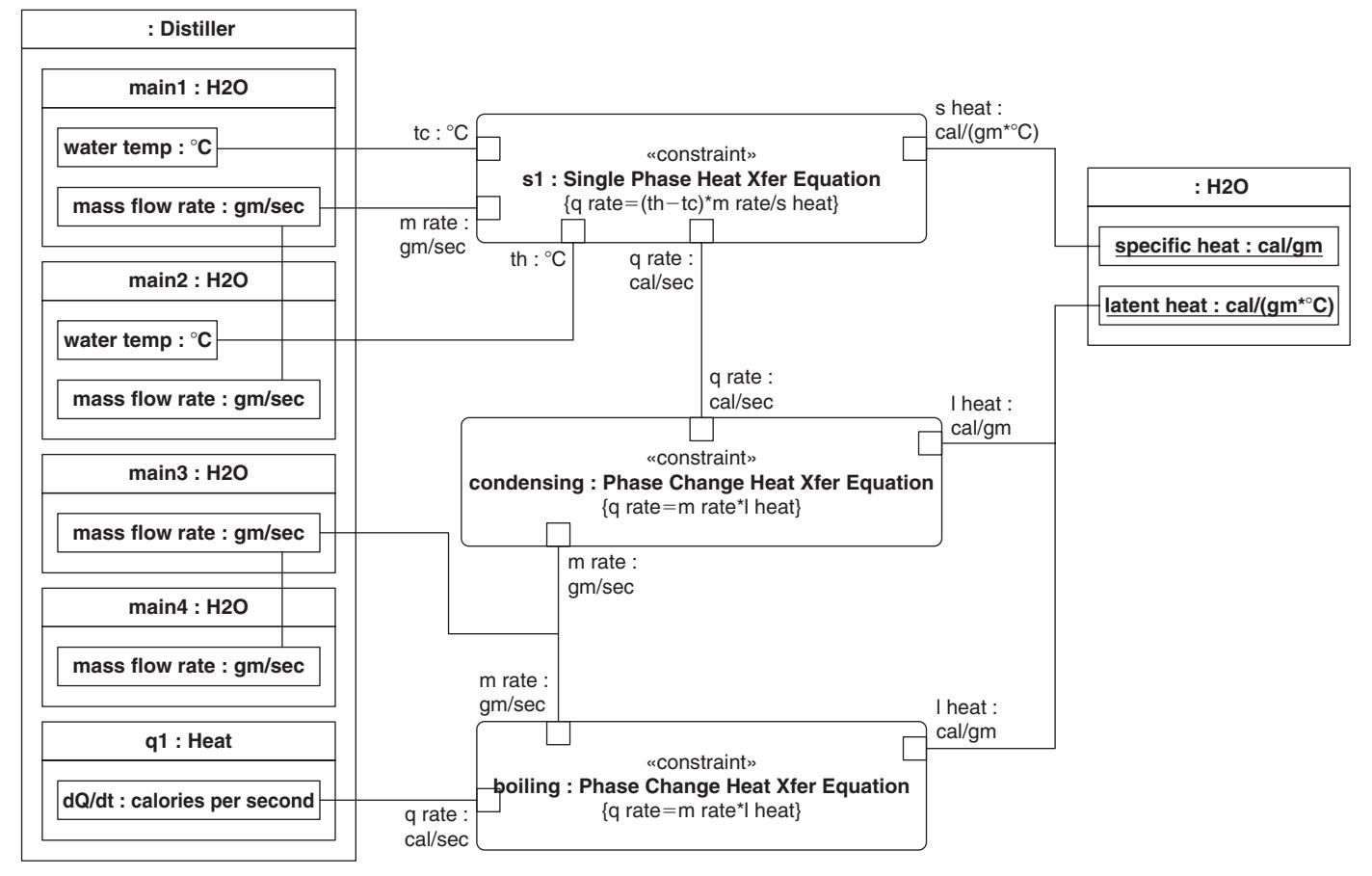


FIGURE 15.23

Defining parametric relationships as a prelude to analysis.

Heat Xfer Equation shows each of these parameters in small square boxes. Binding connectors are used to bind the value properties associated with the *main1* and *main2* mass flow rate and temperature and the specific heat of water to the parameters of this constraint. The *q rate* parameters in different constraints are bound directly to one another, as opposed to being bound to value properties of the item properties. The *q rate* from *condensing : Phase Change Heat Xfer Equation* is bound to the *q rate* for *s1 : Single Phase Heat Xfer Equation*, since the energy used to heat the water comes from condensing steam.

A simple phase change equation is used to determine how much heat needs to be extracted for a given mass flow rate of steam. In this example, the constraint block, *Phase Change Heat Xfer Equation*, is used both for condensing steam and for boiling water. For convenience, this equation is defined only once as a constraint block, and it is used to type the two constraints: *condensing* and *boiling*. Note how both *condensing* and *boiling* constraints have identical parameters but are bound to different properties. Also note that *specific heat* and *latent heat* are invariant properties of *H₂O* and are thus shown underlined.

This parametric diagram defines the mathematical relationships between properties, but it does not specifically define the analysis to be performed. It explicitly relates properties of the items that flow through the distiller. The next step is to perform the analysis by evaluating the equations.

15.7.2 Resolving Heat Balance

The equations and value properties expressed in the parametric diagram were manually entered into a spreadsheet to perform the computation. Figure 15.24 is a table captured from this analysis. The analysis started by assuming unity flow rate into the evaporator (*main2 : H₂O into evap*) and then determining how much water is required to flow through the condenser. The conclusions indicate that to remove enough heat to condense the steam, almost seven times more water mass needs to flow into the system than flows out of it! In the current design, there is no place for that water to go except into the boiler, which will then overflow. This is not a feasible steady-state solution and requires modification to the design.

15.8 Modifying the Original Design

Since analysis has revealed a fundamental flaw in the original distiller design, this section will describe modifications to the design to achieve adequate performance.

15.8.1 Updating Behavior

As shown in the modified activity diagram in Figure 15.25, the design is modified by adding another part called *diverter assembly*: represented as an allocate activity partition, with the action to divert the water called *a5 : Divert Feed*.

table[Package]IsobaricHeatBalance 1 [Results of Isobaric Heat Balance]						
specific heat cal/gm-°C		1				
latent heat cal/cm		540				
satisfies «requirement» WaterSpecificHeat						
satisfies «requirement» WaterHeatOfVaporization						
satisfies «requirement» WaterInitialTemp						
mass flow rate gm/sec	6.8	6.8	1	1	1	
temp °C	20	100	100	100	100	
dQ/dt cooling water cal/sec	540					
dQ/dt steam condensate cal/sec	540					
condenser efficiency	1					
heat deficit	0					
dQ/dt condensate steam cal/sec	540					
boiler efficiency	1					
dQ/dt in boiler cal/sec	540					

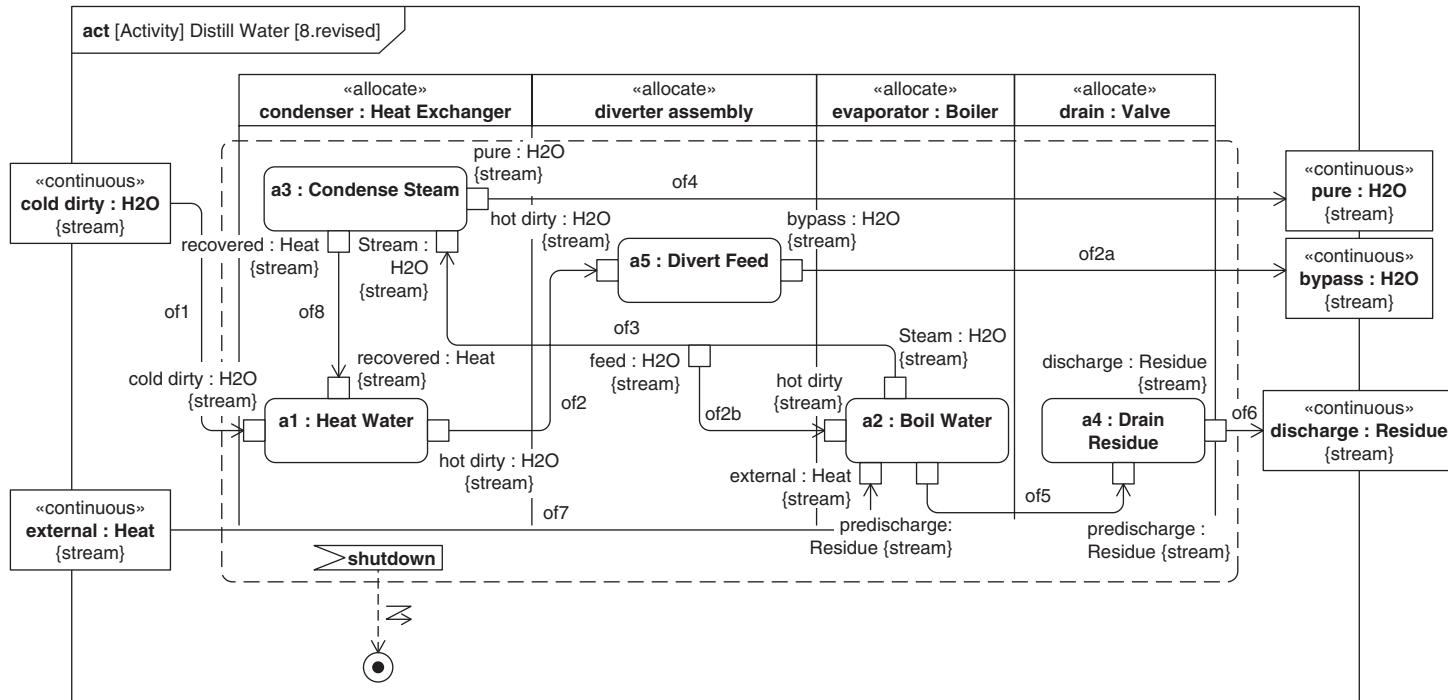
Note: Cooling water needs to have 6.75x flow of steam!
Need bypass between hx_water_out and bx_water_in!

FIGURE 15.24

Analysis reveals heat imbalance in initial design.

15.8.2 Updating Allocation and Structure

The allocate activity partition corresponds to a new part, which includes another usage of the previously defined *Valve* block, that has been added to the system. This new part, its internal structure, and the associated flows are shown in the internal block diagram in Figure 15.26. This assembly is decomposed into a tee fitting to divert most of the flow out of the system, and a valve to throttle the water going into the boiler. The *diverter assembly*: is an untyped part that is a simple collection of parts; it is not defined by a block. Note also the use of nested connectors to avoid the need to use flow ports on the *diverter assembly*.

**FIGURE 15.25**

Revising behavior to accommodate diverting feed water.

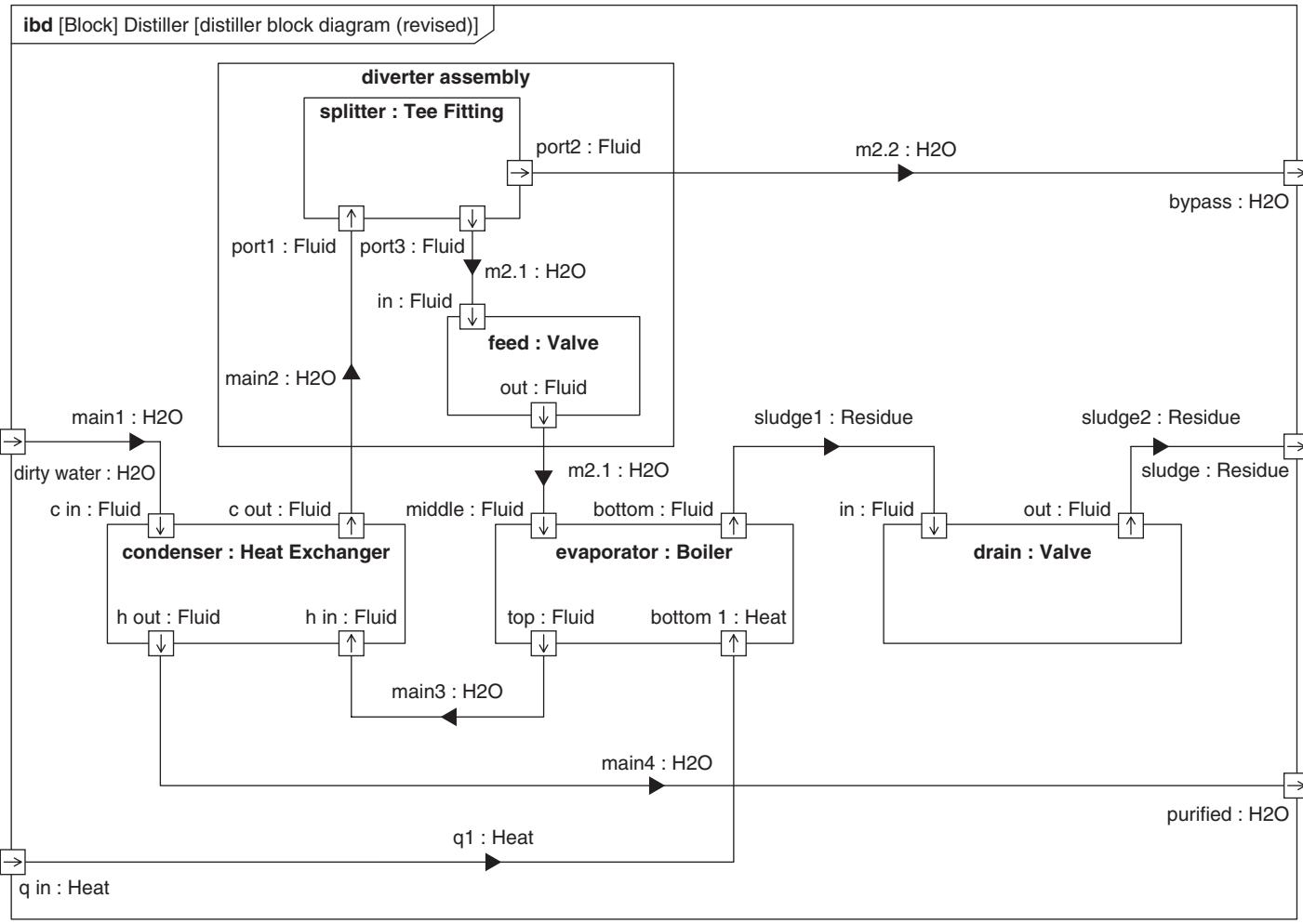


FIGURE 15.26

Revised distiller internal structure with flow diverter.

This modified design enables the *feed : Valve* to be throttled so that the boiler does not overflow, and yet retain enough water flowing through the heat exchanger to condense the steam. Note also the reuse of the block *Valve*. The *drain : Valve* and the *feed : Valve* each have two ports, both of which are defined the same but connected differently. Although not shown here, the block definition diagram is also updated to reflect another composite association between *Distiller* and *Valve* with the new part name.

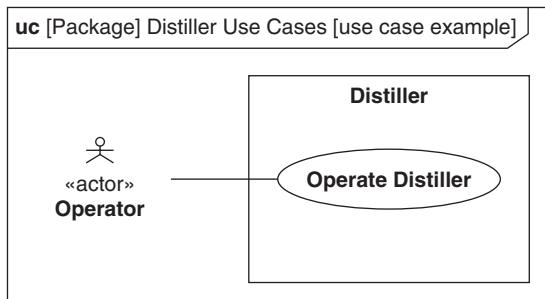


FIGURE 15.27

Defining operator interface using a use case.

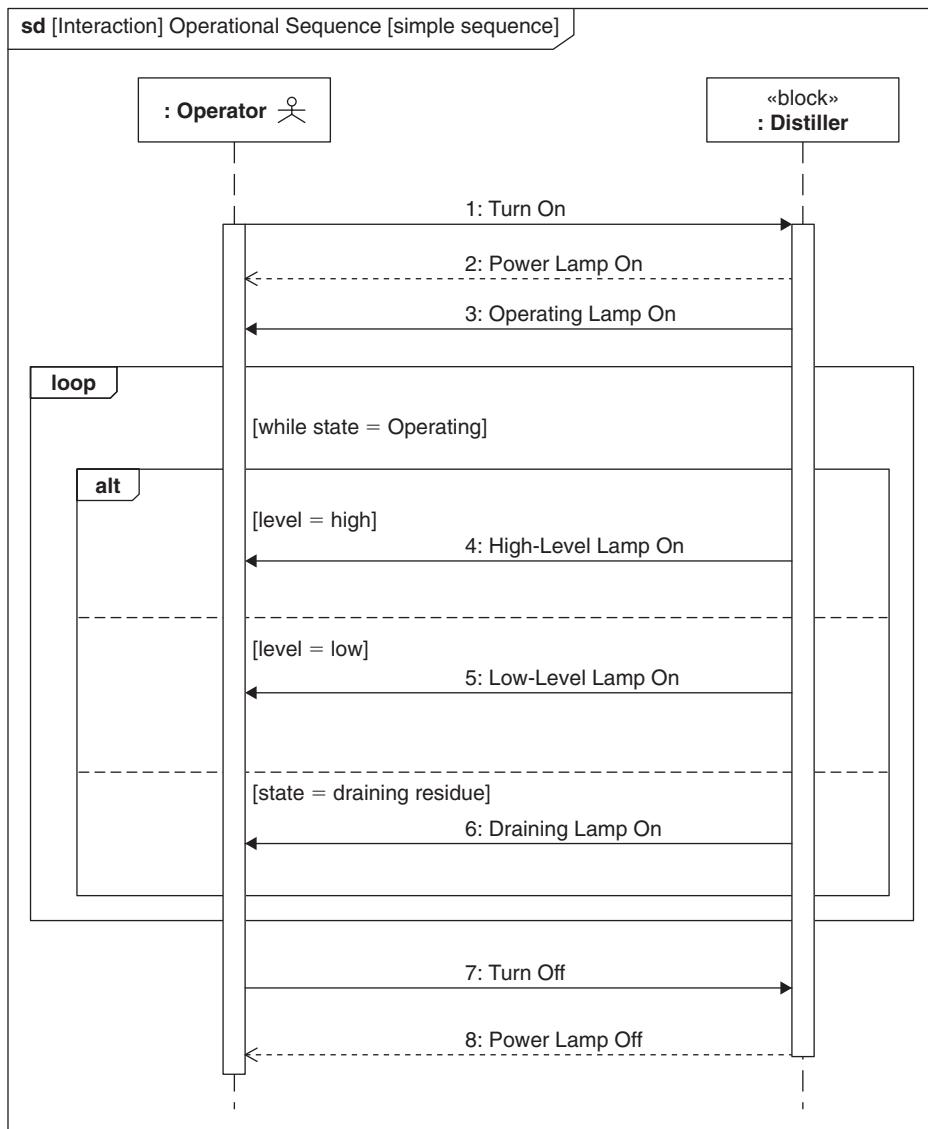
15.8.3 Controlling the Distiller and the User Interaction

Up to this point, the design has not considered how or if a user interacts with the *Distiller*. The following steps update the model to reflect how the operator interacts with *Distiller*, along with other considerations for startup and shutdown of the system.

Figure 15.27 shows a use case diagram that includes model elements contained within the *Distiller Use Cases* package.

It may be appropriate at this point to develop a textual use case description to describe *Operate Distiller*, but the example here does not include this step. The sequence diagram that realizes the use case is shown in Figure 15.28; it describes the expected interaction between the user and *Distiller* when operating the distiller system.

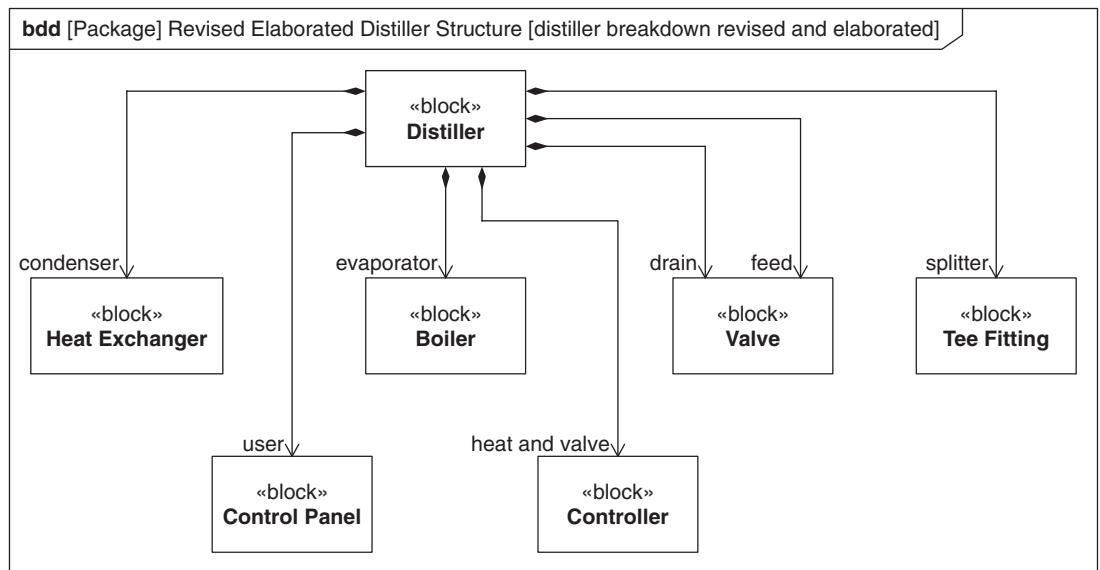
The *Operator* starts by turning the *Distiller* on and observes a *Power Lamp On*. When the *Distiller* reaches operating temperature, the *Operator* observes the *Operating Lamp On*; then the distiller cycles as it produces distilled water. The *Operator* turns the *Distiller* off, and the *Power Lamp Off* signal is returned by the *Distiller*. This interaction indicates that further changes to the design must be made to include the parts needed for the user to provide inputs to the system, the lamps, and the mechanism for automatically controlling the distiller.

**FIGURE 15.28**

Defining operator interaction using a sequence diagram.

15.8.4 Developing a User Interface and a Controller

Figure 15.29 is a block definition diagram, and Figure 15.30 is an internal block diagram that reflects the update to the design to realize the use case. A control panel has been added with the lamps that the operator observes. A controller has

**FIGURE 15.29**

Distiller structural hierarchy with controller and user interface.

been added to ensure that the valves are operated in the proper sequence and that the lamps are turned on and off.

Power input is provided to the heaters in the *Boiler* to convert electrical power to heat. It makes sense to use the controller to provide power to the *Boiler*. A flow specification can now be used to describe the kind of signals expected to pass between the *Controller* and the *Boiler*. The flow specification may include the position of float switches in the boiler to indicate whether the level is high or low.

Figure 15.31 shows the flow specification *Boiler Signals*. Note that it uses two flow properties, *control* and *status*, and the direction is appropriate for the *heat and valve : Controller* in Figure 15.30. The *evaporator : Boiler* uses a conjugate flow port with the same flow specification as the flow port on the *Controller*. The conjugate reverses the direction of the flow properties and makes the connection compatible.

15.8.5 Startup and Shutdown Considerations

Since the system now uses a controller, the startup and shutdown and other aspects of system control can be represented as a state machine diagram for the *Controller*, as shown in Figure 15.32. The states and transitions in the diagram were identified by examining the sequence diagram associated with the *Operate Distiller* use case.

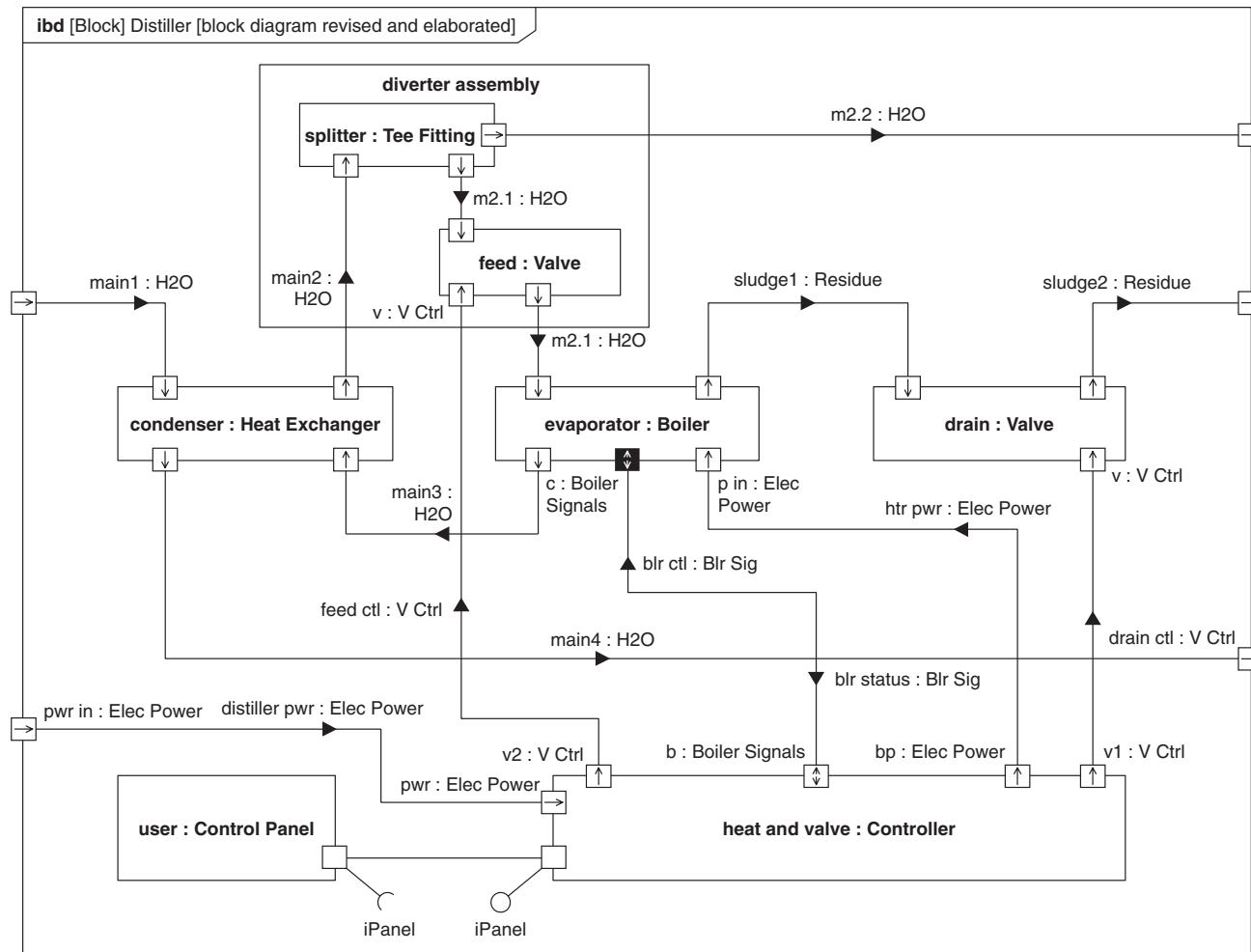
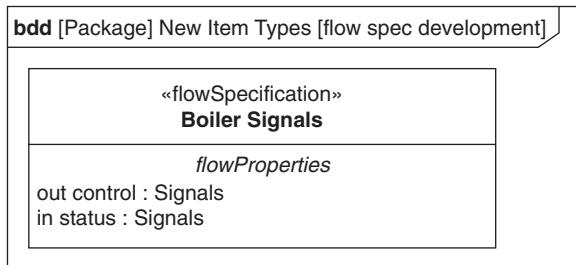


FIGURE 15.30

Distiller internal structure with controller and user interface.

**FIGURE 15.31**

Flow specification for boiler signals.

Starting with the *Distiller* in the *Off* state, in which it is cold and dry, a number of things will have to happen before it begins distilling and producing water. The first step is to fill the boiler. While in the *Filling* state, the *feed : Valve* opens. As soon as the water level in the *Boiler* is adequate to cover the heater coils, the heater can be turned on without damage. The system can now enter the *Warming Up* state, where the boiler heaters are turned on and the boiler begins warming up.

Once the boiler temperature reaches 100°C, the system enters the *Operating* state. In this state, the boiler heaters are still on, but two substates, *Controlling Boiler Level* and *Controlling Residue*, occur in parallel. In this example, control of residue relies on a simple timer to transition between the *Building Up Residue* substate when the *drain : Valve* is closed and the *Purging Residue* substate when the *drain : Valve* is open and dumping the residue. In essence, this state machine periodically blows down the boiler to make sure that not too much sludge builds up.

When controlling the water level in the *Boiler*, one of three substates exist: either *Level OK*, in which case the *drain : Valve* and *feed : Valve* need to both be closed; *Level Low*, which requires more water, so the *feed : Valve* needs to be open; or *Level High*, where the *drain : Valve* needs to be open.

To turn off the *Distiller*, the operator should not just cut the power and walk away. It is necessary to go through a shutdown procedure; otherwise, corrosion will severely limit the lifespan of the *Distiller*. The first step in this procedure is to cool off the system. In the *Cooling Off* state, the heaters are turned off and the *feed : Valve* and the *drain : Valve* opened, allowing cool water to flow freely through the entire system. Once the boiler temperature reaches a safe level, the *Boiler* needs to be drained. In the *Draining* state, the *feed : Valve* is shut while the *drain : Valve* remains open, and all water is drained out of the *Boiler*. Once the *Boiler* is empty, the *Distiller* system can safely be turned off.

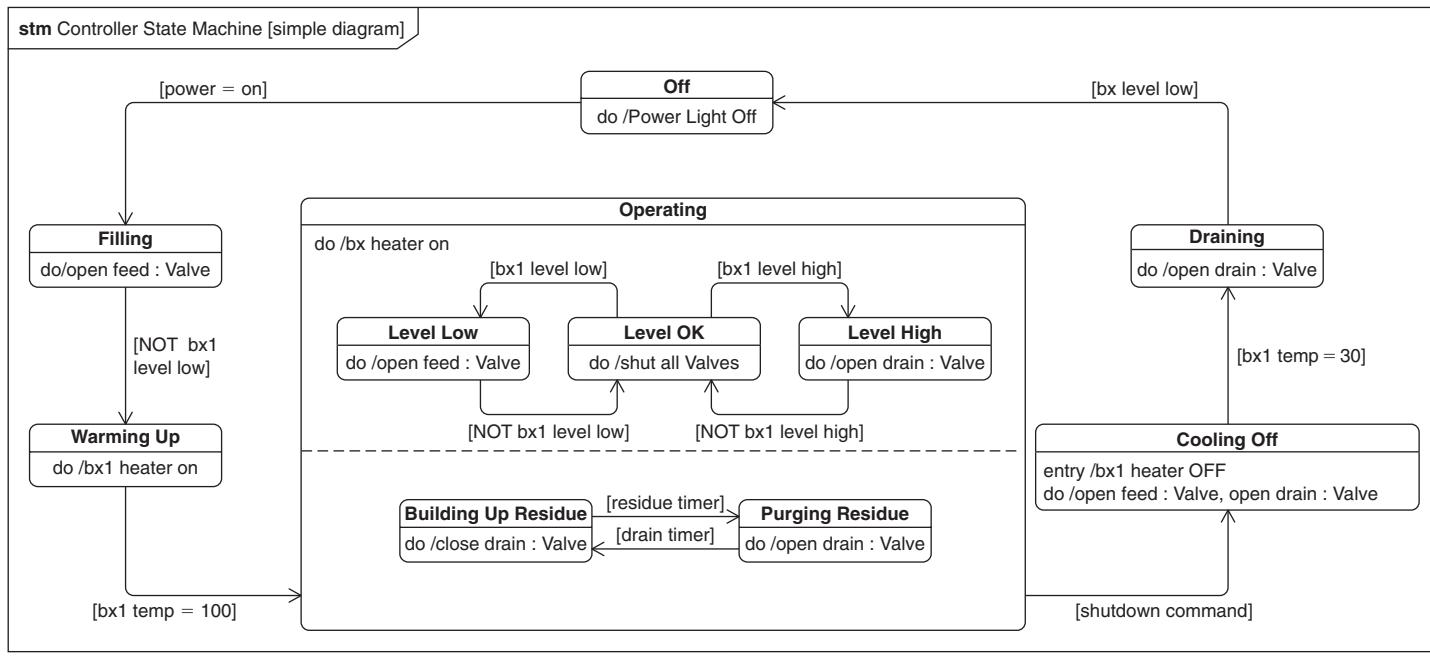


FIGURE 15.32

Controller state machine for distiller.

15.9 Summary

This example shows how SysML can be used to model a system with a traditional functional analysis approach. The problem also illustrates the application of modeling physical systems with limited software functionality. Examples of each SysML diagram are used to support the specification, design, and analysis, along with fundamental SysML language concepts such as the distinction between definition and use.

15.10 Questions

The following questions may best be addressed in a classroom or group project environment.

1. The customer has introduced this new requirement: “The water distiller shall be able to operate at least 2 meters vertically above the source of dirty water.” Show the impact of this new requirement on the system design, as expressed in each of the following modeling artifacts.
 - a. Requirement diagram (relate new requirement to existing requirements)
 - b. Activity diagram (define and incorporate new activities to support the new requirement)
 - c. Block definition diagram (define and incorporate new blocks to support the new requirement)
 - d. Internal block diagram (define flows and interfaces to any new parts necessary to support the new requirement, and any functional and flow allocations from the activity diagram)
 - e. Parametric diagram (describe how the heat balance is affected by this new requirement)
 - f. Use case diagram (describe any changes to the operational scenario)
 - g. Sequence diagram (elaborate any changes to the *Operate Distiller* use case)
 - h. State machine diagram (describe how the *Controller* state machine would be affected by the preceding design changes)
2. Discuss the applicability and physical significance of control flows in the distiller activity model, as shown on Figures 15.9, 15.10, and 15.13. In which situations are control flows useful representations of behavior, and in which ways can they be misleading?

Residential Security System Example Using the Object-Oriented Systems Engineering Method

16

The example in this chapter describes the application of SysML to the development of a residential security system using the Object-Oriented Systems Engineering Method (OOSEM). It demonstrates how SysML can be used with a top-down, scenario-driven process to analyze, specify, design, and verify the system. A scaled-down version of this method was introduced as part of the automobile design example in Chapter 3.

The application of OOSEM, along with the functional analysis method in Chapter 15, are examples of how SysML is applied; but SysML can be applied with other methods as well. The intent of this chapter is, however, to provide a method that readers can adapt to their application to meet their needs.

This chapter begins with a brief introduction to the method and how it fits into the context of the overall development process, and then it shows how OOSEM is applied to the residential security example. The reader should refer to the language description in Part II for the foundational language concepts.

16.1 Method Overview

This section provides an introduction to OOSEM including the motivation and background for the method, a high-level summary of the development process that provides the context for OOSEM, and a summary of the OOSEM system specification and design process within the development process.

16.1.1 Motivation and Background

OOSEM is a top-down, scenario-driven process that uses SysML to support the analysis, specification, design, and verification of systems. The process leverages object-oriented concepts and other modeling techniques to help architect more flexible and extensible systems that can accommodate evolving technology and changing requirements. OOSEM is also intended to ease integration with object-oriented software development, hardware development, and test processes.

In OOSEM and other model-based systems engineering approaches, the system model is a primary output of the design process. The system model artifacts

represent the system's multiple facets such as its behavior, structure, and properties. For a model to have integrity, the various facets must provide a consistent representation of the system, as described in Chapter 2.

OOSEM includes fundamental systems engineering activities such as needs analysis, requirements analysis, architecture, trade studies and analysis, and verification. It has similarities with other methods such as the Harmony process [6, 7] and the Rational Unified Process for Systems Engineering (RUP SE) [9, 10], which also apply a top-down, scenario-driven approach that leverages SysML as the modeling language. OOSEM includes various modeling techniques, such as causal analysis, logical decomposition, partitioning criteria, node distribution analysis, control strategies, and parametrics, to deal with a wide array of system concerns.

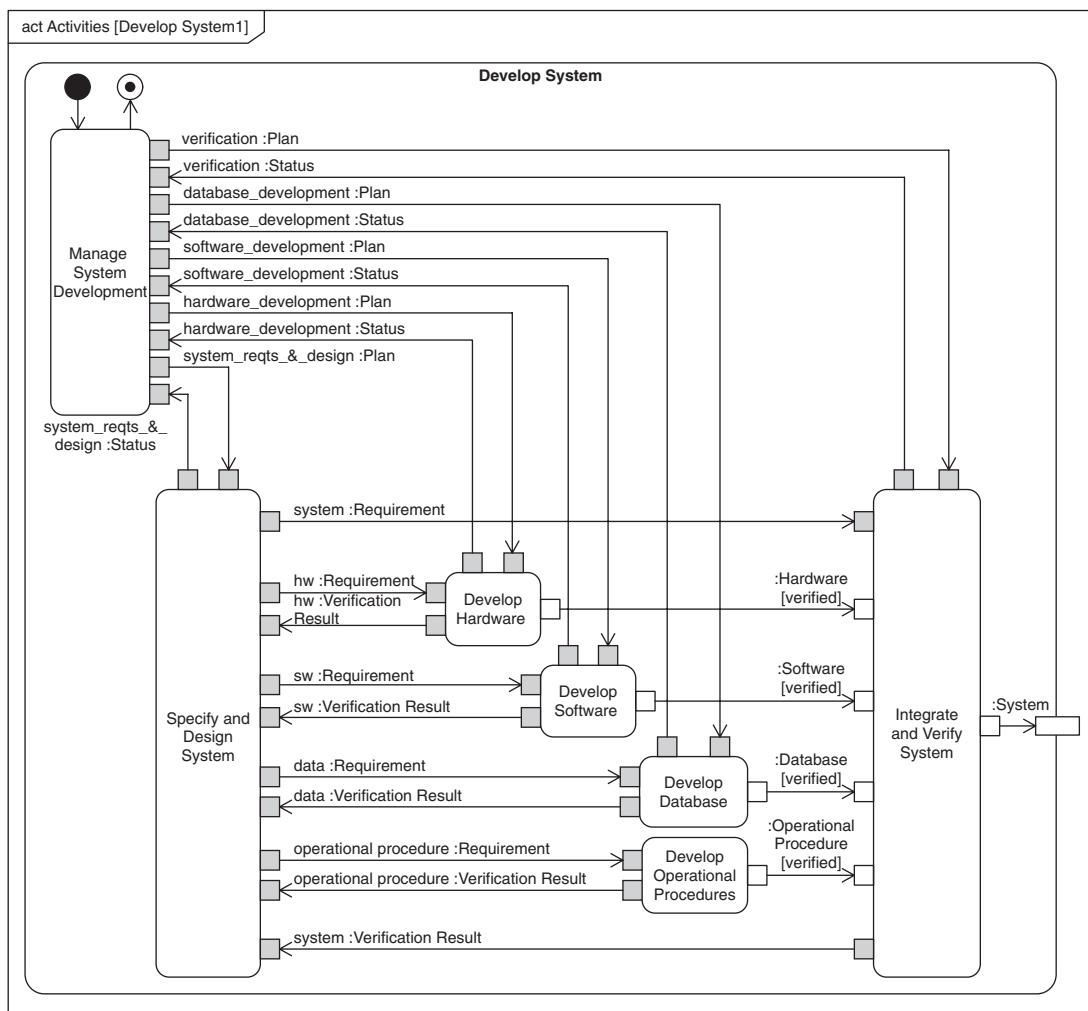
OOSEM was developed in 1998 [36, 37] and further evolved as part of a joint effort between Lockheed Martin Corporation and the Systems and Software Consortium (SSCI), which previously was the Software Productivity Consortium [8]. Early pilots were conducted to assess the feasibility of the method [38], and then it was further refined by the INCOSE OOSEM Working Group beginning in 2002. Tool support has been substantially improved for OOSEM with the adoption of the SysML specification beginning in 2006.

16.1.2 System Development Process Overview

The system life-cycle process includes processes for developing, producing, deploying, operating, supporting, and disposing the system. The successful output of the development process is a verified and validated system that satisfies operational requirements and capabilities and the other life-cycle requirements for production, deployment, support, and disposal.

OOSEM is part of a higher-level development process that was originally based on the Integrated Systems and Software Engineering Process (ISSEP) [39]. A modified version of this process, as it applies to OOSEM, is highlighted in Figure 16.1, and includes the management process, the system specification and design process, the next level development processes, and the system integration and verification process. This process can be applied recursively to multiple levels of a system's hierarchy that is similar to a Vee development process [40], where the specification and design process is applied to successively lower levels of the system hierarchy down the left side of the Vee, and the integration and test process is applied to successfully higher levels of the system hierarchy up the right side of the Vee. This development process is different from a typical Vee process in that it includes both management processes and technical processes at each level of the hierarchy.

Applying the specification and design process at each level results in the specification of elements at the next lower level of the system hierarchy. For example, applying the process at the system-of-systems (SoS) level results in the specification of one or more systems. Applying the process at the system level results in the specification of the system elements, and applying the process at the system-element level results in the specification of the components. The hardware and

**FIGURE 16.1**

System development process.

software development processes are then applied at the component level to analyze component requirements and design, implement, and test them.

The leaf level of the process is the level at which an element or component is procured or implemented. In the automobile design example in Chapter 3, if the automotive design team procures the engine, the team specifies the engine requirements and verifies that the engine satisfies the requirements. On the other hand, if the engine is subject to further design, the process is applied to the next level of engine design to specify the engine components.

The following subsections contain a high-level summary of each process shown in Figure 16.1.

Manage System Development

This process includes project planning, and controlling the execution of the work in accordance with the plan. Project control includes monitoring cost, schedule, and performance metrics to assess progress against the plan, managing risk, and controlling changes to the technical baseline.

The management process also includes selection of the life-cycle model, such as waterfall, incremental, or spiral, that defines the ordering of the activities. Use cases that are defined in the model provide units of functionality that can serve as an effective organizing principle for planning and controlling the scope of work to be accomplished for a particular development spiral or increment.

The management process also includes tailoring the standard process activities and artifacts to meet the project's needs. Tailoring depends on a variety of factors that may include the extent to which the system is a new design (i.e., unpreceded), the system size and complexity, the available time and resources, and the level of experience of the development team. As an example, a system design that is based on a prior design is generally constrained to include significant legacy or predefined commercial off-the-shelf (COTS) components. This can significantly impact which activities are performed and the ordering of the activities. The activities may include early characterization of the COTS components in parallel with other system specification and design activities. The design emphasis is placed on how the COTS components interact to achieve the system requirements, and which additional components are required to interface with the COTS components.

Additional tailoring of the process and its artifacts may be required for specific domains at each level of the system's hierarchy. For the automobile design example, the intermediate element level may require tailoring of the processes and artifacts to develop the power train, body, and steering assembly, including unique types of analysis.

Specify and Design System

This process is implemented by OOSEM, as summarized later in Section 16.1.3. The system specification and design process include activities to analyze the system requirements, define the system architecture, and allocate the system requirements to the next level of design. The next level of design implements the allocated requirements and verifies that the design satisfies the requirements and/or requests updates to the system design to reallocate the requirements as needed. In the residential security example in Section 16.2, the next level of design is assumed to be the component level, where the hardware, software, database, and operational procedures are developed. However, as stated previously, there may be intermediate "element" levels of the system hierarchy.

Develop Hardware, Software, Database, and Operational Procedures

This process includes analysis, specification, design, implementation, and verification of the components. For hardware components, implementation is accomplished by fabricating and/or constructing the component, and for software components, implementation includes coding the software. If there are multiple

intermediate levels of the system hierarchy prior to the component level, the development process in Figure 16.1 is applied recursively to each intermediate level.

Integrate and Verify System

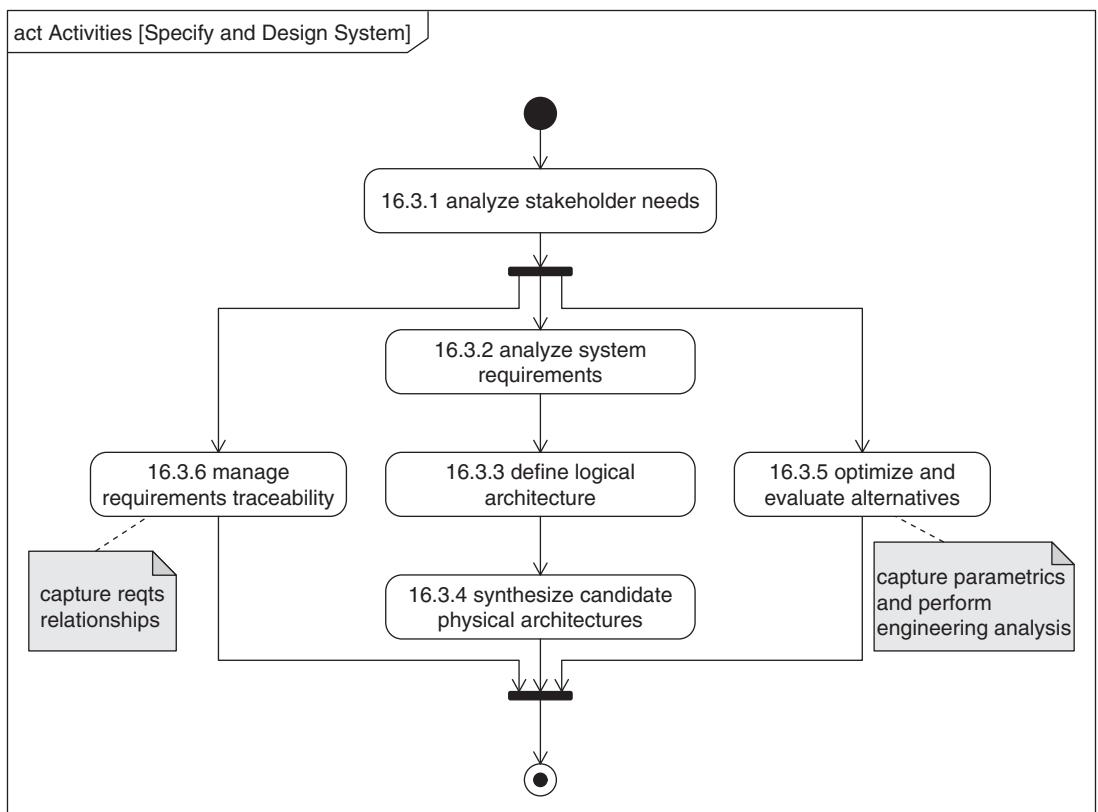
This process integrates the next lower level of system elements or components and verifies that the system design satisfies its requirements. The process includes developing verification plans, procedures, and methods (e.g., inspection, demonstration, analysis, testing), conducting the verification, analyzing the results, and generating the verification reports. OOSEM supports the right side of the Vee by specifying the test cases at each level of design and by integrating the design models into the next higher level of the Vee. Referring to the automobile design example in Chapter 3, the engine component design models are integrated into the engine design model, which in turn is integrated into the automobile system design model as part of the upside of the Vee process. This integrated model can be used to verify that the component designs satisfy their requirements, the engine design satisfies its requirements, and the automobile design satisfies its requirements.

16.1.3 OOSEM System Specification and Design Process

Figure 16.2 is a high-level summary of the OOSEM *Specify and Design System* process. The section in which each activity is addressed is also shown in the diagram. To simplify the process, it does not include the potential loops to reflect the process iterations, nor does it include the inputs and outputs from each activity.

The *Analyze Stakeholder Needs* activity characterizes the as-is system, its limitations and potential improvement areas, and specifies the mission requirements that the to-be system must support. The *Analyze System Requirements* activity specifies the system requirements in terms of its input and output responses and other black-box characteristics. The *Define Logical Architecture* activity decomposes the system into logical components and defines how the logical components interact to realize system requirements. The *Synthesize Candidate Physical Architectures* activity allocates the logical components to physical components that are implemented in hardware, software, data, and procedures. The *Optimize and Evaluate Alternatives* activity is invoked throughout the process to perform engineering analysis that supports system design trade studies and design optimization. The *Manage Requirements Traceability* activity is used to manage traceability from the mission-level requirements to the component requirements. Each of these activities is further elaborated later as part of the example.

The level of detail of the process documentation is tailored according to organizational and project needs. The next level of decomposition for each of the preceding activities is included in the residential security example in the next section. The documentation can be further elaborated to describe the detailed process description for creating each modeling artifact, such as a use case. In addition, the process flows can be further refined to reflect the design iterations and the flow of inputs and outputs. This level of detail is not included in any of the process flows in this example to simplify the process description.

**FIGURE 16.2**

OOSEM *Specify and Design System* process.

16.2 Residential Security Example Overview and Project Setup

The remainder of this chapter describes how OOSEM is applied to the residential security example.

16.2.1 Problem Background

A company called Security Systems Inc. has been providing residential security systems to the local area for many years. Their security systems are installed at local residences and are monitored by a central monitoring station (CMS). The system is intended to detect potential intruders. When an intruder is detected by the security system, operators at the CMS contact the local emergency dispatcher to dispatch police to the residence to intercept the intruder.

Security Systems Inc. had a successful business for many years. In the past several years, however, their sales have significantly dropped and many of their existing

customers have terminated their contracts in favor of competitors. It has become evident to the management of the company that their current system is becoming obsolete in terms of its capabilities, and that they must reestablish their market position. In particular, they have decided to launch a major initiative to develop an enhanced security system (ESS) that is intended to help regain their market share.

The Systems Engineering Integrated Team (SEIT) is responsible for providing technical management for the system development, including technical planning, risk management, managing the technical baseline, and conducting technical reviews. In addition, the SEIT includes team members who are responsible for the system requirements analysis, system architecture design, engineering analysis, and integration and verification of the ESS, as described in Section 1.4 in Chapter 1. The implementation teams are responsible for the system components. This includes analyzing the requirements allocated to the components by the SEIT, and designing, implementing, and verifying that the ESS components satisfy their requirements.

The SEIT selected an incremental development process as its life-cycle model. During the first increment, the SEIT established the incremental project plan and project infrastructure. The plan for the modeling effort included defining the modeling objectives; scoping the model to meet the objectives; selecting and tailoring the method and modeling conventions; selecting, acquiring, and installing the tools; defining the detailed schedule for the modeling activities; staffing the effort; and providing the necessary training.

The SEIT selected OOSEM as their model-based systems engineering method in conjunction with SysML as their graphical modeling language. This was based on the results of an earlier pilot project to assess how well the method and tools would support their needs (refer to discussion on deploying SysML in Chapter 18). They selected tools based on the tool selection criteria described in Chapter 17. The systems development environment includes SysML modeling tools, a UML-based software development environment; hardware design tools; performance analysis tools; testing tools; configuration management tools; a requirements management tool; and other project management tools for planning, scheduling, and risk management. The SEIT and selected members of other implementation teams received rigorous training in SysML, OOSEM, and the use of their selected tools.

The second increment focuses on a breadth-first design approach, which includes analysis of stakeholder needs, specifying the black-box system requirements, and evaluating and selecting the preferred system architecture for the proposed ESS solution. The follow-on increments focus on architecture refinement and implementing the components needed to achieve incremental capabilities that correspond to selected ESS use cases.

The example in this chapter is intended to describe the modeling activities for the second increment. During this increment, the ESS model is used to specify and validate system requirements, architect the solution, and allocate requirements to the ESS hardware, software, and data components, which are either developed by the implementation teams, or procured as COTS products. It is anticipated that there will be significant software and database development, but the hardware components, such as sensors, cameras, processors, and network devices, are primarily COTS. The ESS also requires development of new operational procedures

for the customer and central monitoring station operators that define how to interact with the system.

Only selected diagrams are included to illustrate the approach. In particular, the selected diagrams primarily relate to the intruder-monitoring thread.

16.2.2 Modeling Conventions and Standards

Modeling conventions and standards are required to ensure consistent representation and style across the model. This includes establishing naming conventions for each type of model element, such as packages, blocks, and activities, and for the diagram names. The conventions and standards also identify other stylistic aspects of the language, such as when to use uppercase versus lowercase and when to use spaces in the names. The conventions and standards should also account for tool-imposed constraints, such as limitations on the use of alphanumeric and special characters. It is also recommended that a template be established for each diagram type.

Table 16.1 contains a list of user-defined stereotypes for an OOSEM-specific profile of SysML that is used in this example. The approach for defining a profile is described in Chapter 14.

Further information is detailed in the following list.

Use of Upper- and Lowercase—Uppercase is used for the first letter of each word for all definitions/types, such as blocks and value types, and for packages and requirements, with a space between compound names that have more than one word. *Example:* Surveillance Camera: Lowercase is used for the first letter for parts, properties, item properties, actions, and states with a space between compound names that have more than one word. All other letters are lowercase. *Example:* surveillance camera.

Verb/Noun Form: The verb/noun form is used to name activities, actions, and use cases. *Example:* “Monitor Intruder.”

Pin Names on Activity Diagrams—in>Type Name and out>Type Name are often used. *Examples:* “in:Alert Status” and “out: Dispatch Request.”

Flow Port Names—Flow ports start with *fp* and standard ports start with *sp*.

Tool-Specific Notation—This chapter’s diagrams are generated from a modeling tool. Some of the notation may differ somewhat from the SysML specification that is described in Part II.

16.2.3 Model Organization

The model organization is recognized as a critical aspect of MBSE. The complexity of the system model can quickly overwhelm the users of the model and become intractable, particularly for large teams. This in turn can impact the ability of model developers to maintain a consistent model and the ability to maintain configuration management control of the model. Refer to Chapter 5 for considerations for how to organize the model with packages.

Table 16.1 OOSEM-Specific Profile of SysML User-Defined Stereotypes

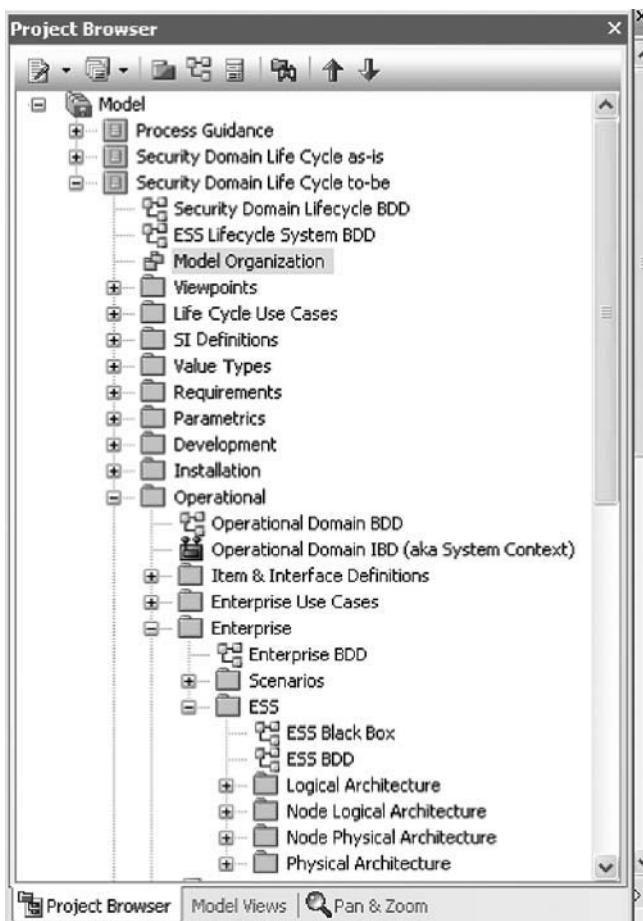
OOSEM Stereotype	Base Class
«composite state»	Block, Property
«configuration item»	Block, Property
«document»	Block, Part
«element»	Block, Part
«file»	Block, Part
«hardware»	Block, Part
«logical»	Block, Part
«node»	Block, Part
«node logical»	Block, Part
«node physical»	Block, Part
«operator»	Block, Part
«procedure»	Block, Part
«mop»	Property
«moe»	Property
«software»	Block, Part
«state»	Block, Property
«status»	Property
«store»	Property
«system of interest»	Block, Part

The OOSEM process includes a standard approach for how to organize the model that is defined by the package structure. Model organization includes a recursive package structure that mirrors the system hierarchy. A package is defined for each block, which is further decomposed to contain the model elements for the next level of decomposition. This package contains the block definition diagram and internal block diagram for the next level of decomposition, and a behavior package that specifies the collaboration among the parts at the next level of decomposition.

The model organization also includes other packages that are not nested within the system hierarchy packages. These packages include requirements, parametrics, input/output definitions, value types, and other model aspects that may be reused at multiple levels of the system hierarchy.

The model organization for this example is highlighted by the package structure shown in the browser view in Figure 16.3. The model shown at the top level of the browser contains three top-level packages called *Process Guidance*, *Security Domain Life Cycle as-is*, and *Security Domain Life Cycle to-be*.

The *Process Guidance* package provides a convenient mechanism to capture process issues, tool issues, and other process information that is identified by the systems engineering team throughout the modeling process. This information

**FIGURE 16.3**

ESS model organization (browser view).

should ultimately be reflected in updates to the organizational standard processes if the information is relevant across projects. For this example, the package also includes the process flows that describe the methodology, such as Figure 16.2. Alternatively, other process-modeling tools may be used to capture process information.

The *Security Domain Life Cycle as-is* package contains the parts of the model that characterize the current system and enterprise in sufficient detail to aid in understanding the limitations to be addressed by the to-be system. Parts of the as-is model may be reused in the to-be model.

The *Security Domain Life Cycle to-be* package contains elements of the model for the to-be enterprise and system. Several nested packages are contained within this package. The *Viewpoints* package contains the viewpoints and associated views for different ESS stakeholders. (Note: Viewpoints and views are described

in Chapter 5.) The *Life-Cycle Use Cases* package contains the use cases that span the systems life cycle. The *SI Definitions* package is an imported library that contains standard units and dimensions.

The *Value Types* package contains additional value types with units and dimensions that are added for use throughout the model. The *Requirements* package contains the requirements for the ESS system from mission-level requirements down to hardware and software component requirements. The requirements are often imported from a requirements management tool. The *Parametrics* package contains the parametric diagrams and associated model elements to support engineering analysis and trade studies.

There are also packages that correspond to other parts of the system's life cycle, including *Development*, *Installation*, *Operational*, and *Support* (not shown). Most of the elaboration of this model is contained within the *Operational* package, since the focus of this example is on the operational system design. The *Operational* package contains nested packages for *Item & Interface Definitions*, *Enterprise Use Cases*, and the *Enterprise*. The *Item & Interface Definitions* elaborate the inputs and outputs and the port definitions used throughout the model. The *Enterprise Use Cases* contain the primary mission use cases for the security system.

