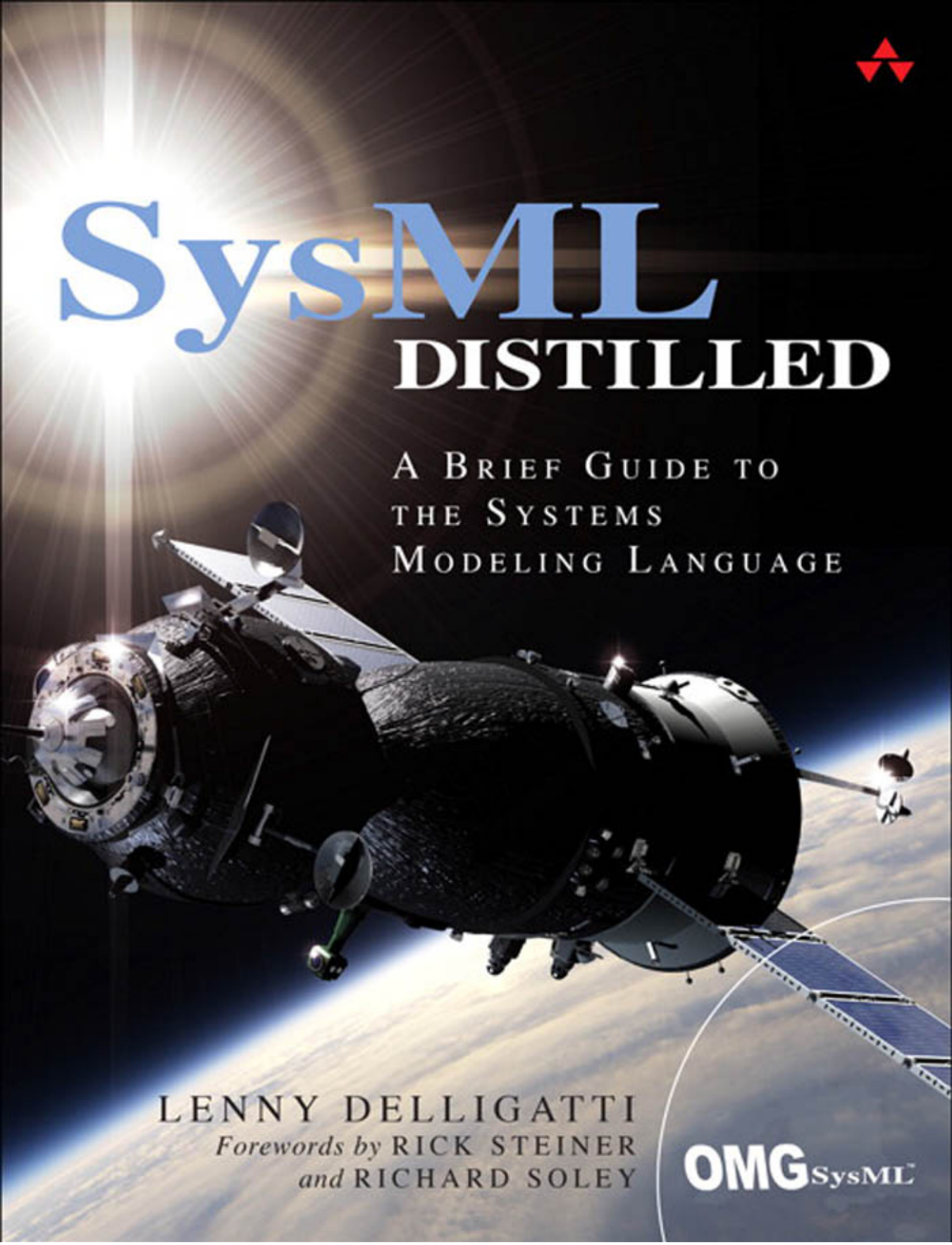

Praise for *SysML Distilled*

"In keeping with the outstanding tradition of Addison-Wesley's technical publications, Lenny Delligatti's *SysML Distilled* does not disappoint. Lenny has done a masterful job of capturing the spirit of OMG SysML as a practical, standards-based modeling language to help systems engineers address growing system complexity. This book is loaded with matter-of-fact insights, starting with basic MBSE concepts to distinguishing the subtle differences between use cases and scenarios to illumination on namespaces and SysML packages, and even speaks to some of the more esoteric SysML semantics such as token flows."

— Jeff Estefan, Principal Engineer, NASA's Jet Propulsion Laboratory

"The power of a modeling language, such as SysML, is that it facilitates communication not only within systems engineering but across disciplines and across the development life cycle. Many languages have the potential to increase communication, but without an effective guide, they can fall short of that objective. In *SysML Distilled*, Lenny Delligatti combines just the right amount of technology with a common-sense approach to utilizing SysML toward achieving that communication. Having worked in systems and software engineering across many domains for the last 30 years, and having taught computer languages, UML, and SysML to many organizations and within the college setting, I find Lenny's book an invaluable resource. He presents the concepts clearly and provides useful and pragmatic examples to get you off the ground quickly and enables you to be an effective modeler."

— Thomas W. Fargnoli, Lead Member of the Engineering Staff, Lockheed Martin

"This book provides an excellent introduction to SysML. Lenny Delligatti's explanations are concise and easy to understand; the examples well thought out and interesting."

— Susanne Sherba, Senior Lecturer, Department of Computer Science, University of Denver

"Lenny hits the thin line between a reference book for SysML to look up elements and an entertaining book that could be read in its entirety to learn the language. A great book in the tradition of the famous *UML Distilled*."

— Tim Weilkiens, CEO, oose

"More informative than a PowerPoint, less pedantic than an OMG Profile Specification, *SysML Distilled* offers practicing systems engineers just the right level of the motivation, concepts, and notation of pure OMG SysML for them to attain fluency with this graphical language for the specification and analysis of their practical and complex systems."

— Lonnie VanZandt, chief architect, No Magic, Inc.

"Delligatti's *SysML Distilled* is a most aptly named book; it represents the distillation of years of experience in teaching and using SysML in industrial settings. The author presents a very clear and highly readable view of this powerful but complex modeling language, illustrating its use via easy-to-follow practical examples. Although intended primarily as an introduction to SysML, I have no doubt that it will also serve as a handy reference for experienced practitioners."

— Bran Selic, president, Malina Software Corp.

"SysML is a rather intimidating modeling language, but in this book Lenny makes it really easy to understand, and the advice throughout the book will help practitioners avoid numerous pitfalls and help them grasp and apply the core elements and the spirit of SysML. If you are planning on applying SysML, this is the book for you!"

— Celso Gonzalez, senior developer, IBM Rational

"*SysML Distilled* is a great book for engineers who are starting to delve into model-based systems engineering. The space system examples capture the imagination and express the concepts in a simple but effective way."

— Matthew C. Hause, chief consulting engineer, Atego and chair, OMG UPDM Group

"I've been deeply involved with OMG since the 1990s, but my professional needs have not often taken me into the SysML realm. So I thought I'd be a good beta tester for Lenny's book. To my delight, I learned a great deal reading through it, and I know you will too."

— Doug Tolbert, distinguished engineer, Unisys, and member, OMG Board of Directors and Architecture Board

"*SysML Distilled* provides a clear and comprehensive description of the language component of model-based systems engineering, while offering suggestions for where to find information about the tool and methodology components. There is evidence throughout the book that the author has a deep understanding of SysML and its application in a system development process. I will definitely be using this as a textbook in the MBSE courses I teach."

— J. D. Baker, OCUP, OCSMP, member of the OMG Architecture Board

"*SysML Distilled* is the desktop companion that many SysML modelers have needed for their bookshelves. Lenny has the experience and certifications to help you through your day-to-day modeling questions. This book is not a tutorial, nor is it the encyclopedic compendium of all things SysML. If you model using SysML, this will become your daily companion, as it is meant to be used regularly. I believe your copy will soon be dog-eared, with sticky notes throughout."

— Dr. Robert Cloutier, Stevens Institute of Technology

"SysML is utilized today in a wide range of applications, including deep space robotic spacecraft and down-to-earth agricultural equipment. This book concisely presents SysML in a manner that is both refreshingly accessible for new learners and quite handy for seasoned practitioners."

— Russell Peak, MBSE branch chief, Aerospace Systems Design Lab, Georgia Tech

"*SysML Distilled* is a wonderfully written, knowledgeable, and concise addition to systems modeling literature. The lucid explanations lead a newcomer by the hand into modeling reasonably complex systems, and the wealth and depth of the coverage of the most-used aspects of the SysML modeling language stretch to even enabling advanced intermediate depictions of most systems. It also serves as a handy reference. Kudos to Mr. Delligatti for gifting the world with this very approachable view of systems modeling."

— Bobbin Teegarden, CTO/chief architect, OntoAge and Board Member, No Magic, Inc.

This page intentionally left blank

SysML Distilled

This page intentionally left blank

SysML Distilled

A Brief Guide to the Systems Modeling Language

Lenny Delligatti

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Delligatti, Lenny.
SysML distilled : a brief guide to the systems modeling language / Lenny Delligatti.
pages cm
Includes bibliographical references and index.
ISBN-13: 978-0-321-92786-6 (paperback : alk. paper)
ISBN-10: 0-321-92786-9 (paperback : alk. paper)
1. Systems engineering—Data processing. 2. Engineering systems—Computer simulation. 3. SysML (Computer science) I. Title.
TA168.D44 2014
620.00285'5133—dc23 2013035922

ISBN-13: 978-0-321-92786-6
ISBN-10: 0-321-92786-9

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, November 2013

This book is dedicated to my wife, Natalie, and my children, Aidan and Noelle—my greatest blessings . . . and my reasons for enduring the many late nights of writing.

This page intentionally left blank

Contents

This page intentionally left blank

Foreword by Rick Steiner

Systems engineering is not an easy subject to teach. Earlier in my career, I was emphatically told that systems engineering could not be taught in a classroom and that it could only be learned through experience. While that hasn't proven to be true, there are certainly concepts within the practice of systems engineering that are both subtle and arcane.

Expressing these concepts in models demands a suitably robust language, which is why a dedicated group of us began development of what would become SysML in early 2002. We attempted to be parsimonious and direct when designing the language, specifically targeting it for use by practicing systems engineers. I'm convinced that the resulting language is both flexible and useful, and I am gratified that it has emerged as a dominant standard for communicating systems-related ideas.

Just like the practice of systems engineering, however, SysML has proven difficult to teach effectively. The scope of systems engineering is remarkably broad, and even though SysML is a relatively compact language, students frequently get overwhelmed with its complexity. Resources for learning SysML and model-based systems engineering have until recently been rather limited, but it's getting better. Formal MBSE and SysML courses are now regularly being taught through several university or extension catalogs, and at least one comprehensive textbook is now available.

An engineer or manager who wants to casually learn the basics of SysML isn't likely to want to take a class. An advanced systems engineer who finds him- or herself in the middle of a project with tight deadlines just doesn't have the time to take a class. It is in both of these situations that this book has the greatest value.

Structured in a manner similar to Martin Fowler's popular *UML Distilled*, this book lays out the fundamentals of SysML diagrams in clear, concise terms. It is written in a casual, lighthearted manner, yet it conveys the gist of each concept and its graphical representation. What I like best about this book is that it keeps me reading, without getting

bogged down in "meta-speak" and "UML-isms." It is sprinkled with humor and practical advice.

This is not a textbook or guidebook for SysML application or MBSE deployment, nor does it describe in detail the methodological rationale for each of the systems engineering concepts it describes. While it does use a consistent satellite example through the chapters, it does not walk the reader through any particular MBSE process. It is not a workbook, nor does it include questions or sample problems for the reader to work out. You as a SysML user or advanced MBSE practitioner may eventually need these other resources, but this book is an excellent start.

This book is a solid, self-paced, lightweight SysML reference guide. The world is ready for this book.

—*Rick Steiner,*
coauthor, A Practical Guide to SysML

Foreword by Richard Soley

Technology Take-Up Takes Time

I had the great luck to attend one of the best technologically focused (and entrepreneurially focused) universities in the world in the 1970s and 1980s. The future, as Steve Jobs might have put it, was invented there, not discovered. It was one of the places where "hackers stayed up late" and helped to create radar, flash photography, and the Internet. Those technologies helped change the world; more importantly, economies flourished through the creation of companies and other organizations that put those technologies to work. The computing explosion that started in the 1960s certainly fared well in the Massachusetts of 1980.

My own contributions during my initial foray into the academic world, eleven years at the Massachusetts Institute of Technology, moved and changed as my academic interests moved and changed, starting with the artificial intelligence field (handwriting recognition was an early focus), moving on in graduate school to computing systems architectures, and finally melding those two interests. Not a small contribution to my focus was being involved in five start-ups during my MIT years (though perhaps it was a large contribution to the length of time I spent at MIT). Artificial intelligence pioneers like Symbolics and Gold Hill Computer were important to my understanding of the *application* of technology; and my own start-up, A.I. Architects (with likely the best systems engineer I have ever met), strongly depended on the collision between the demands of artificial intelligence and the limited computing power of the early personal computing revolution.

Probably the most important single idea that I learned during this period was that the time it takes for technology to come out of the laboratory and into production is far greater than any academic believes. The expert systems of the 1980s, now a primary fixture of diagnostic and other systems worldwide (though generally under the moniker of

"rule-based systems"), were clearly based on systems like PLANNER and CONNIVER from the 1960s. Twenty years seemed like the right rule of thumb; taking a technology through the engineering requirements necessary to stabilize and replicate the approach on an industrial scale, to the market development and integration, takes time.

OMG Objects of Awe

Nevertheless, when the Object Management Group (OMG) started up in 1989, the promise of *object technology* and *distributed objects* was to change the face of computing. As the Internet slowly changed into the World Wide Web, it was clear that consistent, standardized middleware would make it more possible than ever to integrate not only text pages from around the world but also application interoperability. The ability to "mash up" (as we would say twenty years later) computing power and data sources worldwide, using standardized APIs and on-the-wire protocols, would be far simpler with an object-oriented approach.

While OMG did a good job from the beginning in controlling the hype, avoiding the "artificial intelligence winter" that arose from an overhyped AI market in the 1980s, OMG likely didn't do a good enough job of recognizing that technology take-up takes time. It would be fifteen or twenty years before mash-ups became mash-ups, and object-oriented languages (initially C++, itself a good twenty years after Simular; now Java, C#, and Ruby-on-Rails) would permeate the computing milieu. OMG's objects of awe, as with all technologies, would become the quotidian tools of software developers everywhere, but it would take a couple of decades.

Modeling Makes Mavens

In the meantime, another opportunity would come OMG's way, with the proposal in 1996 that the object-oriented analysis and design market (as it was then called) had reached a dead end, an impasse, based not on the inherent technology but rather on the multitudinous approaches (and worse, notations) flooding the market. Even those technology mavens in love with the approach found themselves stymied by too much choice (and too little guarantee of portability and inter-

operability, once a choice was made). The mid-1990s consolidation of the analysis and design market created a vendor-focused market force for the creation of a standard, a force that was widely accepted by the slow-growing user community. The creation of the Unified Modeling Language (UML) standard in 1997, even with only a shared notation and not a shared methodology, was sufficient to coax the market into more than 100 percent CAGR over the next decade and a half. As the application development life cycle is more than just analysis and design, including also development, test, implementation, and maintenance, what had been just for "analysis and design" was soon called *modeling*.

And proof points for modeling, even with nascent standards like UML, abounded within a few years. Early scientific analysis showed 35 percent or more productivity increases using a modeling approach (as opposed to low-level programming language development); perhaps more importantly, as maintenance and support range from 80 percent to 90 percent of the software development life cycle, a couple of key analyses showed that 35 percent productivity increases (or better) could be had *in maintenance and integration*. This acceptance—as of this writing, according to market analysts Gartner & Forrester, including more than 71 percent of all software development teams—led to an explosion of modeling-related standardization at OMG.

Within fifteen years of the availability of the OMG UML standard (and its associated and very powerful parent, the Meta Object Facility, MOF), a fleet of domain-specific modeling languages were standardized. Languages and profiles for defining systems on a chip, for service-oriented architectures (SoaML), for business modeling and analysis (BPMN), for capturing enterprise architectures (UPDM), for defining rule-based systems (SBVR), even for capturing the motivations behind systems development (BMM), all joined the OMG stable. More importantly, most work at OMG shifted to "vertical markets," addressing the needs of healthcare information technology, financial services, life sciences, automotive and other consumer device dependability analysis, and so forth—all based on a view of systems based on high-level models.

Servicing the Spread of Systems

One of the most important horses in that stable is the OMG Systems Modeling Language, SysML. Defined as a "profile" of the UML, SysML

took on the huge task of being the language that could integrate many disparate views of large-systems engineering: not only software and hardware but also requirements, mathematical parameterization, facilities management, design for maintenance, even the management of human and other resources and the behavior of the system under design. The vision I had outlined in 2001, called *model driven architecture*, could come to fruition with such an approach to integrated engineering, and not just for "software architecture," but for the overall structure of complex systems like aircraft carriers and chemical plants. As the IDEF series of specifications had promised in the early 1980s, SysML could truly bring together the expertise necessary from many fields to build well-designed, fit-to-purpose, and *maintainable* large systems.

So here we are, a dozen years removed from the first mention of model driven architecture and coming up on the requisite twenty years since the delivery of the Unified Modeling Language, with a book in hand that integrates the views of experts on how to think about and how to *use* SysML to deliver real systems. Here we find SysML *distilled*: according to the dictionary, metaphorically, its essential meaning or most important aspects extracted and displayed for all to see.

Complex systems development is, by its nature, a team sport. No one person can manage even the gathering of requirements for large systems; the size alone makes such a project complex. Since the real focus of design is simplification along one or more dimensions, we need notations and processes that not only communicate the simplified vision but also allow designers, developers, and engineers to drill down into a system's design and explore, in fractal fashion, the underlying parts of the design, the expectations and requirements, and the integration methodology. It's one thing to know that a notation like SysML—large and complex itself, of course, and including many different tools in its toolbox—can support large systems development; it's quite another to get past the learning curve to be able to effectively use those tools. My father-in-law was well known for using a screwdriver for every handyman task around the house (including driving nails); I prefer to have tools that are fit for purpose and to understand how to use those tools in an integrated way. Further, the SysML modeling language is not intended only to implement large complex systems but also to *communicate their design* to users of those systems; to maintainers of those systems; and to those who may have to debug and integrate extensions, corrections, and changes to those systems.

This book presents that introduction to the toolbox; better, it explains how to use those tools together to gather requirements for, build

designs for, analyze designs of, and *communicate that process* to others in a design team (or future integration team). That's what engineers do, and SysML is the best way to do it.

—Richard Mark Soley, Ph.D.,
chairman and chief executive officer,
Object Management Group, Inc.

This page intentionally left blank

Preface

Why *SysML Distilled*? It's simple: You're busy. You need to know SysML now. You already have some systems modeling work to do. You don't need to know every detail of the language. You just want a book that focuses you on those parts of SysML that are most common and most useful in daily practice. *SysML Distilled* is that book.

You may choose to use this book as a desk reference, reaching for it when you're stuck and you've got a deadline bearing down on you. Or you may choose to dive deep one chapter at a time, adding new modeling skills to your toolbox for the future work coming your way. Or you may decide to read this book cover to cover to prepare for the first two levels of the *OMG Certified Systems Modeling Professional (OCSMP)* certification: *OCSMP Model User* and *OCSMP Model Builder: Fundamental*. This book is designed to serve you in all these ways.

Who Should Read This Book?

SysML is a graphical modeling language that you can use to visualize and communicate the designs of sociotechnical systems on all scales—systems consisting of hardware, software, data, people, and processes. Systems engineers are the ones who are responsible for the specification, analysis, design, verification, and validation of sociotechnical systems. Systems engineers—and students of systems engineering—are therefore the target audience for this book.

But that's an oversimplification. Many authors and teachers have repeated the axiom, "Everything is a system." Allow me to add the corollary: "Every engineer is a systems engineer." No matter your domain or job title, you've likely performed some or all of the systems engineering tasks I've mentioned. The premise of this book is that you can perform those activities more effectively via the standardized medium of an integrated SysML model than you can with

nonstandardized modes of communication in disjoint sets of documents and diagrams. You are a systems engineer—and you want to do your job more effectively. *You* are therefore the target audience for this book.

What do you need to know before you dive in? You should have at least a conceptual understanding of system specification, analysis, design, verification, and validation. Knowing in advance what these activities consist of will help you internalize the ways SysML can help you do them better. The International Council on Systems Engineering (INCOSE) *Systems Engineering Handbook* is the authoritative reference.

You do not need to have any experience with any modeling language to benefit from this book. You may already know that SysML is based on the Unified Modeling Language (UML). In fact, you may have read Martin Fowler's seminal book, *UML Distilled*. I designed *SysML Distilled* to be a companion book for systems engineers, who need to model a wider spectrum of systems beyond that subset—*software* systems—for which UML was created. With that said, you do not need to know UML as a prerequisite for this book. The structure and content of this book make it a self-sufficient primer for learning SysML.

Structure of the Book

This book contains twelve chapters and two appendixes. Chapter 1, "Overview of Model-Based Systems Engineering," introduces the concept of model-based systems engineering (MBSE) and provides the context and the business case for learning SysML. Chapter 2, "Overview of the Systems Modeling Language," discusses why SysML was created and introduces the nine kinds of SysML diagrams that you can create. Chapter 2 also covers general concepts that apply to all nine kinds of diagrams.