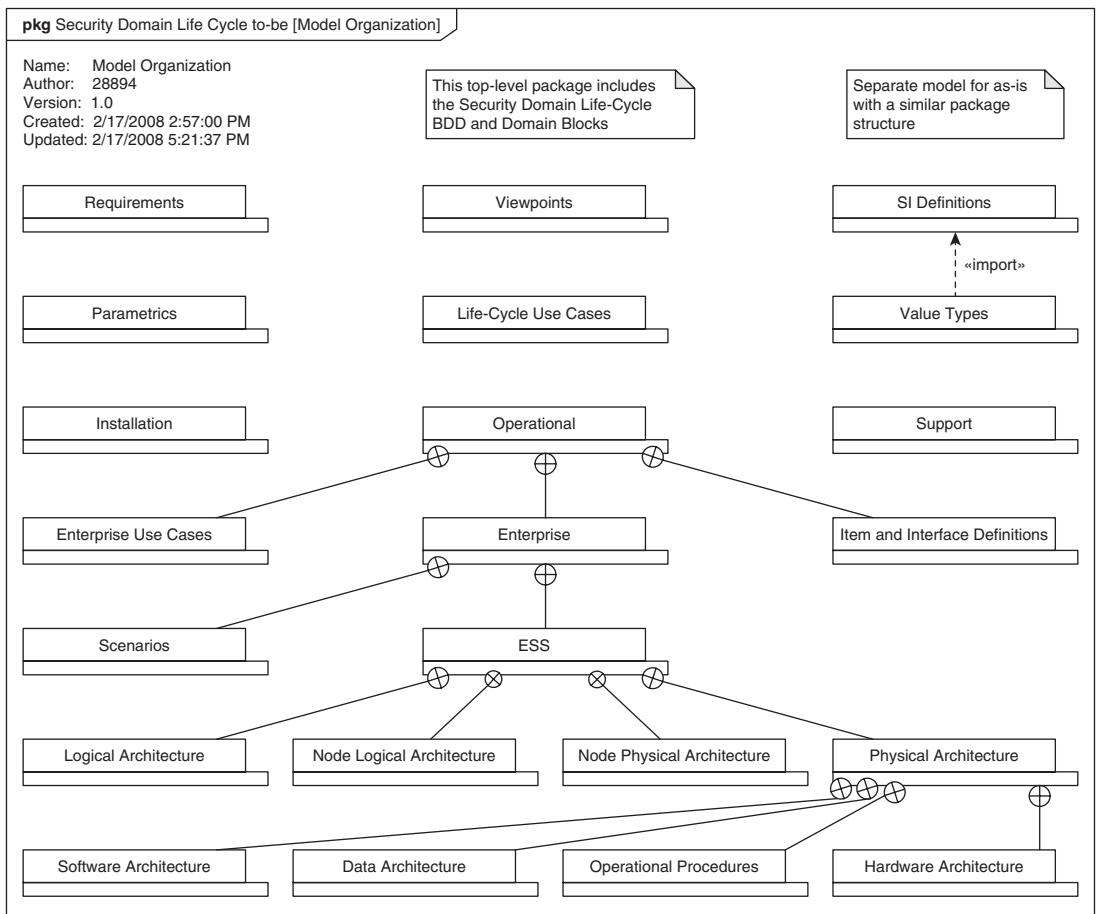
The *Enterprise* package is further nested to mirror the hierarchy of the system as described in the beginning of this section. The *Enterprise* package contains the *Scenarios* package, which describes how the enterprise use cases are realized, and the *ESS* package, which contains the ESS system design model. The ESS is the system that is being specified and designed in this example.

The *ESS* package contains nested packages for the *Logical Architecture*, *Node Logical Architecture*, *Node Physical Architecture*, and *Physical Architecture*. The *Physical Architecture* package in turn contains packages for the *Software Architecture*, *Data Architecture*, *Hardware Architecture*, *Operational Procedures*, and *Operators of the ESS* system (all not shown). Each of the preceding packages contains model elements that are created by applying OOSEM. The contents of each of these packages are described in the following sections.

Diagrams contained in particular packages are highlighted in the browser with special symbols that are unique to each tool. The symbol in Figure 16.3 for the *Operational Domain BDD* within the *Operational* package, represents a block definition diagram in this tool.

As described in Chapter 5, model elements contained in one package can be related to model elements contained in another package. When a model element from another package appears on a diagram, its fully qualified name identifies the package it is contained in. This enables each model element on a diagram to be uniquely identified. The fully qualified name can be shown with the double-colon notation described in Chapter 5, but this is mostly elided in this example to reduce diagram clutter.

The package structure from Figure 16.3 is partially represented in the package diagram in Figure 16.4. The package diagram is called *Model Organization*. As described in Chapter 4, the diagram header includes the type of diagram (**pkg**), the type of diagram element the frame represents (package), the name of the package that is represented by the frame (*Security Domain Life Cycle to-be*), and the

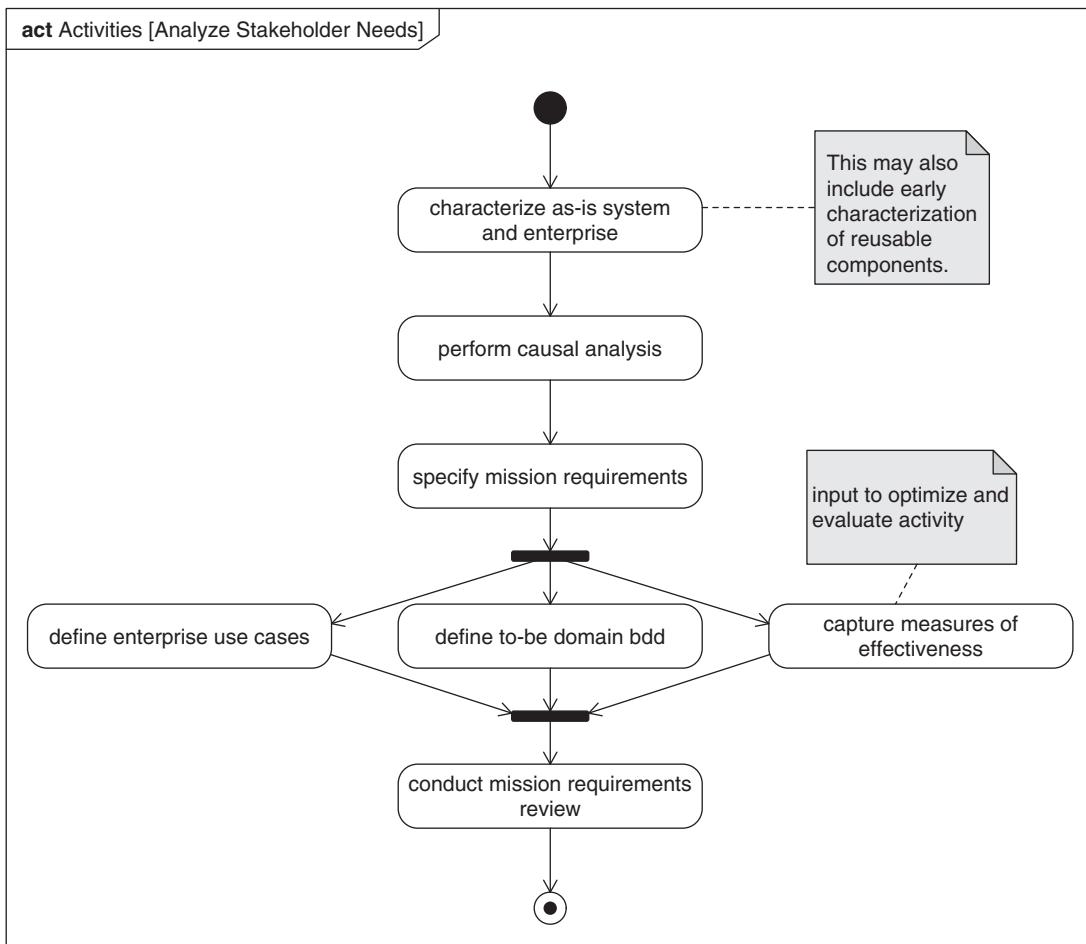
**FIGURE 16.4**

ESS Model Organization (package diagram).

name of the diagram (*Model Organization*). Based on the diagram header information, the diagram frame represents the *Security Domain Life Cycle to-be* package and the diagram contents represent its nested packages.

16.3 Applying the Method to Specify and Design the System

The following subsections elaborate the *Specify and Design System* process and artifacts that were summarized in Section 16.1.3. The subsections correspond to the actions in Figure 16.2. The activities—*Manage Requirements Traceability* and *Optimize and Evaluate Alternatives*—are included toward the end of this section even though they occur as supporting activities throughout this process.

**FIGURE 16.5**

Analyze Stakeholder Needs activity to specify mission requirements.

16.3.1 Analyze Stakeholder Needs

The *Analyze Stakeholder Needs* activity is shown in Figure 16.5. As mentioned previously, this simplified process flow does not include inputs/outputs and iteration. This activity is intended to provide the analysis to understand the stakeholder problems to be solved, and to specify the mission-level requirements that must be satisfied to solve the problem.

This analysis includes assessing the limitations of the current systems by characterizing the as-is system and enterprise and by performing causal analysis to determine the limitations and potential improvement areas from the perspective of each stakeholder. Analysis results are used to derive mission requirements and overall objectives for the to-be system and enterprise, which address the limitations of the

current system and enterprise. The to-be model of the domain, the enterprise use cases, and measures of effectiveness are used to specify mission requirements.

For this example, OOSEM is applied to the design of a single system called ESS. As a result, there is little emphasis on architecting at the SoS or enterprise levels. If this is required, then the additional architecting activities can be applied at the enterprise level [41]. In particular, the OOSEM activities that correspond to *define logical architecture* and *synthesize candidate physical architectures* are inserted between *analyze stakeholder needs* and *analyze system requirements* in Figure 16.2.

Characterize As-Is System and Enterprise

The as-is system, users, and enterprise are characterized at a level sufficient to understand the stakeholder concerns. This involves modeling the current system and enterprise only as required to provide insight into the problem and avoiding excessive modeling of the current system. Causal analysis is performed to determine the limitations and potential improvement areas of the current system. If an as-is solution does not exist, there is obviously nothing to characterize, and one can proceed directly to specifying the mission requirements. However, there is often an as-is solution to the problem that represents a starting point for the analysis.

The as-is domain is shown in the block definition diagram in Figure 16.6. It includes a top-level block called the *Operational Domain as-is*, which provides the context for the other blocks in the domain. This block is decomposed into the *Security Enterprise as-is* and multiple *Sites*.

In OOSEM, an enterprise block is established to represent an aggregation of blocks that collaborate to achieve a set of mission objectives. In this example, the as-is enterprise includes the as-is security system, which is stereotyped as the «system of interest»; the *Emergency Services*, which includes the *Dispatcher* and the *Police*; and the *Communication Network*, which enables communication between the as-is security system and the emergency services. These blocks collaborate to monitor a residence for potential intruders.

The domain block is also composed of multiple sites that are being protected that are external to the enterprise. Each site is composed of a single residence with one or more occupants and may include zero to n intruders.

The domain model helps establish the boundary between the system of interest and the external systems and users that the system either directly or indirectly interacts with. The as-is security system includes multiple site installations, as indicated by the multiplicity on the association end, and a single central monitoring station. Note that the site installation is owned (i.e., black diamond) by the *Security System as-is* and is a reference part (i.e., white diamond) of the *Single-Family Residence*. The reference part provides a mechanism to represent a more complex system boundary, where the part is owned by one block and referenced by another.

An alternative depiction of the as-is domain is shown in Figure 16.7, where the system and external systems are shown in iconic form. This provides a means to communicate a simplified depiction of the as-is domain that can be annotated to informally represent selected interactions and relationships among the entities.

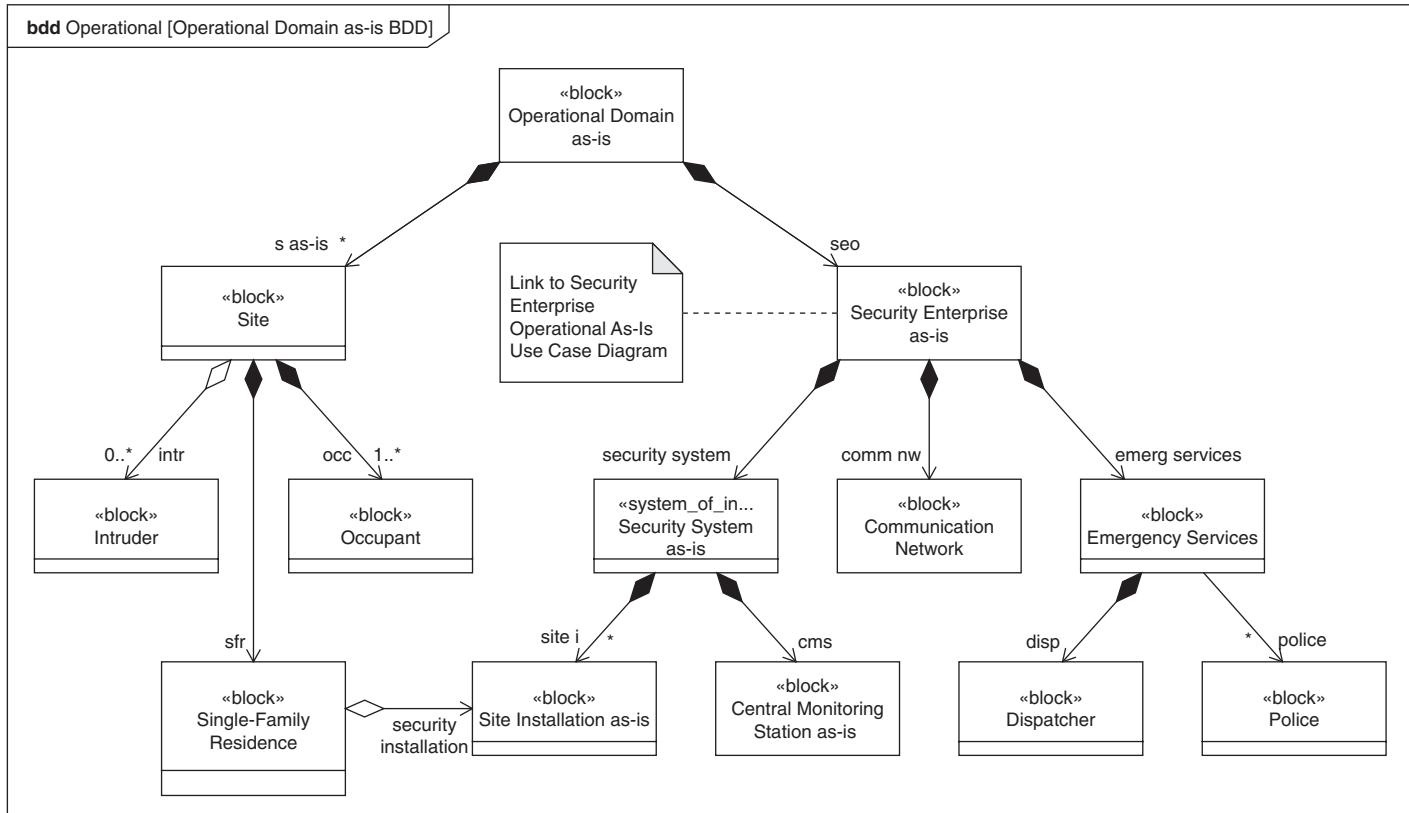
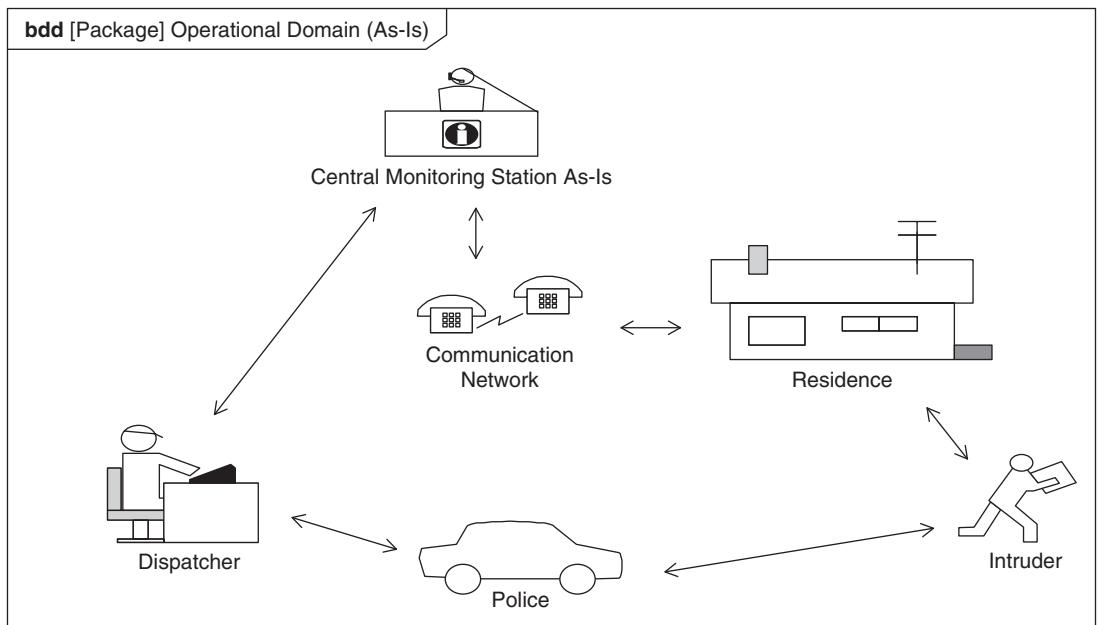


FIGURE 16.6

The as-is operational domain.

**FIGURE 16.7**

ESS as-is domain (iconic representation).

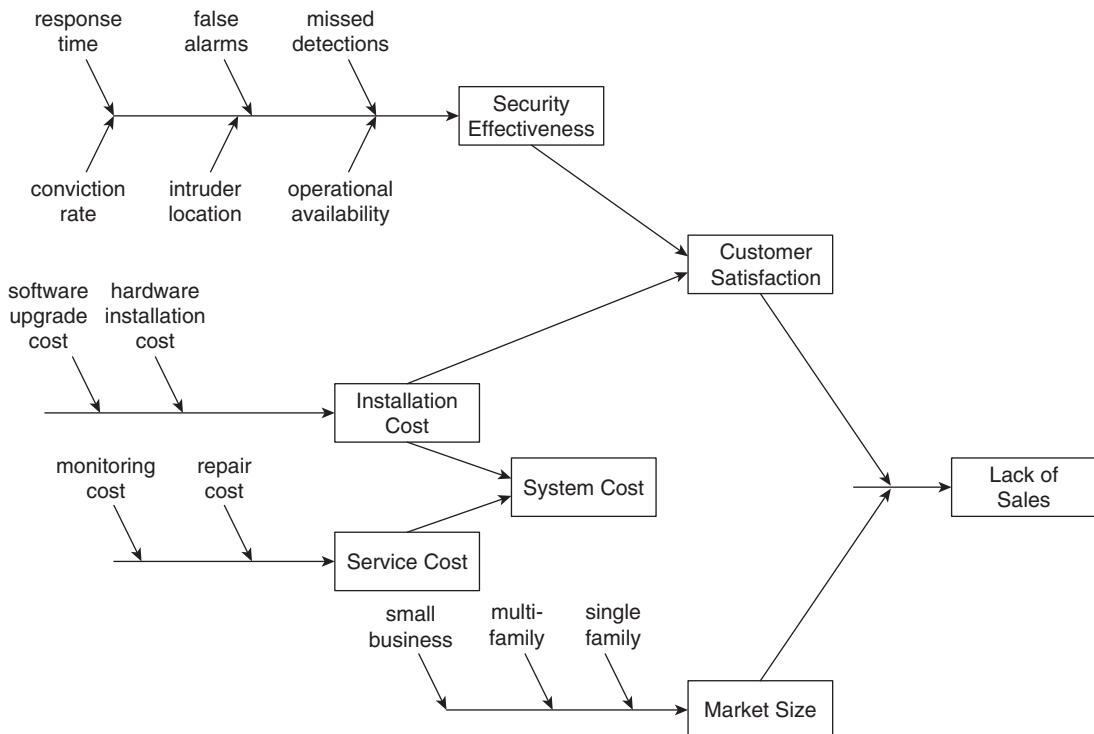
The relationships between the entities could be represented as associations, but for the purpose of this example, it is assumed that they are merely annotations on the block definition diagram; they are represented later as connectors with item flows on the internal block diagram.

Perform Causal Analysis

The as-is system and enterprise are analyzed to assess their capabilities and limitations and to identify potential improvement areas. Other sources of data may be required to support this analysis, including marketing data such as customer surveys and competitive data.

A useful technique for structuring the causal analysis is to use a fishbone diagram to represent a tree of cause-effect dependencies. A fishbone diagram for the *Security Enterprise as-is* is shown in Figure 16.8. The root of the tree represents measures of effectiveness (moe) that can reflect value from the perspective of each stakeholder. The nodes of the tree represent dependent properties that can impact the moes.

Business sales is a moe of particular importance to the company owner, as well as to the investors of Security Systems Inc. The cause-effect dependencies show that sales are impacted by *Customer Satisfaction* and the *Market Size*. *Customer Satisfaction* is measured in terms of *System Cost* and *Security Effectiveness*. *System Cost* is measured in terms of its *Installation Cost* and ongoing *Service Cost*. *Security Effectiveness* is measured in terms of *response time*, *false alarm*, *missed detections*, and other parameters. Moes for other ESS stakeholders—including

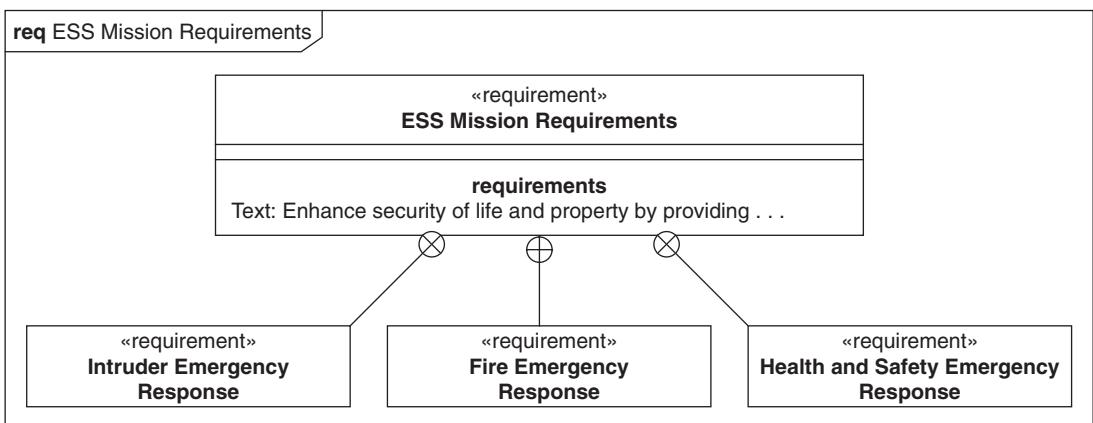
**FIGURE 16.8**

Causal analysis of the *Security Enterprise as-is*.

the customer, the police department, and internal stakeholders such as central monitoring station operators and system installers—should also be considered. One example is the police department's concern regarding false alarms and the associated cost to the city that can be represented by the cause-effect relationship. Although this is not represented as a SysML diagram, an equivalent of a fishbone diagram can be represented by capturing the relationship between the parameters on a parametric diagram.

A weighted value can be assigned to quantify the impact of each cause on the effect similar to a risk or fault tree analysis. Additional engineering analysis is performed to identify the root cause and associated impact on the moes. This analysis may include timeline analysis, reliability analysis, and life-cycle cost analysis; and it may be captured in parametrics diagrams as discussed later.

A primary deficiency identified during the causal analysis in this example is the limited functionality of the current security system relative to the competitive systems. A stakeholder need is identified to extend the functionality beyond intruder detection to include emergency protection for fire and medical emergencies. In addition, it is determined that there is a need to expand the market size for the security systems to protect multifamily residences and small businesses in addition to single-family residences.

**FIGURE 16.9**

ESS mission requirements.

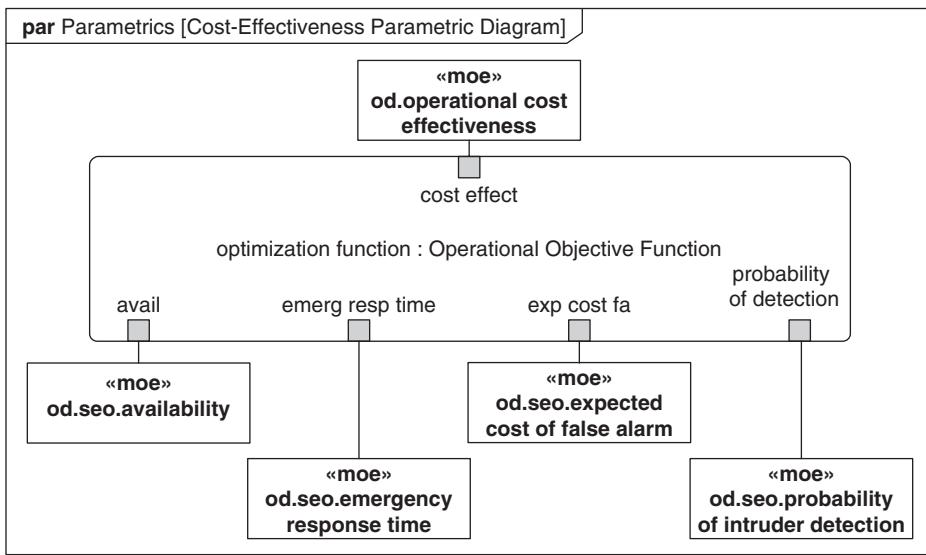
Specify Mission Requirements

Based on the preceding analysis, a prioritized set of mission requirements is defined that address the limitations of the as-is domain. The mission requirements are captured as text requirements, as shown in the requirements diagram in Figure 16.9. The top-level mission requirement for the ESS includes the text statement to “Enhance security of life and property by providing emergency response to theft, burglary, fire, and health and safety.” The mission requirements are contained in the *Requirements* package. The traceability between the mission requirements and lower-level requirements is discussed in Section 16.3.6.

Capture Measures of Effectiveness

Moes reflect mission-level performance requirements and value to the customer and other stakeholders, and they are derived from the causal analysis and related stakeholder needs analysis. The measures of effectiveness are the emergency response time, false alarm rate, operational availability, and total cost of ownership. The target value for each moe is established to achieve a competitive advantage.

Engineering analysis is performed throughout the development effort to support evaluation, selection, and optimization of the design solution. moes are captured in the top-level parametric diagram in Figure 16.10. The moes are represented using the dot notation, described in Chapter 6, to capture the path name to the properties based on the system hierarchy shown in Figure 16.11. An optimization function defines the overall cost effectiveness of the design solution in terms of a weighted sum of the utility associated with each moe. Additional analysis models can be established for each moe and captured in a parametric diagram. This provides a mechanism to flow down the top-level moes to critical system parameters (also known as technical performance measures) as the model is further elaborated. This is discussed further in Section 16.3.5.

**FIGURE 16.10**

ESS top-level parametric diagram.

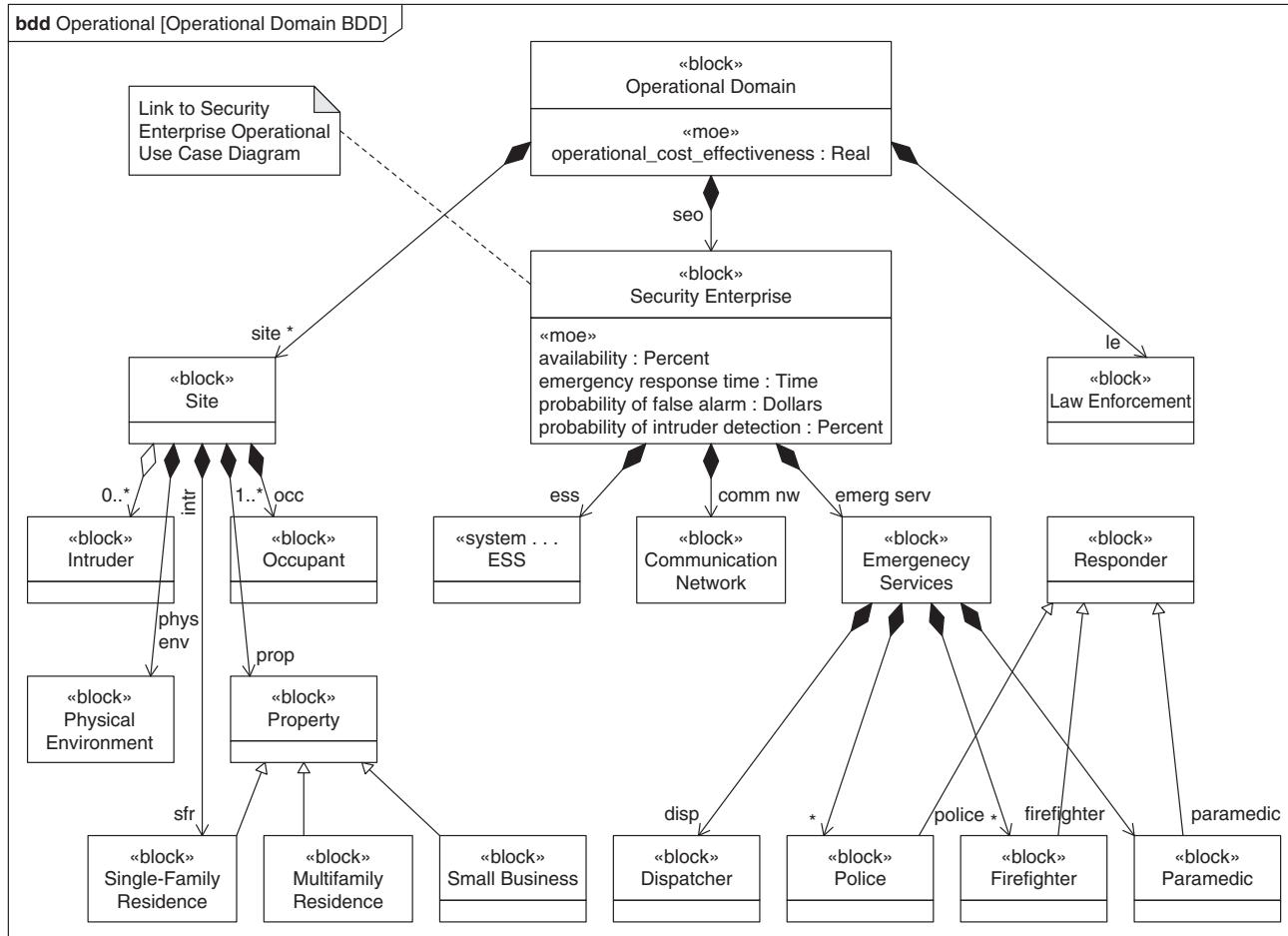
Define To-Be Domain Model

Based on the preceding analysis, we can establish the scope for the to-be system and enterprise. The block definition diagram for the to-be operational domain is shown in Figure 16.11. This diagram is shown in the browser view in Figure 16.3 under the *Operational* package. The diagram represents the hierarchy of blocks with the *Operational Domain* as the top-level block. The to-be operational domain includes significant changes from the as-is operational domain in Figure 16.6, and it reflects the broader set of mission requirements that resulted from the causal analysis.

The *Emergency Services* includes the *Firefighter* and *Paramedic* in addition to the *Police* and *Dispatcher* that were included in the as-is domain. The *Multifamily Residence* and small *Business* have been added as specializations of *Property* along with the *Single-Family Residence* from the as-is domain. The *Physical Environment* has been added since the system must now monitor the environment for fire. In addition, the as-is security system has been replaced by the *ESS* black box, which is the «system of interest» for this development effort.

The *Security Enterprise*, which includes the *ESS*, *Emergency Services*, and *Communications Network*, is responsible for satisfying the mission requirements and providing services to the customer and *Occupants*. The moes are captured as stereotyped value properties («moe») of the *Security Enterprise* block along with their corresponding units. Specific target values and/or value distributions can be specified as well.

In this example, the *Police*, *Firefighter*, and *Paramedic* are all a subclass of *Responder*. As complexity increases, it may be necessary to create a separate block definition diagram for the specialization hierarchies for the external systems.

**FIGURE 16.11**

The to-be operational domain.

Define Enterprise Use Cases

Enterprise use cases are defined to represent each mission objective that corresponds to the mission requirements in Figure 16.9. The objectives are to provide responses to intruders, fire, and medical emergencies, as shown in the use case diagram in Figure 16.12. Each use case is specialized from a more general use case called *Provide Emergency Response*. An additional use case, called *Provide Validated Data*, supports postemergency response actions, such as providing evidence to convict an intruder.

This use case includes two additional use cases—*Assign User Access* and *Provide Query Response*. The *Security Enterprise* is the subject in the use case diagram and is used by the actors to achieve the use case goals (i.e., mission

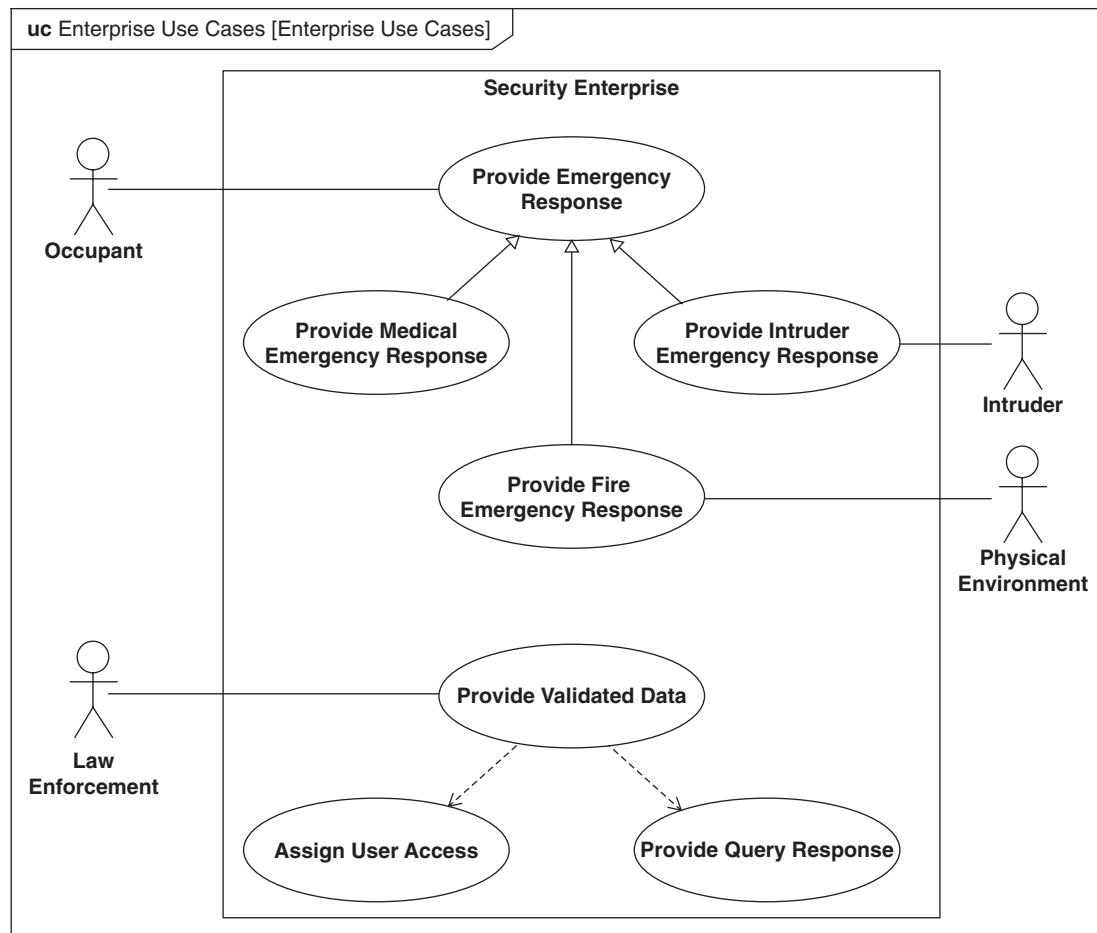


FIGURE 16.12

ESS Enterprise Use Cases.

objectives). The blocks that are external to the enterprise in the *operational domain block definition diagram* are allocated to the actors in the use case diagram. Exception use cases can also be defined; they do not support the mission objectives and are often used to help specify fault-tolerant solutions.

The use cases in this example refine the mission requirements using the refine relationship. An example of the refine relationship is shown in Figure 16.58 in Section 16.3.6. The use cases may also trace to other source documentation such as a concept of operations or marketing data. The enterprise use cases are realized by enterprise scenarios that elaborate the interaction between the actors and parts of the enterprise. This analysis is used to help specify the ESS black-box requirements, as described in the next section.

Each use case may be augmented with a use case description that includes a textual description of each step in the use case scenario. There are many books on how to write and model use cases [34]. The individual steps can be captured as SysML requirements that can be traced to other model elements, such as specific actions in an activity diagram. The use case description may include additional information such as alternative paths and pre- and postconditions, which may be used to express the moes associated with the mission objectives. In this example, the relationship between pre- and postconditions can be used to specify the mission response time. The modeling of use cases is described in Chapter 11.

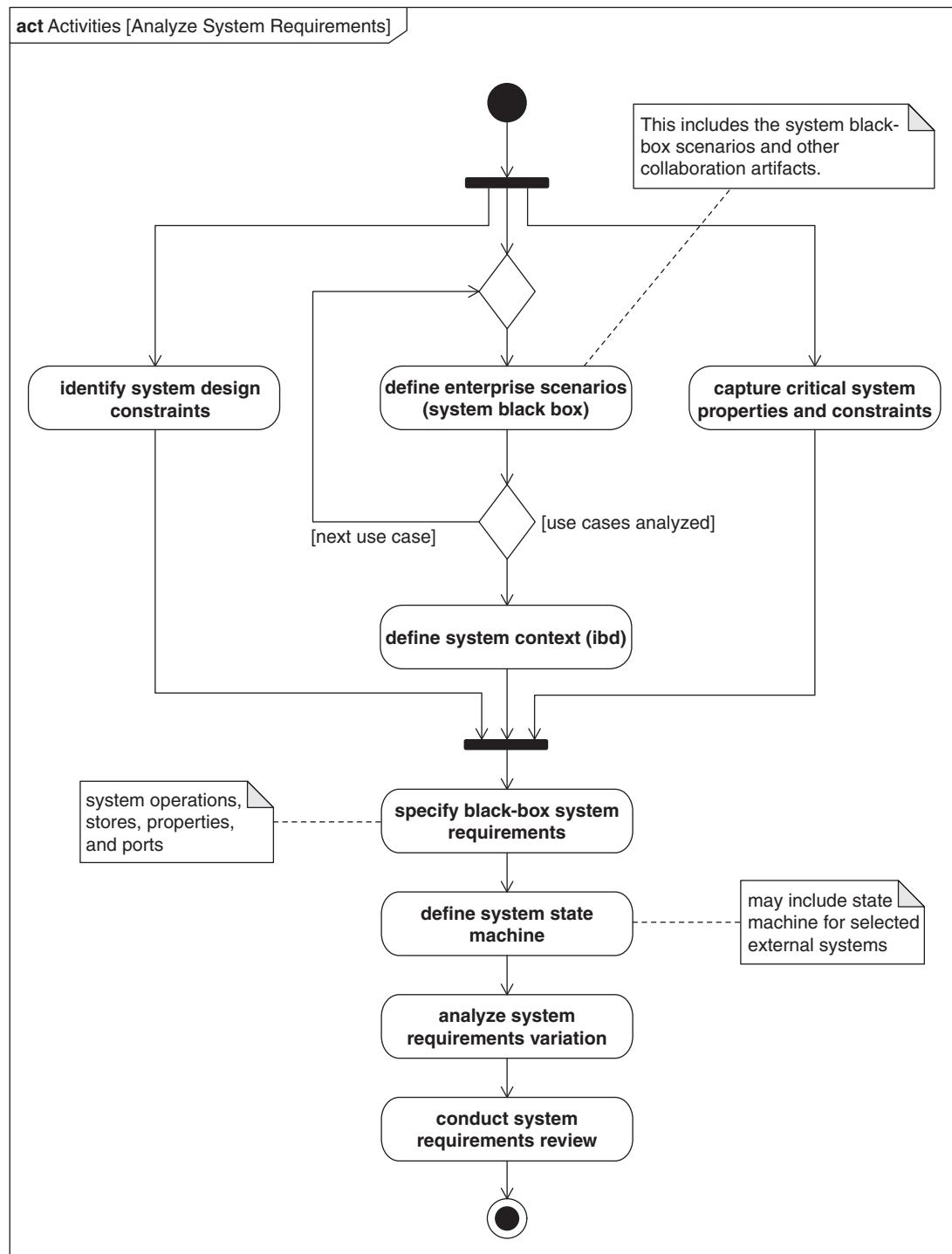
16.3.2 Analyze System Requirements

The *Analyze System Requirements* activity is shown in Figure 16.13. This activity specifies the requirements for the system as a black box in terms of its input and output behavior and other externally observable characteristics. Scenario analyses for each of the enterprise use cases describe how the system interacts with the external systems and users identified in the domain model to achieve the mission objectives.

The scenarios are modeled using either activity diagrams with activity partitions or sequence diagrams. A system context diagram using an internal block diagram is generated to represent the interfaces between the system and the external systems and users. Critical system properties, which can impact the measures of effectiveness, are identified. Based on the analysis, the black-box system requirements can be specified in terms of the system functionality, interface, control, store, and performance requirements. The system state machine augments the system black-box requirements by specifying when system functions or operations are performed. Requirements variation analysis is evaluated in terms of the probability that a requirement will change, and it is input into the design process to ensure that the design can accommodate the change. Design constraints, such as the required use of a COTS component, are also identified and captured, and later imposed on the architecture.

Define Enterprise Scenarios

In this activity, one or more enterprise scenarios are defined for each enterprise use case to specify the interaction between the system and the external systems

**FIGURE 16.13**

Analyze System Requirements activity to specify black-box system requirements.

to achieve the use case goals (i.e., mission objectives). The enterprise scenarios provide the basis for specifying the system behavioral requirements. A complete set of scenarios, which correspond to each primary and alternative path for the use cases, are needed to completely specify the system requirements. This may involve additional refactoring of the use cases to identify common functionality. The modeler should also ensure that the following are addressed:

- High likelihood scenarios
- Performance stressing scenarios and scenarios that significantly impact the moes
- Failure scenarios
- Critical system functionality
- New system functionality
- Interactions that include all external systems and users

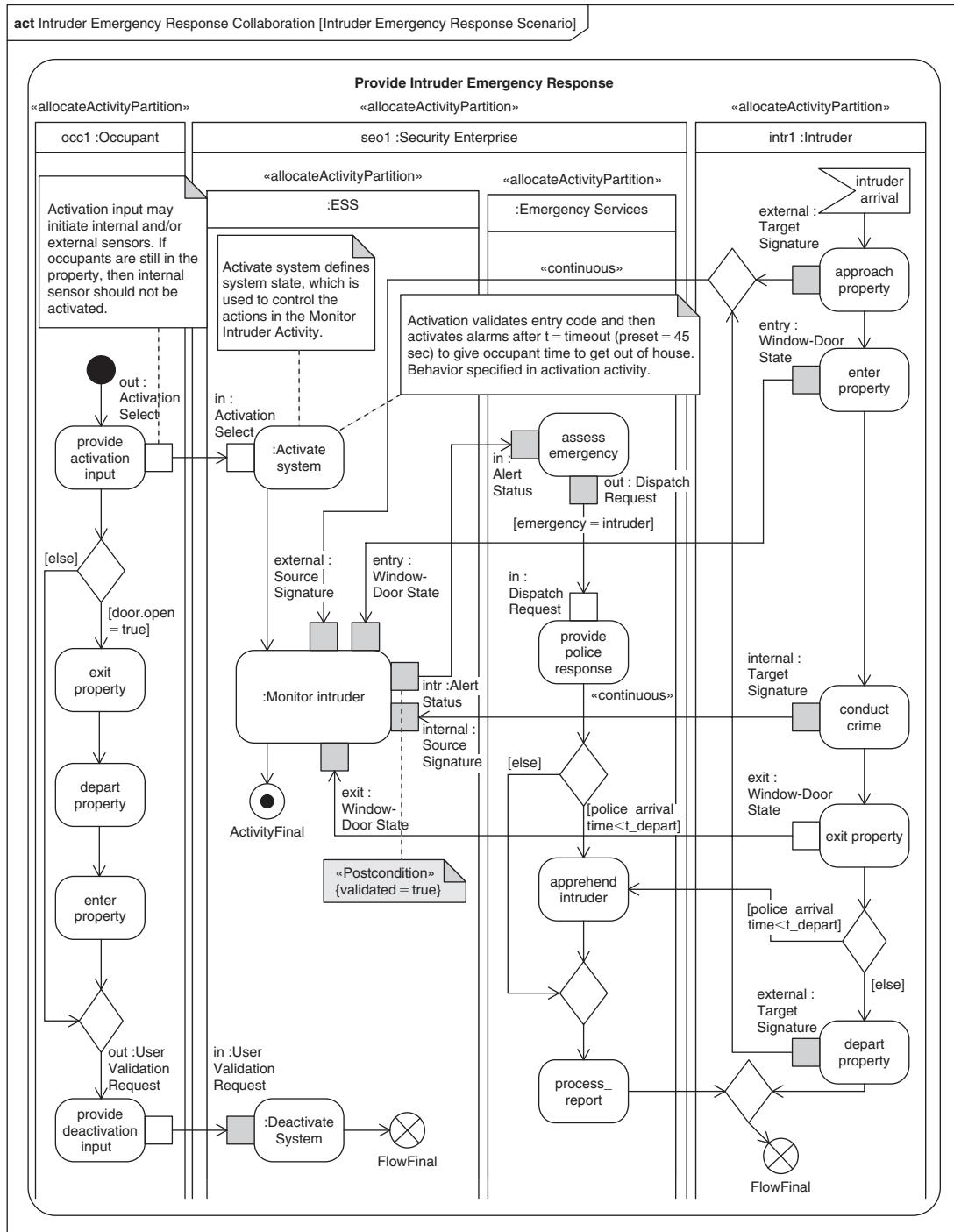
The enterprise scenarios are modeled with activity or sequence diagrams. The activity partitions (also known as swimlanes) in the activity diagram, or the life-lines in the sequence diagram, represent the system and external systems. For this example, the enterprise scenarios are represented with activity diagrams. By applying the «allocate» stereotype to each activity partition, the actions in the activity partition are automatically allocated to the part that is represented by the activity partition. The inputs and outputs of the activity are shown as parameter nodes on the boundary of the activity.

A representative scenario for the enterprise use case, called *Intruder Emergency Response Scenario*, is shown in Figure 16.14. The scenario is represented by an activity diagram with activity partitions for the *ESS*, *Emergency Services*, *Occupant*, and *Intruder*. The actions in each activity partition specify what the corresponding part must do. The *ESS* must activate and deactivate the system in response to the *Occupant* input and must monitor the environment to detect an *Intruder*.

The pre- and postconditions for each action can be specified in terms of constraints. As an example, the postcondition on the intruder alert status must be that the alert has to be validated, $\{validated = true\}$, when the input exceeds a threshold defined by a precondition. The pre- and postcondition constraints can be captured in a parametric diagram to support engineering analysis, such as the analysis of the probability of detection.

The streaming pins on the monitor action (i.e., shaded pins) indicate that the action continues to accept inputs and/or provide outputs as it executes. In this example, the *monitor intruder* action continues to execute as it receives streaming inputs from the *Intruder*. The output control flow from *deactivate system* terminates on a flow final and does not cause the entire activity to terminate.

In addition to the activity or sequence diagram that captures the scenario for the enterprise use case, other artifacts can be created to more completely specify collaboration among the parts. This is called a collaboration process pattern in OOSEM, and it is reused in the logical and physical architecture as well. The pattern includes creation of a block definition diagram, which defines the parts that interact based on the activity partitions; an internal block diagram to capture

**FIGURE 16.14**

Intruder Emergency Response Scenario realizes the enterprise use case.

the interfaces between the interacting parts; a block definition diagram, which captures the input and output definitions (i.e., item definitions); a parametric diagram to capture the input/output constraints including pre- and postconditions; and test cases, which verify that the input/output relation is satisfied. The preceding modeling artifacts can be implemented for each use case scenario to aid in verification, requirements traceability, engineering analysis, and the ability to create executable specifications, as described in Chapter 17.

Define System Context

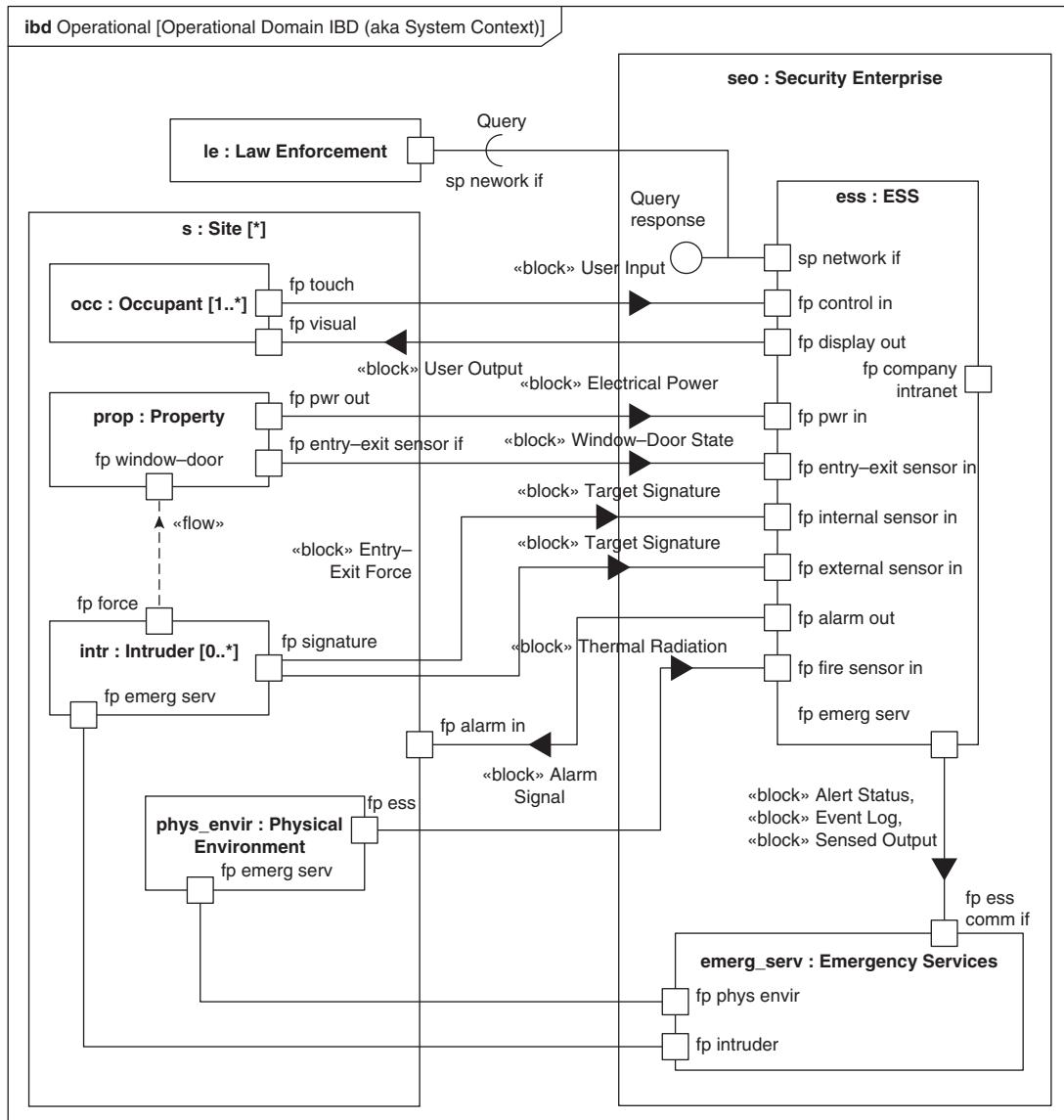
The system context diagram is shown as an internal block diagram in Figure 16.15. This diagram depicts the *ESS* and its interfaces to all external systems and users that participate in the enterprise scenarios. The frame of the internal block diagram represents the *Operational Domain* block. The parts of the *Operational Domain* correspond to the *Security Enterprise* and the enterprise actors from the block definition diagram in Figure 16.11. The *ESS* and *Emergency Services* are nested within the *Security Enterprise*. The inputs/outputs in the activity diagram are allocated to item flows that flow across the connectors between the parts.

Flow ports are defined for each interface on each part. The flow port is typed by a block or flow specification that specifies the type of input/output that can flow through the port. For an item flow to flow in or out of a flow port, the type of the flow port must be the same type or a super class of the item that is flowing. The type of flow port should then represent the most general classification of the input or output item that flows. For a flow specification, this also applies to the type of its flow properties.

To represent the most general classification, the port may be typed by the physical nature of the item that flows, such as material, fluid, video, or an analog or discrete signal, rather than typing the port by the logical content of the item that flows. The type of port may also correspond to a specification of the physical interface such as a USB port on a computer. In that case, any flow that is compatible with the physical nature of the USB specification can flow in or out of the port.

An interface taxonomy, which specifies both a logical and physical classification of interfaces, can be defined. The type of flow port is based on the physical taxonomy. The type of item that flows in or out of the port can subclass from both the logical and physical classification. This enables the item that flows to capture the logical content and have a compatible type with the flow port. A similar approach is used to type the inputs and outputs of the behavior that is bound to the port.

An example of an ESS flow port, shown in Figure 16.15, is *fp external sensor in*, which is typed by *Electromagnetic Signal*. The type of flow port can later be subclassed if it is desired to further constrain what flows in or out of the port. If the external sensor is determined to be a surveillance camera that accepts an optical or an infrared signal input, the type of *fp external sensor in* can be subclassed as an optical signal or an infrared signal, respectively. The item flow is typed by *Target Signature*, which corresponds to the logical content of the item that flows. The *Target Signature* is a subclass of *Electromagnetic Signal* to ensure that its type is compatible with the ESS flow port. The type of the *Intruder* flow

**FIGURE 16.15**

ESS context diagram showing the interfaces between the ESS and the external systems, users, and physical environment.

port on the other end of the connector must also have a compatible type with the item flow and the ESS flow port. An interface specification may also include parametrics to constrain the properties of the connecting ports, such as the sum of the energy input and output flow must equal zero.

Sometimes additional physical encoding of the item that flows needs to be specified. An example of an item that flows is a file that flows from a computer port to a printer port. Assume the logical content of the item is a report and the format of the report is rich text format (RTF). The RTF is the physical encoding of the report and should be reflected in the type of the item. In this case, the item type is RTF, and the report is allocated either to the item type or to item property. Modeling the physical interface characteristics of the ports and flows can be deferred until interface design decisions are made.

The standard port on *ES*, called *sp network if*, specifies an interface to provide a *Query Response* when a *Query* is requested by *Law Enforcement* on the required interface. The *Query* and *Query Response* are the required and provided interfaces, respectively, for the standard ports.

Capture Critical System Properties and Constraints

Critical performance requirements can be captured as a value property of the ESS block or an activity. The performance requirements are derived based on engineering analysis.

One example of a performance analysis is a timeline analysis. The timing diagram in Figure 16.16 specifies the mission timeline for the *Intruder Emergency Response Scenario* in Figure 16.14. The actions from the activity diagram are shown on the *y*-axis and the required time to perform the actions are shown on

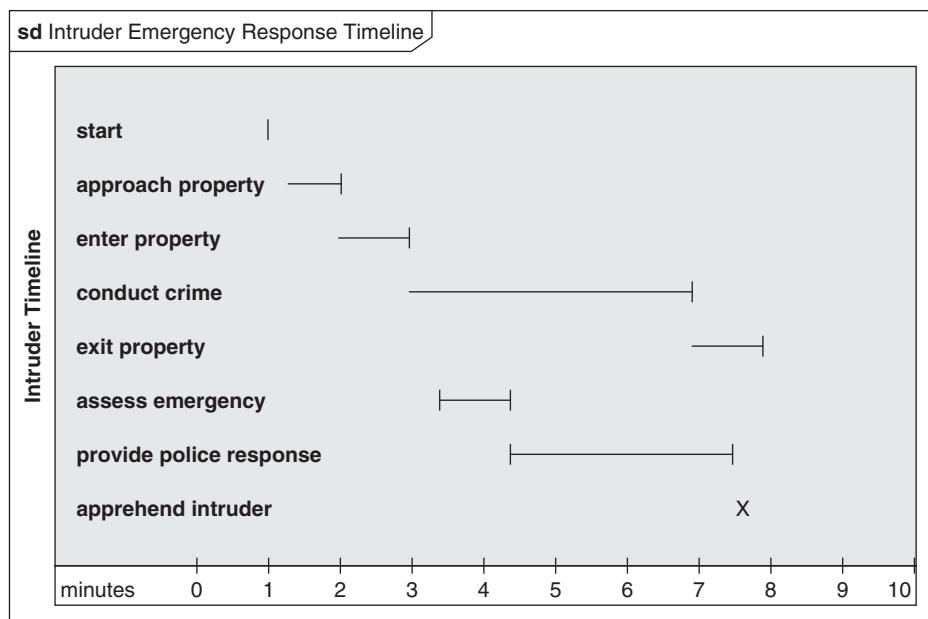


FIGURE 16.16

Intruder Emergency Response Timeline.

the *x*-axis. The timeline is used to allocate time to each action in the scenario in order to satisfy the mission response time that was identified as a moe. In this example, the *intruder detection response time* is the time from the intruder entering the property until the ESS reports the alert to the dispatcher. This is viewed as a critical system property, referred to as a measure of performance, represented as «mop» in the model. The value for this property can be budgeted based on its impact on overall security effectiveness.

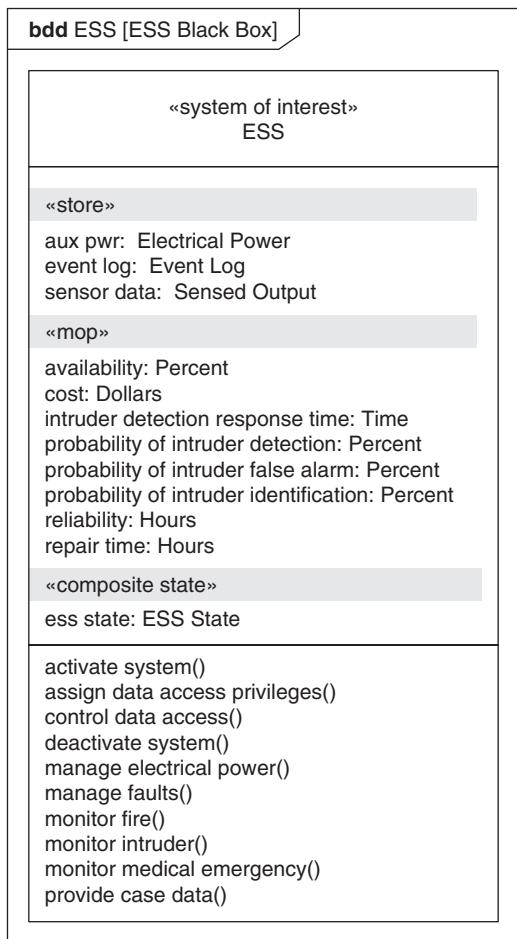
Other critical system properties that require analysis to satisfy requirements may include probability of detection and probability of false alarm. The constraints on these properties are captured in parametric diagrams as part of the engineering analysis described in Section 16.3.5.

Specify Black-Box System Requirements

The application of OOSEM results in the specification of the system based on the scenario analysis, interface definitions, and other engineering analyses performed, as described earlier. The specification is often called a black-box specification in that it defines the system's externally observable behavior and physical characteristics. The black-box specification does not specify the internal characterization, or white-box specification, of the system in terms of how it achieves the black-box specification. Design constraints may augment the black-box specification to constrain how the requirements are implemented, such as the constraint to use a particular COTS component or a particular algorithm in the design.