Chapters 3 through 11 zoom in on the details of each of the SysML diagrams, introducing you to the elements and relationships you can display on them. Although there's occasional overlap in the kinds of elements and relationships that can appear on these diagrams, I focus on each diagram one chapter at a time to effectively group related ideas and help you easily locate a particular topic when you need to. Chapters 3–11 are as follows:

- Chapter 3: "Block Definition Diagrams"
- Chapter 4: "Internal Block Diagrams"

- Chapter 5: "Use Case Diagrams"
- Chapter 6: "Activity Diagrams"
- Chapter 7: "Sequence Diagrams"
- Chapter 8: "State Machine Diagrams"
- Chapter 9: "Parametric Diagrams"
- Chapter 10: "Package Diagrams"
- Chapter 11: "Requirements Diagrams"

The last chapter, Chapter 12, "Allocations: Cross-Cutting Relationships," covers the concept of allocations—a relationship that you can use to relate elements across all nine kinds of SysML diagrams.

The sample diagrams in the figures present various aspects of a single system, the DellSat-77 Satellite System—a system I conceived of entirely for the purpose of writing this book (and I hereby certify that I have not disclosed any proprietary information of any aerospace companies). I chose to focus on a satellite system to demonstrate how you can use SysML to model a complex, real-world sociotechnical system—one other than the classic exemplars (ATMs and cruise control systems) that seem to be prevalent in modeling workshops. And I chose to use a single system as a running example threaded through all chapters to show you how the nine kinds of SysML diagrams present complementary and consistent views of an underlying system model.

The SysML model of the DellSat-77 Satellite System is available for download from my website, www.lennydelligatti.com, on the "Articles and Publications" page. I have made the data files available both in XMI format and in the native formats of various modeling tools. This resource enables self-learners as well as instructors and their students to get hands-on with the system model that appears throughout this book in the modeling tool of their choice.

Appendix A, "SysML Notation Desk Reference," is a concise summary of the graphical notations presented in this book, along with references to the sections where they are discussed in detail. Appendix B, "Changes between SysML Versions," covers the kinds of elements that are introduced in SysML v1.3, the latest version of SysML at the time of this writing.

SysML v1.2 is the version of SysML that is currently assessed on the OCSMP certification exams. The biggest differences between SysML v1.2 and v1.3 are in *ports*—a kind of element that can appear on block definition diagrams (BDDs) and internal block diagrams (IBDs). I cover

BDDs in Chapter 3 and IBDs in Chapter 4. In these chapters, I focus on the SysML v1.2 definition of ports for three reasons:

- They are the predominant kinds of ports in system models on modeling projects that started before the release of SysML v1.3—and many of those projects are still active.
- Some modeling tools lag behind the changes in SysML and have not yet implemented the SysML v1.3 definition of ports.
- The OCSMP certification exams have not been revised since the release of SysML v1.3 and still cover the SysML v1.2 definition of ports.

Never fear, though; I give the SysML v1.3 definition of ports full coverage in Appendix B. If your modeling team is about to create a new system model, I recommend using the new kinds of ports instead of the old ones (assuming your SysML modeling tool supports them).

The order of the chapters is loosely based on the typical frequency of use of the diagrams. It does not reflect the relative value of each kind. It can't. Value is a subjective thing. Your team will determine that based on the modeling method you adopt and the deliverables you produce for your customer.

The order of the chapters also does not reflect—and should not suggest—any particular modeling method. Simply put, this is not a methodology book; rather, it's a language book. In Chapter 1, "Overview of Model-Based Systems Engineering," I discuss the distinction between modeling methods and modeling languages. I list a few well-known modeling methods and point to references that discuss them comprehensively.

My goal in this book is to present you with concise, targeted coverage of the most common and most useful features of SysML—features that are useful no matter which modeling method your team adopts. A key point is that SysML is only a language; it's method independent. I designed *SysML Distilled* to be method independent as well. I want you to come away knowing that SysML is a value-added medium for communication no matter which processes, procedures, or tools your team adopts to do your work and meet your stakeholders' needs.

I hope you find this book a valuable companion in your study of SysML. It's a rich, expressive language—one with enough breadth and depth to let you visualize and communicate all aspects of a system's design. There's a lot to know, but you don't need to know all of it to

create system models that communicate clearly and effectively. Dive in and get what you need. You'll discover how quickly you can put that knowledge to work and deliver value to your customer.

—Lenny Delligatti
Houston, Texas
October 2013

This page intentionally left blank

Acknowledgments

Many talented and dedicated people deserve credit for producing this book. I would like to begin with a special thanks to Jim Thompson—my friend, colleague, spiritual adviser, and partner in weekly sushi catharsis. He spent many months reviewing chapters as I wrote the original manuscript, and he provided valuable and insightful feedback. This book benefited greatly from his keen technical mind and his excellent communication skills.

A special thanks also to Chris Guzikowski at Addison-Wesley. He shepherded this project from its inception and flattened the steep learning curve for this new author. I thank him particularly for the sage advice that got me to the finish line: "Just keep choppin' away at it."

Chris Zahn, development editor at Addison-Wesley, and Betsy Hardinger, copy editor extraordinaire, provided the essential support I needed to hammer this book into its present form. They taught me the art of turning good ideas into good writing and a well-crafted manuscript. The quality of this book is far greater because of their contributions.

Elizabeth Ryan, project editor at Addison-Wesley, coordinated the work of the production team, who created the layout for the book and brought the various pieces together in preparation for printing. They made a complex process look easy and created a polished final product. I'm grateful to them for their hard work.

I'd like to extend my deep appreciation to the exceptional team of engineers and systems modelers who served as technical reviewers for this book: Celso Gonzalez, Robert Cloutier, Susanne Sherba, John Pantone, Michael Engle, and Michael Chonoles. Their expertise and insight enabled me to turn a very rough draft into a significantly more focused final product—one that better serves the systems engineering community. My thanks to all of them.

I would also like to thank the following individuals who graciously agreed to review the revised manuscript and provide endorsements on short notice: Jeff Estefan, Susanne Sherba, Lonnie VanZandt,

Bran Selic, J. D. Baker, Tim Weilkiens, Tom Fargnoli, Robert Cloutier, Matthew Hause, Russell Peak, Doug Tolbert, Celso Gonzalez, and Bobbin Teegarden. These extraordinary people have made significant contributions to the systems modeling community as modeling language developers, modeling certification developers, systems architects, teachers, and expert practitioners in the field. I am honored and humbled to receive their endorsements.

A special thanks to Rick Steiner for writing a foreword for this book. Rick is one of the original creators of SysML, and he continues to serve on the SysML Revision Task Force (RTF)—the team that evolves and improves the SysML specification over time in response to feedback from the systems modeling community. Our profession is richer because of his experience and contributions. My thanks to him for all he's done.

A special thanks also to Richard Soley for his foreword. Richard has led the Object Management Group (OMG) since its inception in 1989. The mark that the OMG has left in the engineering world cannot be overstated. Richard, the OMG staff, and the pantheon of expert engineers who have served as volunteer members of OMG working groups have—without hyperbole—transformed the way we do engineering. The creation of the model-based engineering paradigm and its infusion into the work we do in our profession have enriched those of us who love and practice engineering as well as the customers we serve. My thanks to Richard for his vision and leadership as our community continues to navigate this sea change.

Many brilliant, experienced engineers have contributed to the development of SysML for more than a decade. Any attempt on my part to give each of them due credit individually would fail. I have to resign myself to thanking them collectively for their efforts. It's because of their hard work that we have this rich medium called SysML to communicate our system design ideas to one another. I've strived to make this book a worthy representative of that product.

I am most thankful to my wife, Natalie, and my children, Noelle and Aidan. For two years, they were extraordinarily patient and understanding when I spent long hours on the computer in the evenings and on weekends. They give my life purpose, and I'm deeply grateful to them for their love and support.

About the Author

Lenny Delligatti received his B.S. in electrical and computer engineering from Carnegie Mellon University and his M.S. in computer science systems engineering from the University of Denver. He holds the *OMG Certified Systems Modeling Professional (OCSMP) Model Builder: Advanced* certification, the highest level of certification in SysML and model-based systems engineering (MBSE) methodology. Additionally, he holds the *OMG Certified UML Professional (OCUP): Advanced* certification, the highest level of certification in UML.

Lenny is a senior systems engineer with Lockheed Martin, creating SysML models and serving as the MBSE lead for NASA's Mission Control Center: 21st Century (MCC-21) project at Johnson Space Center. He previously served as an embedded software engineer on NASA's Aircraft Simulation Program (ASP), building VxWorks kernels and developing flight software for NASA's fleet of Gulfstream II in-flight space shuttle simulators. He also served as a software engineer on the Nomad project at Carnegie Mellon University's Field Robotics Center, designing and developing the Sensor Manager subsystem for the Nomad Autonomous Rover.

Lenny is a member of the Object Management Group (OMG) SysML Revision Task Force (RTF) and the OCUP2 Certification Development Team. He also serves as the Education and Outreach Director for the Texas Gulf Coast Chapter (TGCC) of the International Council on Systems Engineering (INCOSE), supporting the professional development of the Houston-area systems engineering community.

In addition to his engineering experience, Lenny served as a Surface Warfare Officer in the U.S. Navy, completing a deployment in support of Operation: Enduring Freedom and two tours of duty in Sasebo, Japan, and Norfolk, Virginia. Following his Navy service, he received formal training in pedagogy at Old Dominion University and earned a license to teach mathematics in the state of Virginia. He served as a mathematics teacher and department head in the Fairfax County public school system before transitioning back into engineering upon his move to Houston, Texas.

Lenny is passionate about engineering and helping engineers develop more effective ways to do engineering. He has created and delivered hundreds of hours of classroom instruction to hundreds of systems and software engineers on the topics of UML, SysML, and MBSE, enabling many to earn OMG certifications and lead MBSE efforts on their projects. He has delivered SysML and MBSE presentations at INCOSE meetings and at American Institute of Aeronautics and Astronautics (AIAA) Technical Symposia at Johnson Space Center.

This page intentionally left blank

Chapter 3

Block Definition Diagrams

The most common kind of SysML diagram is the block definition diagram. You can display various kinds of model elements and relationships on a BDD to express information about a system's structure. You can also adopt design techniques for creating extensible system structures, a practice that reduces the time and cost to change your design as your stakeholders' needs evolve.

3.1 Purpose

The model elements that you display on BDDs—blocks, actors, value types, constraint blocks, flow specifications, and interfaces—serve as types for the other model elements that appear on the other eight kinds of SysML diagrams. We refer to elements that appear on BDDs as **elements of definition**. Elements of definition, in a real sense, form the foundation for everything else in your system model. That's why I'm covering BDDs first.

Elements of definition are important; the structural relationships among them—associations, generalizations, and dependencies—are arguably more important. You display these relationships on BDDs, too. With these relationships, you often create BDDs that convey system decomposition and type classification.

3.2 When Should You Create a BDD?

Often. You should create a BDD often.

That may seem like a glib answer, but it's accurate. BDDs are not tied to any particular stage of the system life cycle or level of design. You and your team will create them (and refer to them) when you perform all the following systems engineering activities: stakeholder needs analysis, requirements definition, architectural design, performance analysis, test case development, and integration. And you often create a BDD in conjunction with other SysML diagrams to provide a complementary view of an aspect of your system of interest.

In short, you should—and will—create BDDs often.

3.3 The BDD Frame

The diagram kind abbreviation for a block definition diagram is *bdd.* The model element type that the diagram frame represents can be any of the following:

- *package*
- *model*
- *modelLibrary*
- *view*
- *block*
- *constraintBlock*

As discussed in Section 2.4, "General Diagram Concepts," the model element that the diagram represents serves as the namespace for the other elements shown on the diagram. A **namespace** is simply a model element that's allowed to contain other model elements; that is, it can have other elements nested under it within the model hierarchy. A namespace, therefore, is a concept that has meaning only within your system model; it has no meaning within an instance of your system.

Many kinds of SysML elements can serve as namespaces. A **package,** however, is the most common kind of namespace for the various elements of definition that appear on BDDs. Therefore, the element that's named in the header of a BDD typically is a package you've created somewhere in the model hierarchy.

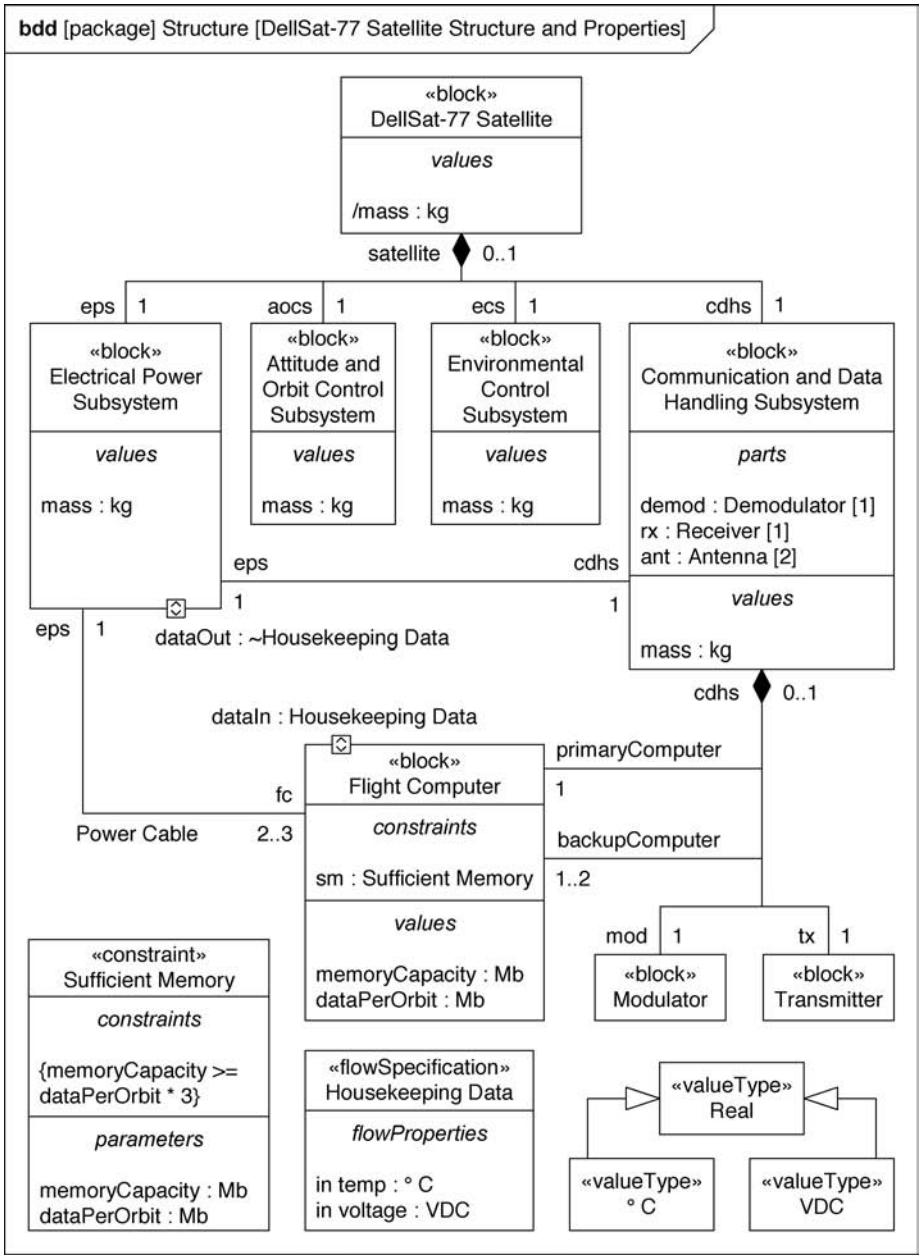

Figure 3.1 *A sample block definition diagram (BDD)*

The name of the BDD in Figure 3.1 is "DellSat-77 Satellite Structure and Properties." The diagram header also tells us that this diagram represents the *Structure* package in the system model. The *Structure* package, therefore, is the namespace for the elements shown on the diagram.

Let's take a look in detail at the kinds of elements and relationships you can display on a BDD.

3.4 Blocks

A **block** is the basic unit of structure in SysML. You can use a block to model any type of entity within your system of interest or in the system's external environment.

Note the distinction between *definition* and *instantiation* (which SysML refers to as "usage"). This distinction is one of the most fundamental system design concepts, and it's a pattern that recurs often in SysML. Some kinds of model elements (e.g., blocks, value types, constraint blocks) represent **definitions** of types; other kinds of model elements (e.g., part properties, value properties, constraint properties) represent **instances** of those types. By analogy, a blueprint of a house is a definition of a type of house; each house a developer builds on a plot of land in accordance with that blueprint is a distinct instance of that type.

With that in mind, I reiterate: A block represents a type of entity, and not an instance. For example, you could create a block named *DesktopWorkstation* in your system model. That block would represent a type that defines a set of properties—such as monitor, keyboard, mouse, CPU, manufacturer, disk space, cost—that are common to all instances. Each desktop workstation that your IT department purchases for each office and cubicle would be a distinct instance of that *DesktopWorkstation* block.

You can easily tell the difference between elements of definition and elements of usage in a system model. Elements of definition have a name only (e.g., *DesktopWorkstation*); elements of usage have a name and a type, separated by a colon (e.g., *SDX1205LJD : DesktopWorkstation*).

The notation for a block is a rectangle with the stereotype «block» preceding the name in the name compartment (as shown in Figure 3.2). You're required to display a block's name compartment. Often you'll display additional optional compartments that convey the features of the block.

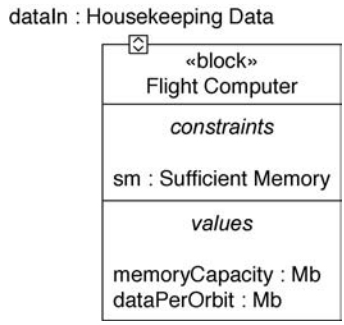

Figure 3.2 *A block*

Features come in two varieties: structural features (also known as properties) and behavioral features. I discuss each category in depth in the next two sections.

Here are the optional compartments that you can display:

- Parts
- References
- Values
- Constraints
- Operations
- Receptions
- Standard ports (in SysML v1.2 and earlier)
- Flow ports (in SysML v1.2 and earlier)
- Full ports (in SysML v1.3)
- Proxy ports (in SysML v1.3)
- Flow properties (in SysML v1.3)
- Structure

The structure compartment is the only compartment that doesn't list features. Rather, it's a graphical compartment that displays a block's internal structure; you can display in that compartment all the same notations you can display on an internal block diagram (IBD). Modelers rarely display this compartment.

Note that even though it's legal to display a block's ports in compartments, it's much more common to display ports as small squares that straddle the border of a block (as shown in Figure 3.2). I discuss ports in detail in Section 3.4.1.5, "Ports."

3.4.1 Structural Features

There are five kinds of **structural features** (also known as **properties**) that a block can own:

- Part properties
- Reference properties
- Value properties
- Constraint properties
- Ports

3.4.1.1 Part Properties

Part properties are listed in the parts compartment of a block (as shown in Figure 3.3). A **part property** represents a structure that's internal to a block. Stated differently, a block is *composed of* its part properties. This relationship conveys ownership.

However, SysML stops short of defining the word *ownership;* this concept has different meanings in different domains. In the hardware domain, ownership typically refers to physical composition. For example, Figure 3.3 conveys that a valid instance of the *Communication and Data Handling Subsystem* block is one that is physically composed of the required parts: flight computers, modulator, demodulator, transmitter, receiver, and antennas. In the software domain, however, ownership

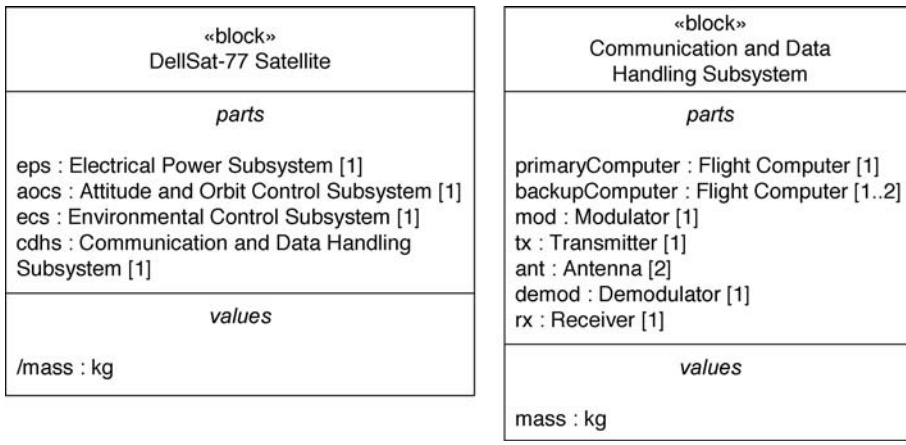

Figure 3.3 *Blocks with part properties*

typically refers to one object's responsibility for the creation and destruction of another object. When memory is allocated for a composite object, memory is allocated for each of its parts, too; similarly, when memory is freed for a composite object, memory is also freed for each of its parts.

But SysML states definitively that *ownership* means that a part property can belong to only one composite structure at a time. However, a part property can be removed from one instance of a composite structure and added to another. For example, I can install a given antenna on only one satellite at a time, and not on two or more simultaneously. But that antenna can be removed from one satellite and reinstalled on another at some point.

When you list a part property in the parts compartment of a block, it appears as a string with the following format:

```
<part name> : <type> [<multiplicity>]
```

The part name is modeler defined. The type generally is the name of a block that you've created somewhere in the system model. The multiplicity is a constraint on the number of instances that the part property can represent within the composite, expressed either as a single integer or as a range of integers.

For example, Figure 3.3 conveys that a valid instance of the *Communication and Data Handling Subsystem* block must be composed of exactly one instance of the *Flight Computer* block—an instance that serves in the role of *primaryComputer.* Additionally, it must be composed of either one or two more instances of *Flight Computer*—instances that serve in the role of *backupComputer.*

If you want a part property to represent an unconstrained number of instances, you can set the multiplicity to 0..*. The asterisk means that there's no upper bound (or more precisely, that you're not specifying an upper bound in the system model). You would read 0..* in English as "zero or more." Alternatively, you can set the multiplicity to *, a shorthand notation for 0..*.