The specification features of a black box that is represented as a block include the following:

- The required functions it must perform and the associated inputs and outputs. The required functions are modeled as activities that are allocated to the block, or as operations of a block whose methods can be activities. The associated inputs and outputs are the inputs and outputs to the activity and the signature of the operation.
- The required external interfaces that enable it to interact with other external systems and users. The interfaces are specified by the ports on the block.
- The required performance, physical, and quality characteristics that impact how well the functions must be performed or a physical characteristic (e.g., weight). These characteristics are specified as value properties with units and dimensions. The value properties may have specific values or probability distributions associated with their values. Constraints on value properties are captured as parametric constraints. OOSEM stereotypes these properties as «mop».
- The required control in terms of input events and preconditions that determine when the functions are performed. The control can be specified in terms of a state machine for the block that specifies which activities are performed in response to different input events and associated preconditions.
- The required items that the system must store including data, energy, and mass. The required stores can be modeled as properties of the block. OOSEM stereotypes these properties as «store».

**FIGURE 16.17**

ESS black-box specification.

The specification features for the *ESS* block are shown in Figure 16.17. In this example, an operation is defined for the *ESS* for each action in the *ESS* activity partition from each scenario that was analyzed. The method of the operation is the corresponding activity. Alternatively, an allocated compartment can be used to show the activities allocated to the block directly. The performance properties, such as *probability of intruder detection*, *intruder detection response time*, and *availability*, are stereotyped as measures of performance «mop». Parametric constraints on these properties can constrain the value properties of the block and/or constrain the inputs and outputs on an activity or operation (i.e., pre- and post-conditions). The ports specify the system interfaces, but are not shown in the figure. The items that are stored, such as the event log and auxilliary power, are stereotyped as «store» properties.

The black-box specification can be traced to the mission requirements as part of the requirements management, described in Section 16.3.6, using the appropriate requirements relationship. Traceability can be defined at a fine-grained feature level or at a less granular level depending on the need.

The black-box specification can be applied at any level of design, including system, element, and component levels. As a result, this approach is used later in the chapter to specify component requirements.

Define System State Machine

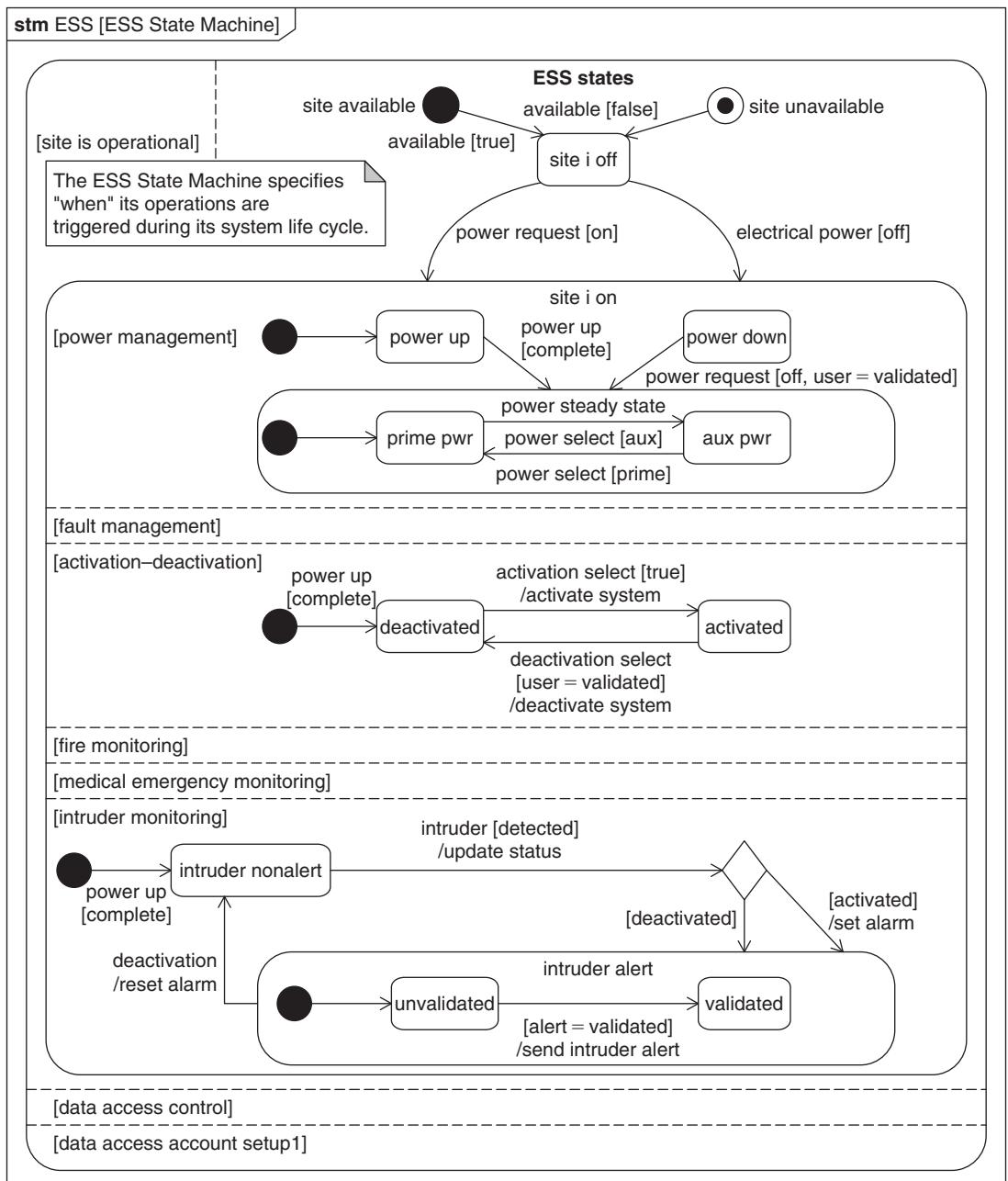
Each scenario defines actions that the ESS system must perform. The ESS state machine specifies the ESS behavior from all scenarios that it participates in. The ESS evaluates the guard conditions in response to an input event to determine whether a transition to a next state is triggered. The guard conditions specify conditions on the input values, current state, and resource availability. If the transition is triggered, the block executes the exit action from the current state, executes the transition behavior (i.e., effect), and enters the next state. It then executes the entry action of the next state followed by its do/behavior, which is defined by an activity. The transition behavior may include a send signal action that can trigger a transition in another system's state machine. The system's logical and physical design must implement the control requirements imposed by the system state machine.

A simplistic state machine specifies the control requirements as a series of statements as follows. If an input event occurs while in the current state and the guard conditions are satisfied, then transition to the next state and execute the selected actions.

A selected portion of the ESS state machine is shown in Figure 16.18. The state machine includes regions for intruder monitoring, fire monitoring, medical emergency monitoring, and fault monitoring to specify that monitoring of these different events can be concurrent. The behavior in the orthogonal regions can result in contention for system resources, such as processor capacity, that must be reconciled through the design process.

As shown in the *intruder monitoring* region, when ESS *power up* is *complete*, it initially transitions to the *intruder nonalert* state. If an intruder is detected, it transitions to the *intruder alert* state. If the system is in the *activated* state at the time of the transition, it sets an alarm. In the *alert* state, the alert is initially *unvalidated*. Once the alert has been validated, the system transitions to a *validated* state and sends the validated intruder alert to *Emergency Services*. The specific approach used for developing this state machine is based on an analysis of the enterprise scenarios, but it is not included here for brevity. Alternative approaches can be used to specify the system state machine as well.

A property of the *ESS* block in Figure 16.17, called *ess state*, is stereotyped as «composite state» and is typed by a block called *ESS State*, which has the same stereotype applied. The value of this property represents the state of the system at any point in time and is determined by the ESS state machine behavior. One can think of the value of this property as the position of a set of switches, where each switch sets a particular system state to true or false. The allowable concurrent,

**FIGURE 16.18**

ESS State Machine.

sequential, and nested states are defined by the state machine diagram. This property can be used in parametrics to capture state-dependent constraints.

Analyze System Requirements Variation

Requirements variation analysis is intended to define the potential change in requirements that can result from different sources, such as a likely change to an external interface, a possible increase in the number of system users, or possible new functionality to stay competitive. For the ESS, potential requirements change can result from an increase in the number of expected site installations, or adding new functionality to monitor carbon monoxide, or to integrate with an automatic sprinkler system to extinguish fires.

Requirements variation is evaluated in terms of the probability that a requirement will change. The results of the analysis are input to the risk analysis to assess the impact of the change and to develop mitigation strategies. The strategy is reflected in the architecture and design approach, such as isolating the source of the changing requirement on the design. A similar strategy can be applied to likely technology changes.

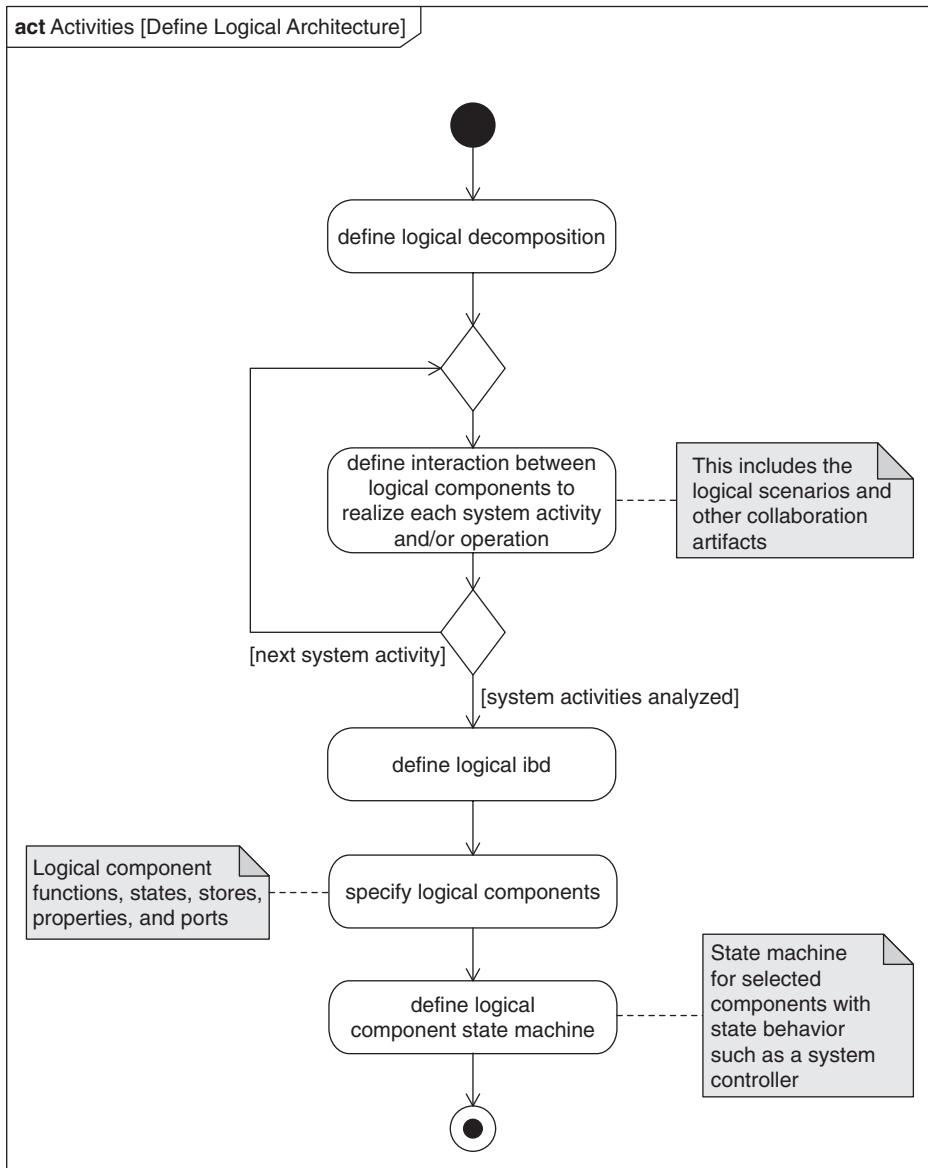
Identify System Design Constraints

Design constraints are those constraints that are imposed on the solution space or ESS white box. These constraints are typically imposed by the customer, by the development organization, or by external regulations. The constraints may be imposed on the hardware, software, data, operational procedures, interfaces, or any other part of the system. Examples may include a constraint that the system must use predefined COTS hardware or software or a specific interface protocol. For the ESS system, much of the legacy central monitoring station hardware constrains the solution, as well as the communications network between the central monitoring station and the legacy site installations.

Design constraints can have a significant impact on the design and should be validated prior to imposing them on the solution. A straightforward approach to address design constraints is to categorize the type of constraints (e.g., hardware, software, procedure, algorithm), identify the specific constraints for each category, and capture them as system requirements in the *Requirements* package, along with the corresponding rationale. The design constraints are then imposed on the physical architecture, as discussed later.

16.3.3 Define Logical Architecture

The *Define Logical Architecture* activity is shown in Figure 16.19. This activity is part of the system architecture design that includes decomposing the system into logical components that interact to satisfy system requirements. The logical components are abstractions of the components that implement the system, which perform the system functionality without imposing implementation constraints. An example of a logical component is a user interface that may be realized by a Web browser or display console, or an entry/exit sensor that may be realized by an optical sensor. The logical architecture serves as an intermediate level of abstraction

**FIGURE 16.19**

Define Logical Architecture activity decomposes the system into logical components, and describes their interactions such that they satisfy the system requirements.

between the system requirements and the physical architecture that can reduce the impact of both requirements and technology changes on the physical design.

The logical architecture definition activity includes decomposing the system into logical components, as described earlier. For each operation the system is

required to perform, a scenario is defined to describe the interaction among the logical components, along with other collaboration artifacts that realize the operation, such as an internal block diagram that shows the interconnection between the logical components. The logical components identified from the initial logical decomposition are subject to refinement based on repartitioning of their functionality and properties. Each logical component is then specified in a similar way, as described for the ESS black-box specification. A logical component may include a state machine as part of its specification if it has significant state-based behavior. The logical components are allocated to the physical architecture, as described in Section 16.3.4.

Define Logical Decomposition

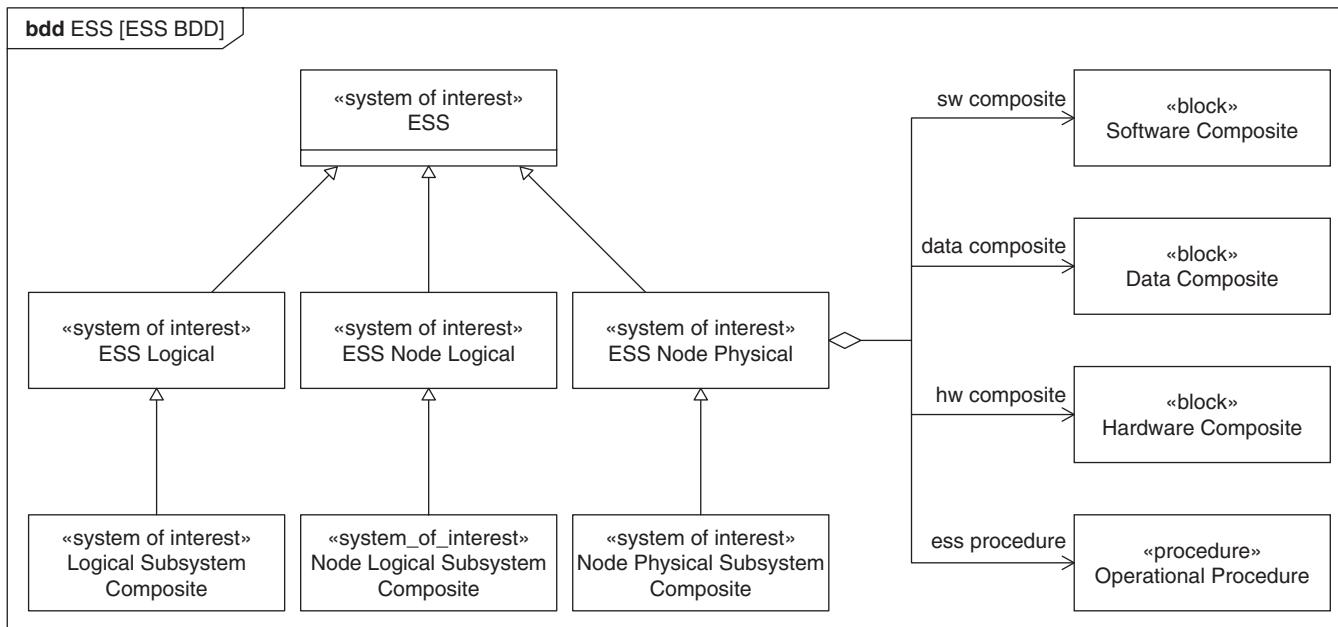
The ESS block is specified as part of the system requirements analysis described in the previous section. In OOSEM, the ESS block is decomposed into both a logical and a physical hierarchy. To maintain separate logical and physical system design hierarchies, a subclass of the ESS block is created for each decomposition of the system hierarchy. In Figure 16.20, the *ESS Logical* block is a subclass of the ESS block that inherits all the ESS features, including its operations (or allocated activities), stores, properties, and ports. The *ESS Logical* block is decomposed into logical components.

The *Logical Subsystem Composite* is a subclass of the *ESS Logical* block. It is used to decompose the ESS into subsystems. In OOSEM, a subsystem corresponds to an aggregate of components that realize an individual ESS operation or activity. The *Logical Subsystem Composite* represents an aggregate of logical subsystems, each of which realizes a particular ESS operation.

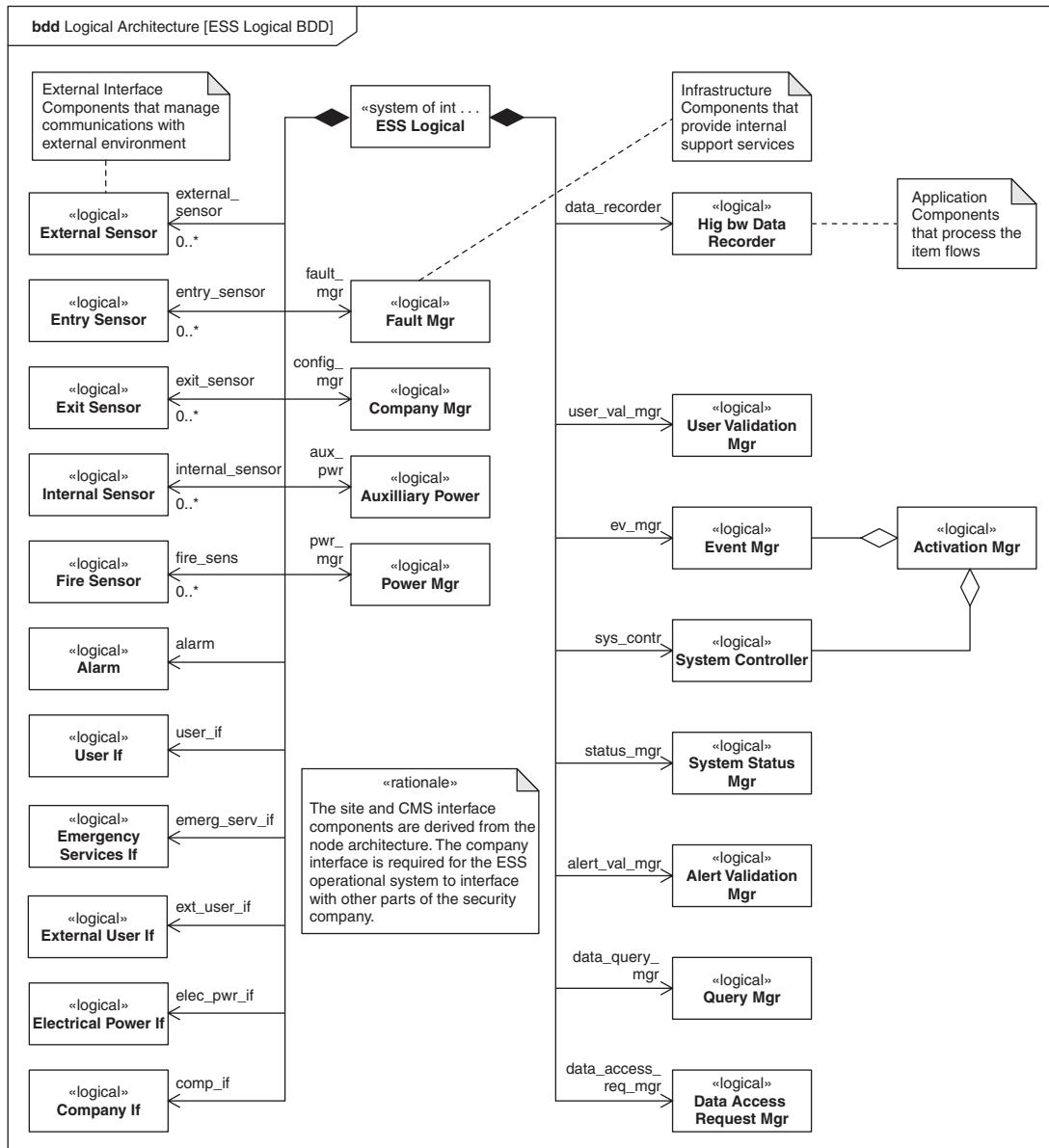
The *ESS Node Logical* and *ESS Node Physical* are also subclasses of the *ESS* block, as shown in the figure. These blocks are used to decompose the ESS system into logical and physical nodes, as described in Section 16.3.4. The nodes represent an aggregate of components at a particular location. For the ESS, the locations correspond to the site installation and the central monitoring station. A logical node aggregates the logical components at a particular location, and the physical nodes aggregate the physical components at a particular location. Using this approach, the system can have multiple decomposition hierarchies, which can be related to one another.

The other blocks in the figure represent composites for other decompositions that are discussed later in this chapter. In particular, the *Node Logical Subsystem Composite* and the *Node Physical Subsystem Composite* aggregate subsystems in a similar manner as described for the *Logical Subsystem Composite*. The software, hardware, and data composites represent aggregates for the hardware, software, and data components of the system, respectively, and the operational procedure is further classified into the types of procedures required to operate the system.

The *ESS Logical* block is decomposed into logical components, as shown in the *ESS Logical* block definition diagram in Figure 16.21. The system is decomposed into three classes of logical components, including *External Interface Components* to manage the interface to each external system or user; *Application*

**FIGURE 16.20**

ESS subclasses for logical and physical decomposition.

**FIGURE 16.21**

ESS logical decomposition into logical components including *External Interface Components*, *Application Components*, and *Infrastructure Components*.

Components, which are responsible for providing the business logic and processing each external item flow from the ESS context diagram in Figure 16.15; and *Infrastructure Components*, which provide internal system support services.

In the ESS logical decomposition, a *User If* is an example of an *External Interface Component*, the *Fault Manager* is an example of an *Infrastructure*

Component, and the *Event Manager* and *System Controller* are examples of *Application Components*. This approach ensures that the system logical architecture includes components with the functionality to communicate with external systems, process the inputs and outputs, and provide internal support services. The part names and multiplicities on the component ends of the composition relationships are also reflected in the *ESS Logical* internal block diagram.

Define Interaction between Logical Components to Realize System Activity and/or Operation

Activity diagrams are defined for each operation or each activity allocated to the *ESS Logical* block. This ensures that each action from the enterprise scenarios that is allocated to the system is realized in the logical design.

Figure 16.22 shows the *Monitor Intruder Activity Diagram* that realizes the *Monitor intruder* operation of the *ESS Logical* block. An enclosing activity is created with the same name as the operation called *monitor intruder*. The inputs and outputs of the enclosing activity match the pins from the *monitor intruder* action in the *Intruder Emergency Response Scenario* in Figure 16.14. The activity partitions represent the parts of the system that are typed by the logical components from the *ESS Logical Block Definition Diagram* in Figure 16.21.

The *External Sensor*, *Entry Sensor*, *Exit Sensor*, and *Internal Sensor* generate *Detections*. The *Event Manager* processes the *Detections* and stores them in the *Event Log*. The *System Controller* then controls the system actions in response to the *Event*. The control actions request the *Status Manager* to provide a status update. If the system has been activated, the controller sends a signal to trigger the alarm, to record the high-bandwidth sensor data, and to request validation of the alert. If the alert is validated, the alert status is communicated to *Emergency Services*.

The logic of this activity diagram is consistent with the system-level behavior defined in the ESS state machine in Figure 16.18. The pattern of behavior for the sensors, event manager, and controller applies to the fire and medical emergency response scenarios as well.

Some of the actions in the activity diagram include streaming inputs and outputs. The *control intruder* action includes a process constraint, which constrains the values of the inputs and outputs that can be captured and used in a parametric diagram for further engineering analysis. The *Event Log* is stored by the *Event Manager* and the external sensor data that are stored by the *High-Bandwidth Data Recorder* as indicated by the data stores in the activity partitions.

A similar set of collaboration artifacts, which can be developed at the ESS system level for each enterprise scenario, can also be developed to further specify the collaboration among logical components. The process for developing the artifacts applies the same collaboration process pattern referred to in Section 16.3.2 to each operation of the *ESS Logical* block. The collaboration artifacts include a block definition diagram, activity diagram, internal block diagram, updates to the item definitions on the block definition diagram, parametric diagram, and test cases.

For each ESS logical operation, a block definition diagram is defined, which aggregates the logical components that interact to realize the system operation.

The logical components correspond directly to the activity partitions on the activity diagram. The block that aggregates the components is referred to as a subsystem. The *Monitor Intruder Subsystem Block Definition Diagram* is shown in Figure 16.23. (Note: The *Monitor Intruder Subsystem* block is a component of the *Logical Subsystem Composite* block shown in Figure 16.20.)

The *Monitor Intruder Subsystem Internal Block Diagram* in Figure 16.24 specifies the structural interconnection between the logical components that interact in the activity diagram. The ports on the enclosing block are the external ESS system interfaces. The parts on the internal block diagram correspond to the part names on the *Monitor Intruder Subsystem Block Definition Diagram*, and to the names of the activity partitions in the activity diagram. The connectors define the interconnection between the parts. The ports on the logical parts and the item flows on the connectors have not been included for brevity. If included, the item flows are allocated from the pins on the actions in the activity diagram using the allocation relationship (refer to Chapter 13 for details). The item properties have the same type as the pins on the activity diagrams. The input and output definitions are specified on the item block definition diagrams.

Define System Logical Internal Block Diagram

The internal block diagram for the *Monitor Intruder Subsystem* showed only the interconnection among parts that participated in the *Monitor Intruder Activity Diagram*. However, there are additional activity diagrams that correspond to each operation of the *ESS Logical* block. Each activity diagram may have different sets of interacting components.

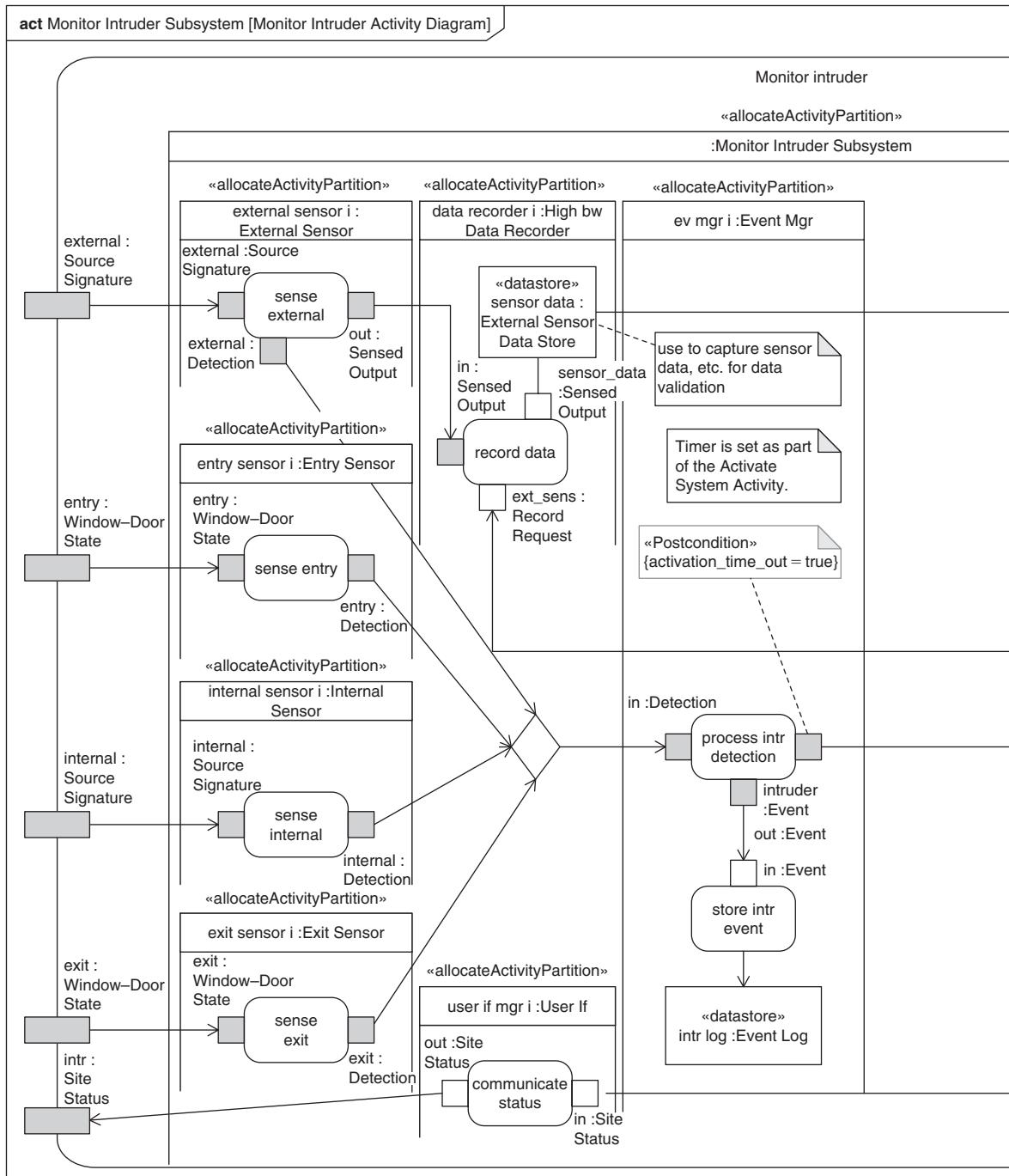
The *ESS Logical Internal Block Diagram* in Figure 16.25 represents the interconnection of all the parts from all the activity diagrams and corresponds to a traditional system block diagram (see page 440). Each subsystem corresponds to a subset of the parts and interconnections on this internal block diagram. The enclosing block represents the *ESS Logical* block. The ports on the *ESS Logical* block are consistent with the ports defined on the ESS in Figure 16.15, enabling the external interfaces to be delegated to the logical parts of the system.

Specify Logical Components

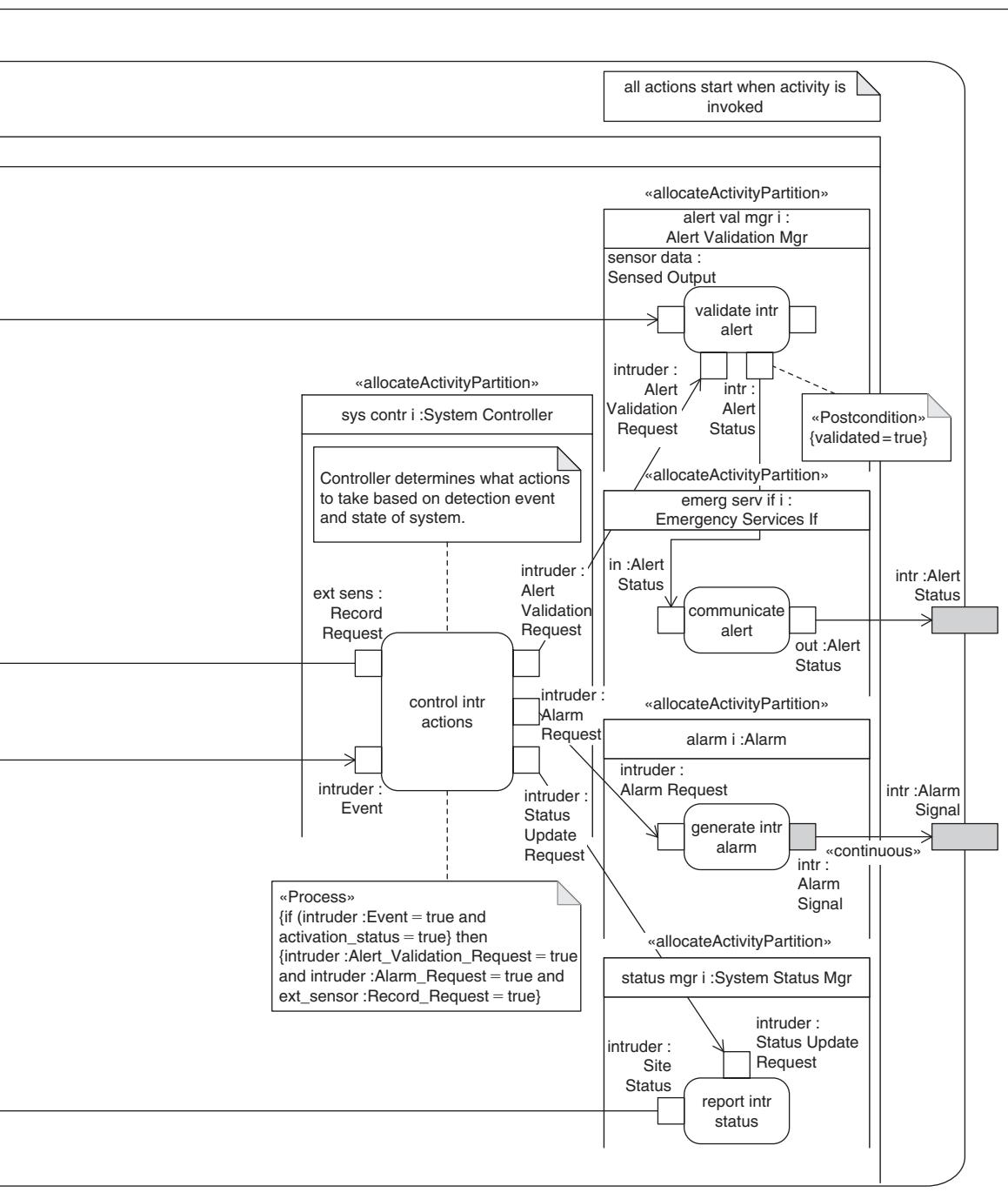
The specification of each logical component includes the specification features that are captured in their respective block in the same way that was described for specifying the *ESS* system block. The actions from the activity diagrams are captured as allocated activities or operations; the logical interfaces can be captured as the component ports; persistent stores are captured as store properties; and performance properties are captured as value properties of the block, or properties of the activity allocated to the block.

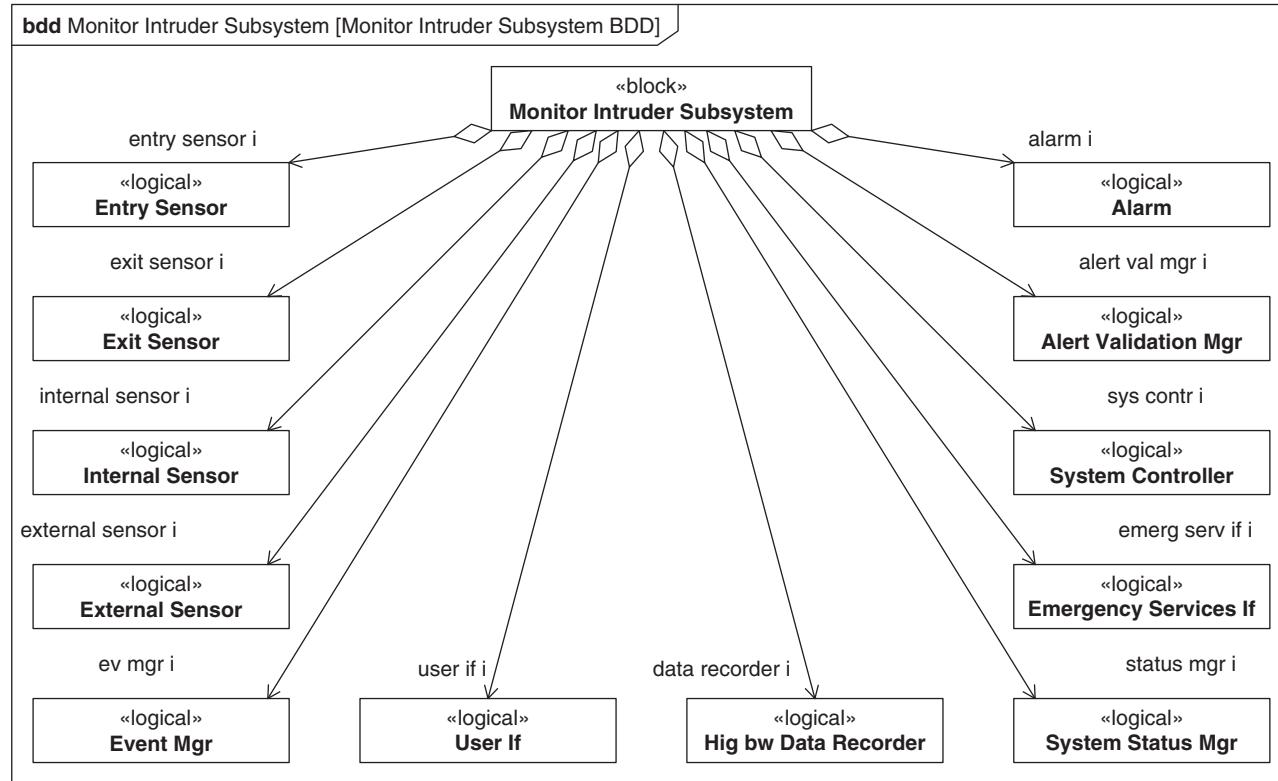
Define Logical Component State Machine

A component specification can include a state machine if it has state behavior that is dependent on input events and preconditions. A simple state-dependent behavior for a component may include a wait state, where the component waits until it receives an input event. The component then transitions to another state to execute

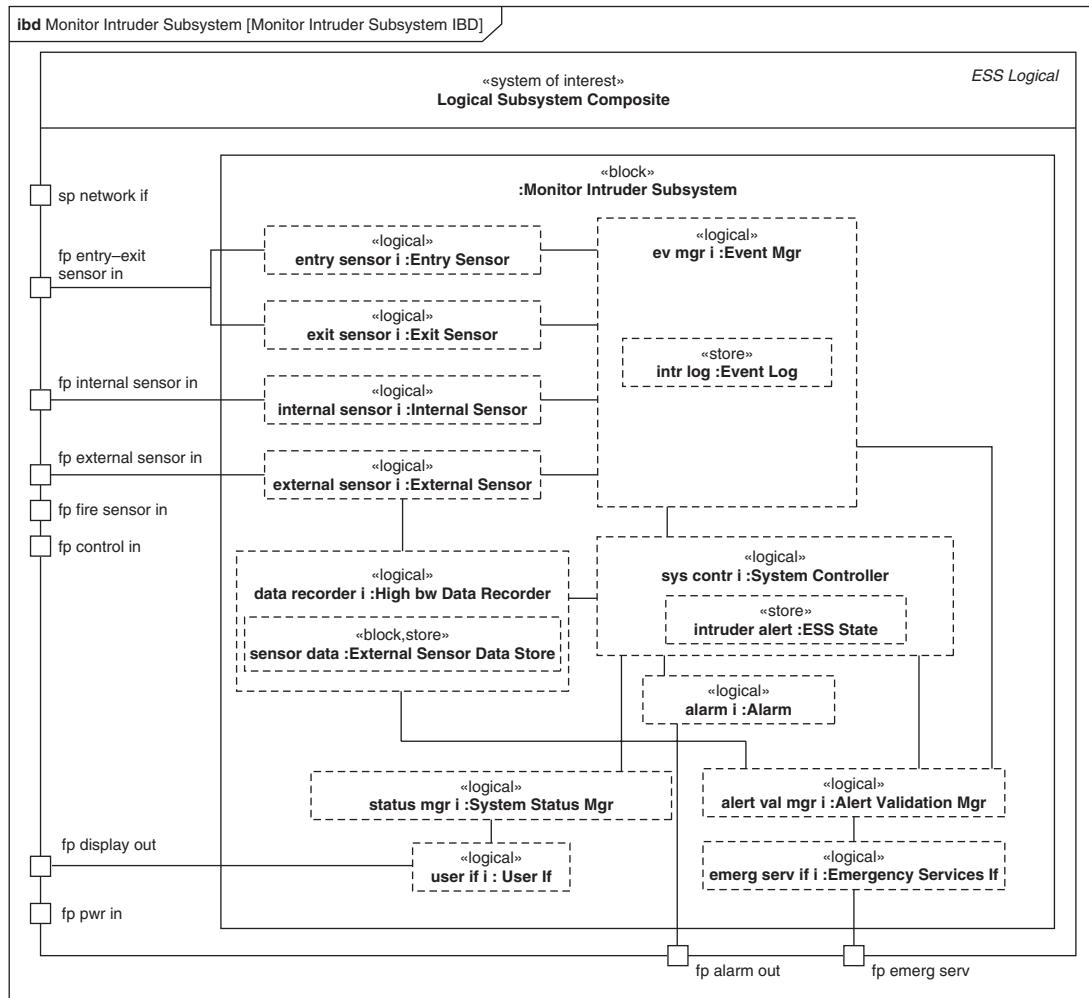
**FIGURE 16.22**

ESS *Monitor Intruder Activity Diagram* is a thread through the logical system design that realizes the *Monitor intruder* operation of the *ESS Logical* block.



**FIGURE 16.23**

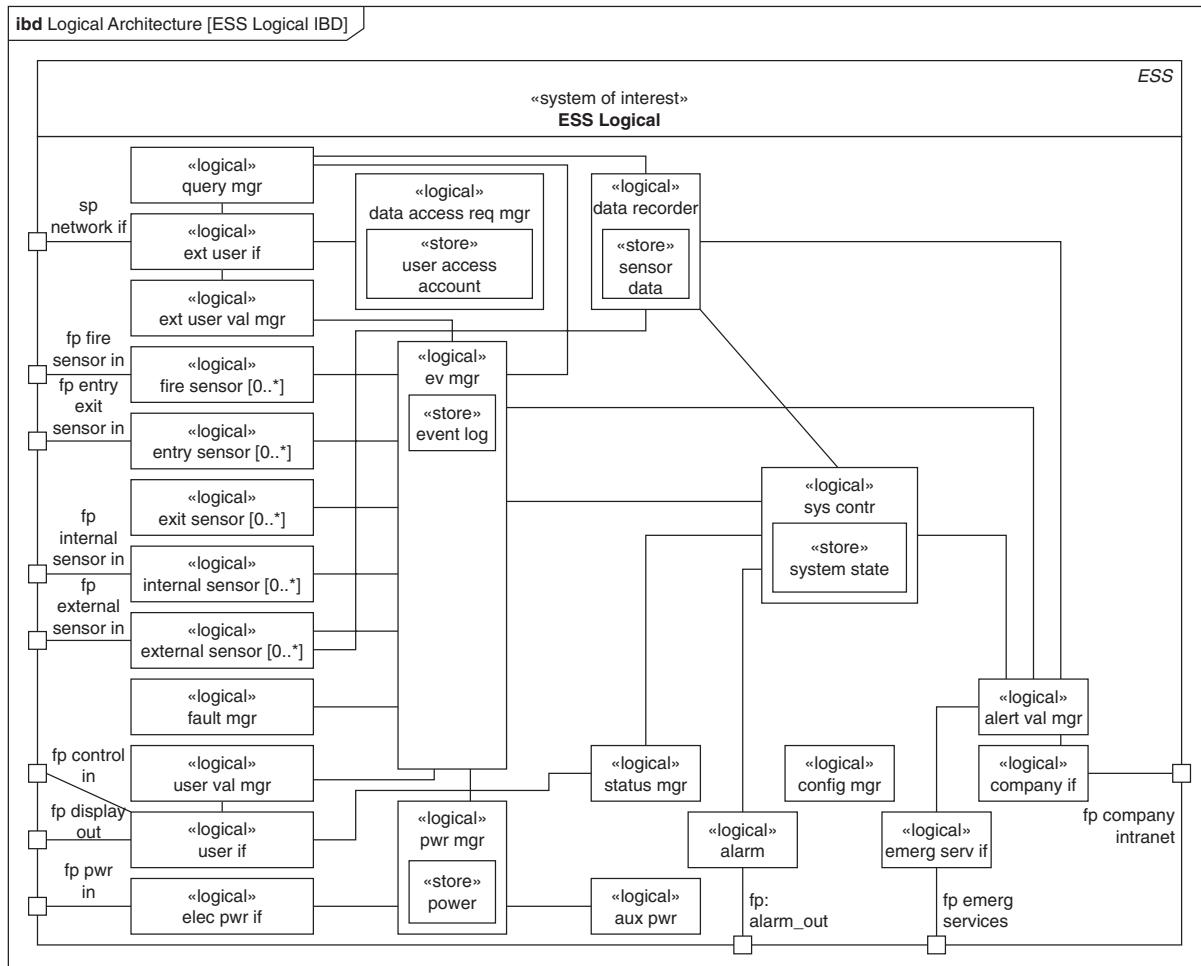
Monitor Intruder Subsystem Block Definition Diagram aggregates the components that interact in the Monitor Intruder Activity Diagram.

**FIGURE 16.24**

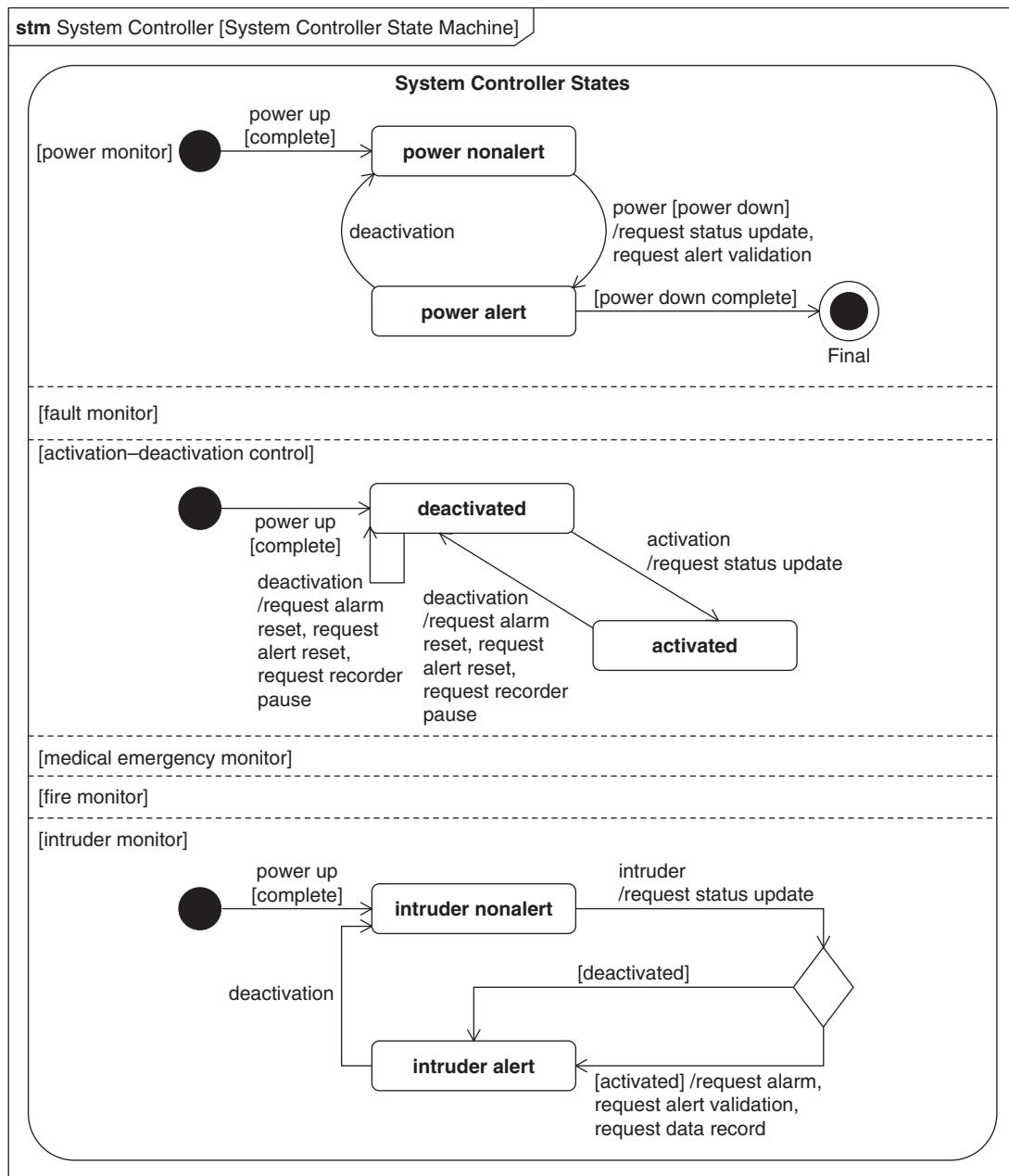
Monitor Intruder Subsystem Internal Block Diagram showing the interconnection of the parts that interact on the *Monitor Intruder Activity Diagram* in Figure 16.22. (Note: The parts are references that are not owned by the subsystem.)

a particular do/behavior that is defined by an activity. It then transitions back to its wait state when the activity is complete and waits for the next triggering event.

For this example, the *Event Manager* and the *System Controller* are logical components that have more complex state-dependent behavior. The *System Controller* is a logical component that is responsible for controlling actions in response to events from the *Event Manager*. Since the controller must respond differently to different events, and its behavior is also dependent on the current state of the system, it is appropriate to represent the controller's behavior with a state machine, as shown in a partial view of its state machine in Figure 16.26.

**FIGURE 16.25**

ESS Logical Internal Block Diagram showing the interconnection between all logical components of the system.

**FIGURE 16.26**

System Controller State Machine.

The system controller includes multiple regions to support concurrent states. As shown in the *intruder monitor* region, when powerup is complete, the *System Controller* transitions to an *intruder nonalert* state. If an intruder event is received, the controller sends a status update and then transitions to the *intruder alert* state. If the system is in the *activated* state, the *System Controller* sends signals to other logical components as indicated on the transition actions. These include a *request alarm* to the *Alarm*, a *request alert validation* to the *Alert Validation Manager*, and a *request data record* to the *High-Bandwidth Data Recorder*. The signals are not sent if the system is not in the *activated* state.

The state machine augments the specification of the logical components. The traceability between the system-level requirements and the logical components is maintained, as discussed in Section 16.3.6.

16.3.4 Synthesize Candidate Physical Architectures

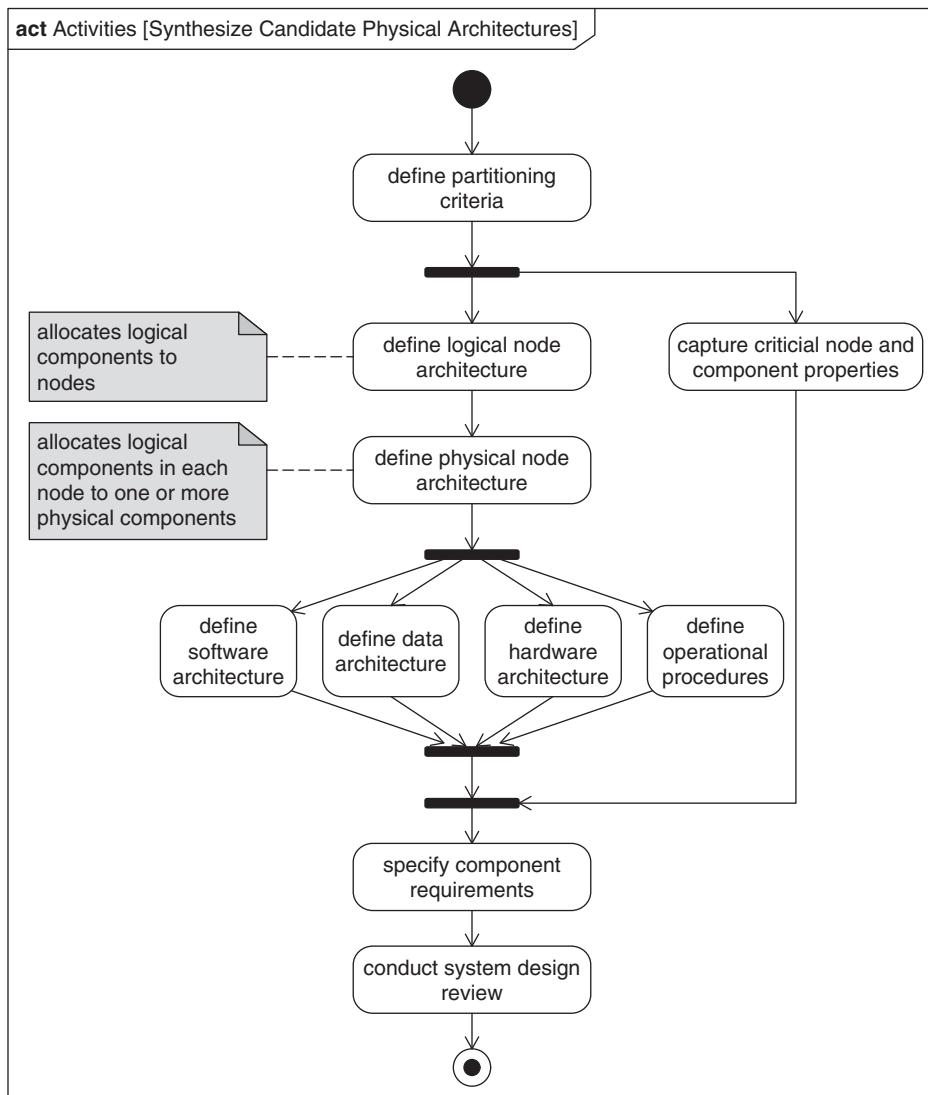
The *Synthesize Candidate Physical Architectures* activity is shown in Figure 16.27. This activity synthesizes the system's alternative physical architectures to satisfy the system requirements. The architecture is defined in terms of its physical components and relationships, and their distribution across system nodes. The physical components of the system include hardware, software, persistent data, and operational procedures. The system nodes represent a partitioning of components based on their physical location or other distribution criteria. If it is not distributed, the system is assumed to consist of a single node.

The partitioning criteria are defined and used to partition the physical components and address concerns such as performance, reliability, and security. A node logical architecture is defined to determine how the logical components, and their associated functionality, persistent data, and control, are distributed across system nodes. A node physical architecture is defined where each logical component in each node is allocated to physical components that may include a combination of hardware, software, and persistent data components, as well as operational procedures performed by operators. System design constraints that were identified in Section 16.3.2 are imposed on the physical architecture.

The software, hardware, and data architecture are established to further partition the physical components based on additional physical domain-specific implementation concerns. The requirements are then specified for each physical component and traced to the system requirements. Engineering analysis and trade studies are performed to evaluate, select, and refine the preferred architecture. It should be noted that trade studies are performed throughout the OOSEM process beginning with *Analyze Stakeholder Needs*.

Define Partitioning Criteria

Partitioning is a fundamental aspect of systems architecting. Partitioning criteria are established to partition functionality, persistent data, and control among the logical and physical components, and to partition the components among subsystems, nodes, and layers of the architecture. Applying partitioning criteria throughout the design process should result in component designs that exhibit maximum

**FIGURE 16.27**

Synthesize Candidate Physical Architectures activity to specify the components of the system.

cohesion and minimum coupling to reduce interface complexity. Applying the criteria should also reduce the impact of requirements and technology changes and more effectively address key requirements such as performance, reliability, maintainability, and security. Some examples of partitioning include the following:

- Refactoring common functionality into shared components
- Partitioning components and functionality based on having the same update rate, or partitioning components with high update rates versus those with low update rates

- Partitioning software components into architecture layers based on the level of dependency of the functionality or services they provide
- Partitioning data into separate repositories based on their security classification level
- Physical partitioning such that lower reliability components are more accessible to ease maintainability
- Physical partitioning of components to reduce the number of moving parts for assembly and disassembly
- Partitioning components based on reuse of common patterns
- Partitioning components based on their likelihood to change
- Partitioning functionality and components based on development considerations such as whether they are part of a particular incremental delivery

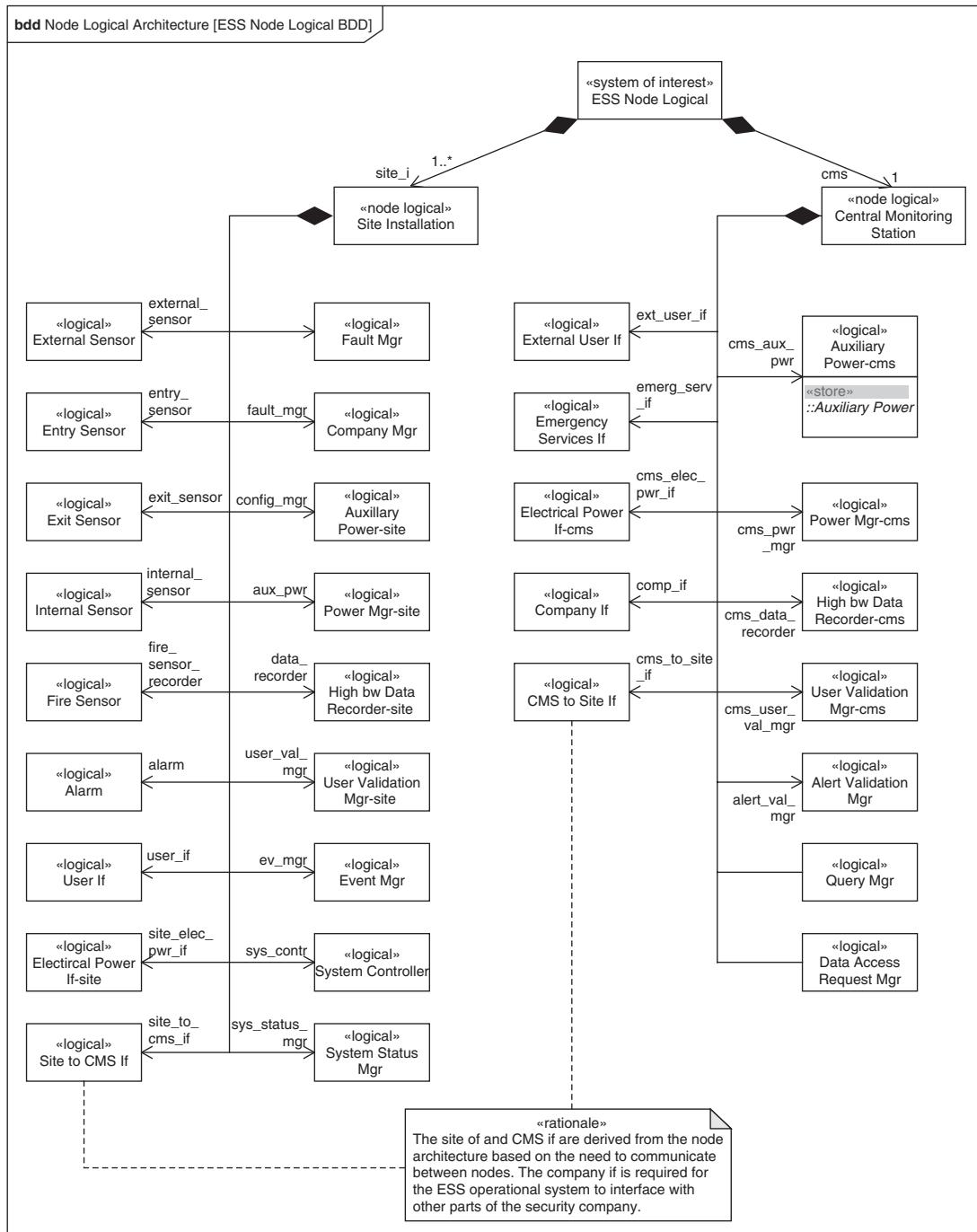
Define Logical Node Architecture

Up to this point, there has been no discussion of how the functionality is distributed across system nodes. A node typically represents a partitioning of components and associated functionality, control, and persistent data based on the geographic location of the components. The node may include a fixed facility or a moving platform such as an aircraft. Many modern systems are distributed across multiple system nodes. Nodes may also be defined based on other criteria such as organizational responsibility (e.g., the people and resources in a particular department). In OOSEM, a logical node represents an aggregation of logical components at a particular location. A physical node represents an aggregation of physical components at a particular location. The logical components at a logical node are allocated to physical components at a physical node, as described later in this section.