If no multiplicity is shown for a part property, the default is 1 (which is equivalent to 1..1). Note that 1 is almost always the default multiplicity in SysML. There is an important exception, however, which I discuss in Section 3.5.2, "Composite Associations."

When a part property has a multiplicity with an upper bound greater than 1 (e.g., 1..2, 0..10, *), we refer to that part property as a **collection** (of instances). The key idea is that *part property* and *instance* are

not synonyms; a single part property may potentially represent multiple instances within a composite if its specified multiplicity allows it.

3.4.1.2 Reference Properties

Reference properties are listed in the references compartment of a block (as shown in Figure 3.4). A **reference property** represents a structure that's external to a block.

Unlike a part property, a reference property does not convey ownership. A reference property can roughly be described as a "needs" relationship; a block with a reference property needs that external structure for some purpose, either to provide a service or to exchange matter, energy, or data. And this implies that some type of connection must exist between them.

Note that the presence of a reference property in a block does not by itself convey its purpose. If you need to convey that purpose, you could do so on an internal block diagram (IBD). I discuss this more in Chapter 4, "Internal Block Diagrams."

When you list a reference property in the references compartment of a block, it appears as a string with the following format:

```
<reference name> : <type> [<multiplicity>]
```

The reference name is modeler defined. The type must be the name of a block or actor that you've created somewhere in the system model. The multiplicity is a constraint on the number of instances that the reference property can represent.

For example, Figure 3.4 shows that the *Electrical Power Subsystem* block has a reference property named *cdhs.* This model conveys that an

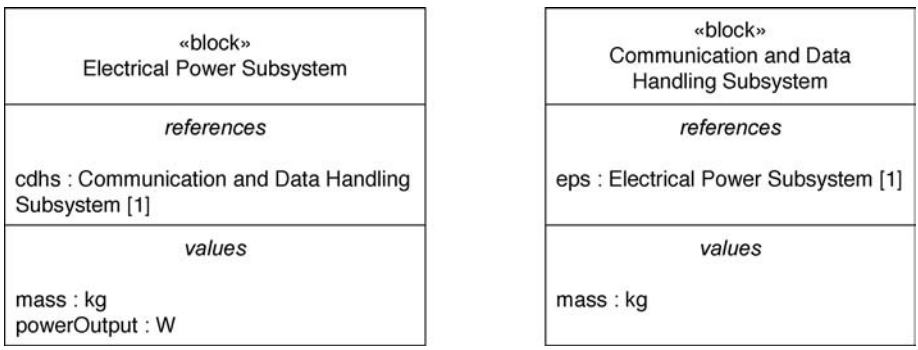

Figure 3.4 *Blocks with reference properties*

instance of *Electrical Power Subsystem* needs exactly one instance of *Communication and Data Handling Subsystem* (to fulfill its design purpose). Again, this view alone doesn't convey what that purpose is; it simply conveys that some type of connection must exist between them.

Like a part property, a reference property's default multiplicity is 1 (if no multiplicity is shown). And like a part property, a reference property is referred to as a collection when its multiplicity has an upper bound greater than 1.

3.4.1.3 Value Properties

Value properties are listed in the values compartment of a block (as shown in Figure 3.5). A **value property** can represent a quantity (of some type), a Boolean, or a string. Most often, though, a value property is something you can assign a number to. Value properties are particularly useful in conjunction with constraint properties to construct a mathematical model of your system (more on this in Chapter 9, "Parametric Diagrams").

When you list a value property in the values compartment of a block, it appears as a string with the following format:

```
<value name> : <type> [<multiplicity>] = <default value>
```

The value name is modeler defined. The type must be the name of a value type that you've created somewhere in the system model. The multiplicity is a constraint on the number of values that the value property can hold. The default value is an optional piece of information; it

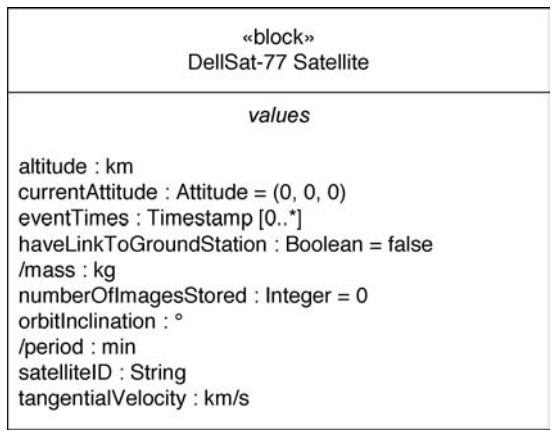

Figure 3.5 *A block with value properties*

represents the value assigned to the value property when an instance of its owning block first gets created.

Figure 3.5 shows that the *DellSat-77 Satellite* block has several value properties. The *eventTimes* value property can hold an unconstrained number of *Timestamp* values (as conveyed by the multiplicity 0..*). *Timestamp* is a value type that exists somewhere in the model hierarchy (more on value types in Section 3.9, "Value Types").

As with a part property and a reference property, a value property's default multiplicity is 1 (if no multiplicity is shown). Similarly, a value property is referred to as a collection when its multiplicity has an upper bound greater than 1.

Some value properties hold values that are assigned, and others hold values that are derived (calculated) from other value properties in the system model. To convey that a value property is derived, you put a forward slash (/) in front of its name. For example, Figure 3.5 shows that the *DellSat-77 Satellite* block owns two derived value properties: *mass* and *period*. This view of the model does not convey the equations used to calculate those derived values, nor does it show which other value properties provide inputs for those equations. You would specify those mathematical relationships using constraint expressions, as discussed in the next section.

3.4.1.4 Constraint Properties

Constraint properties are listed in the constraints compartment of a block (as shown in Figure 3.6). A **constraint property** generally represents a mathematical relationship (an equation or inequality) that is imposed on a set of value properties. This is a higher level of model fidelity than is required on most modeling projects. However, constraint properties are an essential part of constructing mathematical models of

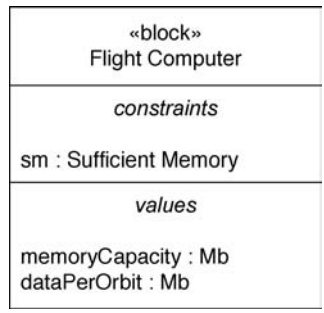

Figure 3.6 *A block with a constraint property*

a system, which you display on parametric diagrams (more on this in Chapter 9).

When you list a constraint property in the constraints compartment of a block, it appears as a string with the following format:

```
<constraint name> : <type>
```

The constraint name is modeler defined. The type must be the name of a constraint block that you've created somewhere in the system model.

A constraint block is simply a special kind of block—one that you create to encapsulate a reusable **constraint expression**. Most often, a constraint expression is an equation or an inequality. For example, Figure 3.7 shows a constraint block named *Sufficient Memory*, which encapsulates the constraint expression

```
memoryCapacity >= dataPerOrbit * 3
```

This constraint block serves as the type for the constraint property *sm* in the *Flight Computer* block (shown in Figure 3.6). This conveys that the values held in the two value properties (*memoryCapacity* and *dataPerOrbit)* must satisfy that mathematical relationship at all times (in a system that's operating nominally).

Note that you're not required to use constraint blocks to impose mathematical relationships on value properties. It's perfectly legal to specify a constraint expression directly in the constraints compartment of a block (as shown in Figure 3.8). You would do this when only one block needs that constraint expression (i.e., when you don't intend to

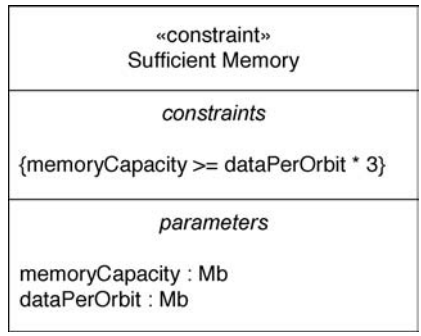

Figure 3.7 *A constraint block*

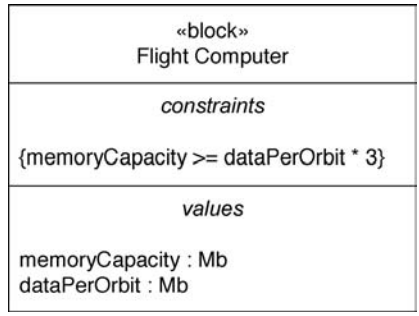

Figure 3.8 *A block with a (non-reusable) constraint*

reuse it in multiple places). As a matter of best practice, though, I recommend that you always encapsulate equations and inequalities in constraint blocks; it enables reuse if the need arises.

I discuss constraint blocks in greater detail in Section 3.10, "Constraint Blocks." Meanwhile, keep in mind these key ideas:

- Blocks can own constraint properties (to constrain value properties).
- Constraint properties are typed by constraint blocks, which generally encapsulate mathematical relationships.

3.4.1.5 Ports

A **port** is a kind of property that represents a distinct interaction point at the boundary of a structure through which external entities can interact with that structure—either to provide or request a service or to exchange matter, energy, or data.

When you add a port to a block, you're modeling a structure as a black box with respect to its environment; the structure's internal implementation is hidden from its clients. Those clients know only the structure's interface (the services it provides and requires, and the types of matter, energy, or data that can flow in and out). Stated differently, a port **decouples** a block's clients from any particular internal implementation.

Encapsulating a block with a set of ports enables you to redesign that block's internal implementation later without impacting the design of the other parts of your system. This practice reduces the time it takes to implement system modifications when the customer's require-

ments change later in the life cycle, and time saving translates into cost saving.

A port can represent any type of interaction point you need to model. For example, it can represent a physical object on the boundary of a hardware object (e.g., a spigot, an HDMI jack, a fuel nozzle, a gauge). It can represent an interaction point on the boundary of a software object (e.g., a TCP/IP socket, a message queue, a shared memory segment, a graphical user interface, a data file). And it can represent an interaction point between two business organizations (e.g., a purchase order, a courier, a website, a mailbox). SysML imposes no constraints on what a port can represent.

SysML v1.2 (and earlier) defines two kinds of ports—standard ports and flow ports—that you can add to a block to specify different aspects of its interface. A **standard port** lets you specify an interaction point with a focus on the services that a block provides or requires; a **flow port** lets you specify an interaction point with a focus on the types of matter, energy, or data that can flow in and out of a block.

Note

SysML v1.3 no longer supports standard ports and flow ports, instead defining two new kinds of ports: full ports and proxy ports. I discuss these in detail in Appendix B. I focus on standard ports and flow ports in this chapter because they continue to be the predominant kinds of ports in system models at the time of this writing. Additionally, the current versions of the OCSMP certification exams cover the concepts of standard ports and flow ports. Moreover, some modeling tools continue to lag behind the changes in SysML and do not yet support full ports and proxy ports.

Standard Ports A standard port models the services (behaviors) that a block provides or requires at an interaction point on its boundary. Most often, you display a standard port as a small square straddling the border of a block (as shown in Figure 3.9). Note that it's legal to list a standard port as a string in the standard ports compartment, but this is an uncommon notation.

A standard port can have a modeler-defined name (e.g., *sp_cdhs*, *sp_eps*) that is displayed as a string floating near the standard port (either inside or outside the block border). A standard port can have

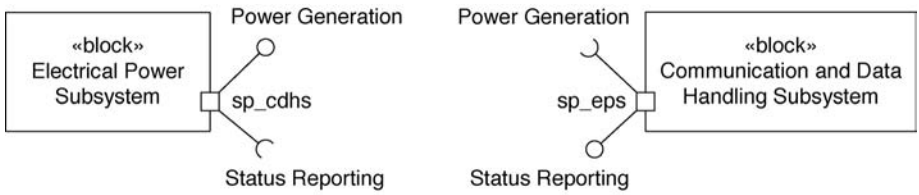

Figure 3.9 *Blocks with standard ports*

one or more types; the types are the **interfaces** you assign to it (e.g., *Power Generation*, *Status Reporting*).

An interface, like a block, is an element of definition—one that defines a set of **operations** and **receptions,** a behavioral contract that clients and providers will conform to. You can display an interface on a BDD as a rectangle with the keyword «interface» preceding the name; you can display its operations and receptions in the second and third compartments. Figure 3.10 displays the *Power Generation* and *Status Reporting* interfaces using this notation.

When you assign an interface to a standard port, you assign it either as a provided interface or as a required interface. A **provided** interface is displayed using the ball notation—the lollipop symbol attached to the standard port (shown in Figure 3.9). A block that provides an interface must implement *all* of the interface's operations and receptions. For example, Figure 3.9 conveys that the *Communication and Data Handling Subsystem* block provides the *Status Reporting* interface, and

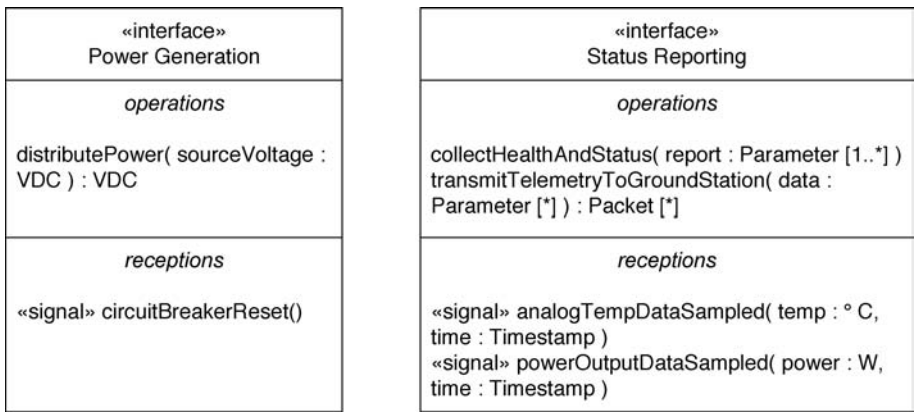

Figure 3.10 *Interfaces*

this means that it implements (can perform) the two operations and the two receptions in that interface.

A **required** interface is displayed using the socket notation—the stick with a semicircle attached to the standard port (shown in Figure 3.9). A block that requires an interface may invoke one or more—but not necessarily all—of its operations or receptions at some point during system operation. For example, Figure 3.9 conveys that the *Electrical Power Subsystem* block requires the *Status Reporting* interface, and this means that it may invoke any (or all) of the four operations and receptions in that interface.

Modeling with standard ports and interfaces is a way to decouple clients and providers, enabling you to design to abstractions rather than specific implementations. This **extensibility** lets you add new providers of interfaces at any time without impacting the existing clients of those interfaces.

Flow Ports A **flow port** models the types of matter, energy, or data that can flow in or out of a block at an interaction point on its boundary. As with a standard port, you most often display a flow port as a small square straddling the border of a block (as shown in Figure 3.11). Unlike a standard port, however, a flow port has a symbol shown inside the small square (more on that soon). It's legal to list a flow port as a string in a compartment—one named "flow ports"—but again, this is an uncommon notation.

A flow port can have a modeler-defined name (e.g., *dataOut*, *dataIn*); it can also have a type (e.g., *Housekeeping Data*). The name and type are displayed as a string floating near the flow port, separated by a colon in the format *name : type.* The type that you specify for a flow port and the symbol that appears inside the square depend on the kind of flow port you're modeling. SysML offers two kinds of flow ports: nonatomic flow ports and atomic flow ports.

Figure 3.11 shows examples of nonatomic flow ports. You add a **nonatomic** flow port (symbolized as < >) to a block when you need to

Figure 3.11 *Blocks with nonatomic flow ports*

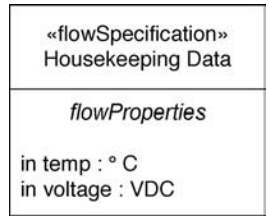

Figure 3.12 *A flow specification*

model *multiple* types of items that could flow in or out via that port. The type of a nonatomic flow port must be the name of a flow specification that you've created somewhere in the system model.

Like a block, a **flow specification** is an element of definition—one that defines a set of **flow properties** that can flow in or out of a nonatomic flow port. You can display a flow specification on a BDD as a rectangle with the stereotype «flowSpecification» preceding the name; you can display its flow properties in a compartment named "flowProperties." Figure 3.12 displays the *Housekeeping Data* flow specification using this notation.

A flow property represents a specific item that can flow in or out of a block via a flow port. Each flow property has a direction, a name, and a type, which are displayed as a string in the following format:

```
<direction> <name> : <type>
```

The direction can be *in*, *out*, or *inout*. The name is modeler defined. The type must be the name of a value type, block, or signal that you've created somewhere in your model hierarchy.

Figure 3.11 shows that the *Flight Computer* block owns a nonatomic flow port named *dataIn*, which is typed by the *Housekeeping Data* flow specification. This model conveys that temperature and voltage values can flow into an instance of *Flight Computer* at some point during system operation.

Figure 3.11 also shows that the *Electrical Power Subsystem* block owns a nonatomic flow port named *dataOut*, which also is typed by the *Housekeeping Data* flow specification. In this case, though, the type, *Housekeeping Data*, has a tilde (~) in front of it. This symbol conveys that the *dataOut* flow port is **conjugated**. This means that the directions of the flow properties in the *Housekeeping Data* flow specification are reversed for that flow port.

Figure 3.13 *Blocks with atomic flow ports*

The other kind of flow port is an atomic flow port. Figure 3.13 shows examples of this kind. You add an **atomic** flow port to a block when you need to model a *single* type of item that could flow in or out via that port. The symbol inside the small square is an arrow that conveys the direction of flow. The type of an atomic flow port must be the name of a value type, block, or signal that you've created somewhere in your model hierarchy.

Figure 3.13 shows that the *Modulator* block and the *Transmitter* block have an atomic flow port named *coupler,* which is typed by the same value type, *Radio Frequency Cycle*. These ports differ only in their direction of flow. This model conveys that a radio frequency signal can flow from a modulator to a transmitter via a coupler—an interaction point at their respective boundaries.

3.4.2 Behavioral Features

All the features I discuss in the preceding section are structural features. On most modeling projects, however, it's not sufficient to specify only the parts, references, constraints, value properties, and ports of a block. They're important, but they convey only one aspect of the design. An equally important aspect is the set of **behaviors** that a block can perform. You convey this aspect of the design by adding behavioral features to a block.

SysML offers two kinds of behavioral features: operations and receptions. I discuss these briefly in the context of interfaces earlier in Section 3.4.1.5. However, they're not limited to interfaces; you can also add operations and receptions to blocks. The decision to add a behavioral feature to a block directly or to an interface (that a block provides or requires) is a matter of your chosen modeling methodology and design principles. SysML does not dictate either course of action, and the format for displaying operations and receptions is the same in either case.

Now let's take a look in detail at each kind of behavioral feature.

3.4.2.1 Operations

An **operation** represents a behavior that a block performs when a client calls it. Stated formally, an operation is invoked by a **call event**.

> **Note**
>
> The term *call event* becomes more meaningful when I discuss events in detail in the context of behaviors in Chapter 6, "Activity Diagrams," Chapter 7, "Sequence Diagrams," and Chapter 8, "State Machine Diagrams." I introduce the term now to establish its connection to the concept of operations. As you study SysML incrementally, remember that diagrams are merely views of an underlying model; what you see on BDDs is related to what you see on other kinds of diagrams, including activity diagrams, sequence diagrams, and state machine diagrams.

Most often, an operation represents a **synchronous** behavior. This means that the caller waits for the behavior to complete before continuing with its own execution. However, SysML doesn't require this; you're free to represent any behavior as an operation—even when the caller doesn't wait for it to complete.

You display an operation on a BDD as a string in the operations compartment of a block (as shown in Figure 3.14). That string has the following format:

```
<operation name> ( <parameter list> ) : <return type>
  [<multiplicity>]
```

The operation name is modeler defined. The parameter list is a comma-separated list of zero or more parameters. (The format for each parameter is shown shortly.) The return type (if any) must be the name of a value type or block that you've created somewhere in your system model. The multiplicity is a constraint on the number of instances of the return type that the operation can return to the caller when it completes.

The parameters in the parameter list represent the inputs or outputs of the operation. Each parameter in the list is displayed with the following format:

```
<direction> <parameter name> : <type> [<multiplicity>] =
  <default value>
```

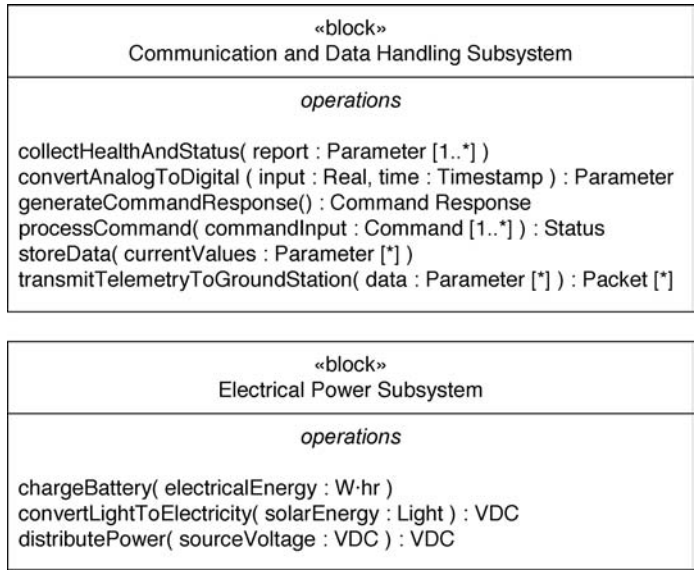

Figure 3.14 *Blocks with operations*

The direction can be *in, out,* or *inout.* The parameter name is modeler defined. The type must be the name of a value type or block that exists somewhere in your model. The multiplicity is a constraint on the number of instances of the type that the parameter can represent. The default value is the value assigned to the parameter if no value is specified as an argument when the operation is called.

Figure 3.14 shows that the *Electrical Power Subsystem* block and the *Communication and Data Handling Subsystem* block own several operations each—operations that represent behaviors that instances of these blocks can perform if called upon during system operation. An example of an operation is *processCommand*. This model conveys that a client can call the communication and data handling subsystem to perform this operation. When it does, the client can pass one or more commands as an input to the operation. And when the operation completes, it will return a status value to the caller.