Functionality, control, and persistent data can be distributed in many ways. A system can be highly distributed such that each node has complete functionality, control, and data and can operate autonomously. Alternatively, the distribution may be highly centralized where most of the functionality, control, and data are within a central node, and the local nodes primarily provide an interface to external systems and users at a particular location. This can provide a cost-effective solution to minimize the required resources across the system. Between a fully distributed solution and fully centralized distribution, functionality, control, and data can be partially distributed across regional and local nodes, where each node performs a subset of the total functionality, control, and data.

Distribution options can include any combination of a central node, multiple regional nodes, and multiple local nodes in each region. Trade studies are typically performed to optimize the distribution approach based on considerations such as performance, reliability, security, and cost. Many types of systems are distributed including information systems with networked communications, electrical power distribution systems, and complex SoS applications such as transportation systems.

The *ESS Node Logical Block Definition Diagram* defines the logical components in a node hierarchy as shown in Figure 16.28. The top block in the block definition diagram is a subclass of the *ESS* block called the *ESS Node Logical*,

**FIGURE 16.28**

ESS Node Logical Block Definition Diagram showing the logical components allocated to the *Site Installation* and *Central Monitoring Station* nodes.

which was shown previously in Figure 16.20. This block is composed of the system logical nodes and stereotyped as «node logical». For the ESS, the nodes represent the *Central Monitoring Station* (CMS), and the *Site Installations* that are installed at *Single-Family Residences*, *Multifamily Residences*, and small *Businesses*. Although not included in this example, a CMS backup facility may be an additional required node to provide disaster recovery and satisfy the system availability requirement.

Each logical node is composed of logical components. A logical component can be distributed to more than one node. However, the logical component may have different requirements in each node, as is the case for the high-bandwidth data recorder that is a component of both the *Site Installation* and the *Central Monitoring Station*. In this case a subclass of the *High-Bandwidth Data Recorder* logical component is defined for each node with its unique characteristics.

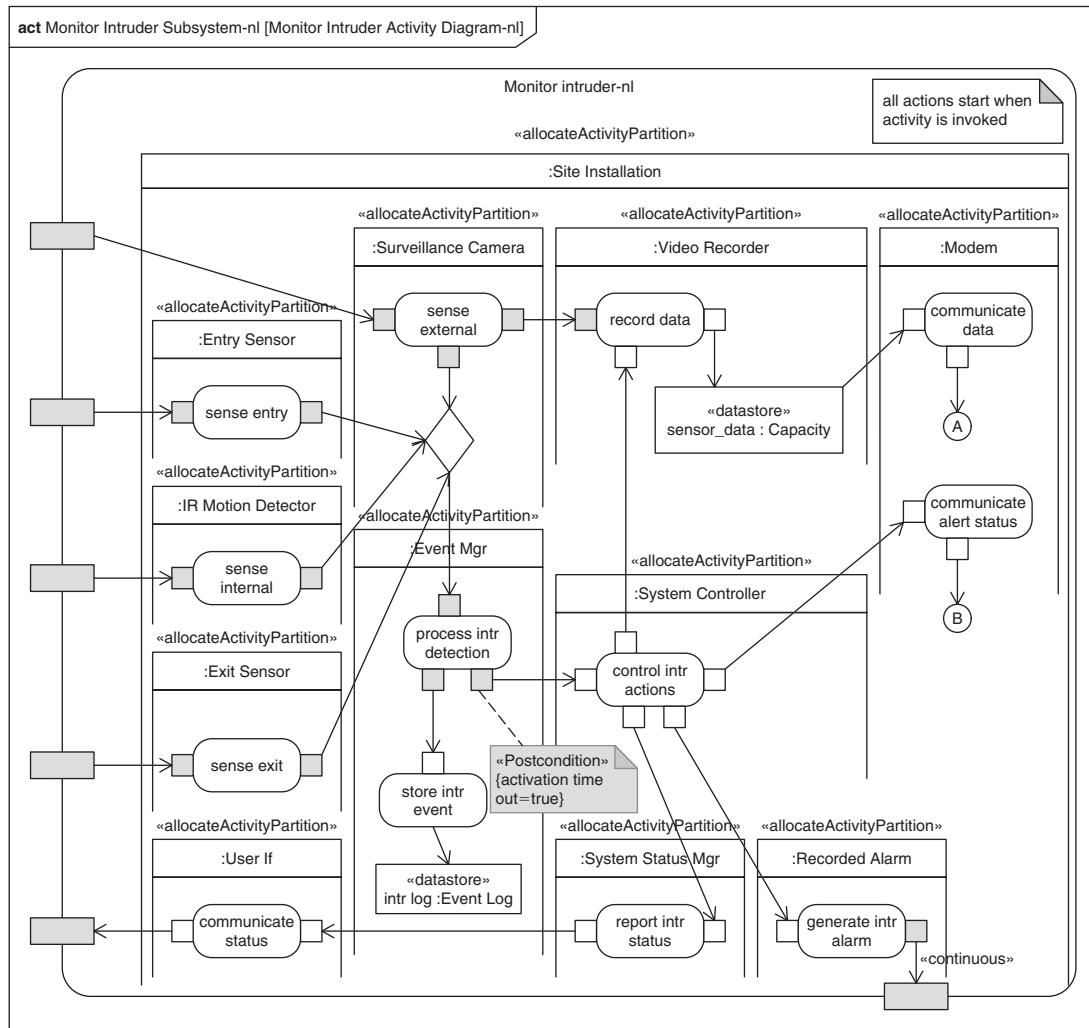
A similar set of modeling artifacts used for defining the *ESS Logical* architecture in the previous section can also be developed for the *ESS Node Logical* architecture. This includes the node logical activity diagrams and internal block diagram. An elaboration of each activity diagram that was created for the *ESS Logical* architecture in the previous section should be created for the *ESS Node Logical* architecture to specify how the activity is executed by the logical components that are distributed across the nodes.

The node logical activity diagrams show the interaction of the components within each node and across nodes. The *Monitor Intruder Activity Diagram-nl* is shown in Figures 16.29 and 16.30. The nodes are represented as activity partitions, and the logical components are nested within their respective node. New logical components are required to support the interaction between nodes that are also included on the *Node Logical Block Definition Diagram*. This activity diagram is consistent with the behavior that was originally specified in the *Monitor Intruder Activity Diagram* as part of the undistributed logical design in the activity diagram in Figure 16.22.

The *ESS Node Logical Internal Block Diagram-nl* in Figures 16.31 and 16.32 shows how the logical components are interconnected within each node and across nodes. This includes the interconnection of parts that are interacting in the *Monitor Intruder Activity Diagram*. Once again, the system external interfaces are maintained on the ports of the enclosing block. The other collaboration artifacts referred to in the previous section can be created for each scenario as well, depending on the process tailoring.

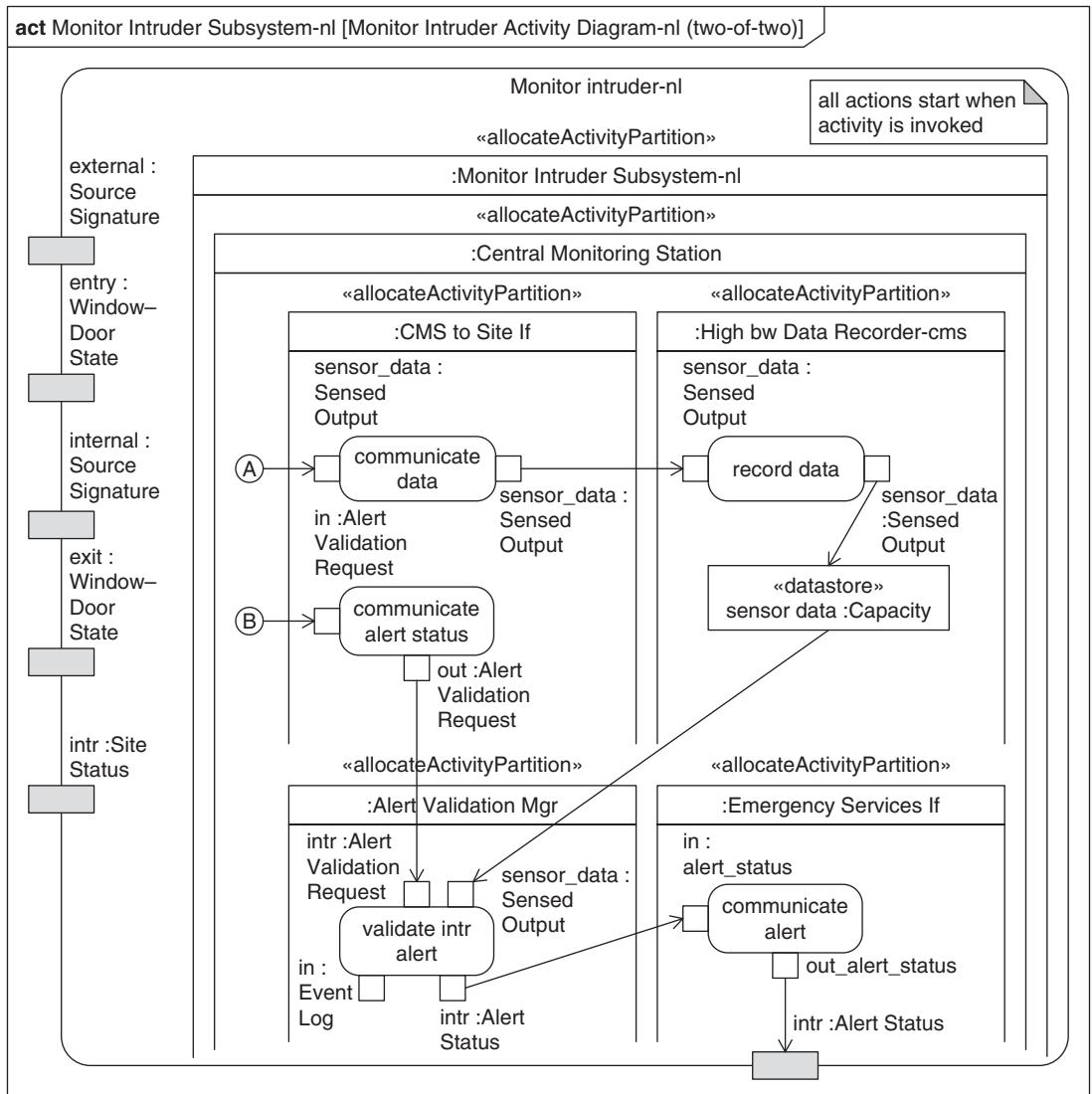
Define Physical Node Architecture

The functionality for the *ESS Logical* architecture is first distributed among the logical nodes and captured in the *ESS Node Logical* architecture as described in the previous section. This was accomplished by distributing the logical components to each of the logical nodes based on partitioning concerns that are somewhat independent of how the components are implemented. For example, it made sense to distribute the *Entry Sensor* to the *Site Installation* node independent of what technology is used to implement the *Entry Sensor*. The logical components in each node are then allocated to physical components in each

**FIGURE 16.29**

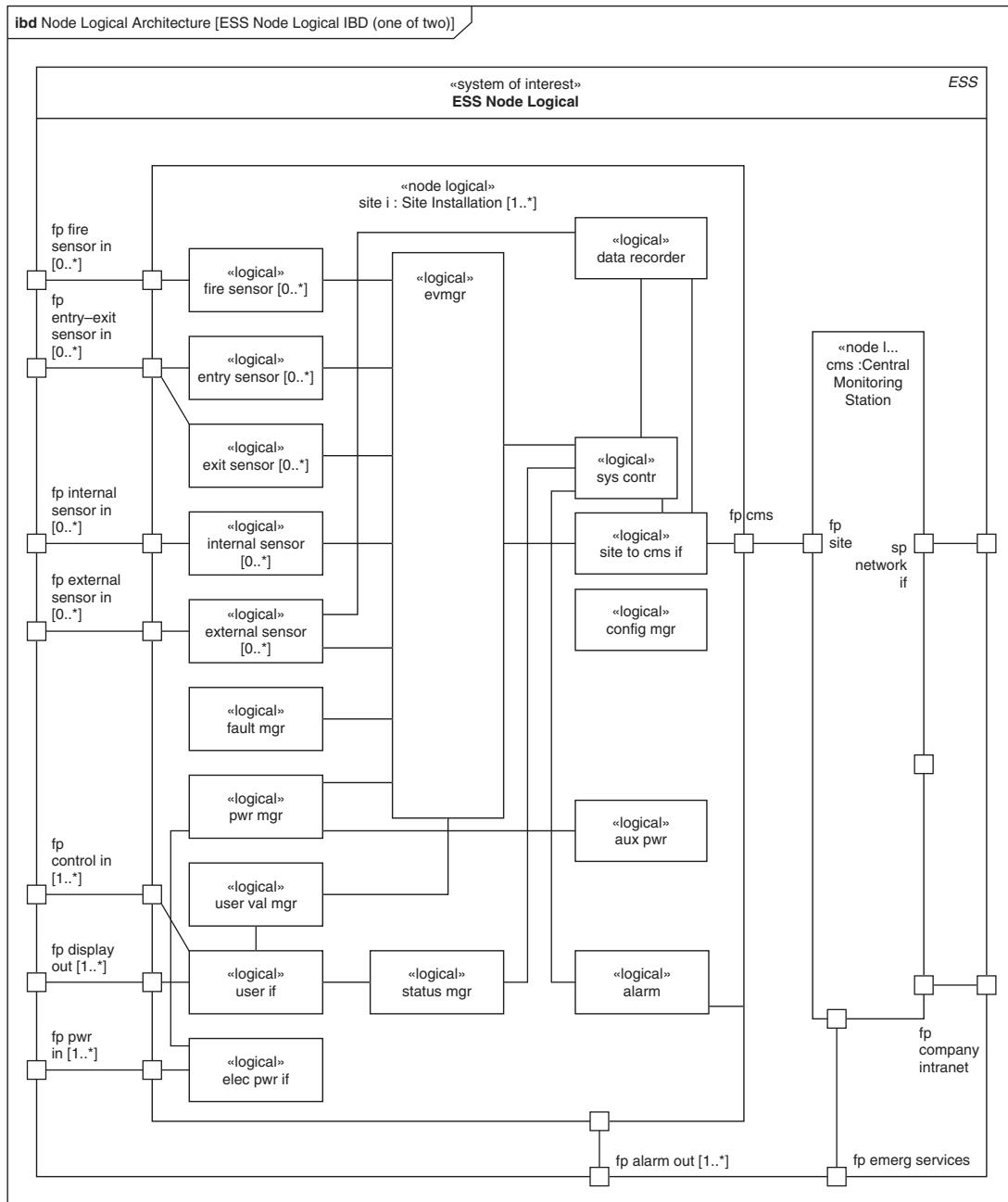
Monitor Intruder Activity Diagram-nl showing the interaction of components within the *Site Installation* node. Additional components have been added to interface between the nodes.

node to constitute the *ESS Node Physical* architecture. The supporting trade-off analysis, which addresses technology and implementation concerns related to performance, reliability, security, and other quality attributes, is addressed as part of this allocation decision. For this example, the *Entry Sensor* is allocated to an *Optical Sensor*. A partial allocation of the logical components to physical components for the *Site Installation* node and the *Central Monitoring Station* node are shown in the allocation tables in Figures 16.33 and 16.34, respectively (see pages 451–452).

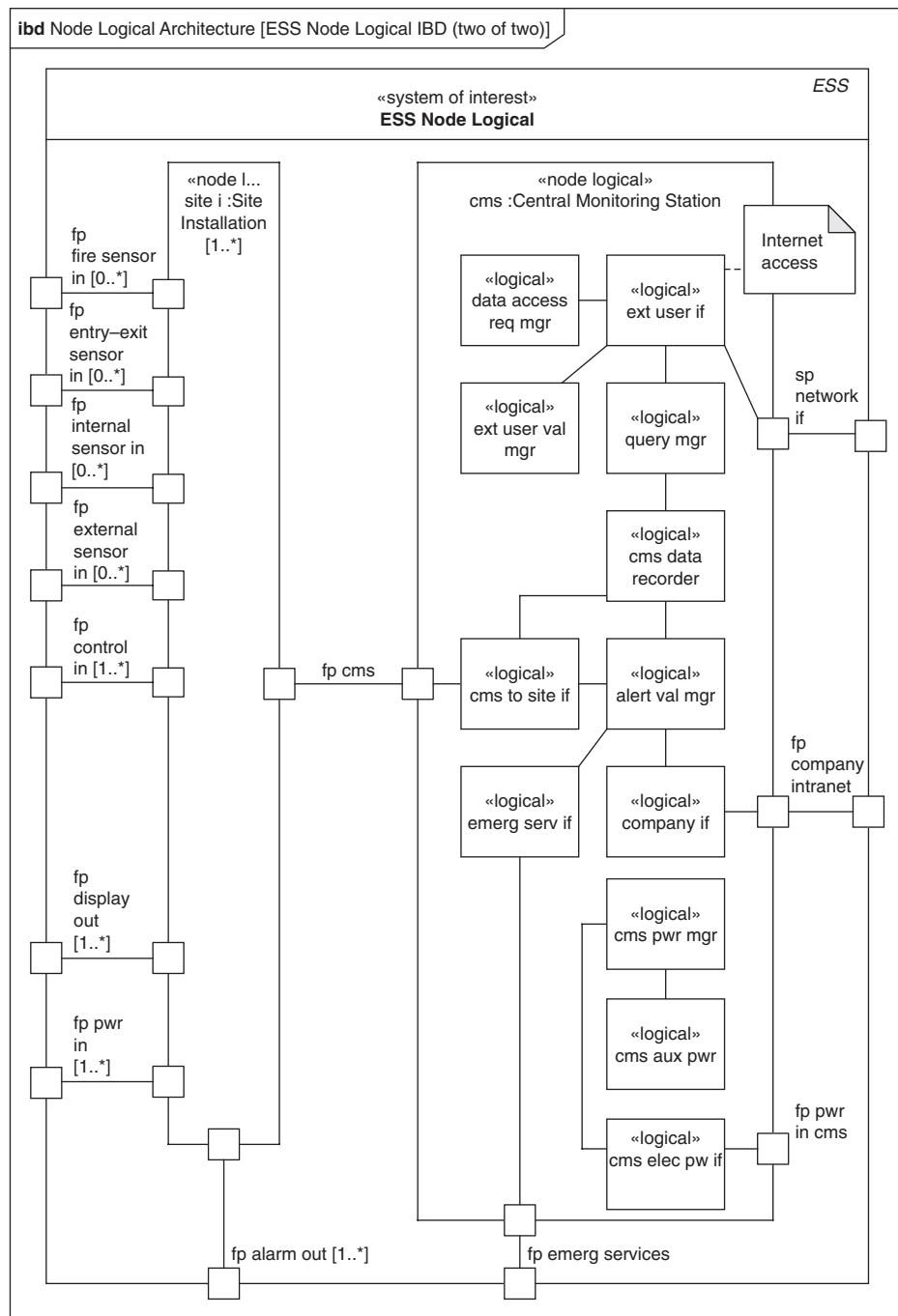
**FIGURE 16.30**

Monitor Intruder Activity Diagram-nl showing the interaction of components within the *Central Monitoring System* node. Additional components have been added to interface between the nodes. This behavior also partially supports the behavior that was specified in the *Monitor Intruder Activity Diagram* in Figure 16.22.

The design constraints that were identified during the system requirements analysis in Section 16.3.2 are imposed on the physical architecture as part of the logical-to-physical allocation. For example, a logical component may be allocated to a particular COTS component that has been imposed as a design constraint. A reference architecture may also constrain the solution space with preselected

**FIGURE 16.31**

ESS Node Logical Internal Block Diagram showing the interconnection between the parts within and across nodes, focused on the *Site Installation*. The parts are represented as activity partitions in the *Monitor Intruder Activity Diagram-nl* in Figure 16.29.

**FIGURE 16.32**

ESS Node Logical Internal Block Diagram showing the interconnection between the parts within and across nodes, focused on the *Central Monitoring Station*. The parts are represented as activity partitions in the *Monitor Intruder Activity Diagram-nl* in Figure 16.30.

From Type	From Name	Relation	To Type	To Name
logical	Alarm	allocate	hardware	Light Alarm
logical	Alarm	allocate	hardware	Sound Alarm
logical	Alarm	allocate	hardware	Recorded Alarm
logical	Alarm	allocate	software	Alarm If
logical	Auxillary Power-site	allocate	software	Battery If
logical	Auxillary Power-site	allocate	hardware	Battery
logical	Company Mgr	allocate	software	Config Mgr
logical	Electrical Power If-site	allocate	hardware	Power Adapter-site
logical	Entry Sensor	allocate	hardware	Magnetic Sensor
logical	Entry Sensor	allocate	software	Optical Sensor If
logical	Entry Sensor	allocate	hardware	Contact Sensor
logical	Entry Sensor	allocate	hardware	Optical Sensor
logical	Event Mgr	allocate	software	File If
logical	Event Mgr	allocate	block	Event Log Data File
logical	Event Mgr	allocate	software	Event Mgr
logical	Exit Sensor	allocate	hardware	Magnetic Sensor
logical	Exit Sensor	allocate	hardware	Contact Sensor
logical	Exit Sensor	allocate	hardware	Optical Sensor
logical	Exit Sensor	allocate	software	Optical Sensor If
logical	External Sensor	allocate	hardware	Electric Fence
logical	External Sensor	allocate	hardware	Surveillance Camera
logical	External Sensor	allocate	software	Camera If
logical	Fault Mgr	allocate	software	Fault Mgr
logical	Fire Sensor	allocate	software	Fire Detector If
logical	Fire Sensor	allocate	hardware	Fire Detector
logical	High bw Data Recorder-site	allocate	hardware	Video Recorder
logical	High bw Data Recorder-site	allocate	software	Video Recorder If
logical	High bw Data Recorder-site	allocate	hardware	DVD
logical	Internal Sensor	allocate	software	IR Motion Detector If
logical	Internal Sensor	allocate	hardware	IR Motion Detector
logical	Power Mgr-site	allocate	software	Pwr Distr Hub If
logical	Power Mgr-site	allocate	hardware	Power Distribution Hub-site
logical	Power Mgr-site	allocate	software	Power Mgr-site
logical	Site to CMS If	allocate	hardware	Router
logical	Site to CMS If	allocate	hardware	Modem
logical	Site to CMS If	allocate	hardware	Network Interface Card
logical	Site to CMS If	allocate	software	Modem If
logical	Site to CMS If	allocate	software	Network If
logical	User If	allocate	hardware	Router
logical	User If	allocate	hardware	Network Interface Card
logical	User If	allocate	hardware	Console
logical	User If	allocate	software	Console If
logical	User If	allocate	software	Network If
logical	User Validation Mgr-site	allocate	software	User Validation Mgr-site

FIGURE 16.33

Allocation of logical components to physical components in *Site Installation* node.

or legacy components. An example of a reference architecture is briefly described later in this section as a multilayered architecture that includes specific types of components associated with each architecture layer—that is, presentation, mission application, infrastructure, and operating system layers.

Alternative physical architectures are identified by allocations of logical components to alternative physical components. The logical-to-physical component

table [Package] CMS [Logical to Physical]

From Type	From Name	Relation	To Type	To Name
logical	Alert Validation Mgr	allocate	software	User If
logical	Alert Validation Mgr	allocate	hardware	Work station-Security Operator
logical	Alert Validation Mgr	allocate	procedure	CMS Security Operator Procedure
logical	Alert Validation Mgr	allocate	software	Alert Mgr
logical	Auxillary Power-cms	allocate	software	Generator If
logical	Auxillary Power-cms	allocate	hardware	Power Generator
logical	CMS to Site If	allocate	software	Network If
logical	CMS to Site If	allocate	software	Modem If
logical	CMS to Site If	allocate	hardware	Modem
logical	CMS to Site If	allocate	software	Firewall
logical	CMS to Site If	allocate	hardware	Router
logical	CMS to Site If	allocate	hardware	Network Interface Card
logical	Company If	allocate	software	Web Browser
logical	Company If	allocate	hardware	Network Interface Card
logical	Data Access Request Mgr	allocate	procedure	CMS Administrator Procedure
logical	Data Access Request Mgr	allocate	block	User Access Account Database
logical	Data Access Request Mgr	allocate	software	Data Access Request Mgr
logical	Data Access Request Mgr	allocate	software	User If
logical	Electrical Power If-cms	allocate	hardware	Power Adapter-cms
logical	Emergency Services If	allocate	hardware	Modem
logical	Emergency Services If	allocate	hardware	Dedicated High bw If
logical	Emergency Services If	allocate	software	Modem If
logical	External User If	allocate	hardware	Network Interface Card
logical	External User If	allocate	software	Firewall
logical	External User If	allocate	hardware	Modem
logical	External User If	allocate	software	Modem If
logical	External User If	allocate	hardware	Router
logical	External User If	allocate	software	Network If
logical	High bw Data Recorder-cms	allocate	hardware	Video Server
logical	High bw Data Recorder-cms	allocate	hardware	Video Storage Device
logical	High bw Data Recorder-cms	allocate	software	Video If
logical	Power Mgr-cms	allocate	software	Power Mgr-cms
logical	Power Mgr-cms	allocate	software	Pwr Distr Hub If
logical	Power Mgr-cms	allocate	hardware	Power Distribution Hub-cms
logical	Query Mgr	allocate	software	DBMS If
logical	Query Mgr	allocate	software	DBMS
logical	User Validation Mgr-cms	allocate	software	Data Access Mgr

FIGURE 16.34

Allocation of logical components to physical components in *Central Monitoring Station* node.

allocations may be based on patterns. The architectural patterns may represent common solutions with associated technologies. For example, the *Event Manager* and *System Controller* constitute a design pattern in the logical design that can be implemented in a selected software design pattern.

Trade studies are performed to select the preferred physical architecture based on selection criteria that optimize the measures of effectiveness and associated measures of performance. In this example, the probability of intruder detection and false alarm may drive the *Site Installation* performance requirements, and the number and type of *Site Installations* that are monitored may drive the *Central Monitoring Station* performance requirements. Performance requirements

must be balanced against availability, cost, and other critical requirements to arrive at a balanced solution based on the trade-off analysis.

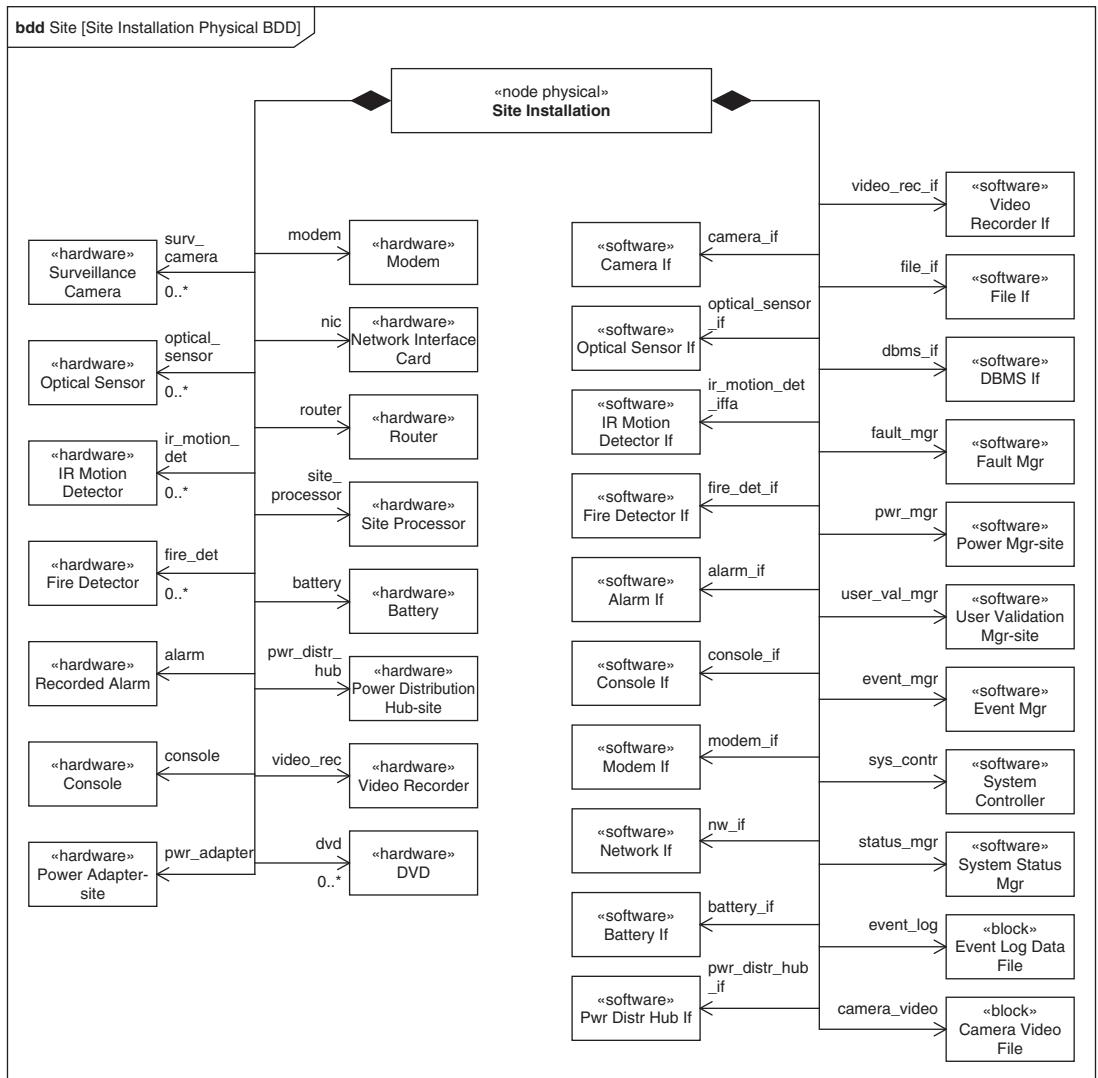
When a logical component is allocated to software, the software component must also be allocated to a corresponding hardware component to execute it. This can also be reflected in the allocation tables in Figures 16.33 and 16.34, although they are not shown. Sometimes, the allocation decision is made at run time. For allocations of software to hardware at run time, the run-time allocation decision process is also modeled as part of the activity diagram or sequence diagram. The run-time allocation methods may require algorithms, such as load balancing, to implement the decision process. In addition to software allocation, persistent data are allocated to hardware components that store the data, and operational procedures are allocated to operators that execute the procedures.

The *ESS Node Physical Block Definition Diagrams* for the *Site Installation* and *Central Monitoring Station* are shown in Figures 16.35 and 16.36, respectively. They are similar to the *ESS Node Logical Block Definition Diagram* in Figure 16.28 except the logical components have been replaced by the physical components to which they were allocated, and the *Site Installation* and *Central Monitoring Station* nodes are physical nodes instead of logical nodes.

Similar modeling artifacts that were created for the ESS node logical architecture in the previous section are created for the ESS node physical architecture, including the collaboration artifacts. This includes the node physical activity diagrams and the node physical internal block diagram. A node physical activity diagram is created for each node logical activity diagram, which in turn represents a realization of a required black-box system behavior. Node physical activity diagrams must support the behavior specified by node logical activity diagrams.

The *Monitor Intruder Activity Diagram-np* for the *Site Installation* and the *Central Monitoring Station* are shown in Figures 16.37 and 16.38, respectively. The activity partitions represent the components of the system's physical architecture. The activity diagram captures the interaction between the hardware and software components, as well as the operators of the system. The activity partitions for the site installation software components are nested within a partition that represents a site installation software configuration item. The software executes on a processing platform, although this is not shown as an activity partition in the activity diagram. Similarly, other activity partitions represent the hardware components and operators.

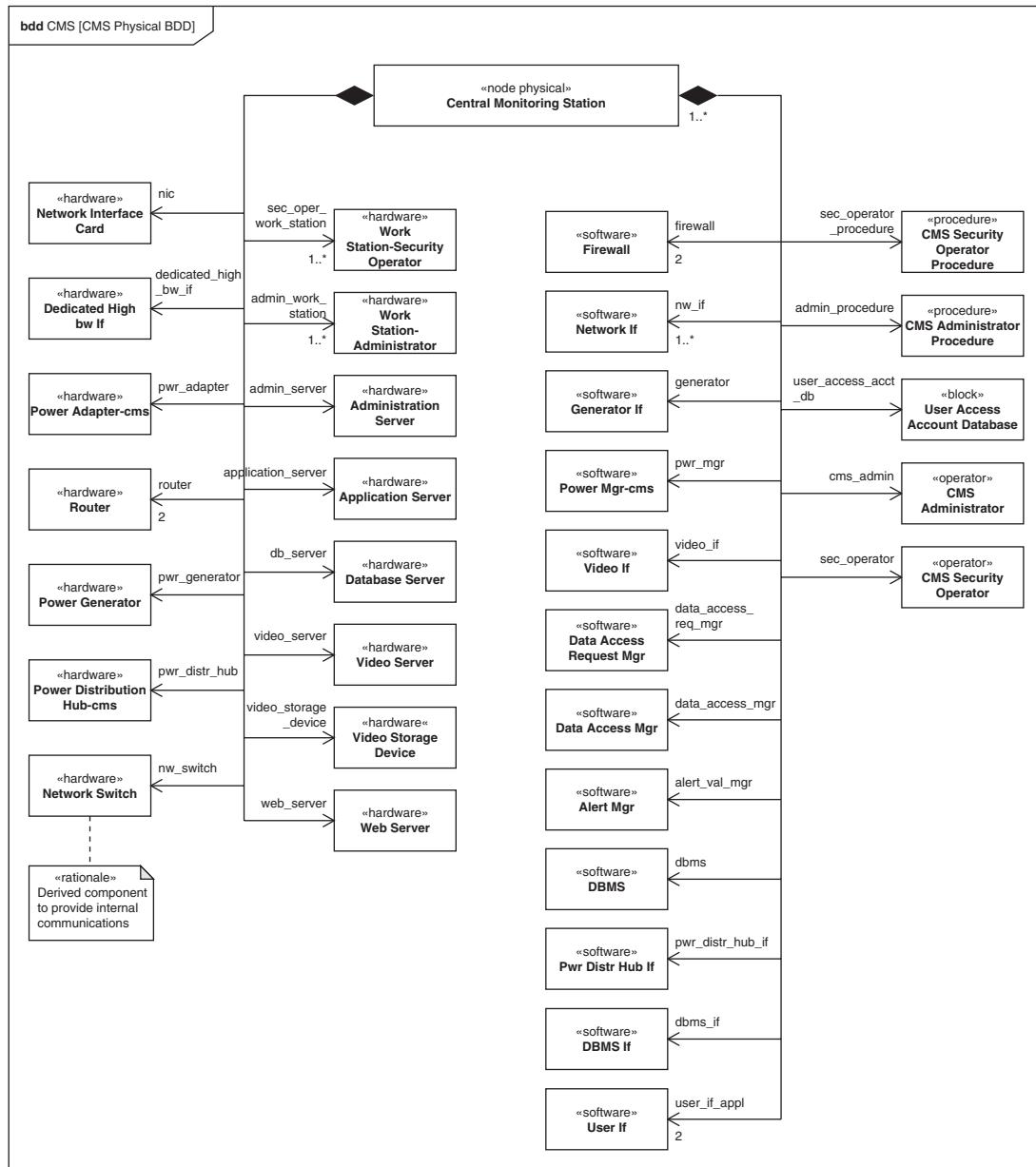
The activity diagram must support the behavior from the corresponding node logical activity diagram, and also support the original behavior specified for the *Monitor intruder* action of the *Intruder Emergency Response Scenario* in Figure 16.14, including its inputs, outputs, and pre- and postconditions. Supporting the behavior of the higher-level abstraction means that the logical behavior described previously is maintained in terms of the inputs, outputs, and flow of control, but that more detail is added to show how this behavior is accomplished when the logical behavior is distributed across nodes. In this example, the *control intruder actions* is accomplished at the *Site Installation* node and the *validate intruder alert* is accomplished at the *Central Monitoring Station* node.

**FIGURE 16.35**

Site Installation Physical Block Definition Diagram showing the hierarchy of physical components in the *Site Installation* node.

As a result, this activity diagram includes communication actions to communicate the control actions from the *Site Installation* to the *Central Monitoring Station*. The communication actions represent the added detail that was not included in the ESS logical activity diagram in Figure 16.22. However, the overall behavior of the logical activity diagram is maintained.

The *ESS Node Physical Internal Block Diagram* in Figures 16.39 and 16.40 (see pages 458–459) show how the physical parts are interconnected within each

**FIGURE 16.36**

Central Monitoring Station Physical Block Definition Diagram showing the hierarchy of physical components in the Central Monitoring Station node.

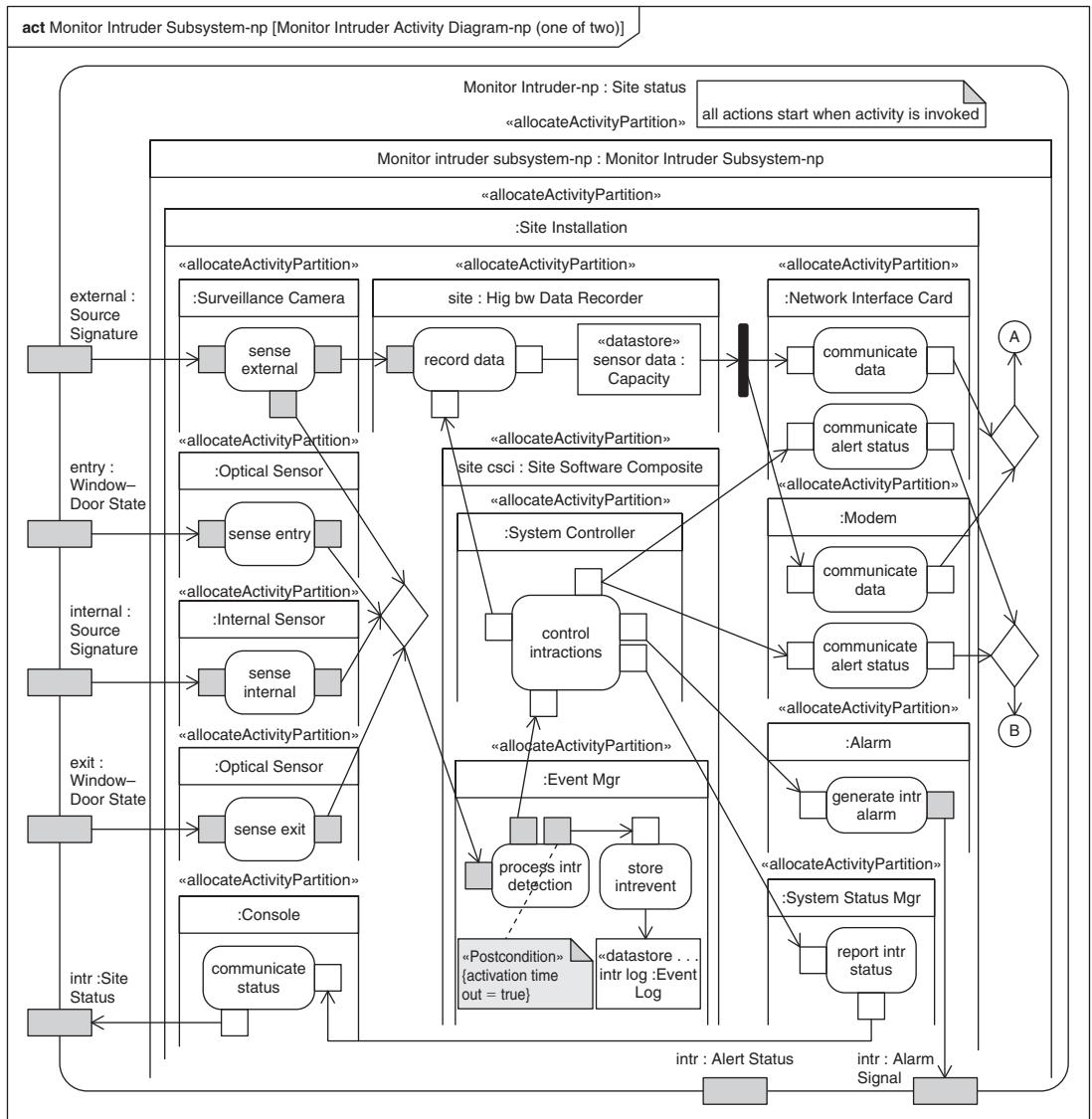
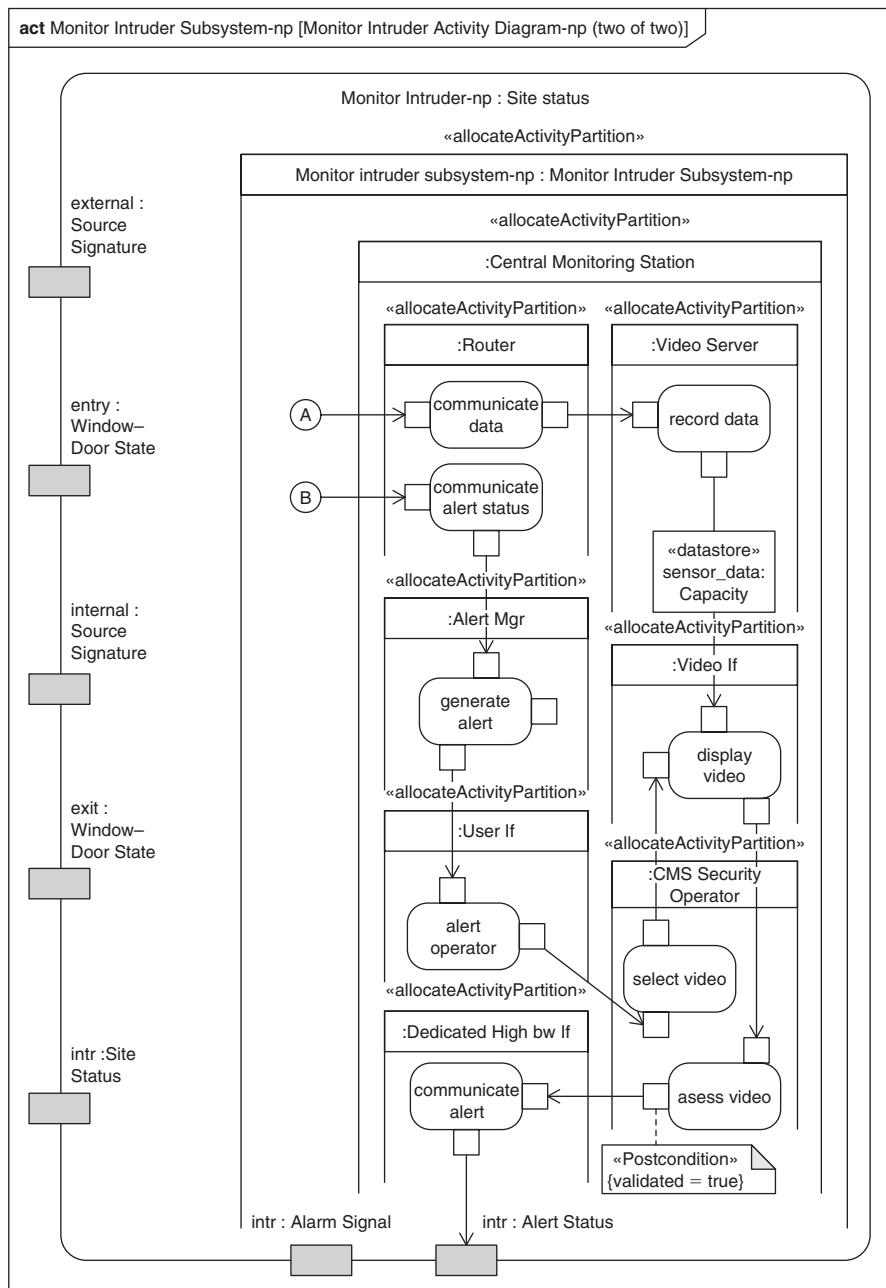


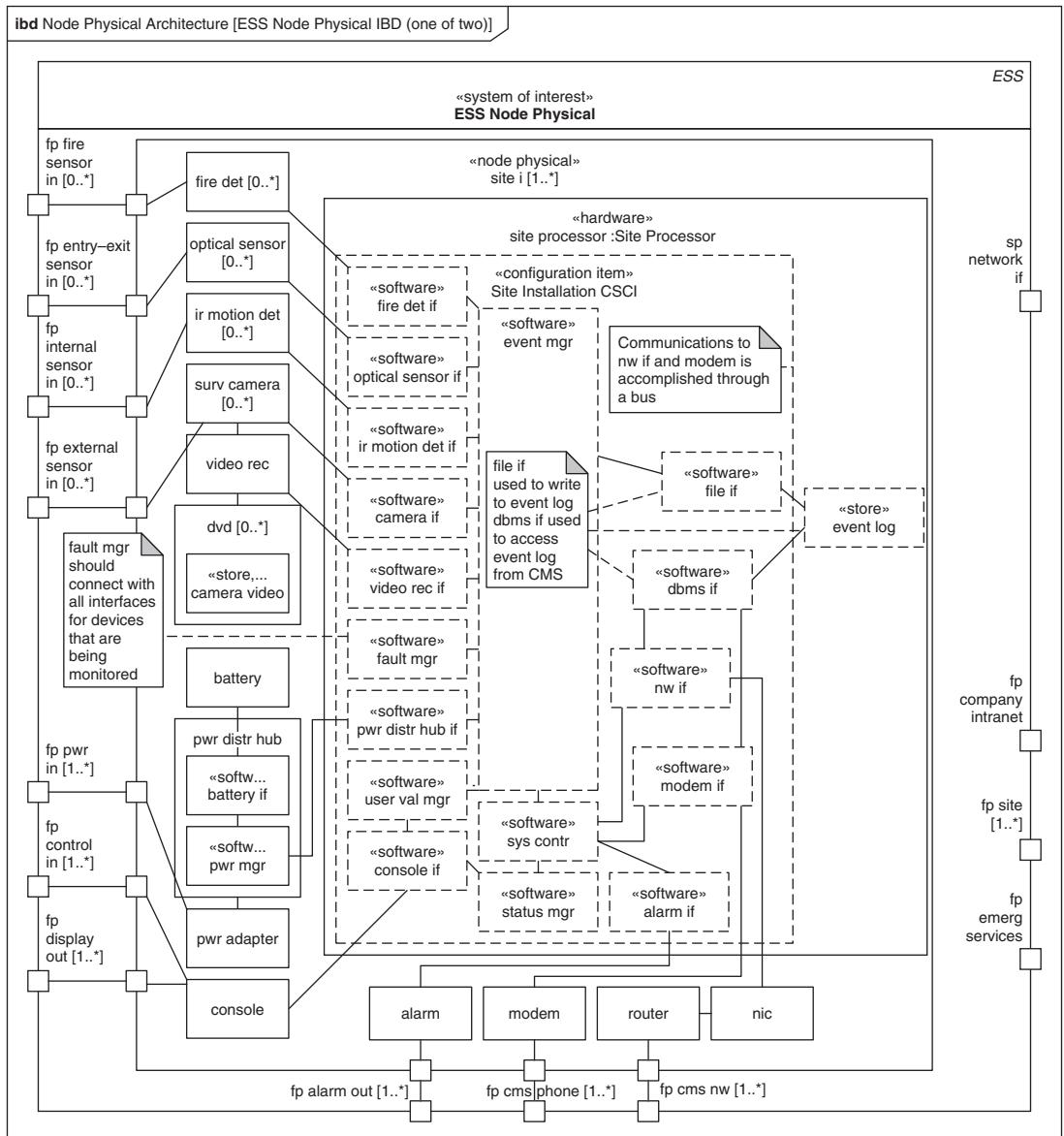
FIGURE 16.37

Monitor Intruder Activity Diagram-np for the *Site Installation Node Physical* supports the behavior of *Site Installation* in the node logical activity diagram in Figure 16.29.

node and across nodes. Similar to the *ESS Logical Internal Block Diagram* and the *ESS Node Logical Internal Block Diagram*, the enclosing block represents a subclass of the ESS block and retains the original ESS external interfaces. The external ports on the *ESS Node Physical* can be further subclassed from the ESS port definitions to specify the physical interface definition such as a USB port on a computer, as described in Section 16.3.2.

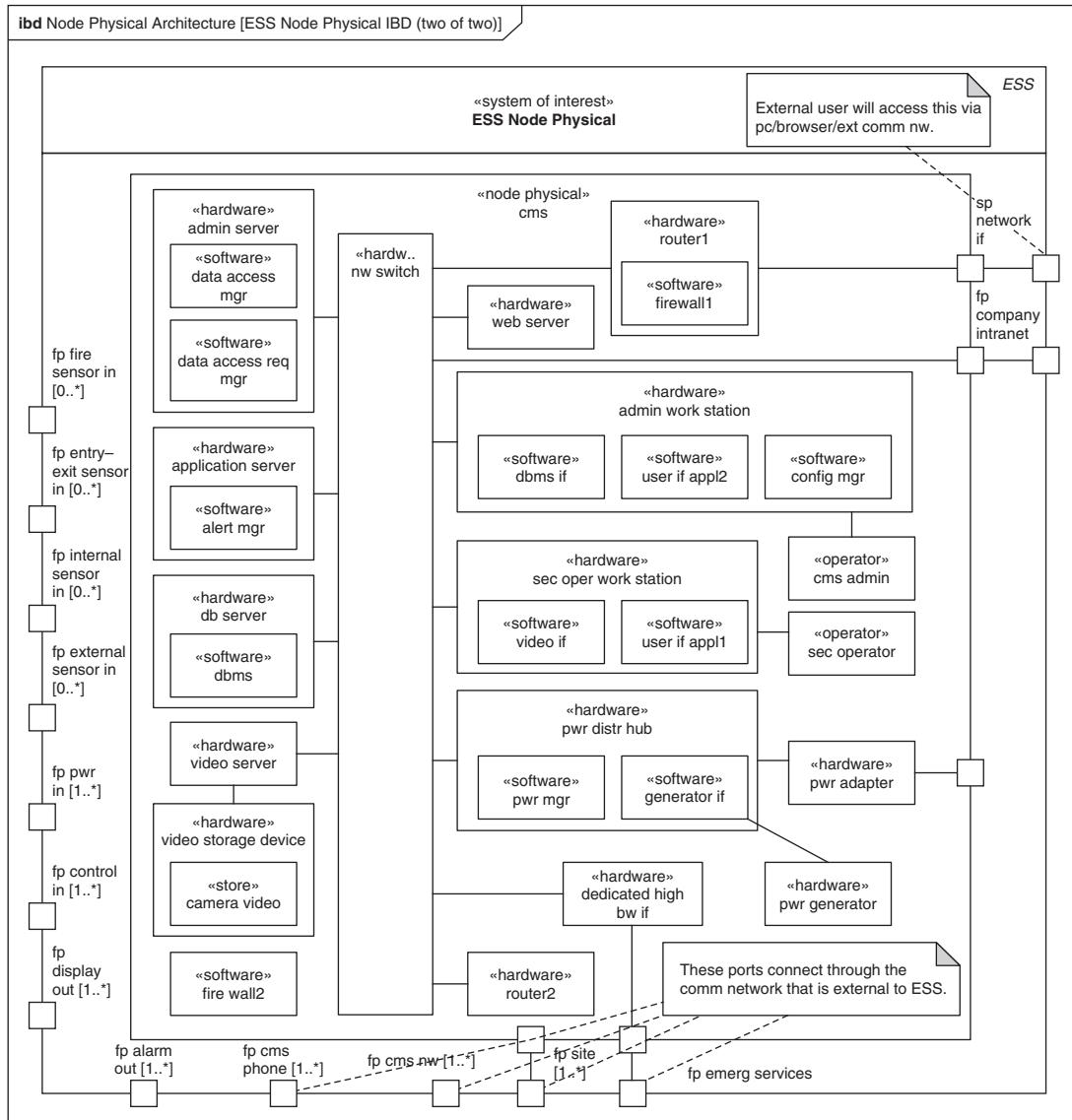
**FIGURE 16.38**

Monitor Intruder Activity Diagram-np for the Central Monitoring Station Node Physical supports the behavior of Central Monitoring Station from the node logical activity diagram in Figure 16.30.

**FIGURE 16.39**

ESS Node Physical Internal Block Diagram for site i :Site Installation.

The internal block diagram defines the interconnection between the parts based on their interaction from all the activity diagrams. The definition of the ports has been deferred pending the detailed interface specifications on the parts. An individual activity diagram, such as the *Monitor Intruder Activity Diagram*, can be viewed as exercising specific interconnection paths through the internal block diagram.

**FIGURE 16.40**

ESS Node Physical Internal Block Diagram for cms:Central Monitoring Station.

The node physical architecture defines the physical components of the system, including hardware, software, persistent data, and other stored items (e.g., fluid, energy) and operational procedures that are performed by operators. The software components and persistent data stores are nested within the hardware component that they are allocated to. The software allocation to hardware

is an abstraction of a UML deployment of a software component to a hardware processor.

The node physical architecture serves as the integrating framework for all components to work together. The following activities support architecting the software, data, and hardware; specialty views of the architecture such as security; and the specification of operational procedures to address their domain-specific concerns.

Define Software Architecture

The software architecture is a view of the overall system architecture that includes the software components and their interrelationships. Software architecting is critical to effectively specify the software components.

The *ESS Software Block Definition Diagram* is shown in Figure 16.41. The *Software Composite* block aggregates all the software and is composed of the *Site Software Composite* and *CMS Software Composite*. The *Software Composite* may be a container rather than a run-time entity that calls other components. This is subject to further refinement through the software architecture process.

The *Site Software Composite* and *CMS Software Composite* in turn aggregate the components that were allocated to the *Site Installation* and the *Central Monitoring Station* software. The hierarchies are shown in the respective software block definition diagrams in Figures 16.42 and 16.43. The initial allocation from the logical-to-physical components may not include the allocation to infrastructure and operating system components that are required to support the application components, but it must be addressed as part of the software architecture design.

These software block definition diagrams, along with the internal block diagram and activity diagrams from the node physical architecture, provide a foundation for defining the software architecture at the system level. The software components may require considerable refinement to address the software-specific concerns and fully specify the software requirements. For example, the software

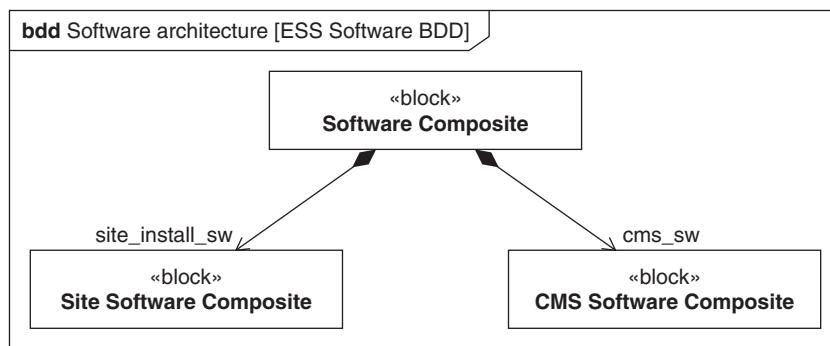
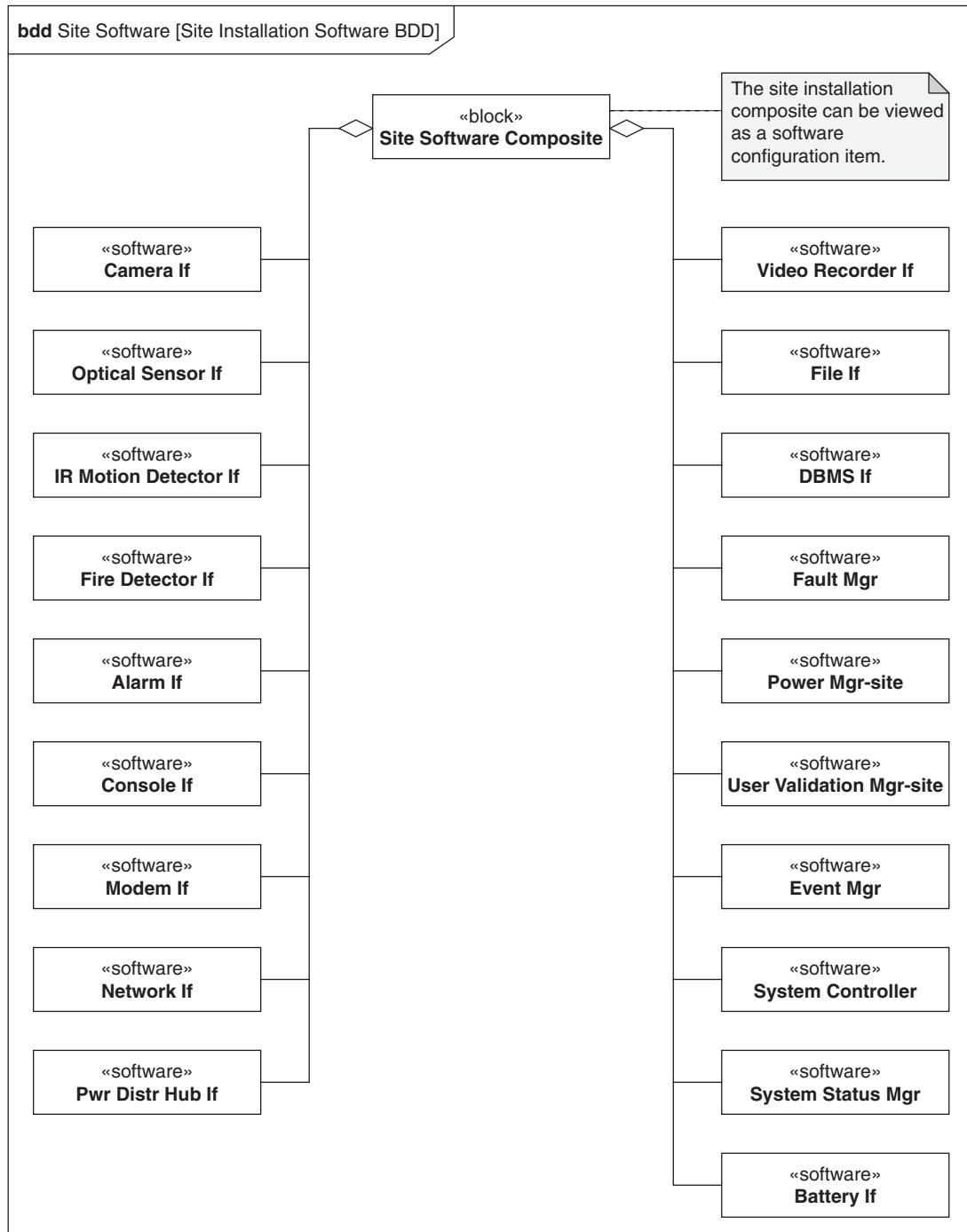
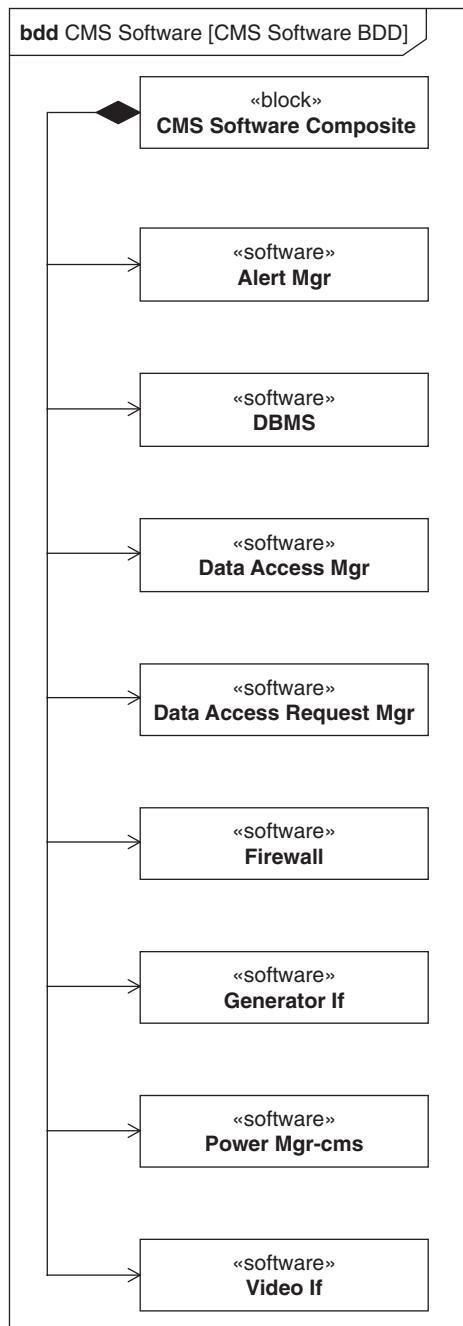


FIGURE 16.41

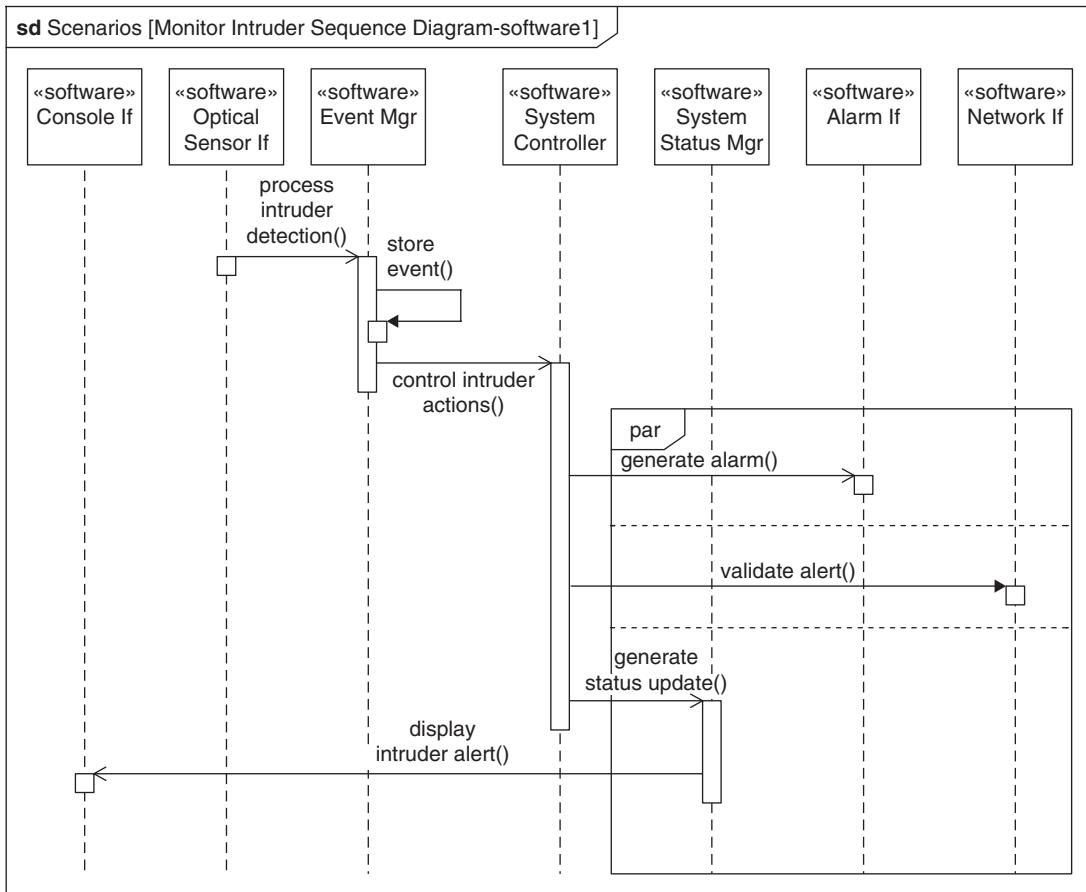
ESS Software Block Definition Diagram aggregates the *Site Installation* software and *Central Monitoring Station* software.

**FIGURE 16.42**

Site Installation Software Block Definition Diagram showing the software components in a typical Site Installation.

**FIGURE 16.43**

CMS Software Block Definition Diagram showing the software components in the *Central Monitoring Station*.

**FIGURE 16.44**

Monitor Intruder Sequence Diagram showing the interaction among the software components.

architecture may include sequence diagrams to refine the interaction between the software components, as shown in the *Monitor Intruder Sequence Diagram* in Figure 16.44. In addition, an internal block diagram, which represents the interconnection of software parts, may be generated for the *Site Software Composite* and *CMS Software Composite*. The interfaces may include standard ports that are typed by required and provided interfaces. Both the sequence diagrams and the internal block diagrams should be consistent with the behavior and structural requirements specified by the node physical architecture activity diagrams and internal block diagrams. The software architecture refinement may be expressed in UML as described later in this section.

Some of the software architecture concerns depend on the application domain. For information systems, the software architecture is often a layered architecture, where each layer includes software components that may depend on a lower layer

for the services it provides. This may include a presentation layer, mission application layer, infrastructure layer, operating system layer, and data layer, as shown in the package diagram in Figure 16.45. The software components from the node physical architecture are further elaborated and partitioned into the different layers.

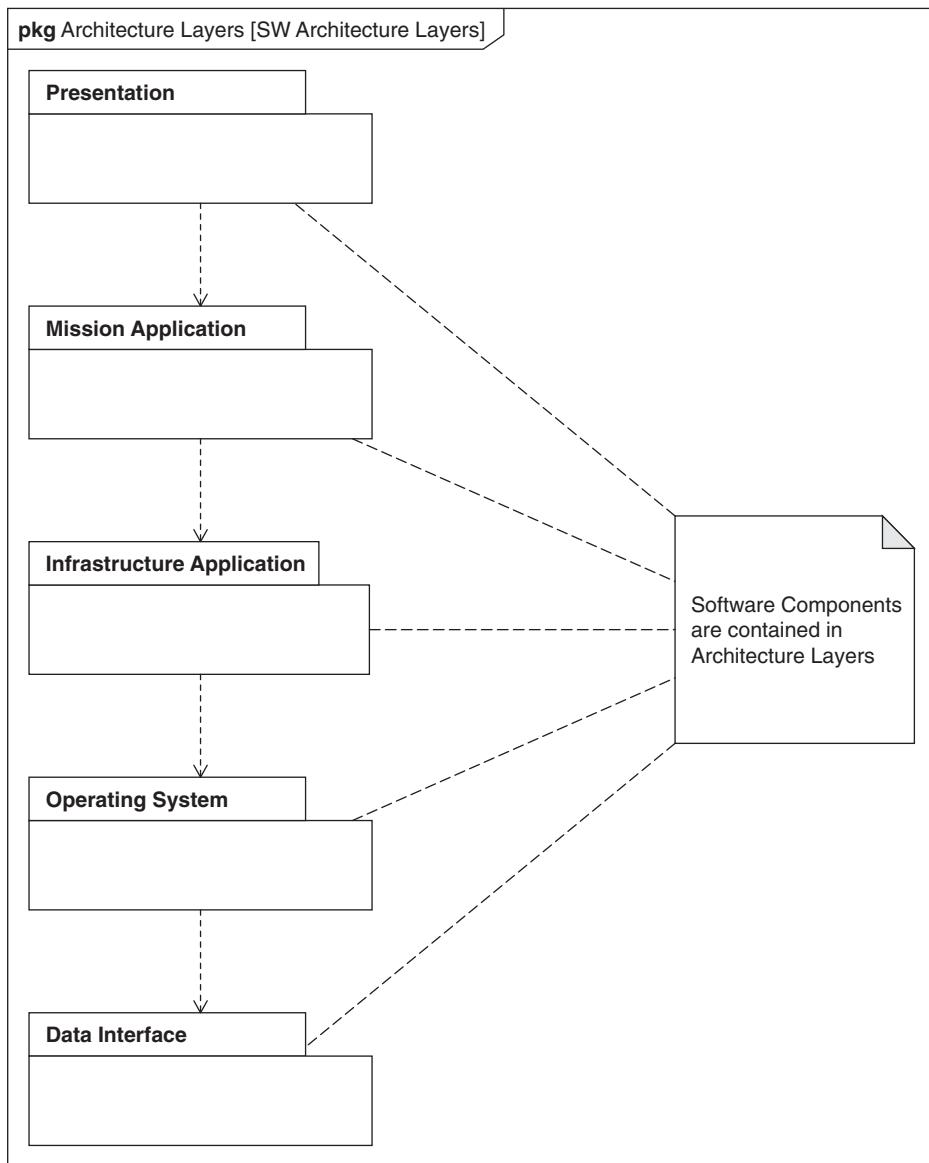


FIGURE 16.45

Layered software architecture on a package diagram showing dependencies between layers.

A reference architecture may be imposed as a design constraint that includes reusable components that provide much of the infrastructure layer, such as messaging, access control services, and database interfaces. For embedded real-time software design, the architecture must also address concerns related to scheduling algorithms and how to address concurrency, prioritization, and contention for bus, memory, and processor resources. It should be noted that the partitioning of software components into packages does not capture all the relationships between the run-time entities that must be addressed to adequately represent the software architecture.

Define Data Architecture

The data architecture is a view of the physical architecture that represents the persistent data, how the data are used, and where the data are stored. The node physical architecture provides the integration framework to ensure that the data architecture is consistent with the overall system design. The persistent data requirements can be derived from the scenario analysis. Persistent data are stored by a logical or physical component and are represented as a property of the component with the «store» stereotype applied. As part of the logical design, the persistent data are encapsulated in the logical component that operates on them. In the physical architecture, the physical data stores, such as a database, are identified.

The data definition types are specified on an *ESS Data Definition Block Definition Diagram* as shown in Figure 16.46. The *Data Composite* aggregates the persistent data definitions, which type the data properties stored by physical components. The *Event Log Data File*, *Camera Video File*, and *User Access*

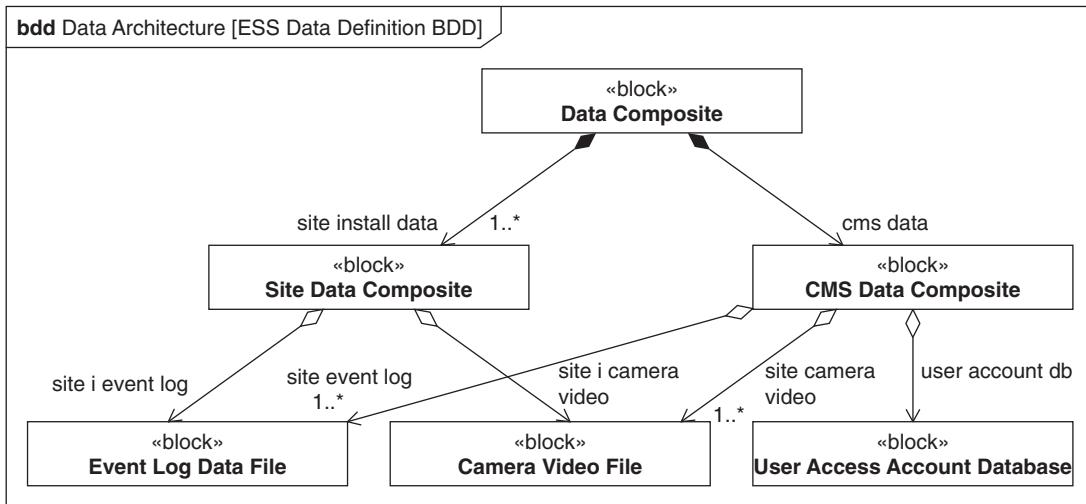


FIGURE 16.46

ESS Data Definition Block Definition Diagram showing persistent data stored by the system at the Site Installation and Central Monitoring Station.

Account Database are examples of types of persistent data that are stored by ESS components. The data definitions can be complex data structures. For example, the *Event Log Data File* includes records of many different types of events, such as intruder power-up events, system activation events, intruder detection events, and others, that were derived from the scenario analysis.

The composition relationship is used as a common pattern within OOSEM to represent different hierarchies. This type of grouping in software is represented by packaging rather than by composition because composition represents run-time composition. The implication from a software perspective is that an instance of a data item of the type *Data Composite* contains one or more instances of the data type *Site Data Composite*, which is not an accurate run-time representation. An alternative modeling approach is to include a package called *Data Composite*, which contains three packages called *Site Data Composite*, *CMS Data Composite*, and *Shared Data Composite* that have the appropriate data types defined within them.

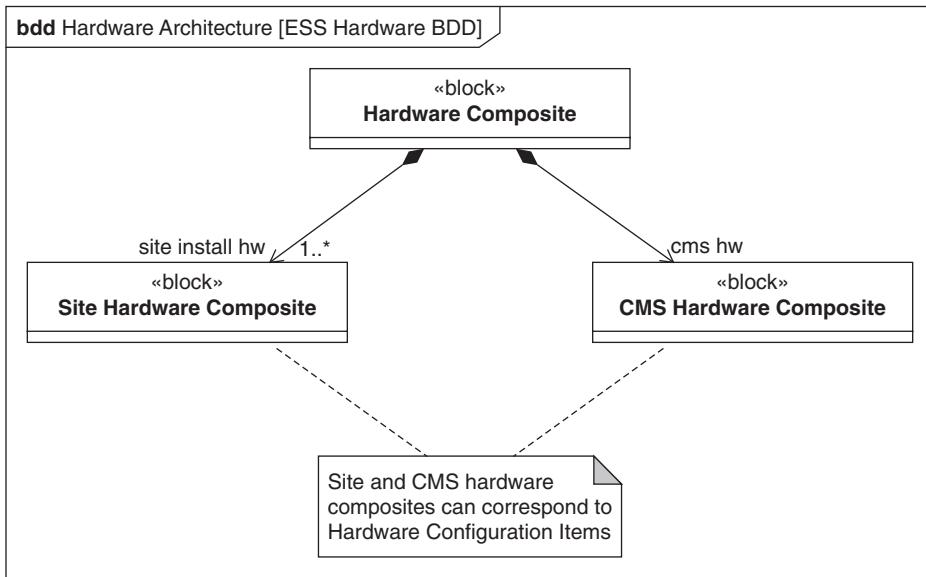
The data architecture may include domain-specific artifacts to refine the data specifications. The data relationships may be specified by an entity relation attribute (ERA) diagram or directly on the data definition block definition diagram using associations among the data definitions. This description can be viewed as the conceptual data model that represents the requirements for implementing the database. The implementation of the conceptual data model is dependent on the technology employed, such as flat file, relational database, and/or an object-oriented database.

There are many other domain-specific aspects of the data architecture that must be considered, such as data normalization, data synchronization, data backup and recovery, and data migration strategies. The selection of the data architecture and the specific technology is determined through trade studies and analyses, as described in Section 16.3.5.

Define Hardware Architecture

The hardware architecture is a view of the physical architecture, which represents the hardware components and their interrelationships. The *ESS Hardware Block Definition Diagram* is shown in Figure 16.47. It aggregates the hardware in a similar way to the *ESS Software Block Definition Diagram* in Figure 16.41.

The *Site Installation Hardware Block Definition Diagram* captures the hardware components in a hierarchical structure, as shown in Figure 16.48. The hardware components are allocated from the logical components in Figure 16.33. The *ESS Node Physical Internal Block Diagram* in Figures 16.39 and 16.40 showed the interconnection of the hardware components. This can be more fully elaborated with more detailed hardware interfaces, including communication protocols and the details of the communications network. The specific selection of the hardware architecture and component technology results from the engineering analysis and trade studies, as described in Section 16.3.5. This includes the performance analysis to support sizing of the hardware components, and reliability, maintainability, and availability analysis to evaluate supportability requirements.

**FIGURE 16.47**

ESS Hardware Block Definition Diagram aggregates the hardware for the Site Installation and Central Monitoring Station.

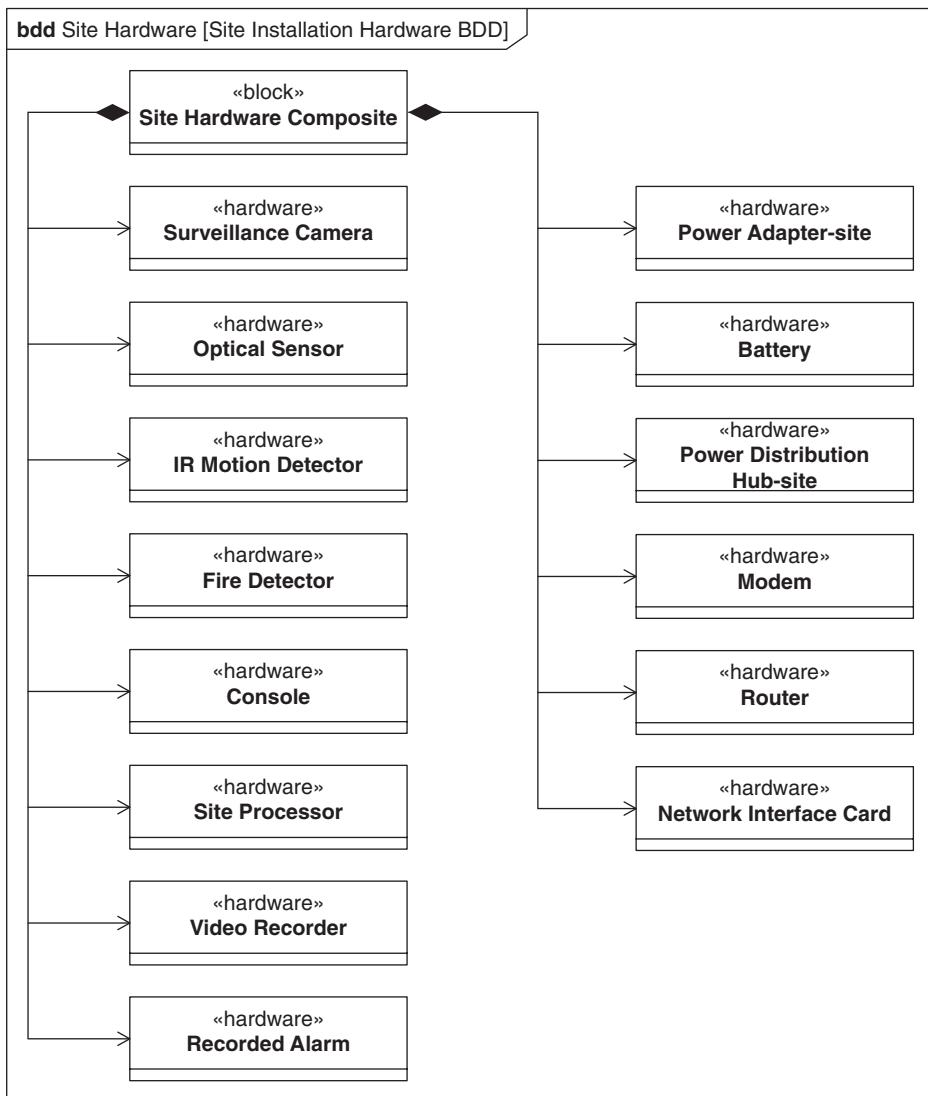
Define Operational Procedures

Operators can be external or internal to the system, depending on how the system boundary is established. For the ESS, the *Occupants* of the property are external to the system, as defined in the *Operational Domain* in Figure 16.11. On the other hand, the *CMS Security Operator* and *CMS Administrator* in Figure 16.49 are considered internal to the ESS. Some logical components are allocated to internal operators to perform selected tasks. Both internal and external operators/users of the system are represented on activity diagrams to describe how they interact with the rest of the system. They are also included in other diagrams like any other external system or system component.

The requirements for what an *Operator* must do to operate the system can be specified in terms of operational procedures, which define the tasks required of each *Operator*. The task analysis, timeline analyses, cognitive analysis, and other supporting analysis are performed to determine levels of task performance that are consistent with the specified skill levels. The ESS *Operational Procedures* are identified in the *ESS Operational Procedures Block Definition Diagram* in Figure 16.50.

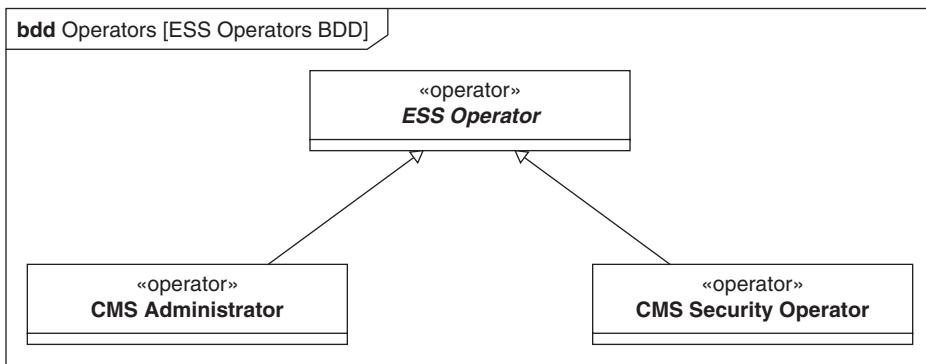
Specify Component Requirements

The node physical architecture, which includes the elaboration of the software architecture, data architecture, hardware architecture, and operational procedures, results in the specification of the components of the system architecture to be implemented in software, data, hardware, and operational procedures, respectively.

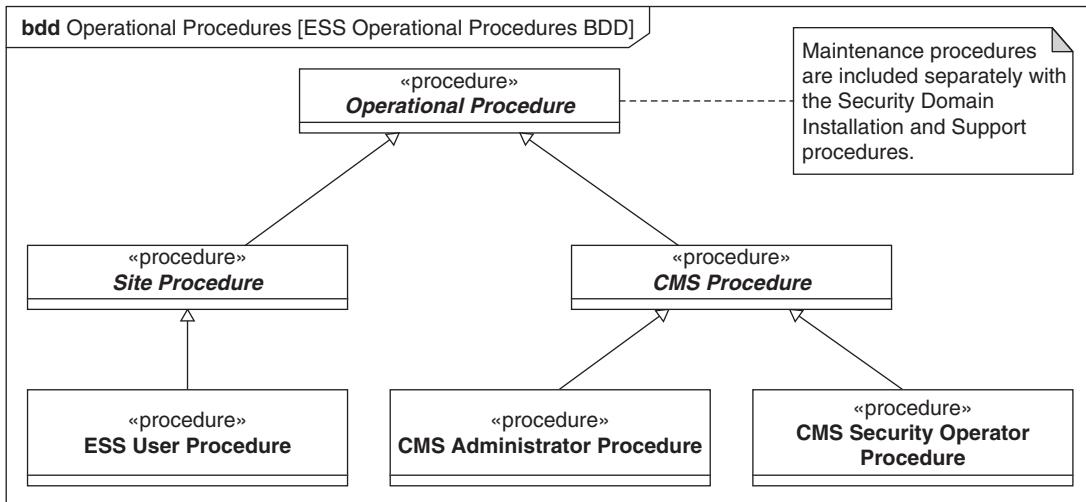
**FIGURE 16.48**

Site Installation Hardware Block Definition Diagram showing the hardware components within the *Site Installation* that were allocated from the logical components in Figure 16.33.

The component specifications are a primary output from systems specification and design process. The component specifications are typically captured as blocks with the appropriate black-box specification features, as described in Section 16.3.2. Examples of a software component specification and hardware component specification model are shown in Figure 16.51.

**FIGURE 16.49**

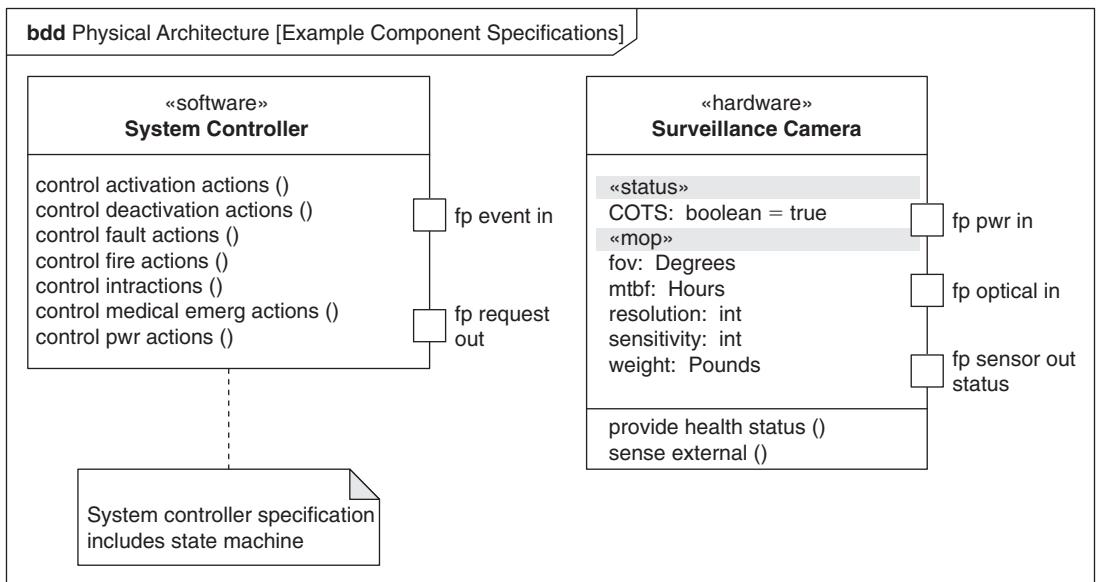
ESS Operators Block Definition Diagram showing the internal ESS Operators.

**FIGURE 16.50**

ESS Operational Procedures Block Definition Diagram.

The software component in the figure is the *System Controller* that is part of the *Site Software Composite*, with the OOSEM «software» stereotype applied. The flow ports show the *event in* port and the *request out* port. Standard ports with required and provided interfaces could have been used instead of flow ports. The controller operations are also specified as features of the block. The state machine for the controller derived from the logical state machine in Figure 16.26 defines the events that trigger the operations. The state machine provides a specification approximately equivalent to a series of statements as follows: {if (*input_i* = *X_j* and current state = *S_m*), then (next state = *S_n*) and do/activity *A_m*}.

The *develop software* process referred to in Figure 16.1 is used to perform software requirements analysis to derive more detailed requirements, perform

**FIGURE 16.51**

Example of hardware and software component specifications.

software design, and implement and test software components. The Unified Modeling Language [28] is used to support this process. Classes can be subclasses of the software component specifications or allocated from the software component specifications and represented on class diagrams. The UML composite structure diagram is used to refine the internal block diagram from the node physical architecture in Figures 16.39 and 16.40 to reflect the interconnection and interfaces between the software components. The software design realizes the software component interfaces, operations, and state machine behavior by introducing more detailed structures and behaviors. The software sequence diagrams, such as the one shown in Figure 16.44, are further elaborated to show the interaction between the lower-level software design components. The UML component diagram and deployment diagram can also be used for software design to show more explicitly how the software is deployed beyond the abstract allocation of software to hardware in Figures 16.39 and 16.40.

The hardware component specification in Figure 16.51 is the *Surveillance Camera* that is part of the *Site Hardware Composite* with the OOSEM «hardware» stereotype applied. The black-box component specification includes functional requirements derived from the scenario analysis, and performance properties with stereotype «mop» whose values are determined through engineering analysis and trade studies, as described in Section 16.3.5. The flow ports are used to specify the interfaces. The development status indicates this is a COTS component.

If software components are allocated to the hardware, they can be represented in an allocation compartment. In addition, a property can also be added

to the hardware component that references a geometric drawing of the component, or customized port types can be defined to represent mechanical interfaces. Additional specification features can be added to address the needs.

Defining Other Architecture Views

There may be other architectural views of the system that address specific stakeholder perspectives, such as a security architecture. The security architecture can be represented as a subset of the node physical architecture, which includes hardware, software, data, and procedures that address security requirements. In this sense, the security architecture is a subset of the overall system architecture.

A viewpoint represents a stakeholder perspective, such as a security architect viewpoint. The viewpoint is used to specify a subset of the model that is of interest to the stakeholder. As described in Chapter 5, a viewpoint includes rules that specify how a particular view is constructed to reflect the stakeholder perspective. The rules can be defined in terms of criteria for querying the model. A view provides a filtered portion of the model that conforms to the viewpoint by returning the model elements in response to the model query.

If the query criteria define all components needed to satisfy the system security requirements, such as the confidentiality, integrity, and availability requirements, the security architecture view includes the model elements that satisfy these requirements. Refer to Section 16.3.6 for an example of how viewpoints and views can satisfy requirements.

16.3.5 Optimize and Evaluate Alternatives

The *Optimize and Evaluate Alternatives* activity is shown in Figure 16.52. This activity is invoked throughout all other OOSEM activities to support engineering analysis and trade studies. This activity includes identifying the analysis that is needed, defining the analysis context, capturing the constraints in a parametric diagram, and performing the engineering analysis.

Chapter 7 describes how to model constraints with parametrics. Chapter 17 includes a discussion of engineering analysis and simulation models and how they fit into the overall modeling environment. SysML enables critical system characteristics in the model to be captured so that they can be analyzed, and it provides a mechanism to integrate the system design models with the multitude of engineering analysis models, such as performance, reliability, and mass properties analysis.

Identify Analyses to Be Performed

The analyses to be performed should support specific analysis objectives, which may include the following:

- Characterize or predict some aspect of the system, such as its performance, reliability, mass properties, or cost
- Optimize the design through sensitivity analysis
- Evaluate and select a preferred solution among alternative design approaches
- Verify a design using analysis
- Support other analyses, such as a risk analysis and mitigation planning

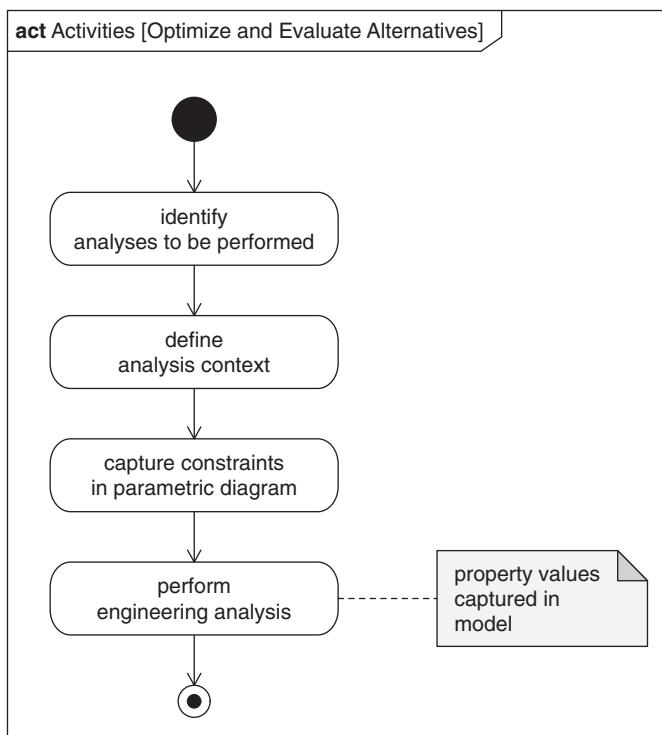


FIGURE 16.52

Optimize and Evaluate Alternatives activity to support trade studies and analysis.

Different types and fidelity of engineering analyses are identified throughout the design process to meet the range of analysis objectives.

Define the Analysis Context

The analysis context is a block definition diagram that defines the constraint blocks that are used in the analysis. The constraint block specifies an equation, such as $\{F = m * a\}$, along with its parameters, which in this case would be "F," "m," and "a."

Figure 16.53 shows the *ESS Analysis Context*. The *Analysis Context* block is composed of constraint blocks that are used to analyze the system. In this example, the constraint block defines an objective function that is used as a basis for evaluating the overall value of the system. The objective function specifies an equation that relates system cost effectiveness to the measures of effectiveness for *availability*, *emergency response time*, *probability of intruder detection*, and *probability of false alarm*.

The *ESS Analysis Context* also includes a reference to the *Operational Domain* block, which is the top block in the system hierarchy. By referencing this block, the analysis equations can be related to any of the properties of the

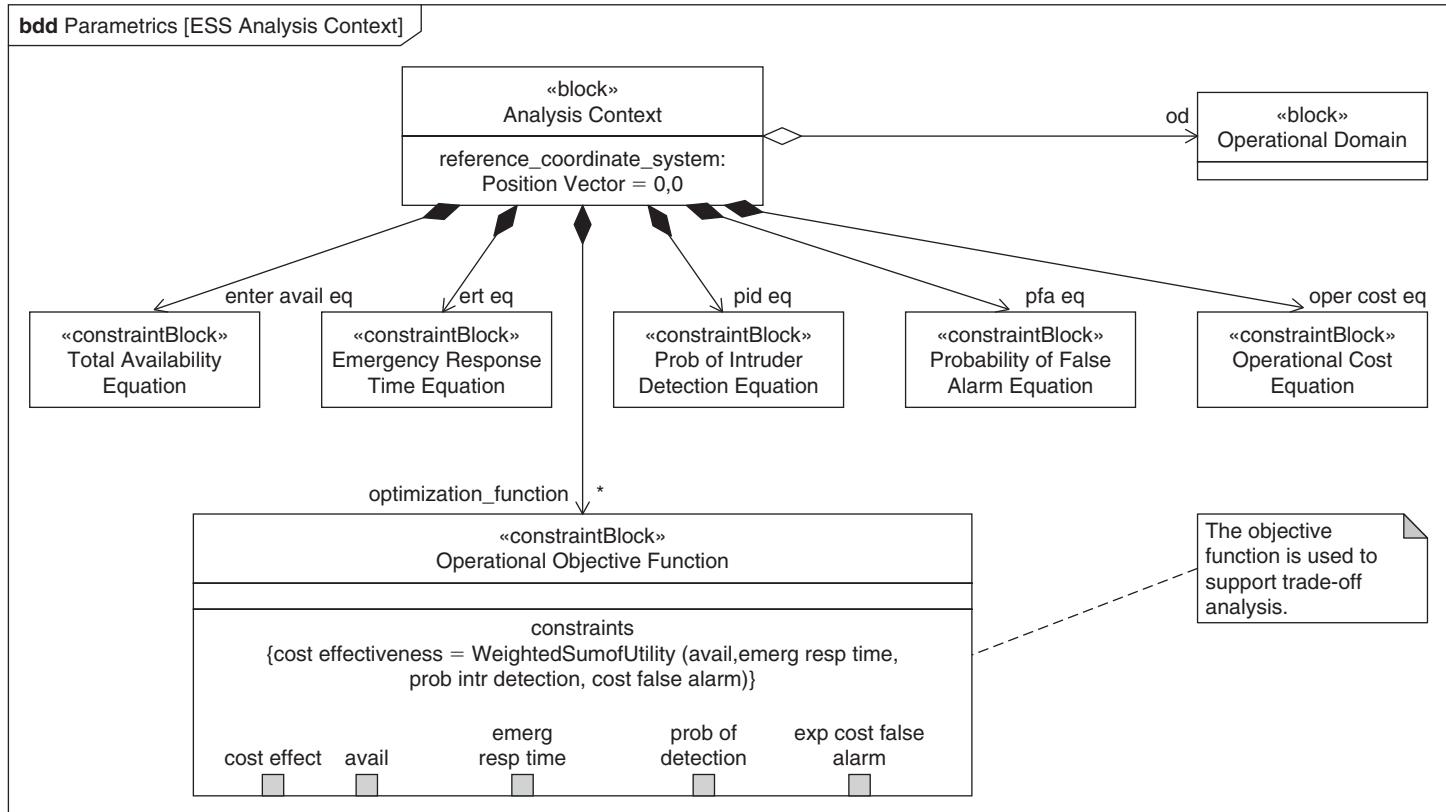


FIGURE 16.53

ESS Analysis Context defines the objective function as the top-level constraint block and other analysis models for each moe.

operational system being analyzed. The context also references the operational domain to support trade-off analysis of ESS alternatives that are part of the operational domain.

Capture Constraints in Parametric Diagram

The parametric diagram enables the integration between the design and analysis models. It does this by binding the parameters of the analysis equations that are defined in the *Analysis Context* to the properties of the system being analyzed.

The top-level parametric diagram for the ESS is discussed in Section 16.3.1 and shown in Figure 16.10. The parametric diagram uses the equations defined in the *ESS Analysis Context* in Figure 16.53. The parametric diagram binds the parameters of the objective function to the moes in the *Security Enterprise* shown in Figure 16.11.

The top-level parametric diagram is used to identify other engineering analyses to be performed. As the system design evolves, additional engineering analysis is needed to evaluate the system design against top-level moes. Figure 16.54 shows a parametric diagram for the availability model that binds the parameters of the availability equation to specific properties of the ESS. The availability property in the figure is also shown in the top-level parametric diagram in Figure 16.10 and represents a moe. The parametric diagrams provide the mechanism to maintain

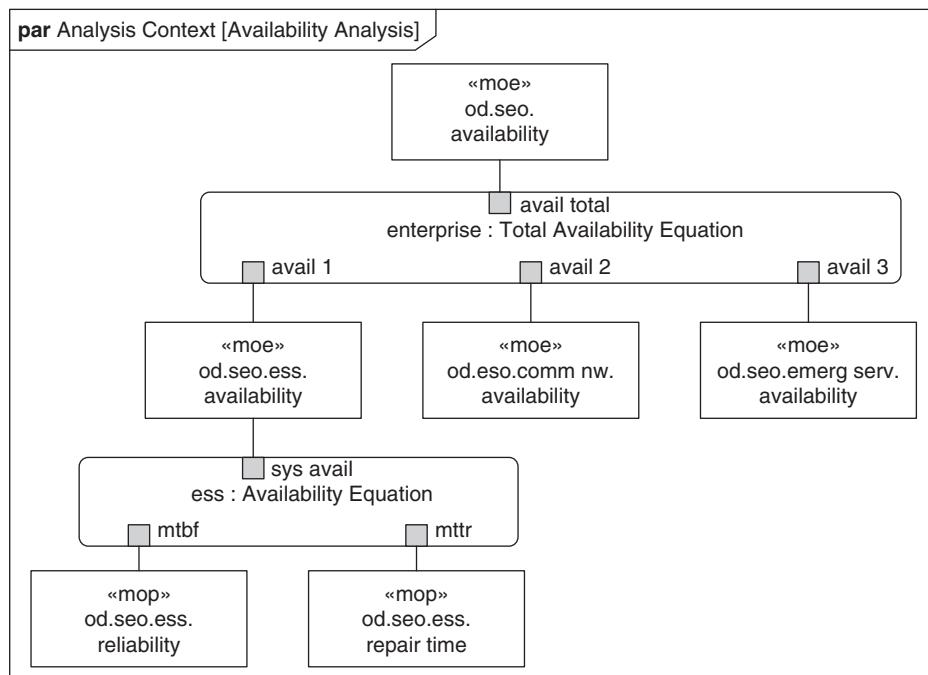


FIGURE 16.54

Availability Analysis model captured in a parametric diagram.

explicit relationships between the moes and their flow down to critical system, element, and component properties.

Parametrics can also be used to constrain inputs, outputs, and the input/output relationship associated with the behavior of a system or component. In the *Intruder Emergency Response Scenario* in Figure 16.14, the *Monitor intruder* action includes pre- and postconditions on the inputs and outputs. A corresponding constraint block can be defined to specify the mathematical relation between the probability of detection of the signal output and the signal-to-noise ratio of the signal input. The constraint block can then be used on a parametric diagram to bind to the system's specific properties to analyze the detection performance.

As described in Section 16.3.2, the state of the system is also shown as a property of the *ESS* block that is stereotyped as «composite state». The value of this property represents the state of the system at any point in time and is determined by the *ESS* state machine behavior. This property can be used in parametrics by binding a state-dependent constraint to the composite state property. For a bouncing ball example, the constraints that apply to the forces on the ball depend on the state of the ball in terms of whether it is in contact with the ground or not. The state-dependent constraint can be conditioned on the state of the ball. For this example, the state-dependent constraint would specify whether the state of the ball is “contact with ground,” then one constraint applies; and if the state of the ball is “not in contact with the ground,” then another constraint would apply.

Perform Engineering Analysis

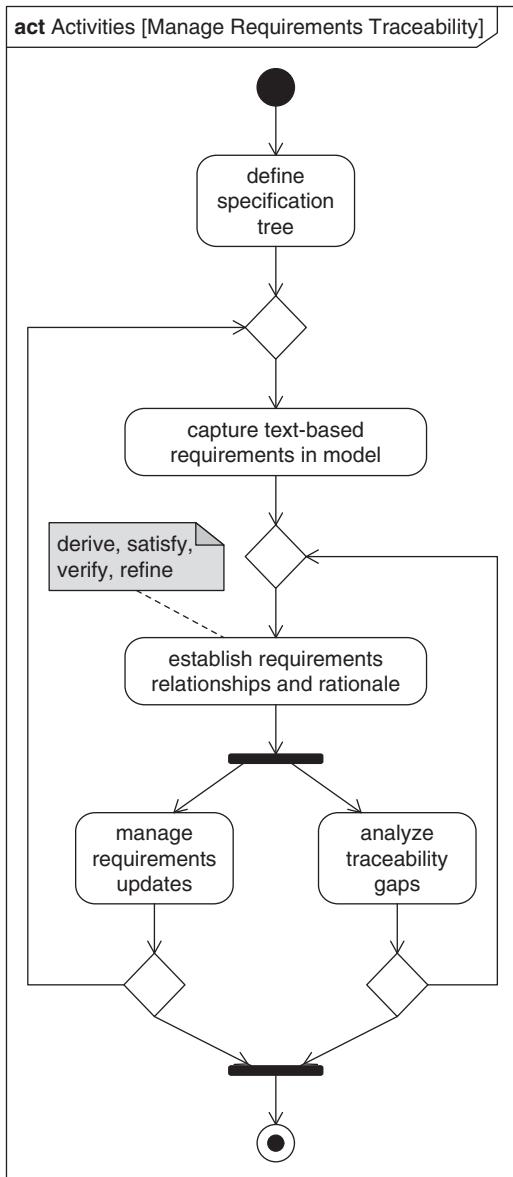
A computational capability is required to execute the equations in the parametric diagram. This can be done manually or with the aid of engineering analysis tools, as described in Chapter 17. The analysis results determine the specific values or range of values of the system properties that satisfy the constraints. The values can be incorporated back into the system design model. As an example, the availability analysis results can show the extent to which the system satisfies its availability requirement. The timeline in Figure 16.16 is another example of analysis results.

16.3.6 Manage Requirements Traceability

The *Manage Requirements Traceability* activity is shown in Figure 16.55. This activity is invoked throughout all other OOSEM activities to establish requirements traceability between the stakeholder requirements and the system specification and design model. This includes defining the specification tree; capturing the text-based requirements in the model; establishing relationships between the text-based requirements and the model elements using derive, satisfy, verify, and refine relationships; and generating the traceability reports. The language concepts for requirements modeling are described in Chapter 12.

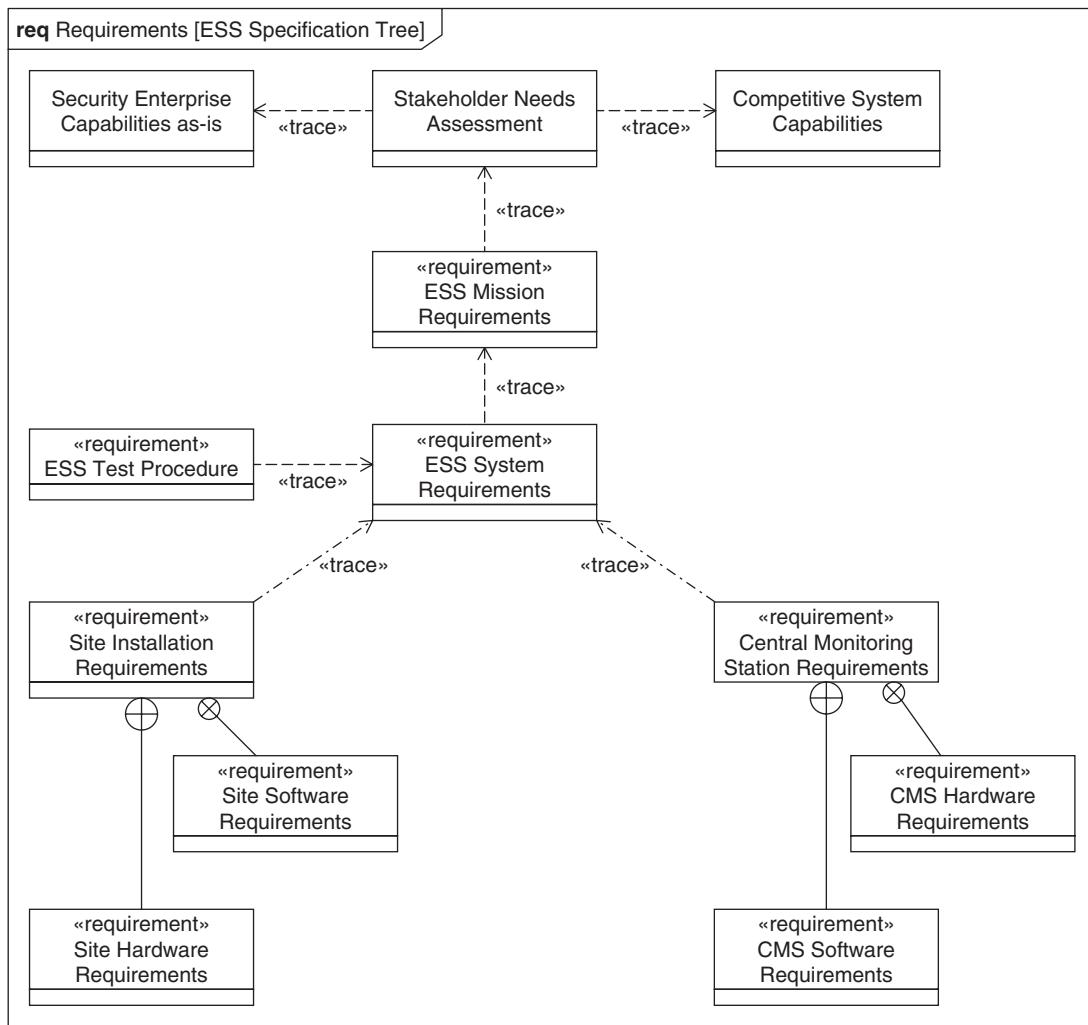
Define Specification Tree

The *ESS Specification Tree* is shown in Figure 16.56. The specification tree shows the specifications at each level of the system hierarchy. A requirement is created

**FIGURE 16.55**

Manage Requirements Traceability activity, intended to maintain traceability between stakeholder requirements and the system specification and design model.

to represent a container for all the requirements contained in each text specification. The specification tree includes the *ESS Mission Requirements*, *ESS System Requirements*, *Site Installation Requirements*, *Central Monitoring Station Requirements*, and *Site and CMS Hardware and Software Requirements*.

**FIGURE 16.56**

ESS Specification Tree on a requirements diagram showing the hierarchy of specifications.

The trace relationship shows the traceability between the specifications at each level. The specification tree also shows traceability from the *ESS Mission Requirements* to a *Stakeholder Needs Assessment* document. A document icon is shown in the upper right corner rather than the stereotype symbol «document», which is an example of a user-defined icon.

The trace relationship is used for coarse-grained traceability that does not include the fine-grained traceability between individual design elements and individual requirements. The fine-grained traceability uses other requirements relationships, as described later in this section.

Capture Text-Based Requirements in Model

The stakeholder requirements are often captured in text specifications external to the modeling environment. The text-based requirements are captured in the model by creating a SysML requirement for each text requirement. Many of the SysML modeling tools provide a mechanism to import text requirements directly into the modeling tool and to maintain synchronization between the source requirements and the requirements in the SysML modeling tool.

The *Requirements* package, which was briefly discussed in Section 16.3.1 and shown in Figure 16.3, contains the requirements. A nested package is created for each specification in the *ESS Specification Tree*, which contains the requirements in the specification.

As an example, the *ESS System Specification* is shown in the requirements diagram in Figure 16.57. The top-level requirement is the *ESS System Requirements*. As mentioned before, this requirement serves as a container for the other requirements in the specification. The containment hierarchy of requirements in each individual specification generally corresponds to the organization of the text-based specification. Each requirement has a name, an id, and text, and may also include additional requirement properties, such as criticality, uncertainty, probability of change, and verification method.

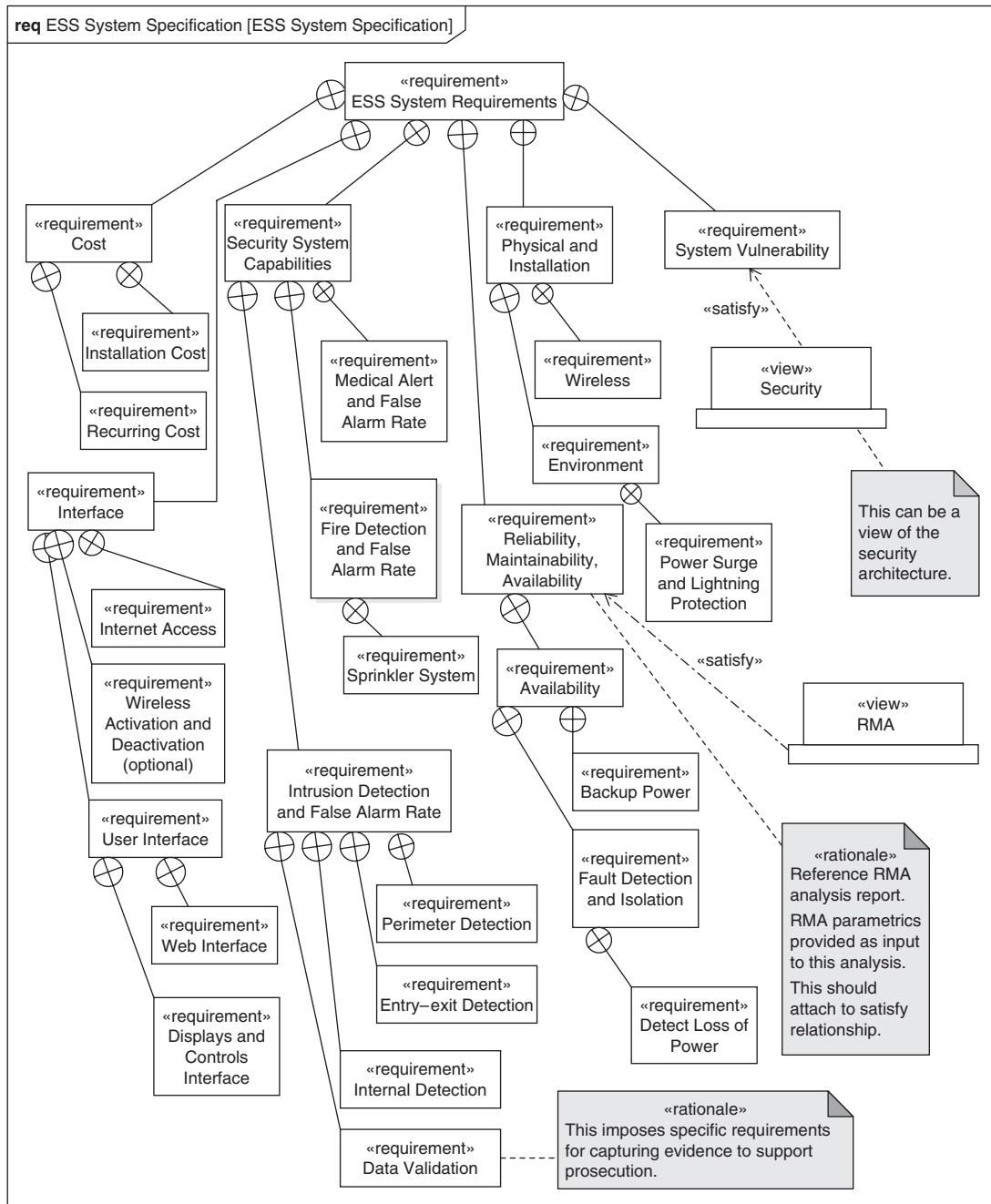
Establish Requirements Relationships and Rationale

Requirements traceability is maintained by establishing relationships between the text-based requirements in the model, and other model elements that correspond to other requirements, design elements, and test cases. The rationale for the relationship can also be captured in the model.

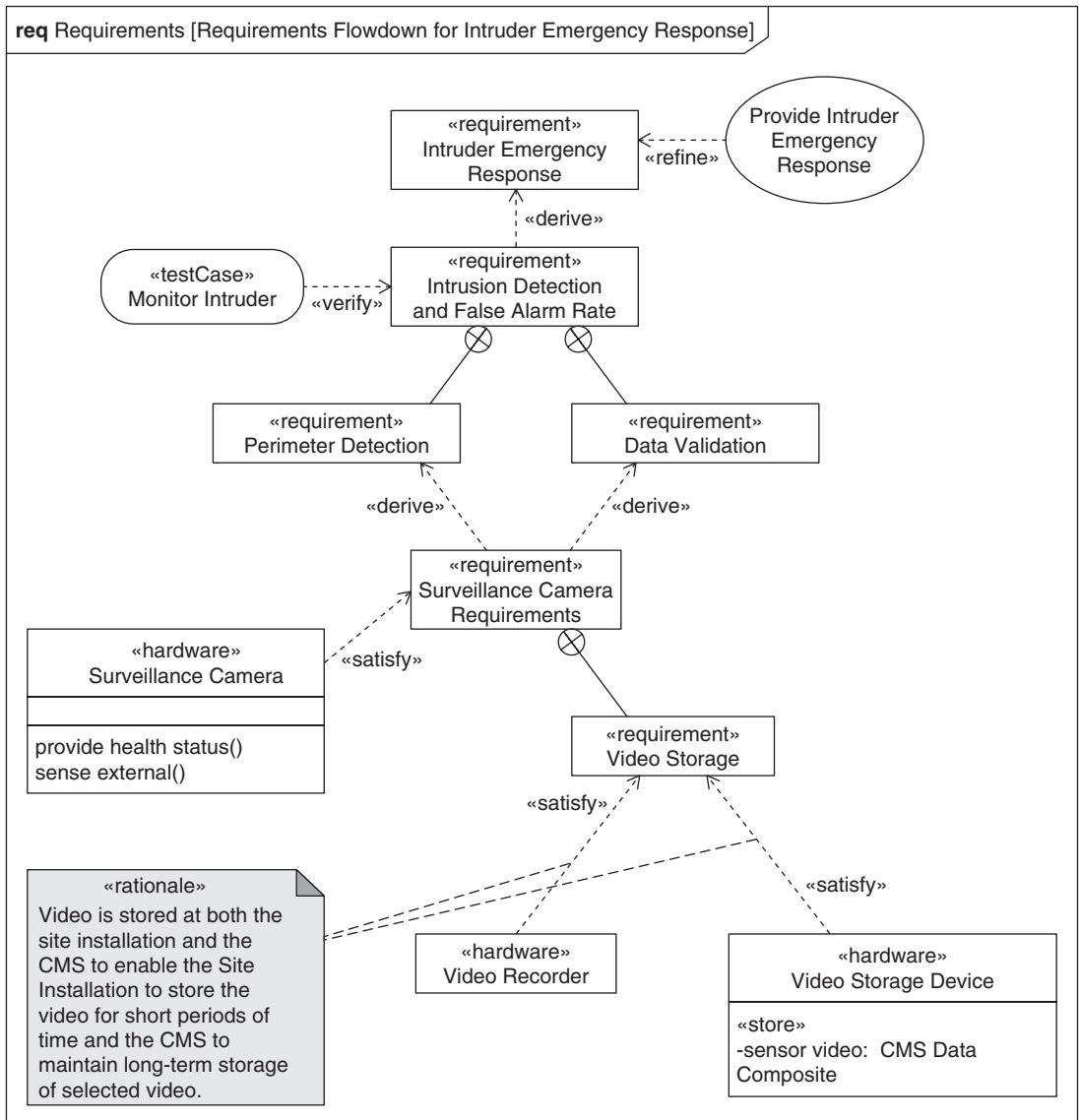
An example of requirements traceability can be seen in the requirements diagram in Figure 16.58, which shows traceability from the mission requirement for *Intruder Emergency Response* to the *Surveillance Camera* component specification. The *Provide Emergency Response* use case refines the requirement by adding a use case description (not shown). The steps in the use case description can also be captured as text requirements that can be related to other requirements, design elements, and test cases as part of the traceability.

The ESS system requirement for *Intruder Detection and False Alarm Rate* is derived from the mission-level requirements. The requirement for *Perimeter Detection* is contained by the *Intruder Detection and False Alarm Rate* requirement. The *Surveillance Camera Requirements* are derived from the *Perimeter Detection* and *Data Validation* requirement. The *Surveillance Camera* is asserted to satisfy the *Surveillance Camera Requirements*. The *Monitor Intruder* test case verifies that the *Intruder Detection and False Alarm Rate* requirement is satisfied. The rationale for storing video at both the *Site Installation* and *Central Monitoring Station* to satisfy the *Video Storage* requirement is shown using the «rationale» stereotype.

The level of granularity at which the traceability is maintained is determined as part of the process tailoring. For example, it may be sufficient to assert that a particular component satisfies a requirement, such as the *Surveillance Camera* in the

**FIGURE 16.57**

ESS System Specification showing the requirements contained in the system specification on a requirements diagram.