A bit of advice: It's good practice to always use a verb phrase (such as *processCommand*) to name an operation; an operation represents a behavior, after all. Also, don't go overboard with the parameter lists; simply adding operations to blocks (without specifying parameters) is often a sufficient degree of model fidelity. If your team needs to specify

parameters for operations within the system model, be judicious about which ones you choose to display on any given BDD; the complete string for an operation could take up a lot of real estate on a BDD if you display even a few parameters.

3.4.2.2 Receptions

A **reception** represents a behavior that a block performs when a client sends a signal that triggers it. Stated formally, a reception is invoked by a **signal event**.

The key distinction between a reception and an operation is that a reception always represents an *asynchronous* behavior. This means that a client sends a signal—which triggers a reception upon receipt—and immediately continues with its own execution; it doesn't wait for the reception to complete (or even, necessarily, to begin).

Another key point is that a **signal** is itself a model element. You can use a signal to represent any type of matter, energy, or data that one part of a system sends to another part—generally for the purpose of triggering a behavior on the receiving end. Like a block, a signal can own properties. Most often, those properties represent data that the signal carries from a client to a target. And when the signal arrives at the target and triggers a reception, the signal's properties become inputs to that reception.

Figure 3.15 displays a signal named *AnalogTempDataSampled*. This signal owns two properties: *temp* (of type ° *C*) and *time* (of type *Timestamp*). When a client generates an instance of this signal during system operation, it can supply values for the two properties. The client can send the signal instance to a target that's receptive to it (e.g., the *Communication and Data Handling Subsystem* block shown in Figure 3.16).

A structure is an eligible target for a signal if it owns a reception that has the same name as the signal. Additionally, the reception must

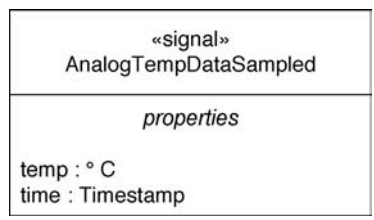

Figure 3.15 *A signal*

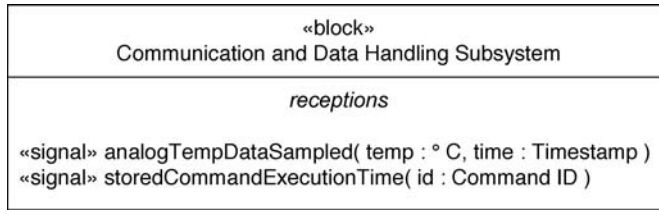

Figure 3.16 *A block with receptions*

have a parameter with a compatible type for each property of the signal. The *Communication and Data Handling Subsystem* block meets these criteria. When an instance of this block in an operational system receives an instance of the *AnalogTempDataSampled* signal, the reception behavior gets invoked, and the values held in the signal's two properties become inputs to that behavior.

When you display a reception in the receptions compartment of a block, the string has the following format:

```
«signal» <reception name> ( <parameter list> )
```

The keyword «signal» must always precede the reception name. As mentioned earlier, the reception name must match the name of the signal in your model that triggers it. You can display as many parameters as necessary in the parameter list. Each parameter in the list is displayed with the following format:

```
<parameter name> : <type> [<multiplicity>] = <default value>
```

The parameter name is modeler defined. The type must be the name of a value type or block that exists somewhere in your model. The multiplicity is a constraint on the number of instances of the type that the parameter can represent. The default value is the value assigned to the parameter if no value is provided in the corresponding property of the signal.

Unlike operations, receptions cannot have return types. Receptions are asynchronous; the client that sent the signal isn't waiting for a reply. For the same reason, the parameters of a reception can only be inputs and never outputs.

3.5 Associations: Another Notation for a Property

Section 3.4, "Blocks," focuses on blocks and the various kinds of properties that blocks can own. Blocks are an important part of a structural model of a system, and the relationships between the blocks are at least as important.

There are three main kinds of relationships that can exist between blocks: associations, generalizations, and dependencies. I discuss generalizations and dependencies in detail in Section 3.6, "Generalizations," and Section 3.7, "Dependencies." This section is devoted to associations.

In discussing reference properties and part properties in Section 3.4.1, "Structural Features," I implicitly address the idea of **associations** between blocks. To reiterate the key points: A reference property represents a structure that's external to a block—a structure that the block needs to be connected to for some purpose. A part property instead represents a structure that's internal to a block—in other words, a structure that the block is composed of.

Reference properties and part properties correspond to two kinds of associations that you often create between blocks and display on BDDs: reference associations and composite associations, respectively. Associations are simply an alternative notation to convey these kinds of structural relationships within a system.

Let's take a look in detail at the two kinds of associations.

3.5.1 Reference Associations

A **reference association** between two blocks means that a connection can exist between instances of those blocks in an operational system. And those instances can access each other for some purpose across the connection.

The notation for a reference association on a BDD is a solid line between two blocks. An open arrowhead on exactly one end conveys unidirectional access; the absence of arrowheads on either end conveys bidirectional access.

The upper BDD in Figure 3.17 displays a reference association between the *Electrical Power Subsystem* block and the *Flight Computer* block. Associations can have several labels. You can optionally display an association name floating near the middle of the line, and you can optionally display a role name and multiplicity on either end of the line. The association name is a modeler-defined string that describes

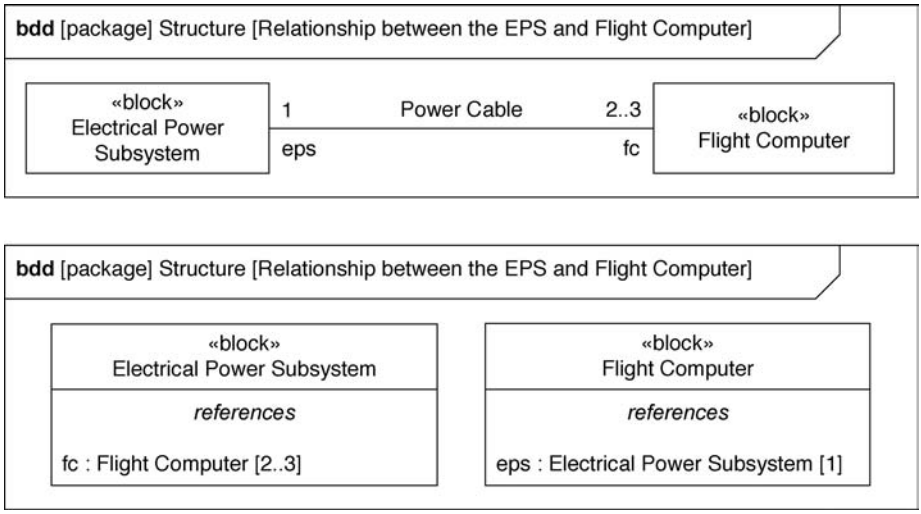

Figure 3.17 *Reference associations and reference properties*

the type of connection that can exist between instances of the two blocks. In Figure 3.17, for example, the name of the reference association shown is *Power Cable*—a name that describes the type of connection that could exist between an electrical power subsystem and a flight computer in a correctly assembled satellite.

Note

I use the phrase "type of connection" deliberately. An association is an element of definition; it can serve as the type for one or more connectors. A **connector** is an element of usage that appears on internal block diagrams (IBDs) (more on this in Chapter 4).

The role name shown on the end of a reference association corresponds to the name of a reference property—one that belongs to the block at the *opposite* end and whose type is the block that it's *next* to. In the upper BDD in Figure 3.17, for example, the role name *eps* represents a reference property that belongs to the *Flight Computer* block and whose type is the *Electrical Power Subsystem* block. The role name *fc* represents a reference property that belongs to the *Electrical Power Subsystem* block and whose type is the *Flight Computer* block. The lower BDD in Figure 3.17 displays an equivalent view of the same model using the references compartment notation instead of reference associations.

Similarly, the multiplicity shown on the end of a reference association (near a role name) corresponds to the multiplicity of that same reference property. This correspondence also is reflected in the two BDDs in Figure 3.17.

Sometimes a block has multiple reference properties of the same type (as shown in the references compartment of the *Flight Computer* block in Figure 3.18). You can convey this equivalently by drawing multiple reference associations between the same two blocks (as shown between the *Flight Computer* block and the *Star Sensor* block in Figure 3.18). Each reference association represents a distinct reference property. Showing both notations on the same diagram is redundant; I'm doing it here to establish the connection between these related concepts.

The choice to use the references compartment notation versus reference associations depends on how much information you need to expose on the BDD. In Figure 3.18, for example, the reference association notation lets me expose the value properties of the *Star Sensor* block; in contrast, the references compartment notation hides all features of the *Star Sensor* block.

Another factor in your decision is the need to specify a type for a connector on an IBD. If you intend to do this, then you need to create a reference association between two blocks and give it a name. The compartment notation would not meet your needs in this case.

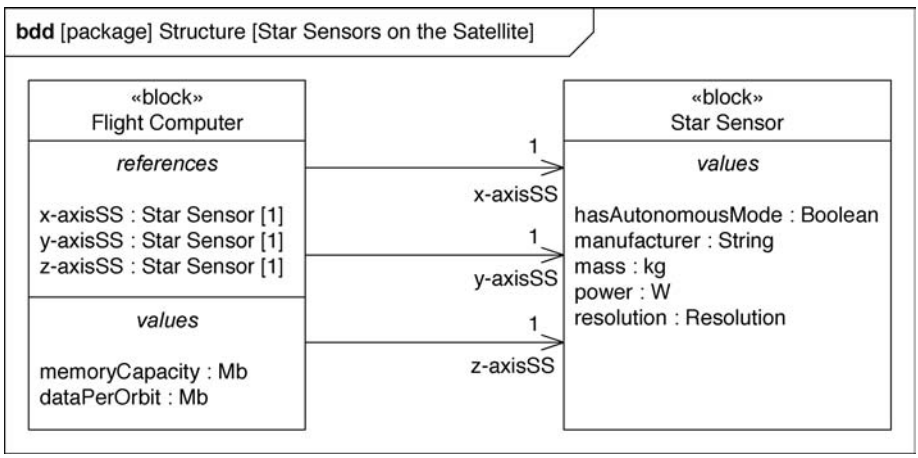

Figure 3.18 *Using reference associations to specify multiple reference properties of the same type*

3.5.2 Composite Associations

A **composite association** between two blocks conveys structural decomposition. An instance of the block at the composite end is made up of some number of instances of the block at the part end.

The notation for a composite association on a BDD is a solid line between two blocks with a solid diamond on the composite end. An open arrowhead on the part end of the line conveys unidirectional access from the composite to its part; the absence of an arrowhead conveys bidirectional access (i.e., the part will have a reference to the composite).