**FIGURE 16.58**

Requirement diagram showing traceability from the *Intruder Emergency Response* mission requirement to component-level requirements and design.

earlier example. Alternatively, it may be necessary to show that a particular feature of a component, such as one of its operations, satisfies a particular requirement. The finer granularity adds precision to the traceability, which may assist in change impact assessment, for example; but it is done at the price of increased effort to establish and maintain the traceability relationships.

Analyze Traceability Gaps

Traceability reports are generated and used to analyze traceability gaps and assess how the system design satisfies the system requirements. Metrics can also be used to determine requirements coverage in terms of both satisfy and verify relationships. The results from this analysis are used to drive updates to the system design and verification and to update the traceability.

The *Viewpoints* package was introduced in Section 16.3.1 and shown in Figure 16.3. Viewpoints and their corresponding views can aid in requirements traceability analysis by providing a means to query the model for the model elements that satisfy a particular set of requirements. This was discussed at the end of Section 16.3.4 as it relates to defining architecture views. A package diagram of viewpoints and conforming views is shown in Figure 16.59. The viewpoints represent different stakeholder perspectives, which include *Emergency Services*, the *Company Owner*, the *Customer*, and selected members of the development team, including the *RMA Analyst* and the *Security Architect*. If selected requirements are used as the basis for defining the view query criteria, each view is essentially a report of the model elements that satisfy the selected set of requirements.

Managing Requirements Updates

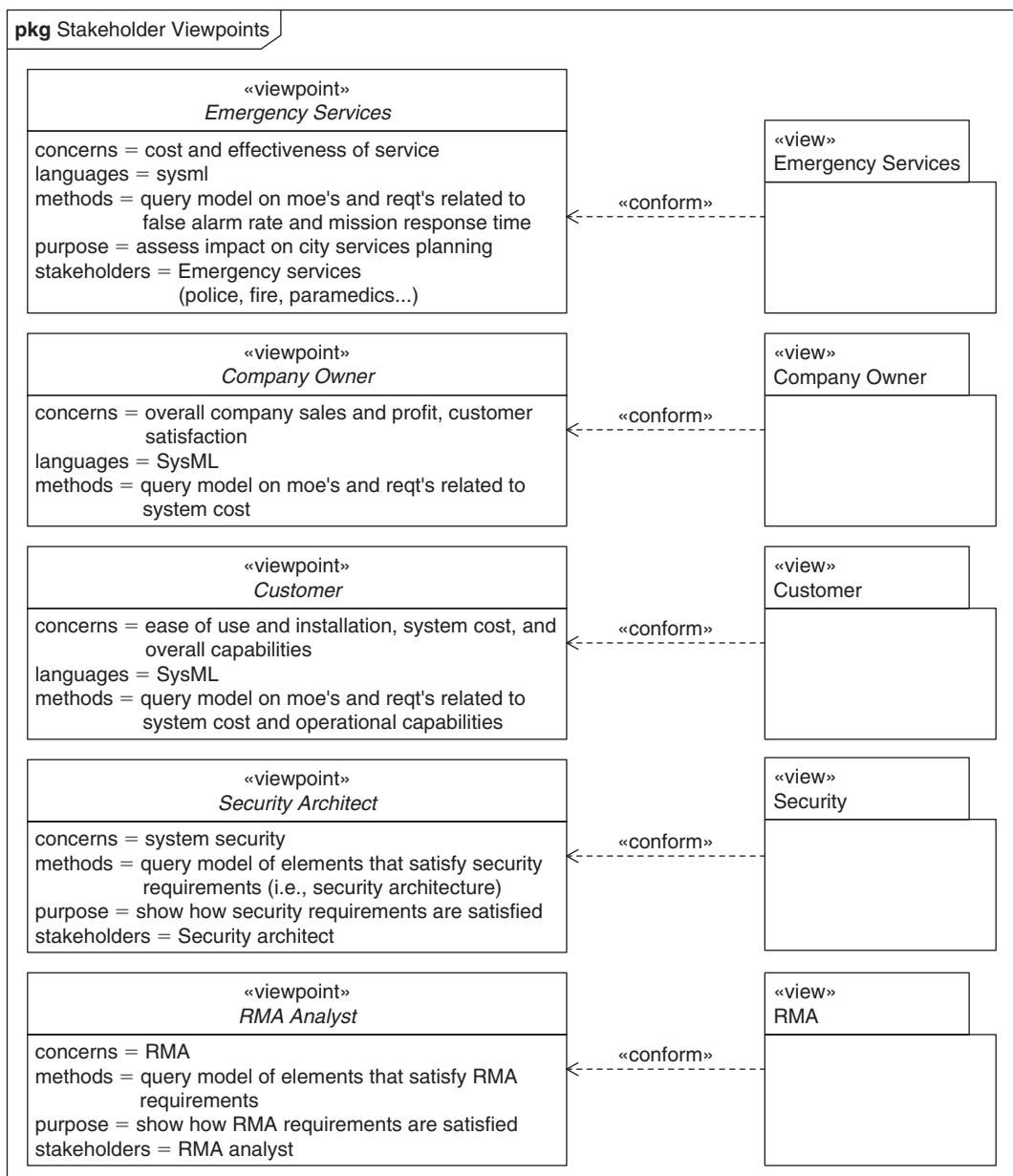
The requirements management activity may result in proposed updates to existing requirements and the generation of new requirements. The model helps to uncover ambiguous, inconsistent, and incomplete requirements that can then be refined by proposing changes to requirements and managing the change through the project change management process.

On larger projects, a requirements management tool is generally used in conjunction with the systems modeling tool. Integration between the two tools is important to ensure that the requirements and their relationships are synchronized between both tools. The change process must determine how changes to requirements are handled. One approach is to make changes to requirements text in the requirements management tool, and to establish the relationships to the model elements and text requirements in the modeling tool. Chapter 17 includes additional discussion on integrating the system modeling tools with the requirements management tool.

16.3.7 Integrate and Verify System

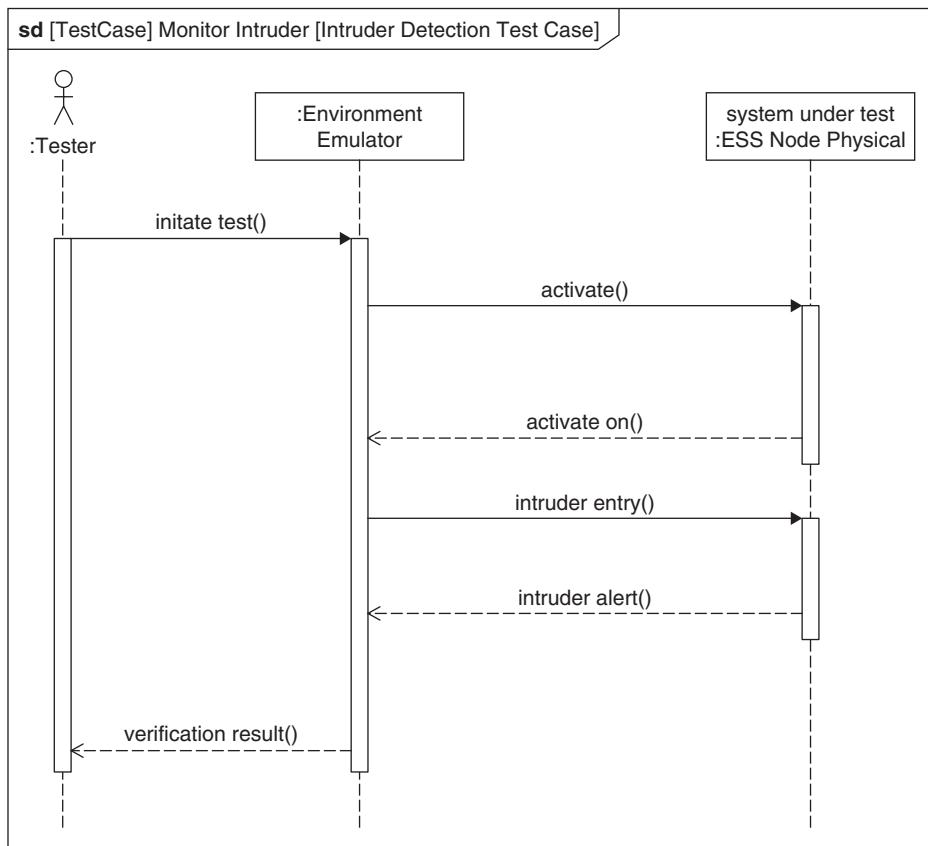
The *Integrate and Verify System* process is part of the system development process described in Section 16.1.2. The goal of this process is to verify that the system satisfies its requirements. System, element, and component verification is typically accomplished by a combination of inspection, analysis, demonstration, and testing. The process includes developing verification plans and procedures, conducting verifications per the procedures, analyzing verification results, and generating verification reports.

OOSEM supports this process in several ways. The system-level use cases, scenarios, and associated requirements are used as a basis for developing test cases

**FIGURE 16.59**

ESS Stakeholder Viewpoints can be used to augment the requirements traceability.

and associated verification procedures. The *Monitor Intruder* test case verifies the *Intrusion Detection and False Alarm Rate* requirement shown in Figure 16.60. The test case is represented as a stereotype of a sequence diagram. The *ESS NodePhysical* is the *system under test*. In this example, an *Emulator* represents the ESS external environment that generates the stimulus for a specific thread

**FIGURE 16.60**

Monitor Intruder test case.

through the system and monitors the response to the stimulus. A tester initiates the test.

The ESS components interact in response to the test stimulus. The results can then be recorded to determine whether the system provides the desired response. An executable system model or the operational hardware and software can be used to generate the response, or some combination of the two. The test case includes a specification of the stimulus and the expected response, which provides the basis for comparison with the verification result to determine whether the system passes or fails. The parametric diagrams used to specify the input/output relations are used to specify the stimulus and expected response values for the test case. The requirements management database is updated to reflect the verification results.

16.3.8 Develop Enabling Systems

To develop a complete capability that supports the entire system life cycle, several enabling systems may need to be developed and/or modified. The enabling

systems include the manufacturing system to produce the system, support systems such as support equipment to maintain the system, and verification systems to test the system. The practice of concurrent engineering demands that these life-cycle considerations be addressed early and in a coordinated fashion. As a result, the enabling systems are developed concurrently with the operational system so that specific concerns, which may impact other parts of the life cycle, are addressed early in the development process.

Figure 16.61 shows the processes for concurrent development of the ESS operational system with the ESS enabling systems for verification and installation. The OOSEM method was applied to the development of the operational system in this example. However, the method and associated artifacts can be tailored and applied to specify and design the enabling systems as well. For very complex enabling systems, the entire method may be applied. For simpler systems, only selected aspects of the method may apply.

As an example, the verification system may be quite complex, such as when precision measurement equipment is required to verify the system. The requirements on the measurement equipment may be more stringent than the requirements on the operational system under test. For this case, a rigorous application of OOSEM may be required. In Figure 16.60, a part of the verification system included the *Emulator* for stimulating the system under testing and monitoring the results. The emulation development may require an application of OOSEM.

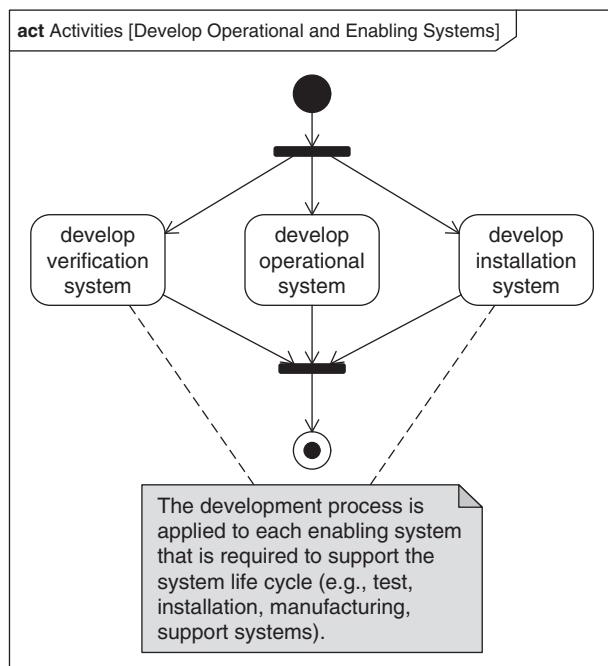
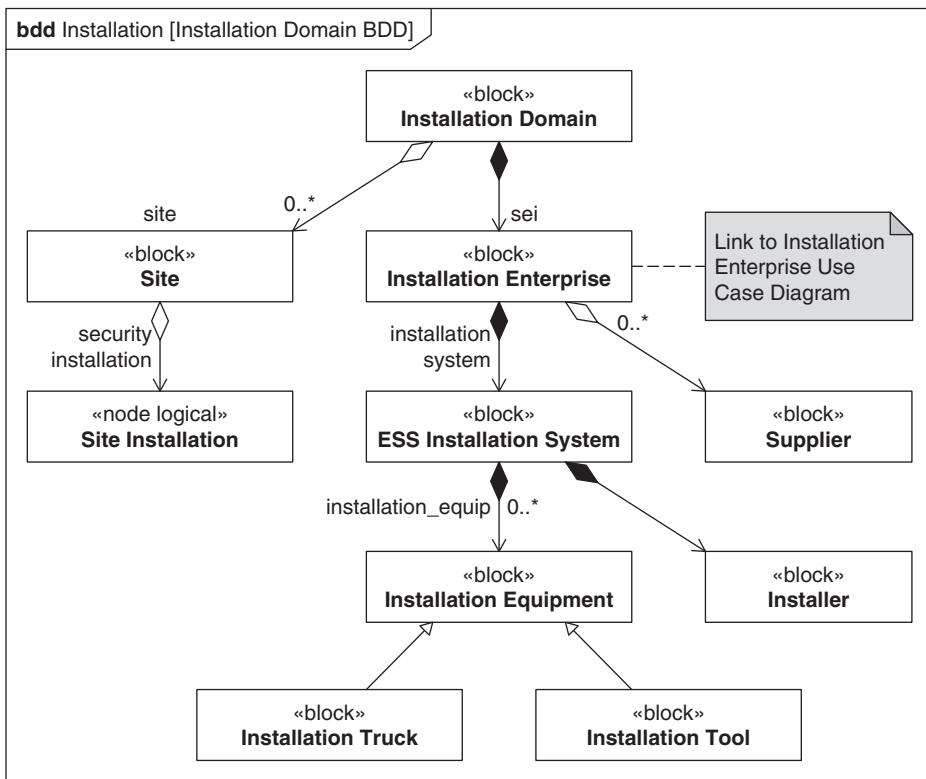


FIGURE 16.61

Concurrent development process of the operational system and enabling systems.

**FIGURE 16.62**

Installation domain block definition diagram, a starting point for the specification and design of the *ESS Installation System*.

The *ESS Installation System* may be complex as well and may warrant the application of OOSEM to its specification and design. The block definition diagram for the ESS installation domain is shown in Figure 16.62. The *Installation Enterprise* includes the *ESS Installation System* and external *Suppliers* that support the installation objectives, as defined by the installation use cases. The *ESS Installation System* includes *Installation Equipment*, such as *Installation Trucks* and *Installation Tools*, and the *Installers*. This serves as a starting point for specifying and designing the *ESS Installation System* in a similar way to the *Operational Domain Block Definition Diagram* in Figure 16.11, which was a starting point for the specification and design of the *ESS* operational system.

16.4 Summary

The example described in this chapter illustrates how SysML is used as part of a model-based systems engineering method, called OOSEM, to solve a systems engineering problem. The top-down scenario-driven method flows the requirements

down from stakeholder needs to component-level specifications, which include hardware, software, persistent data, and operational procedures. The OOSEM approach includes analysis of stakeholder needs, analysis of black-box system requirements, defining the logical architecture, synthesizing candidate physical architectures, and supporting activities to optimize and evaluate alternatives and manage requirements traceability.

The method also supports the verification process in the up side of the Vee development process. The approach illustrates how different aspects of the system are analyzed to address a multitude of concerns related to system functionality, interfaces, performance, distribution, and life-cycle considerations.

OOSEM should be tailored to the particular project objectives and constraints and associated modeling objectives, scope, and tool and resource constraints. The tailoring includes selecting the level of rigor that is applied to each of the OOSEM activities, which modeling artifacts are generated, and to what level of detail.

16.5 Questions

1. Develop the following artifacts for the *Provide Fire Emergency Response* use case shown in Figure 16.12.
 - a. Fire Emergency Response Scenario (equivalent to Figure 16.14)
 - b. Monitor Fire Activity Diagram—Logical (equivalent to Figure 16.22)
2. The customer has introduced the following new requirement: “The ESS shall provide the ability to integrate with a fire-suppression system to extinguish fires when detected with minimal adverse impact to the property.” Describe the impact of this new requirement on the system design by identifying the changes to each of the following modeling artifacts.
 - a. Requirement diagram (Figure 16.57)
 - b. Use Case Diagram (Figure 16.12)
 - c. Fire Emergency Response Scenario (refer to response to Question 1a)
 - d. ESS Context Diagram (Figure 16.15)
 - e. ESS Black-Box Specification (Figure 16.17)
 - f. ESS Logical Decomposition (Figure 16.21)
 - g. Monitor Fire Activity Diagram—Logical (refer to response to Question 1b)
 - h. ESS Logical Internal Block Diagram (Figure 16.25)
 - i. ESS Node Logical Block Definition Diagram (Figure 16.28)
 - j. ESS Node Logical Internal Block Diagram (Figure 16.32)
 - k. Allocation Table for Site Installation (Figure 16.33)
 - l. ESS Node Physical Internal Block Diagram—Site Installation (Figure 16.39)
3. Discuss how the preceding requirements change impacts the analyses to be performed.
4. How are the measures of effectiveness impacted by this requirements change?
5. How does this impact the top-level parametric diagram in Figure 16.10?
6. What additional types of analysis are required, and how can this be reflected in parametric diagrams?

PART

Transitioning to Model-Based Systems Engineering

IV

This page intentionally left blank

Integrating SysML into a Systems Development Environment

17

This chapter describes an approach to and considerations for integrating SysML into a systems development environment. The considerations include the role of the SysML model in the development environment, the logical interfaces between systems modeling tools and other tools in the environment, the specific data exchange mechanisms, and the criteria for selecting a SysML tool. Other aspects of deploying SysML in an organization are discussed in Chapter 18.

17.1 Understanding the System Model's Role in a Systems Development Environment

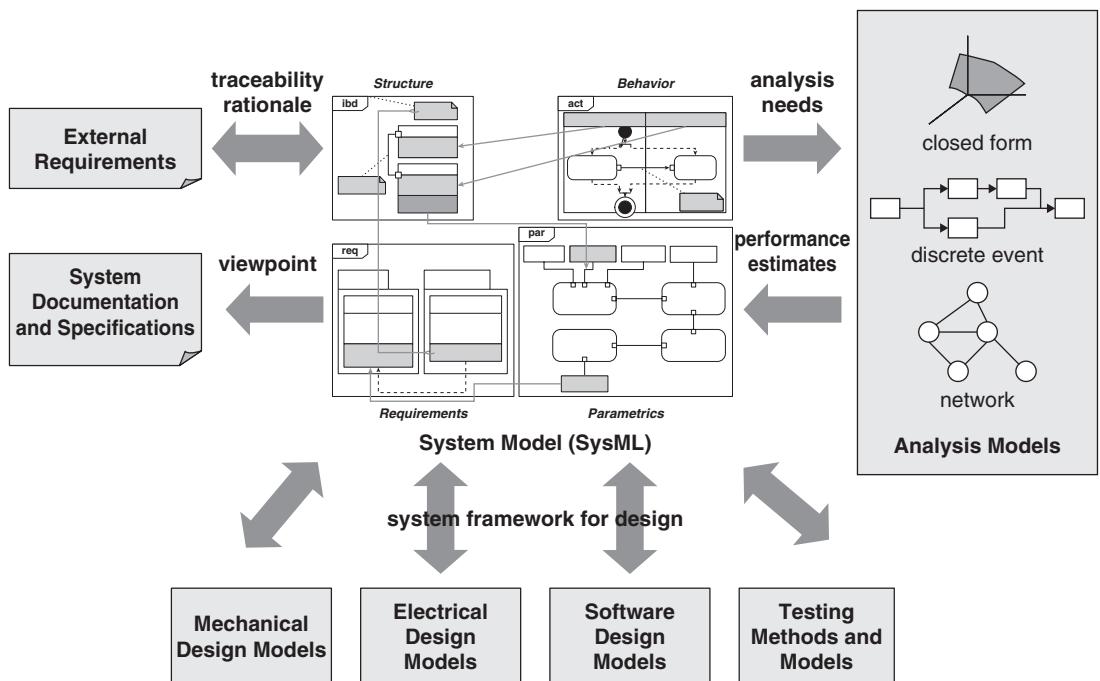
This section describes how the system model must be supported by the system development environment, how it provides an integrating framework for system development, and how it relates to understanding system dynamics via model execution.

17.1.1 Systems Development Environment

A **systems development environment** refers to the tools and repositories used for system development. Typical tools may include the system modeling tools; engineering analysis tools, hardware, software, and test tools; project management tools; and others. Tools and repositories are expected to be computer-based, multiuser, networked applications supported by a computing and network infrastructure. The term integrated systems development environment implies some logical connectivity between these tools and repositories to support collaborative engineering.

17.1.2 The System Model as an Integrating Framework

As discussed in Chapter 2 and shown in Figure 17.1, the system model is a primary product of a model-based systems engineering (MBSE) approach. As such, the application of the specific MBSE approach determines the scope and integrity

**FIGURE 17.1**

The system model as a framework for analysis and traceability.

of the model, and how it relates to other artifacts of the system development process. A system model, represented in SysML, is shown in Figure 17.1. The system model can serve as an integrating framework for other models and development artifacts including external requirements, engineering analysis models, hardware and software design models, and verification models. In particular, it relates the text requirements to the design, provides the design information needed to support the analysis, serves as a specification for the subsystem and component design models, and provides the test case information needed to support verification.

17.1.3 Relation of a System Model to an Executable System Model

As described in previous chapters, the system model captures the requirements, structure, behavior, and parametric constraints associated with a system and its environment. This information can then be queried and analyzed for consistency across the model. The system model information can be used as a basis for building an executable system model to understand and analyze the dynamics of the system. To support this, the static system modeling environment must be augmented by an execution environment. Executable system models can include a **dynamic system model**, a **performance simulation model**, and other **analytical models**.

A **dynamic system model** can be a discrete event simulation that provides a means for dynamic verification and consistency checking of the model. This abstract-level model execution can significantly enhance understanding of system operation. Dynamic system behavior, such as the sequencing of actions, input/output and message flow, and state changes, are often difficult to understand through a static graphical view of the behavior, such as an activity diagram, sequence diagram, or state machine diagram.

Execution of the dynamic system model can be augmented by animation and other visualizations to step through the behavior. Animation can rely on pre-scribed execution of defined scenarios, or it can rely on specific user interaction (e.g., “toggle this input and see what happens”). This capability can help validate functional and interface requirements, validate data types, perform what-if behavior analysis, and explore user interaction concepts. To accomplish this, the dynamic system modeling environment must be able to interpret or compile the semantic information from the SysML model, and extend it into executable code. The primary SysML modeling artifacts that support creation of a dynamic system model are the behavior and structure models.

As stated before, a dynamic system model generally focuses on the sequencing of actions and consistency of inputs and outputs. A **performance simulation**, or continuous stimulation, extends this capability to capture the underlying mathematical relationships associated with resource modeling and physics-based modeling, which is necessary to analyze system performance. The performance simulation may also include the capability to evaluate the stochastic nature of system performance, such as providing a Monte Carlo capability. These simulations are sometimes accompanied by data-analysis tools and sophisticated visualization tools that enable performance to be visualized, much like a video game. The execution environment for this capability must extend beyond the dynamic system model execution environment by providing the ability to simulate the underlying mathematics, such as numerical solutions to differential equations. In addition to the behavior and structure models in SysML, the performance simulation models can leverage SysML parametric models.

A further extension to performance simulation is the distributed simulation capability that enables multiple simulations to be integrated across a network. The High-Level Architecture (HLA) standard supports a distributed simulation capability. A simulation based on HLA requires development of Federated Object Models (FOM), which represent individual simulation modules that can communicate with one another. The Run-Time Infrastructure (RTI) provides the run-time environment for time management, publish/subscribe, messaging, and other features necessary to coordinate the distributed simulation execution.

An **analytic model** is a general class of model used to address a specific type of analysis. Analytic models can be static or dynamic and can include models of reliability, mass properties, and thermal characteristics. Performance simulation can be viewed as a type of analytic model that evaluates performance. The executable analytic model drives either a dynamic simulation or a static constraint solver that solves a set of simultaneous equations. The execution environment must include some computational engine to support this. SysML parametrics,

along with the values of related properties captured in the system model, are intended to be a key input to analysis models.

17.2 Integrating the Systems Modeling Tool with Other Tools

This section discusses integrating the systems modeling tool with other tools, including tool interface definition and interchange standards.

17.2.1 Classes of Tools in a Systems Development Environment

A systems development environment includes a wide spectrum of tools to support the various types of models and other artifacts resulting from the system development process. Figure 17.2 depicts an environment that integrates multiple types of tools to support a development process, which includes systems, hardware, software, verification, engineering analysis, and simulation, along with configuration management and other project management tools. Each class of tool implements specific parts of the overall development process. The tools are briefly summarized next.

Project management tools support planning and control of the overall development effort to ensure effective cost, schedule, and technical performance. These tools may also include workflow engines to control the whole process.

System-of-systems modeling tools support the SoS and enterprise modeling. A typical tool may include support for DoD Architecture Framework (DoDAF) modeling.

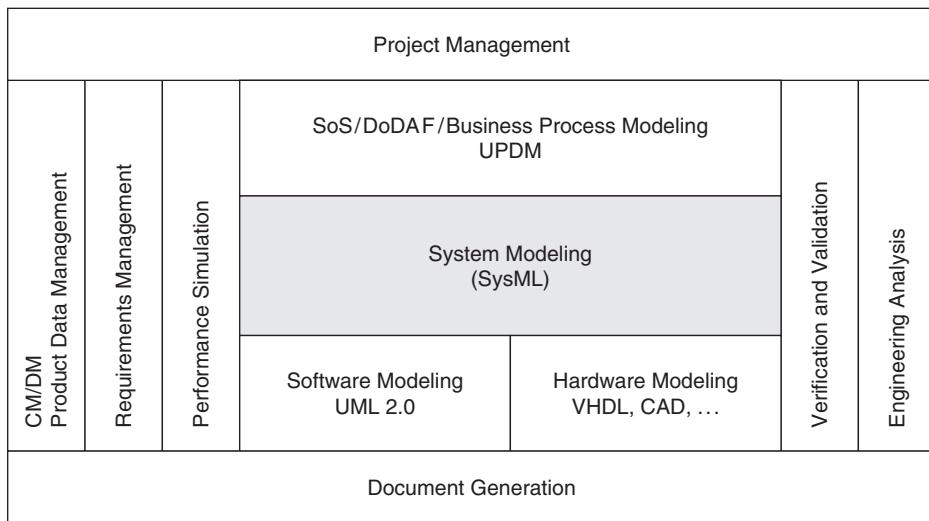


FIGURE 17.2

Interlocking disciplines in an integrated system development environment.

System modeling tools support development of the system model as described earlier. This is assumed to be the primary SysML tool.

Performance simulation tools support dynamic performance analysis and trade-off analysis from the SoS level down to the component level.

Requirements management tools generate, trace, track, and report text-based requirements, and assemble them into specification documents.

Configuration management and **data management** tools ensure that models and other development artifacts (e.g., specifications, plans, analyses, test results) are maintained in a controlled fashion.

Verification tools are used to verify compliance with requirements. The verification environment can vary from simple test tools to complex verification facilities and equipment.

Engineering analysis tools support analysis of the system design from multiple aspects, and often are specialized tools for different disciplines (e.g., reliability, safety, security, cost, and mass properties analysis).

Hardware development tools are used to design, implement, and test hardware components and may include 3D modeling tools, electrical design tools, and so on.

Software development tools are used to design, implement, and test software components and may include UML modeling tools, compilers, debuggers, and so on.

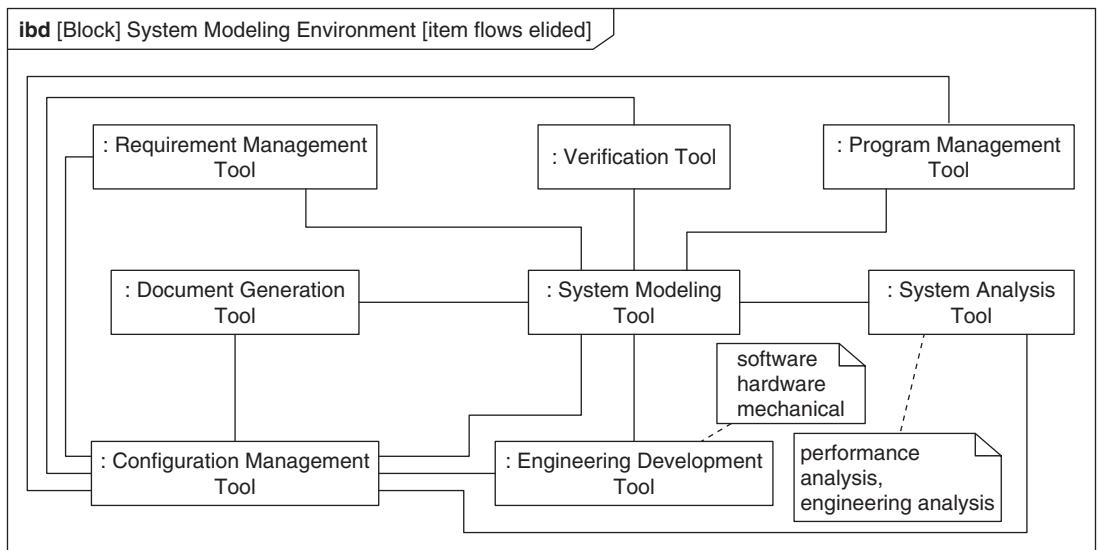
Document generation tools are used to prepare and manage formal documentation of the system design, either as text files or queries, that can be run on demand to collect and format data from the other tools.

17.2.2 An Interconnected Set of Tools

Establishing an integrated systems development environment requires a systems engineering approach in its own right. The full life cycle of the systems development environment should be considered when engineering the environment from its initial procurement, through installation and configuration, operating the environment, and maintaining the environment. Architecting of the environment should include a definition of its interfaces and the standards required to support them. The following discussion focuses on the structure and information flow within the systems development environment.

Figure 17.3 is a notional example of an integrated systems development environment, depicted as an internal block diagram. Note that each class of tool from Figure 17.2 is represented, but that some of the tools, such as the *System Analysis Tool* and *Engineering Development Tool*, include further subclasses; they are abstracted to simplify this diagram.

This example assumes that each class of tool has generic capabilities. A specific tool may fill more than one of these roles, and a specific systems development

**FIGURE 17.3**

Notional interfaces between tools in a systems development environment.

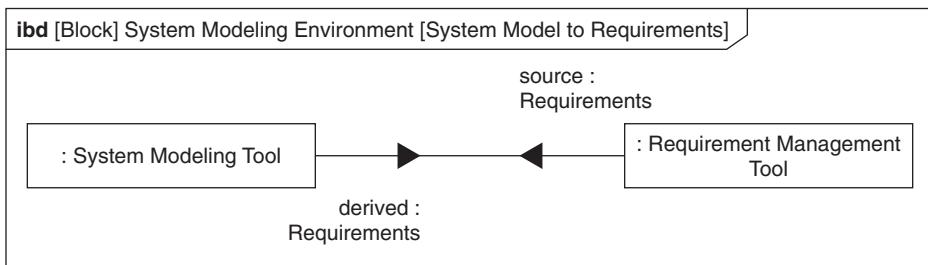
environment may include multiple tools for the same role, depending on user preferences and previous vendor arrangements. Not every class of tool needs to be included for the systems development environment to be useful. In many cases simple office tools (e.g., spreadsheets, word processing, scheduling) may be used. Each of these kinds of tools may have their own file structure or internal database. It is assumed that the *configuration management tool* is able to capture these files or databases in a managed repository throughout the development process.

The following summarizes the logical flow of data between the system modeling tool and the other classes of tools, and it provides a general approach for analyzing the interface requirements for the system modeling tool.

17.2.3 Interface with Requirements Management Tool

Figure 17.4 shows the interface between the *System Modeling Tool* and a *Requirements Management Tool*, which includes the exchange of requirements and their relationships. This can be a two-way exchange of information, but it is highly process dependent, and is a function of which tool is responsible for updating which aspect of the requirements database.

A typical approach to synchronize updates in the *Requirements Management Tool* and the model is to assume that the *Requirements Management Tool* contains all textual requirements in the requirements baseline, and that this baseline is maintained in the *Requirements Management Tool*. The *System Modeling Tool* typically addresses a subset of the total requirements depending on the scope of the model. As a result, the *System Modeling Tool* can be used to propose updates

**FIGURE 17.4**

Interface between *System Modeling Tool* and *Requirements Management Tool*.

to the requirements baseline, but they are formally updated and controlled in the *Requirements Management Tool*.

The derive relationship between the text requirements may be maintained in the *Requirements Management Tool*, as well, because they only involve relationships between text-based requirements. Other requirements relationships, such as the satisfy, verify, and refine relationships between the requirements and the model elements, may be more easily maintained in the *System Modeling Tool*. The responsibility for maintaining the data in each tool must be well defined, and the repositories must be synchronized.

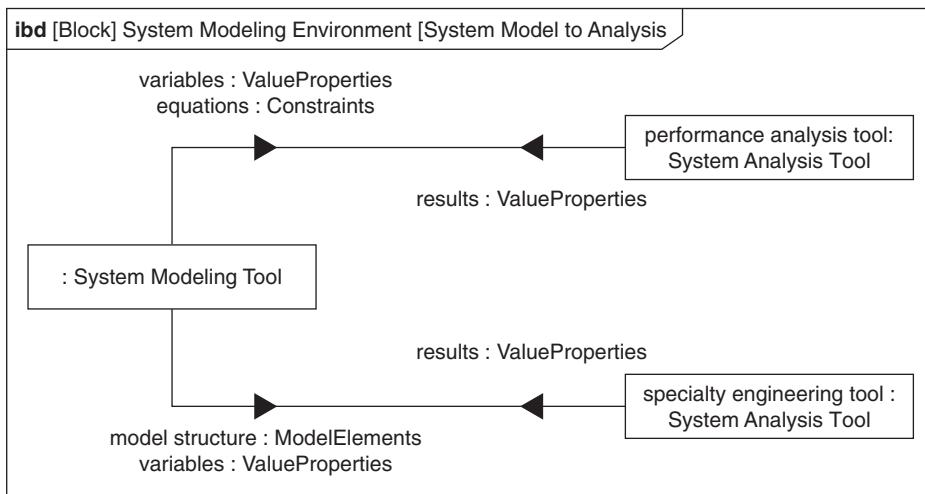
Many modeling tools also interface with other third-party tools that provide an interface between the modeling tool and requirements management tools to support traceability analysis and requirements coverage analysis.

17.2.4 Interface with Performance Analysis and Other Engineering Analysis Tools

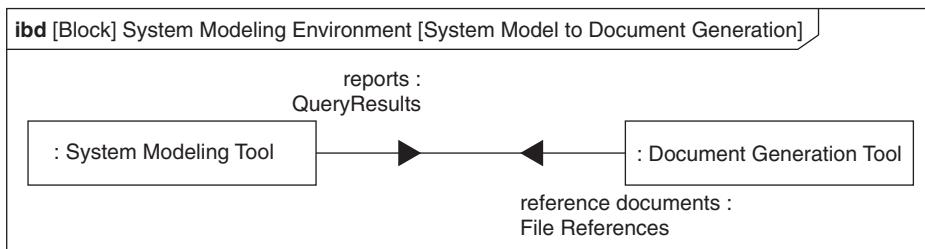
The system model expressed in SysML captures the system design model in terms of behavior, structure, and parametrics. Parametrics are a key feature of SysML that can enhance the integration between design and analysis models. The design model is analyzed in terms of performance analysis and other engineering analysis. As shown in Figure 17.5, the *System Modeling Tool* provides design information to the *System Analysis Tool*. The analysis tool performs the analysis and may provide the analysis results back to the *System Modeling Tool* in terms of property values that can be captured in the system model. For example, the system model may describe a particular network configuration connecting system elements. An analytical model may be derived from the model of system structure and constraints and provide a prediction of overall network performance. Similar types of relationships apply to other executable models, as described in Section 17.1.3.

17.2.5 Interface with Documentation Generation Tool

The specification and design information derived from the system model must be made available in a format that is easily comprehensible by a broad range of

**FIGURE 17.5**

Interface between *System Modeling Tool* and *System Analysis Tool*.

**FIGURE 17.6**

Interface between *System Modeling Tool* and *Document Generation Tool*.

stakeholders (e.g., customers, managers, design engineers, test engineers). Documents are an effective way to organize and communicate system design information to some of the stakeholder community. Document generation should report the model information in a standard and tailorabile format. Note that many SysML tools include some document-generation capability. Figure 17.6 depicts the notional relationship between the *System Modeling Tool* and the *Document Generation Tool*.

17.2.6 Interface with Configuration Management Tool

Successful systems engineering on large projects requires disciplined management of technical baselines. Incremental updates to the system model, requirements, analyses, and other artifacts can be easily overlooked and not properly considered in the design process, resulting in inefficiencies and design quality issues. The development and ongoing update of the technical baseline spans all the information in the systems development environment.

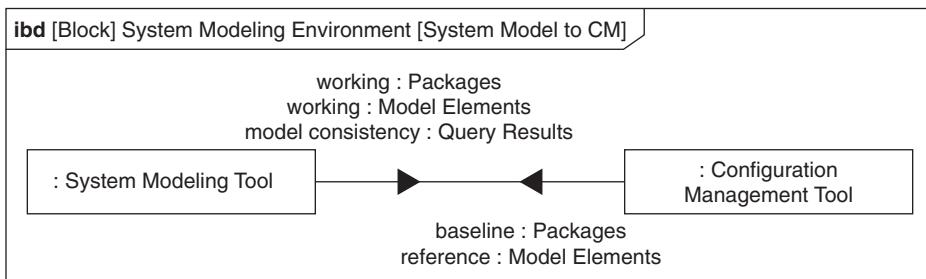


FIGURE 17.7

Interface between *System Modeling Tool* and *Configuration Management Tool*.

The interface between the *System Modeling Tool* and the *Configuration Management Tool* can be described as follows: The *System Modeling Tool* provides packages or other controlled model elements based on the model organization, and the model elements they contain to the *Configuration Management Tool*. The *Configuration Management Tool* in turn controls access to these model elements in the *System Modeling Tool* (or potentially other tools) and the ability to update the model elements either on a check-out or read-only basis. This is depicted in Figure 17.7.

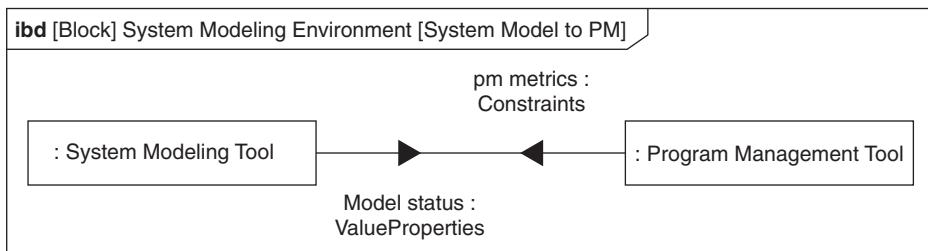
The update of the model requires a disciplined process to ensure that the updates to the technical baseline are properly reviewed to eliminate redundancy and inconsistency with other model elements, requirements, analysis results, plans, constraints, and so on, and to fully understand the impact of the change on the rest of the baseline. The *System Modeling Tool* can be used as a vehicle to check for these inconsistencies and redundancies by using queries and metrics that help reveal them.

The organization of the system model is briefly discussed in Chapter 4. Packages are often used to partition the model and as the unit of configuration control. Typical model organizations are also included in the example problems in Chapters 15 and 16. For large projects, it is usually appropriate to partition the model so that each development team will access and update work in a dedicated part of the model that it controls. Configuration management ensures that each package is appropriately versioned as model elements are updated, and which versions of the constituent packages apply.

Views and viewpoints were described in Chapter 4 and may play an increasingly important role in providing access to the model and for enabling the sharing of current model data between tools and teams.

17.2.7 Interface with Project Management Tool

Project management can leverage information from the system model to assist in planning and control. The model-based metrics described in Chapter 2 are examples of metrics that can be extracted from the model to assess design quality and

**FIGURE 17.8**

Interface between *System Modeling Tool* and *Program Management Tool*.

design progress and to estimate the level of effort required. The following additional data can easily be provided from the system model:

- Number of model elements created or updated within a specific time period
- Number of requirements linked by satisfied or verified relationship within a time period
- Number of use cases realized
- Number of activities allocated to blocks
- Number of analysis results (value properties) identified versus number updated
- And so on

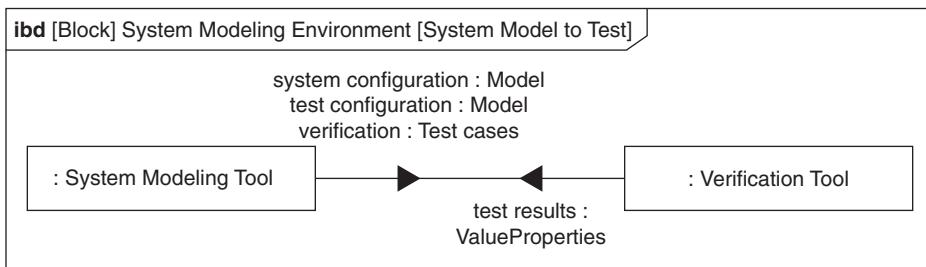
These metrics can be automatically reported from the model, typically by using the scripting capability of a given tool, providing concrete information to assist in managing the development effort.

As shown in Figure 17.8, the *Program Management Tool* can be used to establish metrics in the form of SysML constraints, and the *System Modeling Tool* can then evaluate those expressions and provide model status back to the *Program Management Tool* in the form of computed value properties from the model.

17.2.8 Interface with Verification Tool

Verification planning and conduct is often facilitated by a unique set of tools within a verification environment. This environment is used to verify that each requirement is satisfied, generally by providing a stimulus to the system and monitoring its response to determine whether the requirement is met. Other verification tools are used to support other verification techniques, such as inspection. The verification system environment can be modeled in the system modeling tool along with the operational system it is designed to test, as briefly discussed in Chapter 16.

As shown in Figure 17.9, the *System Modeling Tool* can provide the specific system configuration under test, and a set of test cases to the *Verification Tool*. Once tests have been conducted, test results can be passed back into the *System Modeling Tool* and reconciled with the rest of the model. The results are also passed to the requirements management tool to update the verification status.

**FIGURE 17.9**

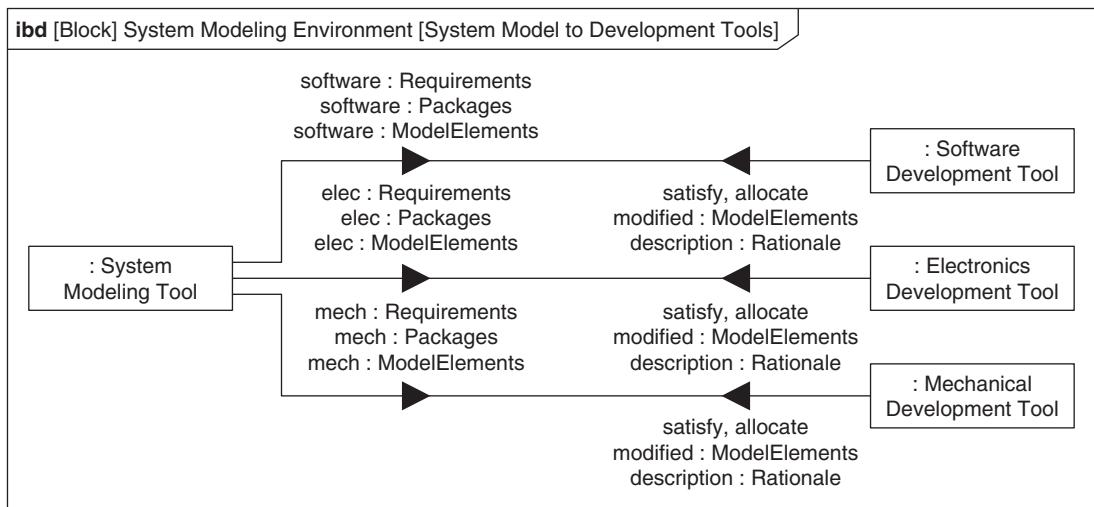
Interface between *System Modeling Tool* and *Verification Tool*.

17.2.9 Interface with Development Tool

A principal reason for developing system models is to specify the requirements and constraints for the system's components, which typically include hardware and software. The interface between the *System Modeling Tool* and hardware and software development tools is a critical one. In particular, the *System Modeling Tool* provides the component specifications to hardware and software *development tools*, which in turn provide design verification data that the hardware and software design models satisfy the specifications. As the state of practice matures, the goal will be to fully integrate the hardware and software component design models back into the system model to verify that the design meets the system requirements.

Figure 17.10 depicts the kinds of information potentially flowing between a *System Modeling Tool* and various hardware and software *development tools*. In each case, the *System Modeling Tool* provides component requirements specific to that domain, as well as model structure (packages and model elements) that provides system context for those requirements. The component black-box specification may be in the form of the component blocks with their features specified, including their interfaces, state machine, behavioral requirements, and value properties. In response, hardware and software *development tools* provide the satisfy relationships to parts of the detailed design with rationale, along with issues that need to be addressed by the system model.

For software design environments using UML, the interface between the *System Modeling Tool* and the UML modeling tool is more a function of the model-based methods because the underlying language concepts have the same roots. In this case the SysML model can be extended and refined in the UML modeling tool to evolve the software design from the specifications. For *hardware development tools*, the interface is more complex since the language structures are different. For these cases, component specifications need to be transformed into the domain of the hardware language. There are mechanisms that are intended to assist in the transformation, such as the ISO STEP standards described in the next section, but this is still work in progress.

**FIGURE 17.10**

Interface between *System Modeling Tool* and hardware and software development tools.

17.3 Data Exchange Mechanisms in an Integrated Systems Development Environment

This section discusses the mechanisms and standards for data exchange between tools.

17.3.1 Data Exchange Mechanisms

Exchange of data between tools in a systems development environment may be accomplished in the following ways:

- Manually (e.g., re-keying the data from one tool into another tool)
- File-based exchange: neutral file format (e.g., csv), native file format (e.g., mdl), or exchange format (e.g., XMI)
- Interaction-based exchange (using APIs or queries to draw data from tools only when required)
- Repository-based exchange (using exchange format files and packaging schemes to manage data independent of the tool)

Selecting which of these approaches to implement between any two tools must take into account the cost of implementing them versus the long-term value tool integration will provide. The following considerations are recommended:

- *How often will data be exchanged between these tools and over what duration?* If it is known in advance that the data need only be exchanged once,

a less sophisticated and lower-cost data exchange may be appropriate. On the other hand, even a simple data exchange, if expected to occur on a frequent basis over an extended period of time, may justify the cost of developing some kind of automated data transfer.

- *How complete or reliable must the data be that are transferred?* For particularly large data transfers, or where transfer errors cannot be tolerated, it may be appropriate to invest in automated data transfer to minimize the possibility of manual error.
- *Is the subject system a single one-off product or the basis for an ongoing product line?* The value of investing in a robust systems development environment with automated data exchange between tools, which facilitates reuse development data, may not be fully realized until the rapid update or development of a follow-on system.

Automation of file-based exchanges between two tools may be accomplished using a “**bridge**,” or purpose-developed software application. In this case some amount of redundant data exist in each tool. The appropriate synchronization of these data is the responsibility of manual procedures, constraints, or the operation of the bridge software, or it can be addressed by model transformations.

An exchange of data between tools may also occur on an as-needed, or interaction basis. This is facilitated by the use of a tool’s **application programming interface** (API) to access and filter data in that tool, and to make them available to another tool or database. This method can be very rapid, repeatable, and reliable, but it is important to understand how the development process anticipates using each tool, the data dependencies between tools, and how often these interactions must occur. Otherwise, tool users may end up competing over various design parameters or characteristics.

A **repository** is a configuration-managed database, accessible to two or more tools that contain the data files the tools share. Repositories generally support multiple tools that file and categorize systems engineering data in a database rather than just lists of data exchange files. Maintaining systems engineering data in this manner enables the use of consistency checkers on the entirety of the repository data rather than relying on consistency checkers in individual tools. Repositories that maintain this kind of systems engineering data can publish a metadata catalog, allowing other tools access to both the data and their meaning.

17.3.2 Role of Data Exchange Standards

The exchange of data between modeling tools has traditionally been accomplished by creating a point-to-point exchange between individual tools using the mechanisms described earlier. This can be costly because each tool requires its own interface mechanism. Implementing point-to-point interfaces can require the development of n^2 interfaces for n tools. In addition, the interface mechanism must be updated as each tool changes. The emphasis for an Integrated Systems Development Environment is on the use of data exchange and other modeling

standards to support tool and model interoperability. Some of the relevant standards related to SysML are briefly discussed next.

XML Metadata Interchange

XMI is short for eXtensible Markup Language (or XML) Metadata Interchange [21] and provides a standard format for interchanging UML and SysML models between tools. XMI is based on three industry standards: the eXtensible Markup Language, the Meta Object Facility (MOF), and the Unified Modeling Language (UML). MOF and UML are modeling and metadata repository standards from the Object Management Group (OMG). XML is a text-based language from the World Wide Web Consortium (W3C) that supports the use of tags to describe structured data. XMI is in essence a set of rules for converting a metamodel, expressed using MOF, UML, and UML profiles into a set of custom tags in XML. Hence SysML, which is a UML profile, also has implicit interchange standards using XMI. However, there may be tool limitations, as well as issues with how the models are used, that can impact the quality of the exchange.

Figure 17.11 shows a simple SysML diagram, where *Block1* is composed of *Block2*, both of which have properties. Figure 17.12 is the equivalent XMI generated from the model. The XMI fragment in Figure 17.12 identifies each model element in terms of its UML metaclass type, unique id, and other information depending on its metaclass.

Note that the id's in Figure 17.12 have been simplified for ease of drawing because globally unique id's would have been very cumbersome for this figure. The diagram frame denotes a package with the name *Parent* that is also captured in the XMI as the owner of both *Block1* and *Block2*. However, the diagram kind, user-defined diagram name, and other diagram information (e.g., symbol positions) are not included in the exchange.

Where the model elements represent SysML concepts, they are extended by instances of SysML's stereotypes, as described in Chapter 14. In this case instances of block reference back to the UML element it extends.

Application Protocol 233

STEP, or the Standard for the Exchange of Product Model Data (more formally known as ISO 10303 [22]), is an international standard for the computer-interpretable representation and exchange of product data. The objective is to provide a mechanism that is capable of describing product data throughout the

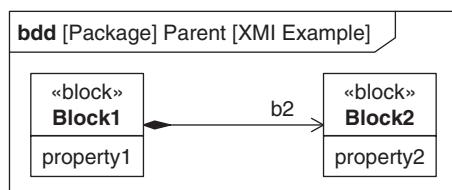


FIGURE 17.11

Simple SysML diagram, as an example for illustrating XMI.

```
<ownedMember xmi:type="uml:Package" xmi:id="ID0" name="Parent" visibility="public">
    <ownedMember xmi:type="uml:Class" xmi:id="ID1" name="Block1" visibility="public">
        <ownedAttribute xmi:type="uml:Property" xmi:id="ID2" name="Property1" visibility="private" />
        <ownedAttribute xmi:type="uml:Property" xmi:id="ID3" name="b2" visibility="private"
            aggregation="composite" type="ID5" association="ID4" />
    </ownedMember>
    <ownedMember xmi:type="uml:Class" xmi:id="ID5" name="Block2" visibility="public">
        <ownedAttribute xmi:type="uml:Property" xmi:id="ID6" name="Property2" visibility="private" />
    </ownedMember>
    <ownedMember xmi:type="uml:Association" xmi:id="ID4" visibility="public">
        <memberEnd xmi:idref="ID3" />
        <memberEnd xmi:idref="ID7" />
        <ownedEnd xmi:type="uml:Property" xmi:id="ID7" visibility="private" type="ID1" association="ID4" />
    </ownedMember>
</ownedMember>
...
<SysML:Block xmi:id="ID8" base_Class="ID1" />
<SysML:Block xmi:id="ID9" base_Class="ID5" />
<SysML:BlockProperty xmi:id="ID10" base_Property="ID2" />
<SysML:BlockProperty xmi:id="ID11" base_Property="ID3" />
<SysML:BlockProperty xmi:id="ID12" base_Property="ID6" />
```

FIGURE 17.12

Equivalent XMI (fragment) for Figure 17.11.

life cycle of a product, independent of any particular system. The nature of this description makes it suitable not only for neutral file exchange but also as a basis for implementing and sharing product databases and archiving.

Application Protocol 233 (AP233) is a STEP-based data exchange standard targeted to support the needs of the systems engineering community; it is consistent with emerging standards in CAD; structural, electrical, and engineering analysis; and support domains. SysML was developed in coordination with the development of the AP233 standard, which has resulted in shared systems engineering domain concepts. It is anticipated that over time SysML tools will be able to leverage AP233 as a neutral format for exchanging SysML models.

Diagram Interchange Standards

An important distinction is made between data interchange and diagram interchange. The preceding standards can exchange model data, but do not explicitly exchange diagram layout information in terms of where the symbols belong and where they appear on a diagram. There is a diagram interchange standard specification from the OMG [42], but it is not widely used. However, if the model information is exchanged and the tool repository is populated with the data, some tools provide a capability to autogenerate the diagram from the model repository. The resultant diagram will not reflect the original diagram layout because that information is not part of the exchange.

Model Transformation

There are clearly many different modeling languages for systems, hardware, and software development as well as domain-specific languages for real-time analysis,

business process modeling, and so on. When the desire is to move data from one modeling language to another, a model transformation is required; this involves mapping of the concepts from one language to the concepts in another language. The transformation may result in data loss or ambiguity. There are standards based on the OMG Meta Object Facility [20] that provide a foundation for these transformations if the metamodel for the language is expressed in a standard MOF format. There are many other approaches to model transformation, and this area will become increasingly important as model-based approaches and domain-specific languages are used more often.

17.4 Selecting a System Modeling Tool

This section focuses on selection of a SysML modeling tool for the systems development environment. A system modeling tool may support SysML to a greater or lesser extent, in accordance with the strengths and weaknesses offered by the tool.

17.4.1 Tool Selection Criteria

The following criteria can form the basis for evaluating and selecting a SysML modeling tool:

- Conformance to SysML specification (latest version)
- Usability
- Document generation capability
- Model execution capability
- Conformance to XMI
- Conformance to AP233
- Integration with other engineering tools (including legacy tools within an existing system development environment)
 - Requirements management
 - Configuration management
 - Engineering analysis tools
 - Performance simulation tools
 - Software modeling tool
 - Electrical modeling tool
 - Mechanical CAD tool
 - Testing and verification tool
 - Project management tools
- Performance (maximum number of users, model size)
- Model checking to verify model conformance (well-formedness rules)
- Training, online help, and support
- Availability of model libraries (e.g., SI units)
- Life-cycle cost (acquisition, training, support)
- Vendor viability

- Previous experience with the tool
- Support for selected model-based method (e.g., scripts that automate certain parts of the method, standard reports, etc.)

17.4.2 Specific Tool Support for the MBSE Method

The specific MBSE method employed may leverage specific SysML features but may not require other features. It is appropriate to ask the following questions to emphasize the features of SysML that a successful tool deployment will need to support.

- *Which behavior representations are the most important?* Activity diagrams? State machines? Sequence diagrams?
- *Will there be a need for item flow representation?*
- *What kind of need will there be for detailed performance analysis and parametric modeling?* Expression of mathematical equations relating parameters of system elements may be a very important part of the system development process and/or method employed.
- *Will there be a need for algorithm specification and development?* It may be important to express information processing algorithms explicitly in mathematical form, using constraint blocks and eventually relating them to specific blocks representing software code.
- *Which architecting principles need to be supported by the tool?*
- *How will allocation be used?* The manner in which allocation is used to guide the development process may dictate a set of constraints and rules associated with allocation relationships. By enforcing or enabling these rules, a tool set can improve the efficiency of the modeling process.

17.4.3 SysML Compliance

According to the SysML specification, a tool can claim its compliance with SysML in terms of compliance to the common subset of UML and the language extensions described by the SysML profile, as described in Chapter 4. Table 17.1 contains an example of a compliance matrix for a notional SysML tool, the form of which is defined in the SysML specification.

This matrix can be used by modeling tool vendors and others to define a given tool's level of compliance with the standard. For each unit of the language, compliance with abstract syntax—underlying language constructs, like metaclasses, stereotypes, constraints, and ability to generate XMI—and compliance with concrete syntax (e.g., graphical notation) are stated.

UML4SysML is the portion of UML that is reused in SysML. The three levels referenced are described in Table 17.2. The matrix can be used as part of the tool-selection process to determine whether critical SysML features and capabilities are included in the tool. For example, if activity modeling with probability

Table 17.1 Example SysML Compliance Matrix Summary for a Notional Tool

Compliance Level	Abstract Syntax	Concrete Syntax
UML4SysML Level 1	YES	YES
UML4SysML Level 2	PARTIAL	YES
UML4SysML Level 3	NO	NO
Activities (without Probability)	YES	YES
Activities (with Probability)	NO	NO
Allocations	PARTIAL	PARTIAL
Blocks	YES	YES
Constraint Blocks	YES	YES
Model Elements (without Views)	YES	YES
Model Elements (with Views)	NO	NO
Ports and Flows (without Item Flow)	YES	YES
Ports and Flows (with Item Flow)	NO	NO
Requirements	YES	YES

Source: *OMG Systems Modeling Language (OMG SysML)*, V1.0, OMG document number formal/2007-09-01, Table 5.5.

Table 17.2 UML4SysML Compliance

SysML Package	UML4SysML Compliance Level
Activities (without Probability)	Level 2
Activities (with Probability)	Level 3
Allocations	Level 2
Blocks	Level 2
Constraint Blocks	Level 2
Model Elements (without View)	Level 1
Model Elements (with View)	Level 3
Ports and Flows (without Item Flow)	Level 2
Ports and Flows (with Item Flow)	Level 3
Requirements	Level 1

Source: *OMG SysML*, V1.0, OMG document number formal/2007-09-01, Table 5.5.

is important to a user's systems engineering approach, along with the ability to export the resulting model in XMI format, then the system modeling tool selected should demonstrate full UML4SysML compliance at Level 3 and activities (with probability) both for abstract and concrete syntax.

After understanding the language features needed and comparing vendors' self-evaluation of their tools compliance with SysML, an evaluation of the tool should be performed based on actual usage. The tool features should be evaluated in the systems development environment envisioned using the methods on typical problems relevant to the domain.

17.5 Summary

Integrating SysML into a systems development environment includes some of the following considerations.

- The system model in SysML is an integral part of the overall system development used to relate text requirements to the design, provide design information needed to support the analysis, serve as a specification for the subsystem and component design models, and provide the test case information needed to support verification.
- System modeling tools do not stand alone but must be integrated into a system development environment that includes many other tools that support requirements management, engineering analysis, hardware and software development, verification, configuration management, and project management.
- A systems engineering approach should be applied to specify the requirements and interfaces for the integrated system development environment.
- Data exchange between tools can be accomplished by manual, file-based, interaction-based, and repository-based mechanisms.
- A standards approach to data and model interchange is the preferred approach to reduce the cost and improve the quality of the data exchange. XMI is a primary data exchange mechanism, but this does not include diagram layout information.
- SysML tool selection should be based on an evaluation against a defined set of criteria that includes both review of vendor information and hands-on use of the tool in the expected environment. Tool compliance to the SysML standard is one critical criterion.

17.6 Questions

1. Why does SysML facilitate establishing a model-based system development environment?
2. Describe how XMI and AP233 are used with SysML.
3. What are five criteria for selecting a SysML tool?
4. What conditions might lead someone to choose a SysML tool with strong structural modeling capability and internal consistency checking over one with probabilistic activity modeling capability? Which level of UML4SysML compliance (Level 1, Level 2, or Level 3) would this preferred tool exhibit?

5. Under which conditions would a tool's capability to model ports, flows, and item flows be important? Which level of UML4SysML compliance (Level 1, Level 2, or Level 3) would this tool exhibit?
6. What can a person do to limit the impact of future tool changes or upgrades on the cost of their system development environment?

Discussion Topics

Describe the role of the system model in the system development environment.

Describe the meaning of the term “executable model” and two different purposes for developing executable models.

Describe how the use of a system model can potentially increase the effectiveness of a system development environment.

Build a matrix listing eight types of tools that can benefit from sharing data with a system modeling tool. In one column, list beneficial information that can flow from the system modeling tool, and in another list information that can flow to the system modeling tool.

Describe four different ways of exchanging data between tools in a system development environment. For each method, describe when it might be most appropriate.

18

Deploying SysML into an Organization

Introducing the use of SysML to an organization and projects should be planned as part of an improvement initiative that addresses the impact on the systems engineering process, methods, tools, and training. This chapter describes how to implement an improvement process to facilitate a smooth and successful transition to SysML.

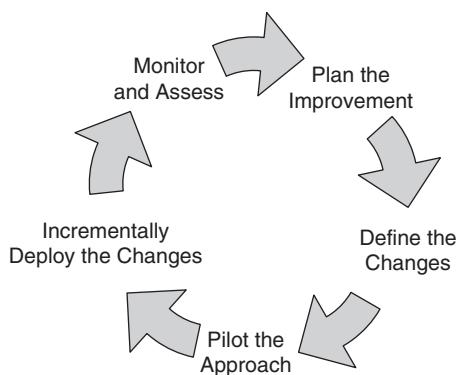
18.1 Improvement Process

Introducing any significant change into an organization requires a well-thought out plan and disciplined implementation to be successful. The introduction of SysML is part of a model-based systems engineering (MBSE) approach. The change to SysML should be implemented using the organization's improvement process that includes changes to the model-based method, tools, and training. Clear responsibility for the improvement should be established, and the expected benefits of the change should be understood and agreed on with stakeholders.

A typical **improvement process** is shown in Figure 18.1. The process includes monitoring and assessing projects to determine issues and improvement goals; developing the improvement plan; defining proposed changes to the process, methods, tools, and training; piloting the approach; and incrementally deploying the improvement. The process should be applied in increments to improve the organization's capability. The steps to implement SysML as part of an improvement initiative are described next.

18.1.1 Monitor and Assess

To introduce a change to improve the organization's capability, a baseline for measuring the improvement should be established. In particular, with respect to introducing SysML and MBSE, the organization should establish how systems engineering is currently being practiced and identify the issues and improvement goals expected from transitioning to SysML/MBSE. The MBSE benefits described in Chapter 2 represent possible motivations for the change. The issues to be addressed and the improvement goals can be used to derive metrics that can be

**FIGURE 18.1**

Improvement process for deploying SysML.

monitored over time. Metrics in turn can be used to determine the effectiveness of the change and to provide an input for follow-up improvement planning.

The maturity of MBSE will vary from project to project in a large organization; it may range from a totally document-based approach on some projects, to some limited use of functional analysis, architecture, and performance simulation modeling on other projects, to pockets of advanced systems modeling that may be integrated across the project. The state-of-practice assessment can provide information about what is working and what is not. The assessment results can be used to identify preferred practices to be shared, and the issues to be addressed by the improvement plan. The results can also be used to identify and select candidate pilot projects and potential target projects for deployment.

An assessment questionnaire can be prepared to support the assessment and may include questions regarding the purpose and scope of MBSE on projects, methods, tools, and training that are being used; how well they are working; and issues and lessons learned. The questionnaire can be administered to organizational and project representatives remotely or through face-to-face meetings. Representation from multiple projects and disciplines should be sufficiently diverse to provide a comprehensive assessment.

Metrics are defined to measure how broadly SysML/MBSE is deployed, and the effectiveness of the improvement. The deployment metrics can include the number and percentage of people trained in SysML/MBSE and the number and type of target projects that are applying SysML/MBSE. The effectiveness metrics measure progress against expected benefits and improvement goals, and how well the issues are being addressed. They should provide quantitative data to help assess the MBSE benefits and the impact on productivity, quality, innovation, and other business objectives.

18.1.2 Plan the Improvement

The improvement plan defines how to accomplish the improvement goals. It should include the activities from the improvement process in Figure 18.1 and a phased approach to develop and deploy changes incrementally into the organization.

The plan should detail the necessary resources and schedule and a commitment to provide the resources.

As with any plan, stakeholder participation is essential in both its formulation and its execution. The stakeholders for MBSE include members of the improvement team responsible for defining the change, as well as the project stakeholders who are expected to implement SysML/MBSE. The stakeholders representation should come from project management, systems engineering, and the development teams including software, hardware, and testing; this group could also include customers and subcontractors. It is important to get representation from all the “communities” early in the process to ensure that their concerns are being addressed, and that there is buy-in to the improvement goals and plan.

18.1.3 Define Changes to Process, Methods, Tools, Metrics, and Training

The improvement will require changes to the organization’s process, methods, tools, metrics, and training. The changes should be defined, documented, reviewed, and approved by the affected stakeholders to ensure that the change is implementable and will achieve the desired results.

Process Changes

It is assumed that the baseline systems engineering process for the organization and/or project is defined. If not, establishing a baseline that reflects the current process is an important first step. The process standards referred to in Chapter 1 provide a starting point for defining the systems engineering process. Sometimes there is a significant disparity between the documented processes for an organization and the way that processes are actually implemented on projects. This is a separate issue that should be addressed, but it is not the focus for this discussion.

The systems engineering processes should be evaluated to determine the impact of SysML/MBSE. This includes impact on both the technical processes and the management processes such as configuration management, review processes, and measurement.

Method Changes

An MBSE method should be evaluated and selected to support the technical processes. There may be methods that are practiced internal to the organization as well as others that are available across industry, as described in Chapter 2. Two example methods are described in Chapters 15 and 16. The criteria for selecting a method may include how well it addresses the concerns of the project, the level of tool support, and the training requirements. The methods should be documented along with an example problem to show how it is applied. The documentation should also include general modeling conventions (e.g., naming conventions and style guidelines) and recommended model organization (refer to Chapter 5 and the examples in Chapters 15 and 16).

Tool Changes

The MBSE tools also need to be evaluated and selected. Criteria for SysML tool selection are included in Chapter 17. The evaluation should also include trial use

of the tool to see how well it addresses the criteria. Documentation should be provided for how the tools are acquired, installed, configured, used, and maintained.

The documentation of the MBSE method should be updated to provide tool-specific guidance on how the selected method is used with the selected tools. This may include general information on the how to create the modeling artifacts in the tool, as well as very detailed practices such as how to specify an interface.

MBSE Metrics

The MBSE metrics should be defined to support the overall goals of the project, as described in Chapter 2. The data-collection approach and reporting should also be defined.

Training Changes

Training is needed to support the language, method, and tools and may require different training courses. SysML training should focus on the language concepts described in Part II. The method documentation referred to earlier can be used to support the method training with examples included such as those in Part III. The introductory tool training may best be provided by the tool vendor to show how the tool is used. However, this may be augmented to include additional training on how the tool is used with the selected method and as part of the specific tool environment, as discussed in Chapter 17.

18.1.4 Pilot the Approach

As with any significant change, the recommendation is to walk before you run. This involves piloting the changes described earlier to validate and refine the approach. Undoubtedly, there will be modifications to the approach based on the results of the pilot project.

A pilot project also requires careful planning, willing participants, necessary resources, and management support. A typical plan for a pilot includes the following:

- Pilot objectives and metrics
- Pilot scope
- Pilot deliverables
- Pilot schedule
- Responsibilities and staffing
- Process and method guidance
 - High-level process flow
 - Model artifact checklist
 - Tool-specific guidance
- Tool support
- Training

The pilot's objectives may include validating that the proposed MBSE method, tools, and training meet the needs of the organization and projects. The scope of

a pilot should support these objectives. A small team should be identified to work on the pilot with a team lead. It is important to maintain continuity among the team members as they work through the pilot.

The selected tools must be acquired, installed, and configured. The pilot team should receive training in the language, method, and tools. The documented method and tool guidance should be provided to the team. In addition, it is preferable that the pilot team includes a member who is skilled in the method and/or tools to provide guidance to other team members.

The pilot project should adequately exercise the method and tools. It is often useful to select a thread through the system and generate at least one artifact for each of the artifacts in the method. The pilot schedule should include milestones for creating the modeling artifacts. The team should also establish a peer-review process to review the artifacts and propose changes to the method, and then use this to help refine the MBSE approach.

The results are captured in a report that includes how well the pilot achieved the objectives, what modifications were made to the proposed approach, and lessons learned, including quantitative data where practical. The OOSEM method described in Chapter 16 was piloted and documented in a reference paper [38] and provides an example of how a pilot was conducted and its results.

Based on a pilot's results, the process, methods, tools, metrics, and training should be updated to reflect the new baseline MBSE approach. The results can serve as training material to be used as part of the broader SysML/MBSE rollout. Some pilot participants can become advocates to help deploy the change to projects.

18.1.5 Deploy Changes Incrementally

The pilot results help to determine the requirements for deploying the SysML/MBSE capability on projects. The pilot provides a more realistic assessment of the type of training required, how long it takes for people to get up to speed, experience in adapting the method and tools, and more realistic expectations of the modeling results.

Project-selection criteria should be established to select a project or projects targeted for the deployment. The criteria may include the project's phase, longevity, size, level of internal and customer support, and the extent to which MBSE benefits can provide recognized value to the project. In addition, the state-of-practice assessment should have helped to identify potential opportunities based on business need and other considerations. Different projects may introduce different scopes for MBSE depending on their current state of practice, their experience level in modeling, and the particular project needs. Ideally, SysML/MBSE is introduced at the start-up phase of a project or at a point in its life cycle that is appropriate to introduce change, for example, at the start of a new development increment. It is important for the project's leadership and customers to be willing advocates for the change.

Realistic expectations should be established in terms of the time, effort, deliverables, and expected results from the modeling effort. The purpose and scope of

the effort should be defined and balanced with project resources, as described in Chapter 2. The MBSE deliverables should be reflected in the project plan, along with milestones for establishing the MBSE infrastructure, including documentation, tools, staffing, and training.

MBSE metrics should be identified to support project objectives. The MBSE metrics in Chapter 2 can serve as a guide along with the lessons learned from the pilot. The approach for data collection should also be defined, including how the data are to be captured from the tools. The reporting of the metrics should be detailed in the project plan, including which metrics, how often they are collected, and how they are used.

The selected tools are acquired, installed, and configured for use. On a larger project, the tools will need to be configured for a multiuser environment. Additional levels of tool integration may be required, as described in Chapter 17. The configuration management approach for controlling a model baseline will need to be clearly defined. The right expertise should be made available to help establish and maintain this environment.

The deployment should include start-up training in the process, methods, and tools. The training should encompass SysML training, MBSE method training, and tool training. The training should use the pilot's documentation and results as part of the training material. Different levels of training may be appropriate for different stakeholders. For example, some of the systems engineering team, which is designated as the core modeling team, may require detailed SysML, MBSE methods, and tool training, whereas other systems engineers and some hardware and software developers may require limited SysML training, which includes the impact on their particular tasks or methods. For example, some of the testers may need to understand how to derive detailed test cases from the model, or the individual who is responsible for requirements management will need to understand how the SysML modeling tool is used with the requirements management tool.

A successful deployment will also require ongoing support from individuals who have expertise in the methods and tools. The improvement metrics should be monitored to assess the MBSE effort. Lessons learned should be captured to further refine the process, methods, and tools and to help further drive the improvement process.

18.2 Summary

SysML is deployed as part of an MBSE approach using the organization's improvement process. Deploying SysML as part of MBSE should consider impacts on the systems engineering process, methods, tools, and training. A successful deployment must be planned, piloted, and incrementally deployed. The success is a key ingredient to motivate other projects to follow. The results should be quantified, where practical, and used as a basis for future improvements.

18.3 Questions

1. When SysML is being deployed, which other aspects of MBSE should be considered?
2. What are the activities in the improvement process?
3. Who are some of the stakeholders in the improvement process?
4. What is the purpose of the monitor and assessment activity?
5. What is the purpose of piloting the MBSE approach?
6. What are some of the up-front project activities that must be planned when deploying SysML to a project?

This page intentionally left blank

SysML Reference Guide

A.1 Overview

This appendix provides a reference guide to the graphical notation for SysML as a set of notation tables. It is organized by diagram kind in the following order:

- Package Diagram
- Block Definition Diagram
- Internal Block Diagram
- Parametric Diagram
- Activity Diagram
- Sequence Diagram
- State Machine Diagram
- Use Case Diagram
- Requirement Diagram

There are also notation tables for the use of allocations and stereotypes, which are used across a number of different diagrams.

It is recommended that you read Section 4.3 in Chapter 4 for an overview of SysML diagrams and their contents before reading this appendix.

A.2 Notational Conventions

Each diagram is described by at least one notation table. For diagrams with many symbols, there are separate tables for nodes and paths, where node symbols are typically rectangles and ovals and path symbols are lines. Package diagrams and block definition diagrams have several subsections to describe different uses of the diagram with corresponding notation tables. The rows in each table are ordered so that their references are encountered in ascending order within the relevant chapter or chapters.

The notation tables have four columns:

- Diagram Element—the name of the diagram element represented in this row, generally identified as a node or path. The term symbol is used when it is neither a node nor a path, such as a text expression in brackets.
- Notation—the graphical notation for the diagram element.
- Description—a description of the SysML concept represented by the diagram element.
- Section—a reference to the section(s) in Part II that contains further explanation of the relevant SysML concept.

The following conventions are used in the tables:

- <Name>—the name of the model element represented by the symbol.
- <Element>—the name of some model element.
- <Type>—the name of some type (Block, ValueType, etc.).
- <String>—any arbitrary text string.
- <Expression>, <ValueSpecification>—a text string intended to represent some kind of regular expression.
- <ElementType>—the keyword representing some kind of model element.
- <Multiplicity>—a representation of multiplicity, thus: <LowerBound>...<UpperBound>, where LowerBound is any natural number and UpperBound is any natural number or “*.”

The names inside the angled brackets are intended to be self-explanatory references to SysML model elements, but occasionally extra explanation is provided in the Description column of a symbol.

It should be noted that various parts of the graphical and textual notation may be elided by a modeler, and the tables do not provide guidance on what can be elided and when. In addition, certain model elements have additional keywords and properties that are listed in the Description column of the relevant symbol.

A.3 Package Diagram

Package diagrams are used principally to describe model organization. They are also used to define SysML language extensions called profiles.

Table A.1 Package Diagram Nodes and Paths

Diagram Element	Notation	Description	Section
Package Node		A package is a container for other model elements. Any model element is contained in exactly one container, and when that container is deleted or copied, the contained model element is deleted or copied along with it.	5.3
Model Node		A model in SysML is a top-level package in a nested package hierarchy. In a package hierarchy, models may contain other models, packages, and views.	5.3
Packageable Element Node		Model elements that can be contained in packages are called packageable elements and include blocks, activities, and value types among others.	5.5
View Node		A view is a type of package that conforms to a viewpoint. The view imports a set of model elements according to the viewpoint methods and is expressed in the viewpoint languages to present the relevant information to its stakeholders.	5.9
Viewpoint Node		A viewpoint describes a perspective of interest to a set of stakeholders that is used to specify a view of a model.	5.9
Containment Path		The containment relationship relates parents to children within a package hierarchy.	5.4
Import Path		An import relationship is used to bring an element or collection of elements into a namespace. Private import is marked by the keyword «access».	5.7
Dependency Path		A dependency relationship indicates that a change to the supplier (arrow) end of the dependency may result in a change to the other end of the dependency.	5.8
Conform Path		Used to assert that a view conforms to a viewpoint.	5.9

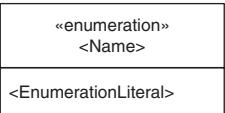
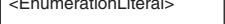
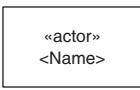
Table A.2 Notation for Describing SysML Extensions on Package Diagrams

Diagram Element	Notation	Description	Section
Metamodel Node		A metamodel describes the concepts in a modeling language, their characteristics and interrelationships.	4.2.2, 14.1.1
Metaclass Node		The individual concepts in a metamodel are described by metaclasses.	4.2.2, 14.1.1
Model Library Node		A model library is a special type of package that is intended to contain a set of reusable model elements for a given domain.	14.2
Stereotype Node		Stereotypes are used to add new language concepts, typically in support of a specific system engineering domain.	14.3
Profile Node		A profile is a kind of package used as the container for set of stereotypes and supporting definitions.	14.4
Generalization Path		A stereotype can be defined by specializing an existing stereotype or stereotypes, using the generalization mechanism.	14.3
Extension Path		The relationship between the metaclass and the stereotype is called an extension, and is a kind of association.	14.3
Association Path		Stereotype properties can be defined using associations.	14.3.1
Reference Path		A reference is special type of import relationship, used to import the metaclasses required by a profile.	14.4.1
Profile Application Path		A profile is applied to a model or package using a profile application relationship.	14.5

A.4 Block Definition Diagram

The block definition diagram is used to define the characteristics of blocks in terms of structural and behavioral features, and the relationships between the blocks, such as their hierarchical relationship. Extensions to the block definition diagram are used to define parametric constraints and also to show a hierarchical view of activities.

Table A.3 Block Definition Diagram Nodes for Representing Block Structure and Values

Diagram Element	Notation	Description	Section
Block Node	<pre>«block» <Name></pre> <pre>parts <Part>:<Block>[<Multiplicity>]</pre> <pre>references <Reference>:<Block>[<Multiplicity>]</pre> <pre>values <ValueProperty>:<ValueType>=<ValueExpression></pre> <pre>operations <Operation>(<Parameter>,...):<Type></pre> <pre>receptions «signal»<Signal>(<Parameter>,...)</pre>	<p>The block is the fundamental modular unit for describing system structure in SysML.</p> <p>Compartments are used to show structural features (parts, references, values) and behavioral features (operations, receptions) of the block. See the following tables in this section for more block compartments.</p> <p>Additional properties on blocks are {encapsulated, abstract}. Abstract may also be indicated by italicizing the <Name>.</p> <p>Additional properties on structural features include: {ordered, unordered, unique, nonunique, subsets <Property>, redefines <Property>}.</p> <p>A forward slash (/) before a property name indicates that it is derived.</p>	6.2, 6.3, 6.5.2
Dimension and Unit Nodes	<pre>«unit» <Name></pre> <pre>dimension =<Dimension></pre> 	A dimension identifies a physical quantity such as length, whose value may be stated in terms of defined units, such as meters or feet. A unit must always be related to a dimension.	6.3.3
Value Type Node	<pre>«valueType» <Name></pre> <pre>values <ValueProperty>:<ValueType>=<ValueExpression></pre> <pre>operations <Operation>(<Parameter>,...):<Type></pre> <pre>dimension=<Dimension> unit=<Unit></pre>	A value type is used to provide a uniform definition of a quantity with units that can be shared by many value properties.	6.3.3
Enumeration Node	 	An enumeration defines a set of named values called literals.	6.3.3
Actor Node	  <Name>	An actor is used represent the role of a human, an organization, or any external system that participates in the use of some system being investigated.	11.3

Blocks have two additional compartments:

- Structure, which has the same symbols as an internal block diagram.
- Namespace, which has the same symbols as a block definition diagram.

Table A.4 Block Definition Diagram Nodes for Representing Interfaces

Diagram Element	Notation	Description	Section
Flow Specification Node	<pre>«flowSpecification» <Name></pre> <pre>flowProperties <Direction> <FlowProperty>:<Item></pre>	A flow specification defines the set of input and/or output flows for a noncomposite flow port. <Direction> may be one of: in, out, or inout.	6.4.3
Interface Node	<pre>«interface» <Name></pre> <pre>operations <Operation>(<Parameters>,...):<Type></pre> <pre>receptions «signal»<Signal>(<Parameter>,...)</pre>	An interface is used to specify the set of behavioral features either required or provided by a standard (service-based) port.	6.5.3
Port Compartments for Block Node	<pre>«block» <Name></pre> <pre>standardPorts <Port>:<Interface></pre> <pre>flowPorts <Direction> <Port>:<Type></pre>	Ports can be shown in separate compartments labeled flow ports and standard ports. <Direction> may be one of: in, out, or inout. Non-atomic flow ports do not have a direction but may have the keyword {conjugated}.	6.4.3, 6.5.2
Nonatomic Flow Port Node	<pre><Name>:<FlowSpecification>[<Multiplicity>] ◻</pre> <pre><Name>:<FlowSpecification>[<Multiplicity>] ◼</pre>	A nonatomic flow port describes an interaction point where multiple different items may flow into or out of a block. A shaded symbol implies a conjugate port.	6.4.3
Atomic Flow Port Node	<pre><Name>:<Item>[<Multiplicity>] →</pre> <pre><Name>:<Item>[<Multiplicity>] ↔</pre> <pre><Name>:<Item>[<Multiplicity>] ←</pre>	An atomic flow port describes an interaction point where an item can flow into or out of a block, or both, as indicated by the direction of the arrow in the Atomic Flow Port Node.	6.4.3
Standard Port Node	<pre><Interface> ○ └─> <Name>[<Multiplicity>]</pre> <pre><Interface> ⊙ └─> <Name>[<Multiplicity>]</pre>	A standard port defines the service-based interaction points on the interface to a block. The shape of the <Interface> symbol indicates whether services are required (socket) by or provided by (ball) the block.	6.5.2
Interface Realization Path	<pre>-----→</pre>	A realization dependency asserts that a block will declare a behavioral feature for each behavioral feature in an interface.	6.5.3
Usage Dependency Path	<pre>-----→</pre>	A uses dependency asserts that a block requires a set of behavioral features defined by an interface.	6.5.3

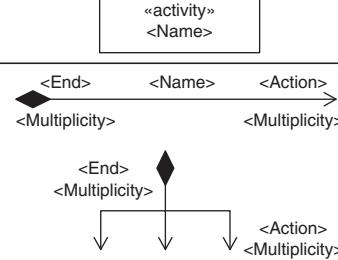
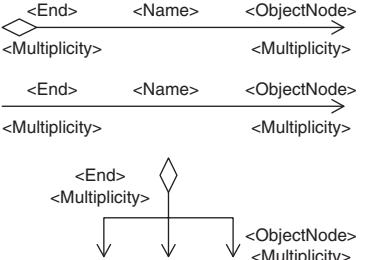
Table A.5 Block Definition Diagram Paths

Diagram Element	Notation	Description	Section
Composite Association Path	<p> $\langle\text{Reference}\rangle \quad \langle\text{Name}\rangle \quad \langle\text{Part}\rangle$ $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$ $\langle\text{End}\rangle \quad \langle\text{Name}\rangle \quad \langle\text{Part}\rangle$ $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$ $\langle\text{Reference}\rangle \quad \langle\text{Multiplicity}\rangle \quad \langle\text{End}\rangle \quad \langle\text{Multiplicity}\rangle$ $\downarrow \qquad \qquad \qquad \downarrow$ </p>	<p>A composite association relates a whole to its parts showing the relative multiplicity at both whole and part ends. A composite association always defines a part property in the whole (indicated by $\langle\text{Part}\rangle$).</p> <p>Where there is no arrow on the nondiamond end of the association it also specifies a reference property to the whole in the part (indicated by $\langle\text{Reference}\rangle$).</p> <p>Otherwise when there is an arrow, the name at the whole end simply gives a name to the association end (indicated by $\langle\text{End}\rangle$).</p>	6.3.1
Reference Association Path	<p> $\langle\text{Reference}\rangle \quad \langle\text{Name}\rangle \quad \langle\text{Reference}\rangle$ $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$ $\langle\text{End}\rangle \quad \langle\text{Name}\rangle \quad \langle\text{Reference}\rangle$ $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$ $\langle\text{Reference}\rangle \quad \langle\text{Name}\rangle \quad \langle\text{Reference}\rangle$ $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$ $\langle\text{End}\rangle \quad \langle\text{Name}\rangle \quad \langle\text{Reference}\rangle$ $\langle\text{Multiplicity}\rangle \qquad \qquad \qquad \langle\text{Multiplicity}\rangle$ $\langle\text{Reference}\rangle \quad \langle\text{Multiplicity}\rangle \quad \langle\text{End}\rangle \quad \langle\text{Multiplicity}\rangle$ $\downarrow \qquad \qquad \qquad \downarrow$ </p>	<p>A reference association can be used to specify a relationship between two blocks. A reference association can specify a reference property on the blocks at one or both ends.</p> <p>The white diamond is the same as no diamond, but profiles can be used to differentiate them by specifying additional constraints.</p>	6.3.2
Association Block Path and Node	<p> $\langle\text{Reference}\rangle \quad \langle\text{Reference}\rangle$ $\langle\text{Multiplicity}\rangle \quad \quad \quad \langle\text{Multiplicity}\rangle$ $\langle\text{Name}\rangle$ $\langle\text{participant}\rangle \{ \text{end} = \langle\text{Reference}\rangle \} \langle\text{Participant}\rangle : \langle\text{Block}\rangle$ </p>	<p>An association block, as the name implies, is a combination of an association and a block, so it can relate two blocks together but can also have internal structure and other features of its own.</p> <p>Participants are placeholders that represent the blocks at each end of the association block, and are used when it is desired to decompose a connector.</p>	6.3.2
Generalization Path	 $\langle\text{GeneralizationSet}\rangle \quad \quad \quad \langle\text{GeneralizationSet}\rangle$ $\dashrightarrow \quad \quad \quad \dashrightarrow$	<p>A generalization describes the relationship between the general classifier and specialized classifier. A set of generalizations may either be {disjoint} or {overlapping}. They may also be {complete} or {incomplete}.</p>	6.6

Table A.6 Additional Notation to Define Parametric Models on Block Definition Diagrams

Diagram Element	Notation	Description	Section
Block Node with Constraint Compartment	<pre>«block» <Name></pre> <pre>constraints {{<Language>}<Constraint>}</pre>	The constraints on a block can be shown in a special compartment labeled constraints. <Constraint> contains an expression preceded by an indication of the language used to express the constraint.	7.2
Constraint Block Node	<pre>«constraint» <Name></pre> <pre>parameters <Parameter>:<Type>[<Multiplicity>]=<ValueExpression></pre> <pre>constraints {{<Language>}<Constraint>} <ConstraintProperty>:<ConstraintBlock>[<Multiplicity>]</pre>	A constraint block encapsulates a constraint to enable it to be defined once and then used in different contexts.	7.3

Table A.7 Additional Notation to Define Activity Models on Block Definition Diagrams

Diagram Element	Notation	Description	Section
Activity Node	<pre>«activity» <Name></pre>	On a block definition diagram, activities are shown using a block symbol with the keyword “activity.”	8.10.1
Activity Composition Path	 <p>The diagram shows a call behavior association between two activity nodes. The top node is a black diamond (end) with multiplicity <End>. The bottom node is a rectangle (name) with multiplicity <Multiplicity>. An arrow points from the top node to the bottom node, labeled <Action> and <Multiplicity>.</p>	Invocation of activities via call behavior actions is modeled using the standard composition association where the calling activity is shown at the black diamond end and the called activity is at the other end of the association.	8.10.1
Object Node Composition Path	 <p>The diagram shows a reference association between an activity node and an object node. There are two paths: one from the activity node to the object node, and another from the object node back to the activity node. Both paths are labeled <End>, <Name>, <ObjectNode>, <Multiplicity>, and <Action> or <ObjectNode>.</p>	Parameters and other object nodes can also be represented on the block definition diagram. By convention, the relationship from activities to object nodes is represented with a reference association.	8.10.2

A.5 Internal Block Diagram

The internal block diagram is used to describe the internal structure of a block in terms of how its parts are interconnected.

Table A.8 Internal Block Diagram Nodes

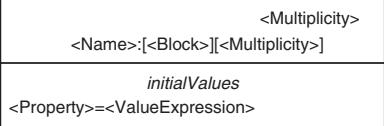
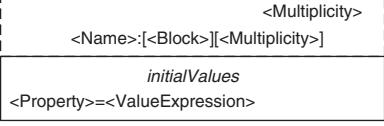
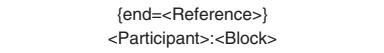
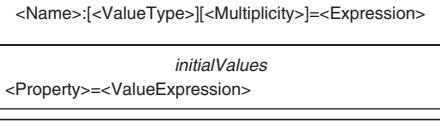
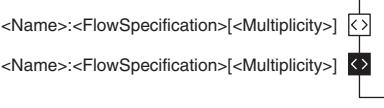
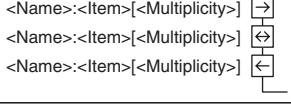
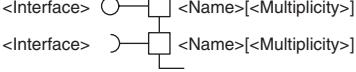
Diagram Element	Notation	Description	Section
Part Node		<p>A part is a property of an owning block that is defined (typed) by another block. The part represents a usage of the defined block in the context of the owning block.</p> <p>Note that a Part Node may have the same compartments as a Block Node. [<Block>] represents a property-specific type.</p>	6.3.1, 6.6.5
Actor Part Node	 <Name>:<Actor>[<Multiplicity>]	An actor part is a property of an owning block that is defined (typed) by an actor.	11.5
Reference Node		<p>A reference property of a block is a reference to another block.</p> <p>Note that a Reference Property Node may have the same compartments as a Block Node. [<Block>] represents a property-specific type.</p>	6.3.2
Participant Property Node		<p>A participant property represents one end of an association block. Using a participant property, a modeler can show the relationship between the internal structure of the association block and the internal structure of its related ends.</p>	6.3.2
Value Property Node		<p>A value property describes the quantitative characteristics of a block.</p> <p>Note that a Value Property Node may have the same compartments as a Value Type Node. [<ValueType>] represents a property-specific type.</p>	6.3.3
Nonatomic Flow Port Node		<p>A nonatomic flow port describes an interaction point that allows multiple different items to flow into or out of a block. A nonatomic flow port is typed by a flow specification. A shaded symbol implies a conjugate port that reverses the items' allowable in and out flow direction.</p>	6.4.3
Atomic Flow Port Node		<p>An atomic flow port describes an interaction point where an item can flow into or out of a block, or both, as indicated by the direction of the arrow in the Atomic Flow Port Node.</p>	6.4.3
Standard Port Node		<p>A standard port describes a service-based interaction point on a block. A standard port is defined by its interface. The shape of the <Interface> symbol indicates whether services are required by or provided by the block.</p>	6.5.3

Table A.9 Internal Block Diagram Paths

Diagram Element	Notation	Description	Section
Connector Path	<pre><End> <Name>:<Association> <End> <Multiplicity> <Multiplicity> <End> <Name>:<Association> <End> <Multiplicity> <Multiplicity></pre>	A connector is used to bind two parts (or ports) and provides the opportunity for those parts to interact, although the connector says nothing about the nature of the interaction.	6.3.1
Connector Property Path and Node	<pre><End> <End> <Multiplicity> <Multiplicity> <Name>:<Association></pre>	More detail can be specified for connectors by typing them with association blocks. An association block, as the name implies, is a combination of an association and a block, so it can relate two blocks together but can also have internal structure and other features of its own.	6.3.2
Item Flow Node	<pre><Name>:<Item>, ... <Name>:<Item>, ...</pre>	An item flow is used to specify the items that flow across a connector in a particular context. An item flow specifies the type of the item that is flowing and the direction of flow. It may also be associated to a property, called an item property, of the enclosing block to identify a specific usage of an item in the context of the enclosing block.	6.4.2

A.6 Parametric Diagram

Parametric diagrams are used to create systems of equations that can be used to constrain the properties of blocks.

Table A.10 Parametric Diagram Notation

Diagram Element	Notation	Description	Section
Constraint Parameter Node	<pre><Name>:<Type>[<Multiplicity>]</pre>	A constraint parameter is a special kind of property that is used in the constraint expression of a constraint block. Constraint parameters do not have direction.	7.3
Constraint Property Node	<pre><Name>:<ConstraintBlock>[<Multiplicity>] {<Constraint>}</pre> <pre>“constraint” <Name>:<ConstraintBlock>[<Multiplicity>] {<Constraint>}</pre>	Constraint properties are defined by constraint blocks and used to bind (i.e., connect) parameters. This enables complex systems of equations to be composed from more primitive equations, and for the parameters of the equations to explicitly constrain properties of blocks.	7.4
Value Binding Path	<pre><Multiplicity> <Multiplicity> <Multiplicity> “equal” <Multiplicity></pre>	Binding connectors connect constraint parameters to each other and to value properties. They express an equality relationship between their bound elements.	7.5

A.7 Activity Diagram

The activity diagram is used to model behavior in terms of the flow of inputs, outputs, and control. An activity diagram is similar to a traditional functional flow diagram.

Table A.11 Activity Diagram Structural Nodes

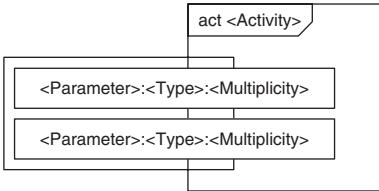
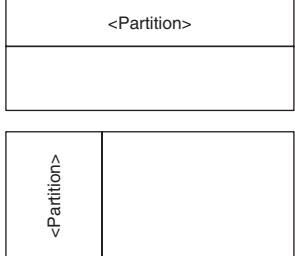
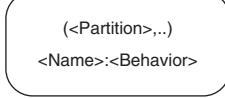
Diagram Element	Notation	Description	Section
Activity Parameter Node		<p>Activity parameter node symbols are rectangles that straddle the boundary of the activity frame.</p> <p>Other annotations include: «noBuffer», «optional», «overwrite», «continuous», «discrete», {rate=<Expression>}.</p> <p>Parameters can be organized into parameter sets, indicated by a bounding box around the parameters in the set. Parameter sets may overlap, and may have an annotation: {probability=<Expression>}.</p>	8.4.1
Interruptible Region Node		An interruptible region groups a subset of the actions within an activity and includes a mechanism for stopping their execution. Stopping the execution of these actions does not effect other actions in the activity.	8.8.1
Activity Partition Node		<p>A set of activity nodes can be grouped into an activity partition (also known as a swimlane) that is used to indicate responsibility for execution of those nodes. <Partition> may be the name of a block or name and type of a part/reference. Partitions may overlap in a grid pattern.</p>	8.9.1
Activity Partition in Action Node		An alternative representation for an activity partition for call actions is to include the name of the partition or partitions in parentheses inside the node above the action name. This can make the activity easier to layout than when using the swimlane notation.	8.9.1

Table A.12 Activity Diagram Control Nodes

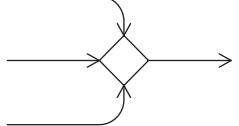
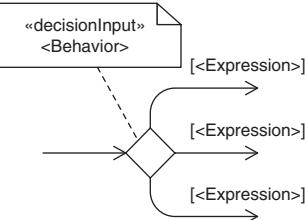
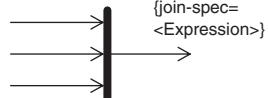
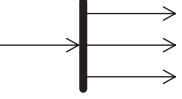
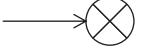
Diagram Element	Notation	Description	Section
Merge Node		A merge node has one output flow and multiple input flows—it routes each input token received on any input flow to its output flow. Unlike a join node, a merge node does not require tokens on all its input flows before offering them on its output flow. Rather it offers tokens on its output flow as soon as it receives them.	8.5.1, 8.6.1
Decision Node		A decision node has one input flow and multiple output flows—an input token can only traverse one output flow. The output flow is typically established by placing mutually exclusive guards on all outgoing flows and offering the token to the flow whose guard expression is satisfied. A decision node can have an accompanying decision input behavior, which is used to evaluate each incoming object token and whose result can be used in guard expressions.	8.5.1, 8.6.1
Join Node		A join node has one output flow and multiple input flows—it has the important characteristic of synchronizing the flow of tokens from many sources. Its default behavior can be overridden by providing a join specification, which can specify additional control logic.	8.5.1, 8.6.1
Fork Node		A fork node has one input flow and multiple output flows—it replicates every input token it receives onto each of its output flows. The tokens on each output flow may be handled independently and concurrently.	8.5.1, 8.6.1
Initial Node		When an activity starts executing a control token is placed on each initial node in the activity. The token can then trigger the execution of an action via an outgoing control flow.	8.6.1
Activity Final Node		When a control or object token reaches an activity final node during the execution of an activity, the execution terminates.	8.6.1
Flow Final Node		Control or object tokens received at a flow final node are consumed but have no effect on the execution of the enclosing activity. Typically they are used to terminate a particular sequence of actions without terminating an activity.	8.6.1

Table A.13 Activity Diagram Object and Action Nodes

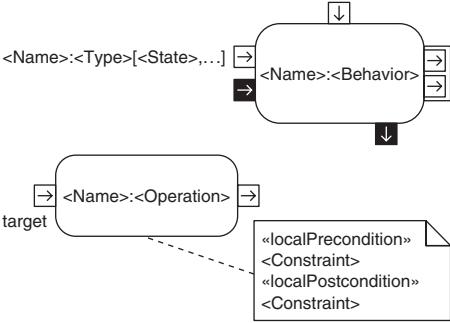
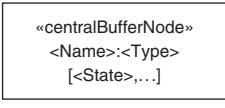
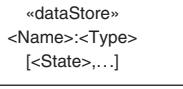
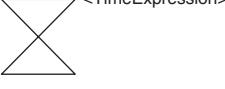
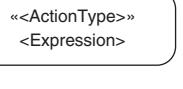
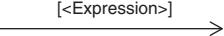
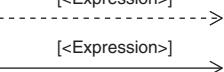
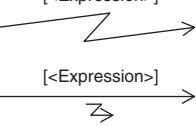
Diagram Element	Notation	Description	Section
Call Action Node	 <p><Name>:<Type>[<State>,...]</p> <p><Name>:<Behavior></p> <p>target</p> <p>«localPrecondition» «Constraint» «localPostcondition» «Constraint»</p>	<p>Call actions can invoke other behaviors either directly or through an operation, and are referred to as call behavior actions and call operation actions, respectively. A call action must own a set of pins that match in number and type of the parameters of the invoked behavior/operation. A called operation requires a target.</p> <p>Streaming pins may be marked as {stream} or filled (as shown).</p> <p>Where the parameters of the called entity are grouped into sets, the corresponding pins are as well. Pre- and postconditions can be specified that constrain the action such that it cannot begin to execute unless the precondition is satisfied, and must satisfy the postcondition to successfully complete execution.</p>	8.1, 8.3, 8.4.2
Central Buffer Node	 <p>«centralBufferNode»</p> <p><Name>:<Type></p> <p>[<State>,...]</p>	A central buffer node provides a store for object tokens outside of pins and parameter nodes. Tokens flow into a central buffer node and are stored there until they flow out again.	8.5.3
Datastore Node	 <p>«dataStore»</p> <p><Name>:<Type></p> <p>[<State>,...]</p>	A datastore node provides a copy of a stored token rather than the original. When an input token represents an object that is already in the store, it overwrites the previous token.	8.5.3
Control Operator Action Node	 <p>«controlOperator»</p> <p><Name>:<ControlOperator></p> <p>(control)</p>	A control operator produces control values on an output parameter, and is able to accept a control value on an input parameter (treated as an object token). It is used to specify logic for enabling and disabling other actions.	8.6.2
Accept Event Action Node	 <p><Event>,...</p>	An activity can accept events using an accept event action. The action has (sometimes hidden) output pins for received data.	8.7
Accept Time Event Node	 <p><TimeExpression></p>	A time event corresponds to an expiration of an (implicit) timer. In this case the action has a single (typically hidden) output pin that outputs a token containing the time of the accepted event occurrence.	8.7
Send Signal Action	 <p>signal [pin]</p> <p>target [pin]</p> <p><Signal></p>	An activity can send signals using a send signal action. It typically has pins corresponding to the signal data to be sent and the target for the signal.	8.7
Primitive Action Node	 <p>«<ActionType>»</p> <p><Expression></p>	Primitive actions include: object access/update/manipulation actions, which involve properties and variables, and value actions, which allow the specification of values. The <Expression> will depend on the nature of the action.	8.12.1

Table A.14 Activity Diagram Paths

Diagram Element	Notation	Description	Section
Object Flow Path		Object flows connect inputs and outputs. Additional annotations include «continuous», «discrete», {rate=<Expression>}, {probability=<Expression>}.	8.1, 8.5
Control Flow Path		Control flows provide constraints on when, and in what order, the actions within an activity will execute. A control flow can be represented using a solid line, or using a dashed line to more clearly distinguish it from object flow.	8.1, 8.6
Object Flow Node		When an object flow is between two pins that have the same characteristics, an alternative notation can be used where the pin symbols are elided and replaced by a single rectangular symbol called an object node symbol.	8.5
Interrupting Edge Path		An interrupting edge interrupts the execution of the actions in an interruptible region. Its source is a node inside the region and its destination is a node outside it.	8.8.1

A.8 Sequence Diagram

The sequence diagram is used to represent the interaction between structural elements of a block, as a sequence of message exchanges.

Table A.15 Sequence Diagram Structural Nodes

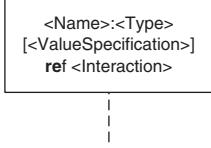
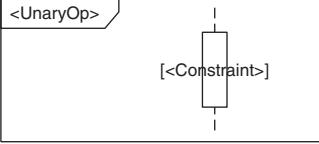
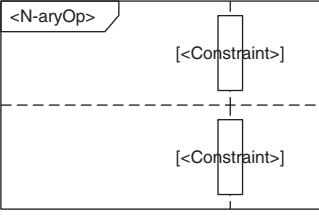
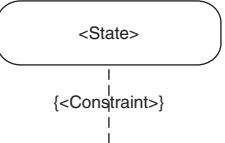
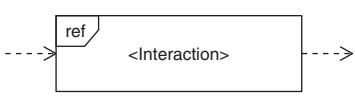
Diagram Element	Notation	Description	Section
Lifeline Node		A lifeline represents the relevant lifetime of an instance that is part of the interaction's owning block, which will either be represented by a part property or a reference property.	9.4
Single-compartment Fragment Node		A combined fragment can be used to model complex sequences of messages. A number of combined fragments have operators with only a single compartment for all operands, shown as <UnaryOp>. These are: seq, opt, break, strict, loop, neg, assert, critical.	9.7.1, 9.7.2
Multi-compartment Fragment Node		<p>Two combined fragments have operators with a compartment per operand, shown as <N-aryOp>. These are par and alt.</p> <p>The lifelines that participate in the fragment overlay on top of the fragment (i.e., are visible) and lifelines that don't participate are obscured behind the fragment. (<i>Note:</i> This is also true of Single-Compartment Fragment Nodes.)</p>	9.7.1
Filtering Fragment Node		There are two combined fragments with filter operators: consider and ignore, shown as <FilterOp>. Inside such a construct, messages that have been explicitly ignored (or not considered) may be interleaved with valid traces.	9.7.2
State Invariant Symbol		A state invariant on a lifeline is used to add a constraint on the required state of a lifeline at a given point in a sequence of event occurrences. The invariant constraint can include the values of properties or parameters, or the state of a state machine.	9.7.3
Interaction Use Node		An interaction use allows one interaction to reference another as part of its definition. The lifelines that participate in the interaction are obscured behind the fragment, and lifelines that don't participate overlay on top of the fragment (i.e., are visible).	9.8

Table A.16 Sequence Diagram Paths and Activation Nodes

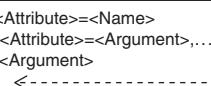
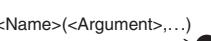
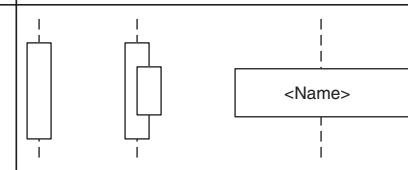
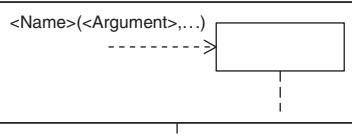
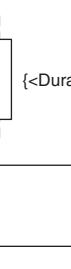
Diagram Element	Notation	Description	Section
Synchronous Message		A synchronous message corresponds to the synchronous invocation of an operation, and is generally accompanied by a reply message.	9.5.1
Asynchronous Message		Asynchronous messages correspond to either the sending of a signal or to an asynchronous invocation (or call) of an operation, and do not require a reply message.	9.5.1
Reply Message		A reply message shows a reply to a synchronous operation call, together with any return arguments.	9.5.1
Found Message Path		A lost message describes the case where there is sending event for the message but no receiving event.	9.5.2
Lost Message Path		A found message describes the case where there is receiving event for the message but no sending event.	9.5.2
Focus of Control (Activation) Node		Focus of control bars or activations are overlaid on lifelines and correspond to executions; they begin at the execution's start event, and end at the execution's end event. When executions are nested, the focus of control bars are stacked from left to right. An alternate notation for activations is a box symbol overlaid on the lifeline with the name of the behavior or action inside.	9.5.4
Create Message Path		The creation of an instance is indicated by the receipt of a create message.	9.5.5
Destroy Event Node		An instance's destruction is indicated by the occurrence of a destroy event.	9.5.5
Coregion Symbol		Within a coregion, there is no implied order between any messages sent or received by the lifeline.	9.7.1

Table A.17 Sequence Diagram Temporal Observation and Constraint Nodes

Diagram Element	Notation	Description	Section
Duration Observation Symbol	 $\langle\text{Name}\rangle=\langle\text{DurationExpression}\rangle$	A duration observation can be used to note the time taken between two instants that represent the occurrence of events during the execution of an interaction.	9.6
Duration Constraint Symbol	 $\{\langle\text{DurationConstraint}\rangle\}$	A duration constraint identifies two events, called the start and end events, and expresses a constraint on the duration between them. A duration constraint can use a duration observation in its definition.	9.6
Time Observation Symbol	 $\langle\text{Name}\rangle=\langle\text{TimeExpression}\rangle$	A time observation is used to note the time at some instant during the execution of an interaction.	9.6
Time Constraint Symbol	 $\{\langle\text{TimeConstraint}\rangle\}$	A time constraint identifies a constraint that applies to the time of occurrence of a single event in the interaction execution. A time constraint can use a time observation in its definition.	9.6

A.9 State Machine Diagram

State machine diagram are used in SysML to describe the state-dependent behavior of a block throughout its life cycle in terms of its states and the transitions between them.

Table A.18 State Machine Diagram State Nodes

Diagram Element	Notation	Description	Section
State Machine with Entry- and Exit-Point Pseudostate Nodes	<p>The notation shows a rectangular state machine node labeled 'stm <StateMachine>' containing two circular pseudostates. One is a standard entry-point pseudostate (circle with a cross) and the other is an exit-point pseudostate (circle with a double circle). Labels '<Name>' are placed near each pseudostate.</p>	A state machine may have entry- and exit-point pseudostates, which are similar to junctions. On state machines, entry-point pseudostates can only have outgoing transitions and exit-point pseudostates can only have incoming transitions.	10.6.5
Atomic State Node	<p>The notation shows a rounded rectangular state node labeled '<State>'. Inside, there is a list of behaviors: 'Entry/<Behavior>', 'Exit/<Behavior>', 'Do/<Behavior>', '<Event>-[<Constraint>]/<Behavior>', and '<Event>/defer'.</p>	A state represents some significant condition in the life of a block, typically because it represents some change in how the block responds to events. 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 activity that executes once the entry behavior has completed and continues to execute until it completes or the state is exited.	10.3
Composite State with Entry- and Exit-Point Pseudostate Nodes	<p>The notation shows a large rounded rectangular state node labeled '<State>'. It contains two smaller regions, each with its own entry and exit points. Labels '<Name>' are at the boundaries.</p>	A composite state is a state with nested regions; the most common case is a single region. A composite state may have entry- and exit-point pseudostates that act like junction pseudostates. Entry points have incoming transitions from outside the state and exit points have the opposite.	10.6.1
Composite State Node with Multiple Regions	<p>The notation shows a large rounded rectangular state node labeled '<State>'. It contains two separate regions, each with its own entry and exit points. The regions are connected by dashed lines, indicating they are orthogonal. Labels '<Name>' are at the boundaries.</p>	A composite state may have many regions, which may each contain substates. These regions are orthogonal to each other and so a composite state with more than one region is sometimes called an orthogonal composite state.	10.6.2
Sub-State Machine Node with Connection Points	<p>The notation shows a rounded rectangular state node labeled '<State><StateMachine>'. It contains a transition ending on a submachine state, which then starts a referenced state machine. Labels '<Name>' are at the boundaries.</p>	A state machine may be reused using a kind of state called a submachine state. A transition ending on a submachine state will start its referenced state machine. Transitions may also be connected to connection points on the boundary of the state.	10.6.5

Table A.19 State Machine Diagram Pseudostate and Transition Nodes

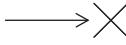
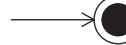
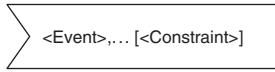
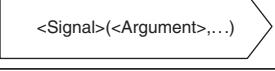
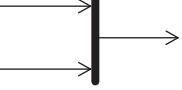
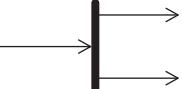
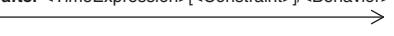
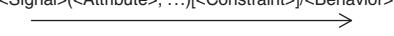
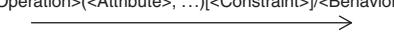
Diagram Element	Notation	Description	Section
Terminate Pseudostate Node		If a terminate pseudostate is reached, then the behavior of the state machine terminates.	10.3
Initial Pseudostate Node		An initial pseudostate specifies the initial state of a region.	10.3
Final State Node		The final state indicates that a region has completed execution.	10.3
Choice Pseudostate Node		The outgoing transitions of a choice pseudostate are evaluated once it has been reached.	10.4.2
Junction Pseudostate Node		A junction pseudostate is used to construct a compound transition path between states.	10.4.2
Trigger Node		This node represents all the transition's triggers, with the descriptions of the triggering events and the transition guard inside the symbol.	10.4.3
Action Node		<EffectExpression> describes the effect of the transition, either the name of a behavior or the body of an opaque behavior.	10.4.3
Send Signal Node		This node represents a send signal action. The signal's name, together with any arguments that are being sent, are shown within the symbol.	10.4.3
Join Pseudostate Node		A join pseudostate has a single outgoing transition and many incoming transitions. When all of the incoming transitions can be taken, and the join's outgoing transition is valid, then all the transitions happen.	10.6.2
Fork Pseudostate Node		A fork pseudostate has a single incoming transition and many outgoing transitions. When an incoming transition is taken to the fork pseudostate, all of the outgoing transitions are taken.	10.6.2
History Pseudostate Node		A history pseudostate represents the last state of its owning region, and a transition ending on a history pseudostate has the effect of returning the region to the state it was last in.	10.6.4

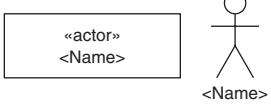
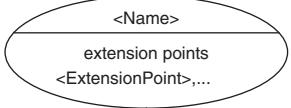
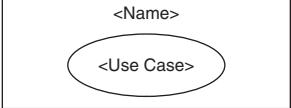
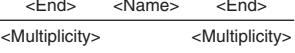
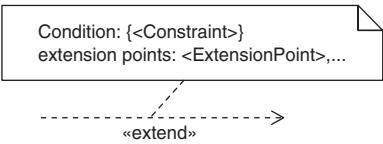
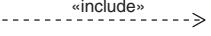
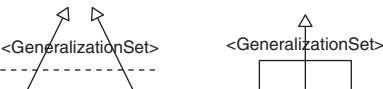
Table A.20 State Machine Diagram Paths

Diagram Element	Notation	Description	Section
Time Event Transition Path	after <TimeExpression>[<Constraint>]<Behavior>  at <TimeExpression>[<Constraint>]<Behavior> 	Time events indicate either that a given time interval has passed since the current state was entered (after), or that a given instant of time has been reached (at). The transition can also include a guard and effect.	10.4.1
Signal Event Transition Path	<Signal>(<Attribute>, ...)[<Constraint>]<Behavior> 	Signal events indicate that a new asynchronous message has arrived. A signal event may be accompanied by a number of arguments, which may be assigned to attributes. The transition can also include a guard and effect.	10.4.1
Call Event Transition Path	<Operation>(<Attribute>, ...)[<Constraint>]<Behavior> 	Call events indicate that an operation on the state machine's owning block has been requested. A call event may also be accompanied by a number of arguments, which may be assigned to attributes. The transition can also include a guard and effect.	10.5
Change Event Transition Path	when <Expression>[<Constraint>]<Behavior> 	Change events indicate that some condition has been satisfied (normally that some specific set of attribute values hold). The transition can also include a guard and behavior/effect.	10.7

A.10 Use Case Diagram

The use case diagram is used to model the relationships between the system under consideration or subject, its actors, and use cases.

Table A.21 Use Case Diagram Notation

Diagram Element	Notation	Description	Section
Actor Node		The users and other external participants in an interaction with a subject are described by actors. An actor represents the role of a human, an organization, or any external system that participates in the use of some subject. Actors may interact directly with the subject or indirectly with the system through other actors.	11.1, 11.3
Use Case Node		Use cases describe the functionality of some system in terms of how its users use that system to achieve their goals. A use case may define a set of extension points, that represent places where it can be extended.	11.1, 11.4
Subject Node		The entity that provides functionality in support of the use cases is called the system under consideration, or subject, and is represented by a rectangle. It often represents a system that is being developed.	11.4
Association Path		Actors are related to use cases by associations. The multiplicity at the actor end describes the number of actors involved, and the multiplicity at the use case end describes the number of instances in which the actor or actors can be involved.	11.4
Extension Path		The extending use case is a fragment of functionality that extends the base use case and is not considered part of the normal base use case functionality. It often describes some exceptional behavior in the interaction between subject and actors, such as error handling, which does not contribute directly to the goal of the base use case. The arrow end of the extension relationship points to the base use case that is extended.	11.4.1
Inclusion Path		The inclusion relationship allows a base use case to include the functionality of an included use case as part of its functionality. The included use case is always performed when the base use case is performed. The arrow end of the include relationship points to the included use case.	11.4.1
Generalization Path		Use cases and actors can be classified using the generalization relationships. Scenarios and actor associations from the general use case are inherited by the specialized use case.	11.4.1

A.11 Requirement Diagram

The requirement diagram is used to graphically depict hierarchies of requirements or to depict an individual requirement and its relationship to other model elements.

Table A.22 Requirement Diagram Nodes

Diagram Element	Notation	Description	Section
Requirement Node	<pre>«requirement» <Name></pre> <pre>text = "<String>" id = "<String>"</pre> <pre>satisfiedBy «<ElementType>><Element></pre> <pre>derived «requirement»<Requirement></pre> <pre>derivedfrom «requirement»<Requirement></pre> <pre>refinedBy «ElementType»<Element></pre> <pre>master «requirement»<Requirement></pre> <pre>verifiedBy «<testCase>><TestCase></pre>	A requirement specifies a capability or condition that must (or should) be satisfied, a function that a system must perform, or a performance condition a system must achieve. Each requirement includes predefined properties for its identification and textual description. SysML includes specific relationships to relate requirements to other requirements as well as to other model elements, which include deriving requirements, satisfying requirements, verifying requirements, refining requirements, and copying requirements. The compartment notation is one method for displaying a requirement relationship between a requirement and another model element.	12.1, 12.3, 12.4, 12.5.2
Requirement Related-Type Node	<pre>«<ElementType>> <Name></pre> <pre>refines «requirement»<Requirement></pre> <pre>satisfies «requirement»<Requirement></pre>	Requirements can be related to model elements that may appear in different hierarchies or on different diagrams. These relationships can be shown using the compartment notation when the requirements and related model elements do not appear on the same diagram.	12.5, 12.11, 12.13
Trace Compartment	<pre>tracedTo «ElementType»<Element></pre> <pre>tracedFrom «ElementType»<Element></pre>	The trace relationship can be shown using compartment notation when the requirements and related model elements do not appear on the same diagram.	12.5, 12.14
Package Node	<pre><Name></pre> <pre><Name></pre>	Requirements can be organized into a package structure. Each package within this package structure may correspond to a different specification, each containing the text-based requirements for that specification.	12.8
Test Case Node	<pre>«testCase» <Name></pre> <pre>verifies «requirement»<Requirement></pre>	A test case can represent any method for performing the verification, including the standard verification methods of inspection, analysis, demonstration, and testing.	12.12

Table A.23 Requirement Diagram Paths

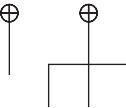
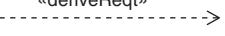
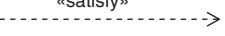
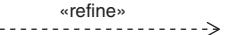
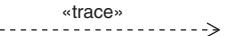
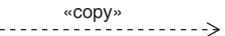
Diagram Element	Notation	Description	Section
Containment Path		The containment relationship is used to represent how requirements are contained in specifications (packages), or how a complex requirement can be partitioned into a set of simpler requirements without adding or changing their meaning.	12.9
Derivation Path		A derive relationship occurs between a source requirement and a derived requirement, based on analysis of the source requirement.	12.10
Satisfaction Path		A satisfy relationship is used to assert that a model element corresponding to the design or implementation satisfies a particular requirement.	12.11
Verification Path		A verify relationship is used between a requirement and a test case or other named element to indicate how to verify that the requirement is satisfied.	12.12
Refinement Path		The refine relationship is used to reduce ambiguity in a requirement by relating it to another model element that clarifies the requirement.	12.13
Trace Path		A trace relationship is a general-purpose way to relate a requirement and any other model element, useful for relating requirements to documents, etc.	12.14
Copy Path		The copy relationship relates a copy of a requirement to the original requirement, to support reuse of requirements.	12.14.1

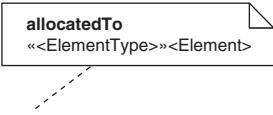
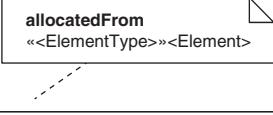
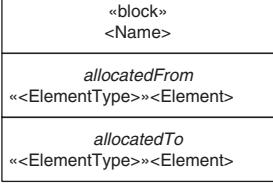
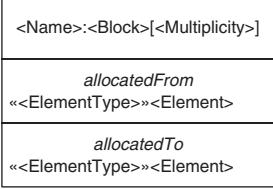
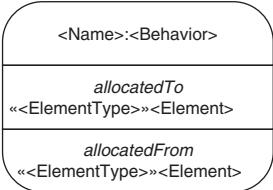
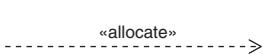
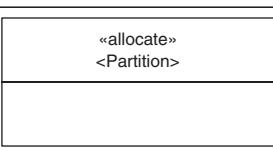
Table A.24 Requirement Diagram Callouts

Diagram Element	Notation	Description	Section
Trace Callout		This callout notation is an alternative notation for depicting trace relationships. It is the least restrictive notation in that it can be used to represent a relationship between any requirement and any other model element on any diagram type.	12.5.3, 12.14
Derivation Callout		This callout notation is an alternative notation for depicting derive relationships.	12.5.3, 12.11
Verification Callout		This callout notation is an alternative notation for depicting verify relationships.	12.5.3, 12.12
Satisfaction Callout		This callout notation is an alternative notation for depicting satisfy relationships.	12.5.3, 12.11
Refinement Callout		This callout notation is an alternative notation for depicting refine relationships.	12.5.3, 12.13
Master Requirement Callout		This callout notation is an alternative notation for depicting copy relationships.	12.5.3, 12.14.1
Rationale Callout		A rationale is typically associated with either a requirement, or a relationship between requirements. It can also be applied throughout the model to capture the reason for any type of decision.	12.6
Problem Callout		A problem is a particular kind of comment used to identify or flag design issues in the model.	12.6

A.12 Allocation

SysML includes several notational options to provide flexibility for representing allocations of model elements across the system model. The graphical representations are similar to those used for relating requirements to other model elements.

Table A.25 Notation for Allocations

Diagram Element	Notation	Description	Section
Allocated To Callout		The callout notation can be used to represent the opposite end of the allocation relationship for any model element. In this case the callout box is anchored to an element that is allocated to the element name in the callout box.	13.3
Allocated From Callout		The callout notation can be used to represent the opposite end of the allocation relationship for any model element. In this case the callout box is anchored to an element that is allocated from the element name in the callout box.	13.3
Block Node with Allocation Compartments		The compartment notation identifies the element at the opposite end of the allocation relationship in a compartment of the model element. When used on a block, it explicitly indicates allocation of definition to/from the block.	13.3
Part Node with Allocation Compartments		The compartment notation identifies the element at the opposite end of the allocation relationship in a compartment of the model element. When used on a part, it explicitly indicates allocation of usage to/from the part. An inferred allocation (part typed by a block, which in turn has an activity allocated to it) should not be depicted by a compartment on the part.	13.3
Call Action Node with Allocated To Compartment		When an allocation compartment is used on an action, it explicitly indicates allocation of usage to/from the action. An inferred allocation (action typed by an activity, which in turn is allocated to a block) should not be depicted by a compartment on the action.	13.3
Allocation Path		This allocation relationship can be depicted directly when both ends of the allocation relationship are shown on the same diagram. The arrowhead represents the "allocatedTo" end.	13.3
Allocate Activity Partition Node		The presence of an allocate activity partition on an activity diagram implies an allocate relationship between any action node within the partition and the part represented by the partition. This provides allocation of usage (action to part), but not allocation of definition (activity to block). The alternative activity partition notation (Activity Partition in Action Node in Table A.11) can also be used.	13.7

A.13 Stereotypes

Stereotypes may be applied to elements on any diagram, and SysML has a generic notation across all diagrams. Information about applied stereotypes can be shown either inside node symbols, as part of name strings, or using callout notation.

Table A.26 Notation for Stereotyped Elements

Diagram Element	Notation	Description	Section
Name Compartment with Keywords and Properties	<pre>«<Stereotype>» {<Property>=<Value>,...} <Name></pre>	A stereotyped model element is shown with the name of the stereotype in guillemets, followed by any values for the stereotypes properties and then the name of the model element. Multiple stereotypes and their properties may be shown before the model element name.	14.6
Name Compartment with Keywords	<pre>«<Stereotype>,...» <Name></pre>	If no stereotype properties are shown, then multiple stereotype names can appear in a comma-separated list within one set of guillemets.	14.6
Name String with Keywords and Properties	<pre>«<Stereotype>»{<Property>=<Value>,...}<Name> label <Stereotype>»{<Property>=<Value>,...}<Name> ...</pre>	If the model element is represented by path symbol (e.g., a line), the stereotype name and properties are shown in a label next to the line and before the name of the element. Stereotype keywords and properties can also be shown for elements in compartments, when they are shown before the element name.	14.6
Stereotype Callout	<pre>«<Stereotype>» <Property>=<Value> ...</pre>	Irrespective of the symbol representing a model element, the values for applied stereotypes properties can always be shown using callout notation. Property values from multiple stereotypes can be shown in a single note symbol.	14.6
Node with Stereotype Compartment	<pre>«<Stereotype>,...» <Name></pre> <pre>«<Stereotype>» <Property>=<Value> ...</pre>	Where a symbol supports compartments, the values for the properties of an applied stereotype can be shown in a compartment specific to that stereotype.	14.6

References

- [1] Object Management Group, *OMG Systems Modeling Language (OMG SysML™)*, V1.0, OMG document number formal/2007-09-01, September 2007; available at <http://www.omg.sysml.org>.
- [2] ANSI/EIA 632, *Processes for Engineering a System*, American National Standards Institute/Electronic Industries Alliance, 1999.
- [3] IEEE Standard 1220-1998, *IEEE Standard for Application and Management of the Systems Engineering Process*, Institute for Electrical and Electronic Engineers, December 8, 1998.
- [4] ISO/IEC 15288:2002, *Systems Engineering—System Life Cycle Processes*, International Organization for Standardization/International Electrotechnical Commission, November 15, 2003.
- [5] Estefan, Jeff, *Survey of Candidate Model-Based Systems Engineering (MBSE) Methodologies*, Rev. A, May 25, 2007.
- [6] Douglass, Bruce P., *The Harmony Process*, I-Logix Inc. white paper, March 25, 2005.
- [7] Hoffmann, Hans-Peter, *HarmonySE/SysML Deskbook: Model-Based Systems Engineering with Rhapsody*, Rev. 1.51, Telelogic/I-Logix white paper, Telelogic AB, May 24, 2006.
- [8] Lykins, Friedenthal, Meilich, Adapting UML for an Object-Oriented Systems Engineering Method (OOSEM), *Proceedings of the INCOSE International Symposium*. Minneapolis, July 15–20, 2000.
- [9] Cantor, Murray, RUP SE: The Rational Unified Process for Systems Engineering, *The Rational Edge*, Rational Software, November 2001.
- [10] Cantor, Murray, *Rational Unified Process® for Systems Engineering*, RUP SE Version 2.0, IBM Rational Software white paper, IBM Corporation, May 8, 2003.
- [11] Ingham, Michel D., Rasmussen, Robert D., Bennett, Matthew B., and Moncada, Alex C., Generating Requirements for Complex Embedded Systems Using State Analysis, *Acta Astronautica*, 58(12):648–661, June 2006.
- [12] Long, James E., *Systems Engineering (SE) 101, CORE®: Product & Process Engineering Solutions*, Vitech training materials, Vitech Corporation, Vienna, VA, 2000.
- [13] Zachman, John A., A Framework for Information Systems Architecture, *IBM Systems Journal*, 26(3):276–292, 1987.
- [14] C4I Architecture Working Group, *C4ISR Architecture Framework Version 2.0*, December 18, 1997.
- [15] U.S. Department of Defense, DoD Architecture Framework (DoDAF), Version 1.5, April 23, 2007; available at http://www.defenselink.mil/cio-nii/docs/DoDAF_Volume_II.pdf.
- [16] Ministry of Defence, *Architecture Framework (MODAF)*, Version 1.1, June 4, 2007.
- [17] ANSI/IEEE Std. 1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, American National Standards Institute/Institute for Electrical and Electronic Engineers, September 21, 2000.
- [18] The Open Group, *The Open Group Architecture Framework (TOGAF)*, Version 8.1.1, Enterprise Edition. New York: VanHaren, 2007; available at <http://www.opengroup.org/bookstore/catalog/g063v.htm>.
- [19] *Standard for Integration Definition for Function Modeling (IDEF0)*, Draft Federal Information Processing Standards, Publication 183, December 21, 1993.
- [20] Object Management Group, *Meta Object Facility Core Specification*, Version 2.0, OMG document number formal/06-01-01, January 2006.

- [21] Object Management Group, *MOF 2.0/XMI Mapping XMI Metadata Interchange Specification*, Version 2.1.1, OMG document number formal/2007-12-10, December 2007.
- [22] ISO TC-184 (Technical Committee on Industrial Automation Systems and Integration), SC4 (Subcommittee on Industrial Data Standards), *ISO 10303-233 STEP AP233*; available at <http://www.ap233.org/ap233-public-information>.
- [23] Object Management Group, *Model-Driven Architecture (MDA) Guide*, v1.01, June 12, 2003; available at <http://www.omg.org/mda/>.
- [24] Wymore, W., *Model-Based Systems Engineering*. Boca Raton, FL: CRC Press, 1993.
- [25] International Council on Systems Engineering (INCOSE), *Systems Engineering Vision 2020*, Version 2.03, TP-2004-004-02, September 2007.
- [26] Object Management Group, *Object Constraint Language (OCL)*, Version 2.0, OMG document number formal/06-05-01, May 2006.
- [27] Object Management Group, *UML for Systems Engineering RFP*, OMG document number ad/03-03-41, March 28, 2003.
- [28] Object Management Group, *Unified Modeling Language (OMG UML): Superstructure, v. 2.1.2*, OMG Available Specification, OMG document formal/2007-11-02, November 2007.
- [29] Object Management Group, *OMG SysML™ Requirements Traceability Matrix*, OMG document number ptc/2007-03-09, March 2007.
- [30] Haskins, Cecilia (ed.), *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, v. 3.1, INCOSE-TP-2003-002-03.1, International Council on Systems Engineering, August 2007.
- [31] Reisig, Wolfgang, *A Primer in Petri Net Design*. New York: Springer-Verlag.
- [32] Wagenhals, Haider, and Levis, Synthesizing Executable Models of Object Oriented Architectures, *Journal of International Council of Systems Engineering*, 6(4):266-300, 2003.
- [33] Bock, Conrad, SysML and UML 2.0 Support for Activity Modeling, *Journal of International Council of Systems Engineering*, 9(2):160-186, 2006.
- [34] Cockburn, Alistair, *Writing Effective Use Cases*. Boston: Addison-Wesley, 2000.
- [35] Object Management Group, *UML Testing Profile*, v. 1.0, OMG document number formal/05-07-07, July 2005.
- [36] Friedenthal, Sanford, Object Oriented Systems Engineering, in *Process Integration for 2000 and Beyond: Systems Engineering and Software Symposium*. New Orleans: Lockheed Martin Corporation, May 1998.
- [37] Meilich, Abe, and Rickels, Michael, An Application of Object-Oriented Systems Engineering to an Army Command and Control System: A New Approach to Integration of Systems and Software Requirements and Design, *Proceedings of the INCOSE International Symposium*, Brighton, England, June 6-11, 1999.
- [38] Steiner, Rick, Friedenthal, Sanford, Oesterheld, Jerry, and Thaker, Guatam, Pilot Application of the OOSEM Using Rational Rose Real Time to the Navy CC&D Program, *Proceedings of the INCOSE International Symposium*, Melbourne, July 1-4, 2001.
- [39] Rose, Susan, Finneran, Lisa, Friedenthal, Sanford, Lykins, Howard, and Scott, Peter, *Integrated Systems and Software Engineering Process*. Herndon, VA: Software Productivity Consortium, 1996.
- [40] Forsberg, Kevin, and Mooz, Harold, Application of the "Vee" to Incremental and Evolutionary Development, *Proceedings of the Fifth Annual International Symposium of the National Council on Systems Engineering*, St. Louis, July 1995.
- [41] Izumi, L., Friedenthal, S., and Meilich, A., Object-Oriented Systems Engineering Method (OOSEM) Applied to Joint Force Projection (JPF), a Lockheed Martin Integrating Concept (LMIC), *Proceedings of the INCOSE International Symposium*, June 2007.
- [42] Object Management Group, *Diagram Interchange*, v. 1.0, OMG document number formal/06-04-04, April 2006.

Index

A

Abstract syntax, 63, 66–67, 337, 505

Accept event action

 description of, 191–192, 209
 node, 529*f*

Accept signal action, 191, 193

Accept time action node, 529*f*

Action(s)

 accept event. *See* Accept event action
 accept signal, 191, 193
 call. *See* Call actions
 control operators used to enable and disable,
 189–191
 definition of, 171, 174
 example of, 175
 node for, 535*f*
 with nonstreaming input and output, 189
 primitive, 209–210, 210*f*
 requirements for, 175–176
 send signal, 191–192, 209, 220, 252, 529*f*
 tokens created by, 171, 174–175

Action pins, 326

Activations, 224, 224*f*

Activities

 behavior depicted by, 172
 as block behaviors, 203–204
 in block context, 202–206
 communicating between, 172
 continuous, 210–211
 control flow in. *See* Control flow
 definition of, 171
 do, 206, 245
 executing, 208–211
 function of, 128, 171, 176
 invocation, composite associations used
 to model, 206
 as methods, 204–205
 node, 524*f*
 signals used to communicate between, 192*f*
 use case with, 276

Activity composition node, 524*f*

Activity diagram

 allocation on, 309*f*
 automobile system application of, 41–42,
 47, 48*f*
 definition of, 171–172
 description of, 30, 30*f*

example of, 173*f*

frame label for, 173

invocation actions on, 180*f*

nodes, 527–529*f*

paths, 530*f*

purpose of, 527

residential security system, 436–437*f*,
 456–457*f*

use case and, 279–280

water distiller case study of, 367–368, 368*f*

Activity final node, 188, 528*f*

Activity flow, 204*f*

Activity hierarchy

 block definition diagrams used to model,
 206–208

 description of, 174, 174*f*

Activity parameters

 description of, 176–178
 nodes, 177, 179, 186, 527*f*

Activity partitions

 allocate, 322–323, 375*f*, 376
 description of, 41, 200–202, 201*f*, 279
 node, 527*f*

Actor

 associations used with, 272
 definition of, 269–270
 node, 521*f*, 537*f*
 system users represented using, 270–271

Actor part node, 525*f*

Actual gates, 234

Alias, 88

Allocate activity partitions, 322–323, 375*f*, 376

Allocation

 asymmetric, 316
 of behavior, 312
 behavioral, 307
 of definition. *See* Allocation of definition
 definition of, 307
 evaluation of, across user model, 331–332
 of flow. *See* Flow allocation
 of function. *See* Functional allocation
 functional, 307
 between independent structural hierarchies,
 327–329
 inferred, 320
 logical-physical, 312, 448
 notation for, 308–311, 541*f*

- Allocation (*continued*)
 - of properties, 313
 - reference property relationships shown through, 108
 - of requirements, 311
 - software-hardware, 312
 - of structure. *See Structural allocation*
 - water distiller case study of. *See Water distiller system, allocation*
 - Allocation matrix, 325
 - Allocation of definition
 - description of, 308, 314–317, 315*f*, 316*t*
 - functional, 318, 320–322, 321*f*
 - structural, 329
 - Allocation of usage
 - description of, 308, 315*f*, 315–316, 316*t*
 - functional, 318, 319*f*
 - structural, 327–328
 - Allocation relationship
 - balance of, 332
 - in callout notation, 310, 310*f*
 - in compartment notation, 310
 - completeness and consistency evaluations, 332
 - creation of, 308
 - description of, 307
 - in matrix format, 311, 311*f*
 - Alt/else, 229–230
 - Analysis context
 - definition of, 169
 - description of, 163–165
 - trade study as, 167
 - Analysis models, 164
 - Analytic model, 491
 - Application programming interface, 501
 - Application Protocol, 233, 503
 - Architectural frameworks, 12–13
 - Architecture Team*, 10
 - Assembly connector, 126
 - Assert, 232
 - Assessment questionnaire, 510
 - Association(s)
 - with actors, 272
 - composite. *See Composite associations*
 - definition of, 110
 - reference. *See Reference associations*
 - Association blocks
 - description of, 112–113
 - node, 523*f*
 - path, 523*f*
 - Association path, 520*f*, 537*f*
 - Asymmetric allocation, 316
 - Asynchronous digital subscriber line connection, 111
 - Asynchronous message, 220–222, 532*f*
 - Asynchronous requests, 204
 - Atomic flow ports
 - description of, 122–123
 - node, 522*f*, 525*f*
 - Atomic state node, 534*f*
 - Automobile design
 - activity diagram, 41–42, 47, 48*f*
 - block definition diagram, 34, 36*f*, 37, 45, 46*f*, 56
 - internal block diagram, 43–45, 47, 49*f*, 50
 - parametric diagram, 52–53, 53*f*
 - requirement diagram, 34, 35*f*, 58*f*
 - sequence diagram, 39, 40*f*
 - state machine diagram, 42–43
 - systems engineering application to, 5–9
 - use case diagram, 37–39
- B**
- Base use case, 273
 - Behavior
 - classifier, 128–129
 - description of, 203–204
 - entry, 206
 - execution of, 223–225
 - exit, 206
 - main, 128–129
 - opaque, 128
 - state machine, 241
 - use cases elaborated with, 276–281
 - Behavioral allocation
 - description of, 307, 312
 - to structure, 317–323
 - Behavioral features
 - block response to request for, 144
 - classification and, 135, 143–144
 - description of, 129–130
 - Behavior port, 126, 131
 - Binding connectors, 52, 156
 - Black-box interaction, 235
 - Black-box specification, 425–427, 426*f*, 499
 - Block
 - association. *See Association blocks*
 - behavioral features of, 129–130
 - constraint. *See Constraint block*
 - definition of, 34, 95, 97
 - example of, 97
 - properties of, 52
 - structural elements of, 215
 - symbol for, 97, 99
 - value properties, 116–117, 159
 - whole-part relationship for, 101
 - Block composition hierarchy
 - on block definition diagram, 103*f*
 - part properties used to model, 100–108
 - Block configurations, 141–143, 142*f*, 159–161

- Block definition diagram
activity hierarchies, 174f, 206–208
airplane example application of, 68
allocation on, 309f
association blocks on, 112
automobile system application of, 34, 36f, 37, 45, 46f, 56
block composition hierarchy on, 103f
block configuration modeled on, 141–143, 142f
classification hierarchy on, 137f
compartments, 522
constraint blocks on, 149–150, 150f
description of, 30, 30f
example of, 96f
generalization set on, 138f
header of, 96
model library components represented on, 340
names on, 99
nodes, 521–522f, 524f
object nodes modeled using, 206, 208
parameters modeled using, 206, 208, 524f
part properties on, 103f
purpose of, 95–96, 521
reference association on, 108–109, 109f
residential security system, 438f, 454–455f, 460–462f, 465f, 467–469f, 485f
value types modeled on, 113, 115–116
variant configurations modeled on, 139f
water distiller case study of, 376–377
- Block node, 521f, 524f
- Break, 232
- Bridge, 501
- C**
- Call actions
description of, 171, 209
node, 529f
- Call behavior actions
control operator invoked by, 190–191
definition of, 178
function of, 176
name strings of, 185
pins, 178–179
- Call events
description of, 246, 252–253
transition path, 536f
- Call operation action, 205, 220
- Callout notation
for allocation relationships, 310, 310f
for requirements relationship, 291, 291f
- Causal analysis, 409–410, 412–413
- Central buffer nodes, 186, 529f
- Change events
description of, 192, 246–247, 263
transition path, 536f
- Child elements, 79
- Choice pseudostate
description of, 250, 251
node, 535f
- C4ISR standards framework, 12
- Classification
behavioral features and, 135, 143–144
of block, 136
hierarchies of, 134–144
overlapping, generalization sets for modeling of, 136–138
for reuse, 134
variants modeled using, 138–139
- Classifier behavior, 128–129, 202
- Classifiers, 134
- Clock, 136, 227
- Clock skew, 162
- Cohesion metrics, 25–26
- Collaboration artifacts, 434, 453
- Combined fragments
definition of, 216, 229
interaction operators, 229–233, 231f
- Compartment notation
for allocation relationships, 308, 310f
for requirements relationships, 290, 291f
- Completion events, 246
- Compliance matrix, 505, 506f
- Component Design, Implementation, and Test*, 9
- Component developers, 17
- Components package, 85
- Component specifications, 284
- Composite associations
definition of, 102
description of, 164, 174, 206
part properties, 102–104
path, 523f
symbol for, 102
- Composite state
definition of, 254
node, 534f
orthogonal, 255
with single region, 254–255
- Compound transition, 250
- Concept of operations
description of, 12
document-based systems engineering use of, 16
- Concrete syntax, 63, 505
- Configuration management tools, 493, 496–497
- Conform path, 519f
- Conjugate port, 123
- Connection points, 261

- Connector(s)
 assembly, 126
 associations used to define features of, 110
 definition of, 105
 delegation, 126
 modeling of, 106–107
 parts connected on internal block diagram
 using, 105–106, 106*f*
 path, 526*f*
 Connector allocation, 323–325, 324*f*
 Connector property node, 526*f*
 Connector property path, 526*f*
 Consider, 232
 Constraint
 definition of, 149
 duration, 226–227
 encapsulation of, in constraint blocks,
 152–154, 168
 state, 198
 stereotypes with, 342–346
 summary of, 168
 time, 226–227
 time-dependent, 161
 value properties of block, 159
 Constraint block
 analysis models, 164
 on block definition, 149–150, 150*f*
 composite associations between, 154
 composition used to build, 154–155
 constraints encapsulated in, 152–154, 168
 definition of, 149, 156
 description of, 51
 features of, 149
 item flows constrained using, 163
 libraries of, 164
 node, 524*f*
 parametric diagram, 150–151, 155–159
 value properties of block constrained
 using, 159
 Constraint expression, 149, 151–153
 Constraint parameters
 binding of, using parametric diagram, 155–159
 characteristics of, 153–154
 definition of, 152
 derived, 153
 node, 526*f*
 ordered, 153
 unique, 153
 Constraint properties
 description of, 152, 156, 163, 169
 node, 526*f*
 Constructive Systems Engineering Cost Model, 26
 Contained elements, 79
 Containment hierarchy, 84, 93, 284, 294–295
 Containment path, 519*f*, 539*f*
 Containment relationship, 83, 86, 294–295
 Context diagram, 44, 277, 277*f*
 Continuous activities, 210–211
 Continuous flow, 171, 194, 371, 373*f*
 Continuous state, 263–264
 Control flow
 allocation of, to connectors, 323, 325
 description of, 172
 order of action execution specified using,
 187–191
 path, 530*f*
 schematic diagram of, 189*f*
 Control nodes
 control logic depicted with, 188–189
 description of, 171–172
 Control operators
 action node, 529*f*
 description of, 189–191
 Control tokens, 172, 187
 ControlValue, 190
 Copy path, 539*f*
 Copy relationship, 303, 304*f*
 Coregion
 definition of, 229
 symbol for, 532*f*
 Cost function, 166
 COSYSMO. *See* Constructive Systems
 Engineering Cost Model
 Coverage property, 138
 Create messages
 description of, 225
 path, 532*f*
 Creation occurrence, 225
 Critical, 232
 Criticality property of requirement, 286
 Critical performance requirements, 424–425
 Cross-cutting relationships, 289–291
- D**
- Data architecture, 465–466
 Data exchange
 mechanisms of, 500–501
 standards for, 13, 501–504
 Data interchange standards, 13
 Data management tools, 493
 Data store nodes, 186, 529*f*
 Decision node, 182–183, 528*f*
 Decomposition of lifelines, 235–238
 Deep history pseudostate, 259
 Default value, 118
 Definition, allocation of
 description of, 308, 314–317, 315*f*, 316*t*
 functional, 318, 320–322, 321*f*
 structural, 329
 Delegation connector, 126

- Delegation port, 126, 127*f*
- Department of Defense Architecture Framework, 12
- Dependencies, 89–90, 93
- Dependency path, 519*f*
- Deployment, 509–514
- Derivation callout, 540*f*
- Derivation path, 539*f*
- Derived property, 100, 117–118
- Derive relationship, 296
- Design constraints, 7–8, 429, 448
- Destroy event node, 532*f*
- Destroy messages, 225
- Destroy occurrence, 225
- Development tools, 499
- Diagram(s). *See also specific diagram*
- interchange standards for, 503
 - UML, 69
- Diagram content, 33, 73–75
- Diagram description, 72
- Diagram frames, 33, 70–71
- Diagram header, 33, 71–72
- Diagram kind, 71
- Diagram name, 72
- Diagram usage, 72
- Dimensions
- definition of, 116
 - nodes, 521*f*
- Direct notation, 290
- Discrete rate, 194
- Discrete state, 263–264
- Do activity, 206, 245
- Documentation, 512
- Document-based approach
- characteristics of, 15
 - MBSE vs., 15–16
 - specification tree, 16
- Document-based systems engineering
- concept of operations document used in, 16
 - limitations of, 16
- Document generation tools, 493, 495–496
- Domain-specific language, 346
- Duration constraint
- description of, 226–227
 - symbol for, 533*f*
- Duration observation
- description of, 226
 - symbol for, 533*f*
- Dynamic system model, 491
- E**
- EIA 632, 12
- Electrical engineering, 16–17
- Element import, 87
- Elements. *See Model elements*
- Enabling systems, 483–485
- End event, 226
- Engineering analysis tools, 493, 495
- Enhanced Functional Flow Block Diagrams, 172, 208, 367
- Enterprise use cases, 417–418
- Entry behavior, 206
- Entry point pseudostate, 242, 259, 261*f*
- Enumeration node, 521*f*
- Events
- call, 246, 252–253, 536*f*
 - change, 246–247, 536*f*
 - completion, 246
 - definition of, 219
 - end, 226
 - receive message, 221
 - send message, 221
 - signal, 246–247, 536*f*
 - start, 226
 - time. *See Time events*
- Exception use cases, 418
- Executable specification, 208
- Executable system model, 490
- Executions, 223–225
- Exit behavior, 206
- Exit point pseudostate, 242
- Extension, 341
- Extension path, 520*f*, 537*f*
- Extension points, 273
- Extension relationships
- for stereotype, 349
 - for use case, 273–274
- External transition, 247
- F**
- File-based data exchange, 500
- Filtering fragment node, 531*f*
- Final state node, 535*f*
- Flow(s)
- continuous, 171, 194, 371, 373*f*
 - control. *See Control flow*
 - discrete, 171
 - item. *See Item flows*
 - object. *See Object flows*
 - parallel, 371
 - between ports, 127–128
- Flow allocations
- behavioral, 323
 - control flows, 323, 325
 - description of, 312, 323
 - functional, 383*f*
 - item flows, 323
 - object flows, 323–327
 - structural, 329–331
 - water distiller system, 371, 375*f*, 379, 381*f*, 382

- Flow-based simulation stereotype, 344*f*, 350*f*
 Flow charts, 188
 Flow final node, 188, 528*f*
 Flow order, 196
 Flow ports
 atomic, 122–123, 522*f*, 525*f*
 connecting of, on internal block diagram, 124–126
 description of, 45, 95, 120, 203, 422
 nonatomic, 123–124, 522*f*, 525*f*
 Flow property, 123
 Flow rates, 194, 196
 Flow specification
 definition of, 123
 illustration of, 124
 node, 522*f*
 Focus of control bars, 224
 Focus of control node, 532*f*
 Fork node, 182, 528*f*
 Fork pseudostate
 description of, 242, 255, 257
 node, 535*f*
 Formal gates, 234
 Found messages
 description of, 222
 path, 532*f*
 Functional allocation
 allocate activity partitions used to model, 322–323
 behavior allocated to structure using, 317–323
 of definition, 318, 320–322, 321*f*
 definition of, 307, 312
 flow, 383*f*
 of usage, 318, 319*f*
 water distiller case study of, 371, 375*f*
 Functional requirements, 6–7, 200
- G**
 Gates, 234
 Generalization, 134, 341
 Generalization path, 520*f*, 523*f*, 537*f*
 Generalization set, 138, 138*f*
 General-purpose systems modeling domain, 65–66
 General-purpose systems modeling language, 335
 Guard, 229, 246–247
 Guard expression on object flows, 182
 Guillemets, 37, 73, 349
- H**
 Hardware development tools, 493
 Harmony, 12, 398
 Hierarchical state, 254
 High-Level Architecture, 491
- History pseudostate
 description of, 258–259, 260*f*
 node, 535*f*
- I**
 Icon symbols, 74, 74*f*
 IEEE 1220, 12
 IEEE 1471-2000 standard, 13, 91
 Ignore, 232
 Import path, 519*f*
 Import relationship, 87–88
 Improvement process, 509–514
 Included use case, 273
 Inclusion path, 537*f*
 Inclusion relationship for use case, 273
 Initial node, 188, 528*f*
 Initial pseudostate
 description of, 244
 node, 535*f*
 Instances, 97
Integration and Test Team, 10
 Integration Definition for Functional Modeling, 13
 Interaction-based data exchange, 500
 Interaction operators, 229–233, 231*f*
 Interaction references, 234–235
 Interactions
 black-box, 235
 context for, 216–218
 definition of, 215, 238
 with lifelines, 219*f*
 lifelines used to represent participants in.
 See Lifelines
 messages connected to frame of, 235
 sequence diagram representation of, 216
 size of, 234
 use case with, 276
 weak sequencing, 222–223
 Interaction use
 description of, 234
 node, 531*f*
 Interface
 adding to standard ports, 131–132
 definition of, 130
 modeling, 130
 node, 522*f*
 Interface realization path, 522*f*
 Interface taxonomy, 422
 Internal block diagram
 allocation on, 309*f*
 automobile system application of, 43–45, 47, 49*f*, 50
 block configuration detailed modeled on, 143
 connecting parts on, 105–106, 106*f*
 description of, 30, 30*f*, 97

- example of, 98*f*, 216–217, 218*f*
flow ports connected on, 124–126
item flows on, 122*f*
nested parts shown on, 106, 107*f*
nodes, 525*f*
part properties modeled on, 104, 105*f*
paths, 526*f*
purpose of, 97, 525
reference properties modeled on, 110
residential security system, 439–440*f*,
 449–450*f*, 458–459*f*
standard ports connected on, 132–133
- Internal transition, 247
- International System of Units, 116
- Interruptible regions
- description of, 193–194, 195*f*, 374*f*
 - node, 527*f*
- Interrupting edge
- description of, 193
 - path, 530*f*
- ISO 10303. *See* STEP
- ISO 15288, 12
- Item, 120
- Item flows
- allocation of, 323
 - definition of, 95
 - description of, 163
 - function of, 121
- heat balance analysis in water distiller system,
 382, 384–386
- on internal block diagram, 122*f*
- modeling of, 120–121
- node, 526*f*
- object flows allocated to, 325–326
- properties associated with, 379
- Item property, 121
- J**
- Join node, 182, 188, 528*f*
- Join pseudostate
- description of, 242, 256–257
 - node, 535*f*
- Join specification, 182, 183*f*, 188
- Junction pseudostate
- description of, 250
 - node, 535*f*
- K**
- Keywords, 73
- L**
- Lifelines
- with activations, 224*f*
 - asynchronous messages exchanged
 between, 222*f*
- decomposition of, 235–238
- definition of, 218
- events, 219
- executions, 223
- interaction with, 219*f*
- messages exchanged between, 220, 222*f*
- nested, 237, 238*f*
- node, 531*f*
- nonoverlapping, 230*f*
- occurrences, 219
- overlapping, 230*f*
- selector expression, 218
- state invariants on, 233–234
- symbol for, 219
- synchronous messages exchanged between,
 222*f*
- Links, 105
- Logical architecture, 429–442
- Logical connector, 328
- Logical decomposition, 431–434
- Logical node architecture, 444–446, 445*f*
- Logical–physical allocation, 312, 448
- Logical structure, 327
- Loop, 229–230
- Lost messages
- description of, 222
 - path, 532*f*
- M**
- Main behavior, 128–129
- Master requirement callout, 540*f*
- MATLAB, 152
- Matrices
- allocation relationships depicted as, 311, 311*f*
 - compliance, 505, 506*f*
 - description of, 75–76
 - requirements relationships depicted as,
 293, 294*f*
- MBSE. *See* Model-based systems engineering
- Measures of effectiveness, 166, 412, 414
- Members
- definition of, 87, 93
 - visibility of, 87, 93
- Merge node, 183, 528*f*
- Message(s)
- asynchronous, 220–222, 532*f*
 - call and send, 221
 - create, 225
 - destroy, 225
 - exchanging of, between lifelines, 220, 222*f*
 - filtering of, 233*f*
 - found, 222
 - lost, 222
 - reply, 221
 - synchronous, 220–222

- Message overtaking, 223
- Metaclasses
- definition of, 66, 338
 - model elements and, 68*f*
 - node, 520*f*
 - in reference metamodel, 343
 - stereotypes based on, 341, 347
- Metamodels
- concepts associated with, 337–339
 - definition of, 66–67, 337
 - node, 520*f*
 - reference, 339, 341, 343
 - UML4SysML*, 338, 338*f*, 341, 505–506
- Meta Object Facility, 13, 337, 504
- Method(s)
- activities as, 204–205
 - definition of, 21, 134, 202, 223
 - MBSE, 21
 - modeling of, 134
- Metrics
- description of, 25–27, 497, 514
 - improvement uses of, 510
- Ministry of Defence Architectural Framework, 12–13
- Model(s)
- breadth of, 23
 - completeness of, 23
 - consistency of, 23–24
 - containment hierarchy, 84, 93
 - criteria necessary to meet purpose, 22–25
 - definition of, 21, 79, 81, 92
 - description of organization, 58, 92–93
 - depth of, 23
 - features of, 21
 - fidelity of, 23
 - good, 22
 - hierarchy of, 81–83
 - integration with other models, 25
 - interchange of, 60
 - organization of, 82–85, 404–408
 - in package hierarchy, 81–82
 - requirements representation in, 285–287
 - scope of, 22–23
 - self-documenting, 24–25
 - stereotypes applied when building, 348–352
 - in SysML, 21
 - understandability of, 24
 - water distiller case study, organization of, 362–364
- Model-based metrics, 25–27
- Model-based systems engineering (MBSE)
- definition of, 17
 - description of, 3, 15
 - document-based approach vs., 15–16
 - history of, 16–17
 - improvements resulting from use of, 20–21
- mathematical formalism for, 17
- method, 21
- model repository, 18–20
- purpose of, 17
- steps involved in, 361–362
- SysML used in support of, 31–32
- system model. *See* System model
- system modeling tool for, 505
- transitioning to, 20–21
- water distiller case study application of, 361–362
- Model Driven Architecture, 13
- Model elements
- definition of, 68, 338
 - description of, 59, 68*f*
 - diagrammatic representation of, 71–72
 - importing of, into packages, 87–89, 407
 - packageable. *See* Packageable elements
 - in package diagram, 80
 - in package hierarchy, 91
 - qualified name for, 86, 93
 - stereotyped, 336, 337*f*, 349–350
 - symbols, 349
- Model element type, 72
- Modeling conventions, 24
- Modeling language, 65, 335
- Modeling standards, 13
- Modeling tools, 339
- Model libraries
- definition of, 81, 335, 339
 - node, 520*f*
 - reusable constructs provided using, 339–340
- Model node, 519*f*
- Model repository, 18–20
- Model semantics, 23, 63, 67
- Model standards, 13
- Moe, 166–167, 414–415, 452, 472–473
- MOF. *See* Meta Object Facility
- Multicompartment fragment node, 531*f*
- Multidisciplinary systems engineering team
- description of, 9–10
 - schematic diagram of, 10*f*
- Multiple inheritance, 136
- N**
- Name clash, 87
- Name compartment with keywords, 542*f*
- Namespace
- definition of, 79, 85, 93, 151
 - packages as, 85–86
 - purpose of, 85
 - target, 87
 - uniqueness rules, 85
- Name string with keywords and properties, 542*f*
- Neg, 232
- Nested lifelines, 237, 238*f*

- Nested packages on package diagram, 84f, 86
Nested requirements, 295
Nested structures, 106–107
Node symbols, 73, 74f. *See also specific node*
Node with stereotype compartment, 542f
Nonatomic flow ports
 description of, 123–124
 node, 522f, 525f
Nonfunctional requirement, 284
Nonoverlapping lifelines, 230f
Nonstreaming activity parameter, 177
Notation
 allocation, 308–311
 callout. *See Callout notation*
 compartment. *See Compartment notation*
 definition of, 69, 337
 direct, 290
 matrices, 75–76
 requirements relationships depicted using, 290
 table, 75
 transition, 247–249, 252, 252f
 trees, 76
Note symbols, 74–75, 75f
- O**
- Object access actions, 209
Object constraint language, 24
Object flows
 allocation of
 to connector, 323–325, 324f
 to item flow, 325–326
 description of, 171
 function of, 179
 guard expression on, 182
 node, 530f
 order of, 196
 path, 530f
 pins and parameters connected using, 181f
 rates of, 194, 196
 routing, 182–186
Objective function, 166, 415, 472–473
Object manipulation actions, 209
Object nodes
 activity parameter nodes, 177, 179, 186
 block definition diagram modeling of, 206, 208
 composition path, 524f
 connecting of, 182
 description of, 179
 pins. *See Pins*
 state constraint on, 198
Object-oriented systems engineering method.
 See OOSEM
Objects, 97
Object update actions, 209
Occurrences
 creation, 225
 definition of, 219
 destroy, 225
OCL, 343
OOSEM
 description of, 12, 397
 design process in, 397–398
 development of, 398
 model organization, 404–408
 package structure of, 405
 residential security example of. *See Residential security system*
system development
 design levels, 400
 hardware components, 400
 integration, 401
 management process, 400
 overview of, 398–400
 software components, 400–401
 specifications, 400
 verification, 401
system model, 397–398
system requirements, 400
system specification and design process, 401, 402f
Opaque behavior, 128
“Opaque” constructs, 209
Open Group Architecture Framework, 13
Operands, 229
Operation, 129
Operation calls, 252–253
Opt, 229
Ordered constraint parameters, 153
Ordering property, 196
Orthogonal composite state, 242, 255
Overlapping lifelines, 230f
Overlap property, 138
- P**
- Package(s)
 components, 85
 definition of, 59, 81, 93
 dependency between, 89–90, 93
 model elements imported into, 87–89
 as namespaces, 85–86
 nested, 84f, 86
 node, 519f, 538f
 top-level, 82
Packageable elements
 definition of, 81
 dependencies between, 89–90, 93
 in model library, 339–340
 node, 519f
 on package diagram, 85
Package diagram, 519–520f
 dependencies on, 90
 description of, 29, 336

- Package diagram (*continued*)
 model elements contained in, 80
 model library components represented on, 340
 model organization represented using, 363/
 nested packages on, 84/
 nodes, 519-520/
 packageable elements on, 85
 packages defined in, 80-82
 paths, 519-520/
 purpose of, 518
 residential security system, 464/
 sample, 30/
 stereotypes depicted on, 341/
 Package hierarchy
 definition of, 92
 model elements in, 91
 model in, 81-82, 84/
 organizing of, 82-85
 purpose of, 91
 Package import, 87
 Par, 229-230
 Parallel flow, 371
 Parameters
 for activities, 176-178
 block definition diagram modeling of, 206, 208
 constraint. *See* Constraint parameters
 object flows used to connect, 181/
 operations, 129
 optional, 176
 required, 176
 Parameter sets
 definition of, 185
 routing object flows from, 185-186
 Parametric diagram
 automobile system application of, 52-53, 53/
 constraint blocks, 150-151, 155-159, 169
 definition of, 150
 description of, 30, 30/
 model organization using, 57-59
 nodes, 526/
 power distribution equation using, 158/
 purpose of, 526
 residential security system, 474/
 Parametrics, 474-475
 Participant property node, 525/
 Partitioning, 442-444
 Part node, 525/
 Part properties
 block composition hierarchies modeled using,
 100-108
 on block definition diagram, 103/
 composite associations, 102-104
 connecting of, 105-106
 definition of, 95, 100, 218
 on internal block diagram, 104, 105/
 Path symbols, 73-74, 74/
 Performance simulation model, 490-491
 Performance simulation tools, 493
 Physical structure, 327-329
 Pilot project, 512-513
 Pins
 action, 326
 call behavior action, 178-179
 definition of, 174-175
 object flows used to connect, 181/
 Polymorphism, 134, 144
 Ports
 behavior, 126, 131
 conjugate, 123
 definition of, 95
 delegation, 126, 127/
 description of, 45, 377-379
 flow. *See* Flow ports
 flow modeling between, 127-128
 function of, 120
 standard. *See* Standard port
 Postconditions, 197
 Preconditions, 197
 Primitive action node, 529/
 Primitive actions, 209-210, 210/
 Probabilistic flow, 196-197
 Probability distribution, 118-119
 Probes, 352
 Problem callout, 291-292, 540/
 Profile(s)
 definition of, 335, 341, 346
 example of, 336/
 node, 520/
 reference metamodel for, 347
 stereotypes from, 346, 348-349
 in UML, 67
 user model application of, 347-348
 uses of, 346
 Profile application
 description of, 347
 path, 520/
 Project management tools, 492, 497-498
 Properties
 coverage, 138
 default value assigned to, 118
 definition of, 95, 99
 derived, 100, 117-118
 part. *See* Part properties
 purpose of, 99-100
 redefining, 136
 reference, 100, 218
 value. *See* Value properties
 Property derivation, 100, 117-118
 Property-specific type, 139-140
 Pseudostates
 choice, 250, 251/
 definition of, 242, 244

- entry point, 242, 259, 261*f*
exit point, 242, 259, 261
fork, 242, 255, 257, 535*f*
history, 258–259, 260*f*, 535*f*
initial, 244
join, 242, 256–257
junction, 250
terminate, 244
transitions routed using, 249–251, 251*f*
- Q**
Qualified name, 86, 93
- R**
Rationale callout, 540*f*
Rationale for requirements relationships, 291–292
Rational Unified Process for Systems Engineering, 12, 398
Realization dependency, 90, 131
Receive message event, 221
Receptions, 129, 221
Redefinition, 136
Reference associations
 on block definition diagram, 108–109, 109*f*
 definition of, 108–109
 path, 523*f*
 symbol for, 109
Reference clock, 162
Referenced sequence diagram, 39–40
Reference metamodel
 definition of, 339, 341, 343
 for profile, 347
Reference node, 525*f*
Reference path, 520*f*
Reference properties
 definition of, 100, 108, 218
 internal block diagram used to model, 110
 noncomposite relationships between blocks
 modeled using, 108–113
Reference relationship, 347
Refine dependency, 90
Refinement callout, 540*f*
Refinement path, 539*f*
Refine relationship, 300–302, 302*f*, 418
Region(s)
 definition of, 243–244
 multiple, 255–258
 single, 254–255
Relationship
 allocation. *See* Allocation relationship
 containment, 294–295
 reference, 347
 requirements. *See* Requirements
 relationships
 satisfy, 298
 verify, 298–300, 300*f*
- Reply message, 221, 532*f*
Repository-based data exchange, 500
Requirement(s)
 allocation of, 311
 criticality property of, 286
 deriving, 296–298
 expressing of, 283
 function of, 283
 model representation of, 285–287
 nested, 295
 nonfunctional, 284
 package structure organization of, 294
 risk property of, 286
 sources of, 283
 specification for, 283
 stereotypes, 286–287, 288*f*
 text-based, 283
 verification status, 285–286
 water distiller case study, 359–360, 364–367
- Requirement ambiguity, 298–300
Requirement diagram
 automobile system application of, 34, 35*f*, 58*f*
 callouts, 540*f*
 description of, 30, 30*f*, 284–285
 example of, 286*f*
 header for, 285
 nodes, 538*f*
 paths, 539*f*
 purpose of, 538
 residential security system, 479–480*f*
 water distiller case study of, 364, 365*f*
- Requirement node, 538*f*
Requirement related type node, 538*f*
Requirements allocation, 311
Requirements analysis, 284
Requirements categories, 286–287
Requirements management tools, 283–284, 493–495
Requirements relationships
 callout notation for, 291, 291*f*
 compartment notation for, 290, 291*f*
 cross-cutting, 289–291
 depiction of, 290–293
 diagram used to represent, 284
 direct notation for, 290
 matrix depiction of, 293, 294*f*
 rationale for, 291–292
 refine, 300–302, 302*f*
 residential security system, 478, 480
 tabular depiction of, 292–293, 294*f*
 trace, 303
 types of, 287–289
 verifying of, 298–300
- Requirements table, 292, 293*f*
Requirements Team, 10
Requirements traceability, 9, 16, 56–57, 297

- Requirements tree, 366
 Requirements variation analysis, 429
 Residential security system, 402–408
 activity diagram, 436–437f, 456–457f
 block definition diagrams, 438f, 454–455f,
 460–462f, 465f, 467–469f, 485f
 engineering analysis, 414
 internal block diagram, 439–440f, 449–450f,
 458–459f
 model development, 408
 analyses, 471–474
 Analyze Stakeholder Needs activity,
 409–418
 Analyze Systems Requirements activity,
 418–429
 as-is system, 409–412
 black-box specification, 425–427, 426f
 causal analysis, 409–410, 412–413
 component requirements, 467–471
 constraints, 474
 critical performance requirements,
 424–425
 data architecture, 465–466
 Define Logical Architecture activity,
 429–442
 design constraints, 429
 enabling systems, 483–485
 engineering analysis, 475
 enterprise scenarios, 418, 420–422
 enterprise use cases, 417–418
 hardware architecture, 466
 Integrate and Verify System, 481–485
 logical decomposition, 431–434
 logical node architecture, 444–446, 445f
 Manage Requirements Traceability
 activity, 475–481
 measures of effectiveness, 412, 414
 mission requirements, 414
 operational procedures, 467
 Optimize and Evaluate Alternatives
 activity, 471–475
 partitioning criteria, 442–444
 physical node architecture, 446–460
 requirements relationships, 478, 480
 Requirements Variation analysis, 429
 security architecture, 471
 software architecture, 460–465
 specification tree, 476–477, 477f
 state machine, 427–429, 441f
 Synthesize Candidate Physical
 Architecture activity, 442–471
 system context, 422–424
 text-based requirements, 478
 to-be domain model, 415, 416f
 traceability gaps, 481
 trace relationship, 477
 trade studies, 452
 verification procedures, 481–485
 modeling conventions and standards, 404
 model organization, 404–408
 package diagram of, 408f, 464f
 parametric diagram, 474f
 problem background, 402–404
 requirements diagram, 479–480f
 sequence diagrams, 463, 463f
 stakeholder needs activity, 409–418
 Systems Engineering Integrated Team, 403
 Risk property of requirement, 286
 Role name, 102–104
 Routing
 of object flows, 182–186
 of transitions using pseudostates, 249–251, 251f
 RUP SE. *See* Rational Unified Process for Systems
 Engineering
- S**
- Satisfaction callout, 540f
 Satisfaction path, 539f
 Satisfy relationship, 298
 Scenarios, 272
 Selector expression, 218
 Semantics, 63, 67
 Sending event, 216
 Send message event, 221
 Send signal action, 191–192, 209, 220, 252, 529f
 Send signal node, 535f
 Seq, 229
 Sequence diagram
 automobile system application of, 39, 40f
 description of, 30, 30f, 215
 example of, 217f
 interaction representation by, 216
 message exchanges in, 40
 nodes, 531–532f
 paths, 532f
 purpose of, 531
 referenced, 39–40
 residential security system, 463, 463f
 time representation on, 225–228
 use case elaborated with, 277–279, 279f
 water distiller case study use of, 390, 391f
 Service-oriented approach, 215
 Shallow history pseudostate, 259
 Signal events
 description of, 246–247
 transition path, 536f
 Signals, 130, 191–193, 220–221
 Single compartment fragment node, 531f
 SI units, 116, 117f
 Sizing parameters, 26

- Software architecture, 460–465
Software development tools, 493
Software engineering, 12
Software–hardware allocation, 312
Specialization of stereotypes, 342, 353–354
Specification
 component, 284
 definition of, 283
 systems, 284
Specification tree, 16, 284, 294, 476–477, 477*f*
Stakeholders
 description of, 5–9
 requirements, 359–360
Standard ports
 connecting of, on internal block diagram, 132–133
 definition of, 95, 120
 description of, 45
 interfaces added to, 131–132
 node, 522*f*, 525*f*
Standards
 architectural frameworks, 12–13
 data interchange, 13
 evolution of, 11–12
 frameworks, 12
 model, 13
 modeling, 13
 software engineering and, 12
 systems engineering, 11–13
 taxonomy of, 11*f*, 11–12
Start event, 226
State
 composite. *See* Composite state
 continuous, 263–264
 definition of, 244–245
 discrete, 263–264
 entry and exit behaviors, 245, 261, 261*f*
 hierarchical, 254
 submachine, 242, 259, 261–263
 transitioning between. *See* Transition
State analysis method, 12
State charts, 242
State constraint, 198
State hierarchies
 composite states, 254–258
 description of, 254
 nested, transition firing order in, 257–258
State invariants
 description of, 233–234, 234*f*
 symbol for, 531*f*
State machine(s)
 behavior of, 241
 description of, 43
 discrete, 264*f*
 interactions between, 242
 operation calls, 252–253
 overview of, 241–242
 pseudostates. *See* Pseudostate
 regions, 243–244
 residential security system, 427–429, 441*f*
 scenarios represented by, 280–281
 schematic diagram of, 243*f*
 use case with, 276
 water distiller case study use of, 395*f*
State machine diagram
 automobile system application of, 42–43
 description of, 30, 30*f*, 242
 example of, 243*f*
 frame label of, 242
 nodes, 534–535*f*
 paths, 536*f*
 purpose of, 534
 use case and, 280–281
 water distiller case study of, 370, 370*f*
STEP, 502–503
Stereotype(s)
 application of, during model building, 348–352
 callout, 542*f*
 constraints added to, 342–346
 definition of, 33, 67, 335
 extension relationships, 349
 flow-based simulation, 344*f*
 function of, 341
 metaclasses as basis for, 341, 347
 model elements, 336, 337*f*, 349
 node, 520*f*
 notation for, 542*f*
 package diagram depiction of, 341*f*
 profile. *See* Profile
 properties added to, 342–346
 requirements, 286–287, 288*f*
 specialization of, 342, 353–354
 subclassing, 286–287
 in user model, 345
Streaming activity parameter, 177
Strict, 232
Strict property of profile application
 relationship, 347
Structural allocation
 of definition, 329
 description of, 312–313
 flow, 329–331
Subclasses, 37, 134
Subject node, 537*f*
Submachine state, 242, 259, 261–263
Subset, 136
Superclass, 134
Surveillance system case study, 76
 modeling conventions used in, 77
 package diagram for, 82*f*

- Swimlane, 200, 322–323
- Symbols**
 - activity final node, 188
 - activity partition, 200
 - asynchronous message, 221
 - call behavior action, 178
 - callout, 291
 - composite association, 102
 - duration constraint, 227
 - flow final node, 188
 - fork, 183
 - icon, 74, 74*f*
 - initial node, 188
 - join, 183
 - lifeline, 219
 - model element, 349
 - node, 73, 74*f*
 - note, 74–75, 75*f*
 - path, 73–74, 74*f*
 - submachine state, 261–262, 262*f*
 - synchronous message, 221
 - time constraint, 227
 - transition, 247–249, 252, 252*f*
 - use case, 272
- Synchronous message, 220–222, 532*f*
- Synchronous requests, 204
- SysML**
 - automobile design application of, 32–60
 - description of, 3
 - representation of systems, 29
- SysML diagrams.** *See also specific diagram*
 - content of, 73–75
 - description, 72
 - frames, 33, 70–71
 - header, 33, 71–72
 - icon symbols, 74, 74*f*
 - keywords, 73
 - name, 72
 - node symbols, 73, 74*f*
 - note symbols, 74–75, 75*f*
 - path symbols, 73–74, 74*f*
 - purpose of, 30
 - summary of, 29–30
 - taxonomy of, 69–70, 70*f*
 - usage, 72
- SysML language**
 - applications of, 31–32
 - architecture of, 65–69, 429–471
 - diagram overview, 29–31
 - purpose, 29
 - semantics of, 63, 67
 - specification, 63–64
- SysML model**
 - critical properties, 25
 - description of, 21
- SysML profile, 64
- SysML specification, 13
- System(s)**
 - complexity of, 11
 - design of, 7, 8*f*, 9
 - hierarchy of, 65
 - requirements, 6
 - use case for describing functionality of, 271–276
 - users of, 270–271
- System boundary, 6, 6*f*
- System context, 277, 422–424
- System Integration and Test*, 4, 4*f*
- System life cycle, 5–6
- System model**
 - analytic, 491
 - description of, 17–18, 18–19*f*
 - dynamic, 491
 - executable, 490
 - purpose of using, 22
 - in systems development environment, 489–492
 - types of, 490
- System modeling domain**
 - general-purpose, 65–66
 - mapping between concepts in, 67
- System modeling tools**
 - description of, 492–494
 - evaluation of, 511–512
 - interactions, 494–499
 - selection of, 504–507
- System of systems**
 - allocations for modeling of, 307
 - definition of, 3, 11
 - modeling tools for, 492
 - OOSEM, 398
- Systems Analysis Team*, 10
- Systems development environment**
 - data exchange in, 500–501
 - definition of, 489
 - system model's role in, 489–492
- Systems engineering**
 - application of, 5–9
 - automobile industry application of, 5–9
 - configuration management tool, 496–497
 - definition of, 4
 - industries that use, 4
 - management plan, 16
 - methods, 12
 - model-based. *See Model-based systems engineering*
 - motivation for, 3–4
 - object-oriented. *See OOSEM*
 - process of, 4–5, 398–402
 - Rational Unified Process for, 12, 398

- schematic diagram of, 4f
summary of, 13-14
- Systems Engineering Integrated Team, 403
- Systems engineering manager, 10
- System Specification and Design*, 4-5, 399-402
- Systems specification, 284
- System under consideration, 272
- T**
- Tables
- description of, 75
 - requirements relationships depicted in, 292-293, 294f
- Target namespace, 87
- Terminate pseudostate
- description of, 244
 - node, 535f
- Test case
- description of, 299-300
 - node, 538f
- Test signal, 191
- Text-based requirements
- description of, 283
 - residential security system, 478
- Time constraint
- description of, 226-227
 - symbol for, 533f
- Time events
- description of, 192, 246-247
 - transition path, 536f
- Time observation
- description of, 226, 228
 - symbol for, 533f
- Time representation using sequence diagram, 225-228
- Time varying properties, 161-162
- Tokens
- description of, 171, 174-175, 193
 - discarding of, 196
 - overwriting of, 196
- Trace, 219
- Trace callout, 540f
- Trace compartment, 538f
- Trace dependency, 90
- Trace path, 539f
- Trace relationship, 303, 477
- Trade studies, 166-168, 452, 471-475
- Training, 512
- Transition
- compound, 250
 - definition of, 242
 - external, 247
 - firing order of, in nested state hierarchies, 257-258
 - internal, 247
 - naming of, 248
 - notation for, 247-249, 252, 252f
 - purpose of, 245
 - triggers, 246
- Transition effect, 206, 247, 249f
- Transition guard, 246-247
- Trees, 76
- Trigger node, 535f
- Triggers, 246
- U**
- UML
- description of, 13, 63-64
 - diagrams, 69
 - profile in, 67
 - reusable portion of, 64
 - timing diagram, 53-54
- UML4SysML*, 338, 338f, 341, 505-506
- Unique constraint parameters, 153
- Uniqueness rules, 85
- Units
- definition of, 56, 116
 - nodes, 521f
- Usage
- allocation of, 315f, 315-316, 316t
 - definition of, 101
- Usage dependency path, 522f
- Use case(s)
- with activities, 276
 - activity diagram and, 279-280
 - actor. *See Actor*
 - base, 273
 - behaviors added to, 276-281
 - classification of, 274
 - context diagrams and, 277, 277f
 - definition of, 271
 - description of, 38-39, 269
 - enterprise, 417-418
 - exception, 418
 - extension relationship, 273-274
 - included, 273
 - inclusion relationship, 273
 - with interactions, 276
 - node, 537f
 - relationships, 273-274
 - requirements analysis supported with, 284
 - residential security system case study of, 417-418
 - scenarios, 272
 - sequence diagram and, 277-279, 279f
 - with state machine, 276
 - state machine diagram and, 280-281
 - system functionality described using, 271-276
- Use case description, 271-272, 275-276, 284, 418

- Use case diagram**
 automobile system application of, 37–39
 description of, 30, 30*f*, 269–270
 example of, 270*f*, 274, 275*f*
 header for, 269–270
 nodes, 537*f*
 paths, 537*f*
 purpose of, 537
 water distiller case study use of, 390, 390*f*
- Use dependency**, 90
- User model**
 allocation evaluation across, 331–332
 components of, 67–68
 definition of, 67, 335
 profiles applied to, 347–348
 stereotypes in, 345
- Users**, 270–271
- Uses dependency**, 131
- Utility function**, 166
- V**
- Value actions**, 209
- Value binding path**, 526*f*
- Value properties**
 blocks with, 116–117
 definition of, 95, 100, 113
 description of, 152
 node, 525*f*
 property-specific type, 139–140
 purpose of, 113
 time varying properties, 161–162
- Value types**
 block definition diagram used to model, 113, 115–116
 definition of, 113
 dimensions added to, 116
 node, 521*f*
 units added to, 116
- Variants**, 138–139
- Vee development process**, 398
- Verdict**, 299
- Verification callout**, 540*f*
- Verification path**, 539*f*
- Verification status for requirement**, 285–286
- Verification tools**, 493, 498, 499*f*
- View**
 description of, 92
 node, 519*f*
- Viewpoint**
 description of, 91–92
 node, 519*f*
- Visibility of members**, 87, 93
- Vitech Model-Based Systems Engineering Method**, 12
- W**
- Water distiller system**
 activity diagram, 367–368, 368*f*
 allocation
 of actions, 379, 381*f*, 382
 activity partitions, 375*f*, 376
 flow, 371, 375*f*, 379, 381*f*, 382
 functional, 371, 375*f*, 383*f*
 updating, 387–390
- blocks**
 block definition diagram of, 376–377
 internal block diagram of, 379, 380*f*
 ports on, 377–379
- continuous flow**, 371, 373*f*
- controller**, 391–392, 392–393*f*
- design modifications**, 386–394
- functional allocations**, 371, 375*f*, 383*f*
- functional hierarchy** in, 370*f*
- heat balance** in, 382, 384–386
- internal block diagram**, 379, 380*f*
- interruptible region**, 374*f*
- item flow heat balance analysis**, 382, 384–386
- MBSF approach**, 361–362
- modeling behavior**, 367–376
- model organization**, 362–364
- parallel flow**, 371
- performance analysis**, 382–386
- problem statement**
 activity diagram depiction of, 367, 368*f*
 description of, 359–361, 364
- requirements**, 359–360, 364–367
- sequence diagram** for, 390, 391*f*
- stakeholder requirements**, 359–360
- startup and shutdown**, 392, 394
- state machine diagram**, 370, 370*f*
- state machine for**, 395*f*
- structure**
 hierarchy of, 376*f*, 392*f*
 modeling of, 376–382
 updating of, 387–390
- use case diagram** for, 390, 390*f*
 user interface, 391–392, 392–393*f*
- Weak sequencing**, 222–223, 229
- Whole-part relationship**, 101
- X**
- XMI**, 13, 60, 69, 502
- XML Metadata Interchange**, 13, 60, 69, 502
- Z**
- Zachman framework**, 12