Figure 3.19 displays four examples of composite associations from the *DellSat-77 Satellite* block to the subsystem blocks. (It's permissible and common practice to overlap the solid diamonds on the composite end.) This BDD conveys that a correctly manufactured and assembled DellSat-77 satellite will be composed of one electrical power subsystem, one attitude and orbit control subsystem, one environmental

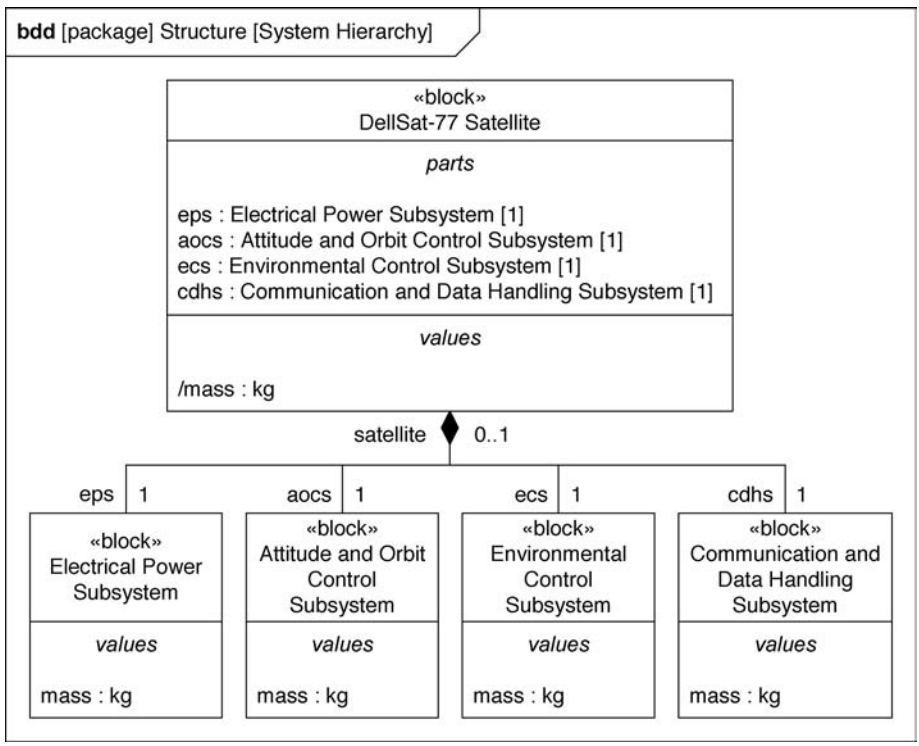

Figure 3.19 *Composite associations and part properties*

control subsystem, and one communication and data handling subsystem. The possible numbers of instances are conveyed by the multiplicities on the part ends of the four composite associations.

The role name shown on the part end of a composite association corresponds to the name of a part property—one that's owned by the block at the composite end and whose type is the block at the part end. In Figure 3.19, for example, the role name *aocs* represents a part property that's owned by the *DellSat-77 Satellite* block and whose type is the *Attitude and Orbit Control Subsystem* block. This correspondence is equivalently reflected in the parts compartment of the *DellSat-77 Satellite* block. It's redundant to show both the parts compartment notation and composite associations on the same diagram; I'm doing it here to reinforce the connection between these concepts.

The multiplicity on the part end of a composite association is not restricted; a composite structure can be made up of an arbitrary number of instances of parts—however many a system requires.

However, the multiplicity on the composite end is restricted. A part—by definition—can belong to only one composite at a time. Therefore, the upper bound of the multiplicity on the composite end must always be 1 (as shown in Figure 3.19). The lower bound of that multiplicity can be either 0 (zero) or 1. A lower bound of 0 conveys that a part can be removed from its composite structure; a lower bound of 1 conveys that it cannot be removed (it must be attached to a composite structure at all times in a valid instance of a system).

In Section 3.4.1.1, I state that 1 is almost always the default multiplicity for elements in SysML. However, there is an important exception to this rule, and here it is: The default multiplicity on the *composite* end of a composite association is 0..1. (On the part end, however, the default multiplicity is the usual case, 1.)

Sometimes a block has multiple part properties of the same type (as shown in the parts compartment of the *Communication and Data Handling Subsystem* block in Figure 3.20). You can convey this equivalently by drawing multiple composite associations between the same two blocks (as shown from the *Communication and Data Handling Subsystem* block to the *Flight Computer* block in Figure 3.20). Each composite association represents a distinct part property.

The same factor that I discuss at the end of Section 3.5.1, "Reference Associations," affects your choice to use either the parts compartment notation or a composite association. You should use a composite association when you need to expose the features of the block that types a part; you should use compartment notation instead when those features are not the focus of the diagram.

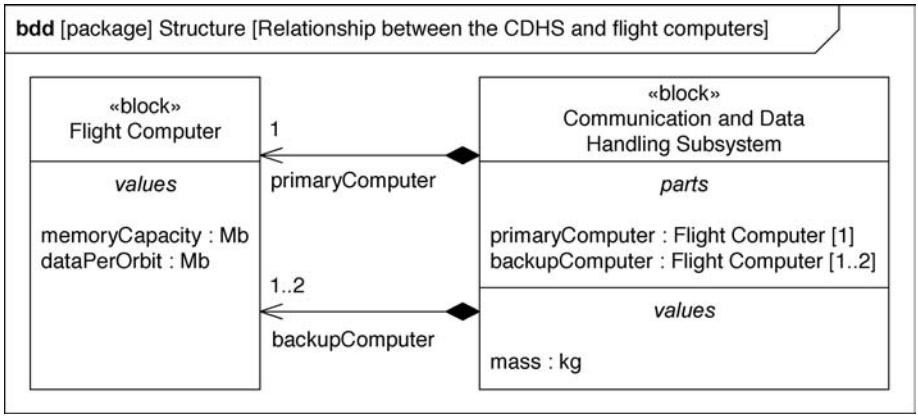

Figure 3.20 *Using composite associations to specify multiple part properties of the same type*

3.6 Generalizations

A **generalization** is another kind of relationship you typically display on BDDs. This relationship conveys **inheritance** between two elements: a more *generalized* element, called the **supertype**, and a more *specialized* element, the **subtype**. You use generalizations to create classification trees (type hierarchies) in your system model.

The notation for a generalization is a solid line with a hollow, triangular arrowhead on the end of the supertype. This relationship is read in English as "is a type of" going from the subtype to the supertype. For example, the BDD in Figure 3.21 shows a generalization from the *Gyroscope* block to the *Sensor* block (among others). This relationship conveys that a gyroscope *is a type of* sensor.

When a supertype has more than one subtype shown on the same BDD, modelers often overlap the hollow, triangular arrowheads on the supertype end to conserve space on the diagram (as shown in Figure 3.21). Purists will tell you that overlapping the arrowheads actually conveys a special grouping of subtypes called a **generalization set**. This is a slightly more advanced feature of the language that you may find useful later. For now, feel free to overlap the arrowheads purely to enhance the readability of your diagrams.

One key point is that generalizations are transitive. The model displayed in Figure 3.21 shows that a star mapper is a type of star sensor, and a star sensor is a type of sensor. Therefore, a star mapper is a type of sensor. Type hierarchies in your model can be arbitrarily deep.

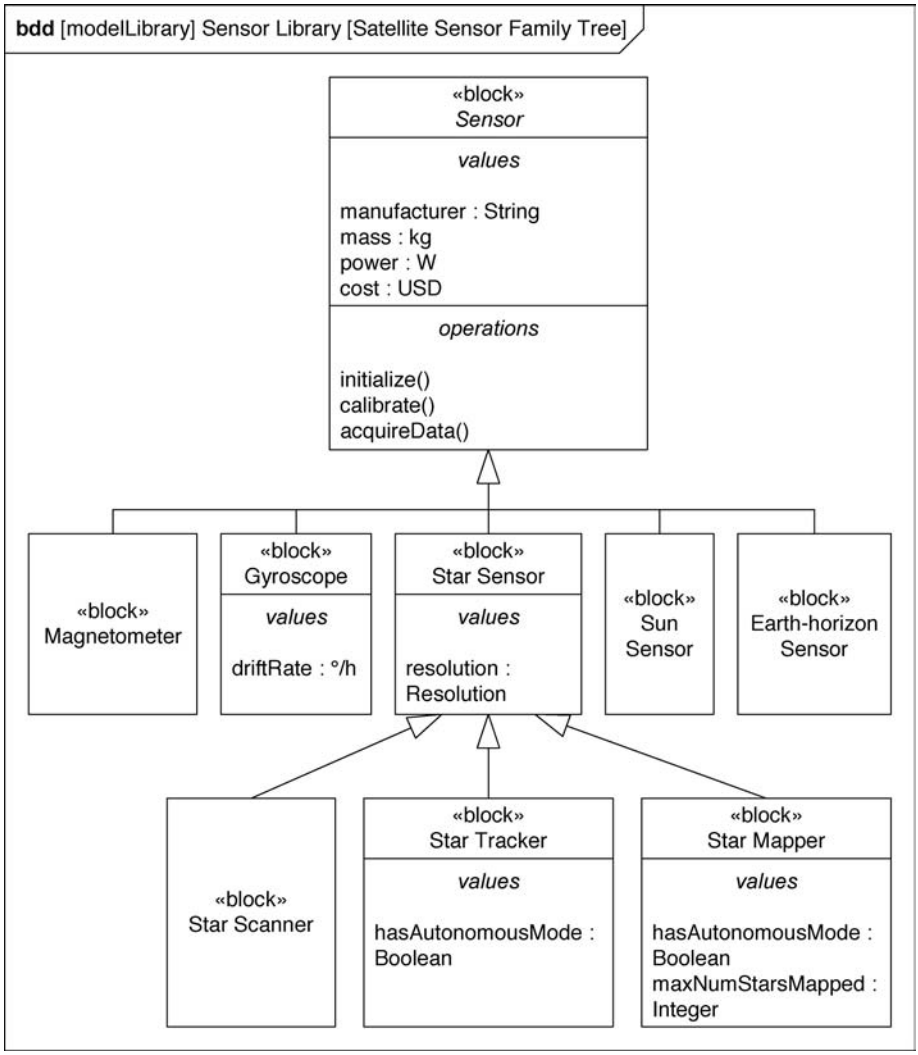

Figure 3.21 *Generalization relationships between blocks*

A generalization conveys that a subtype inherits *all* the features of its supertype: the structural features (properties) and the behavioral features (operations and receptions). In addition to the features it inherits, a subtype may have other features that its supertype doesn't have. For this reason, modelers often refer to a subtype as a **specialization** of its supertype.

For example, the *Star Sensor* block is a specialization of the *Sensor* block. It inherits the four value properties and three operations from the

Sensor block, and then it adds a fifth value property, *resolution,* that the *Sensor* block doesn't have. Similarly, the *Star Mapper* block inherits the five value properties and three operations from the *Star Sensor* block, and then it adds two new value properties (*hasAutonomousMode* and *maxNumStarsMapped*), which neither of its supertypes have.

You create generalizations to define **abstractions** in your system design. A supertype (such as *Sensor*) is an abstraction of its subtypes; it factors out those features that are common among the subtypes. Abstractions let you define a common feature (such as the *initialize* operation) in one place within the model—in the supertype—and that common feature propagates down the type hierarchy to all the subtypes. Then, if you later need to change that common feature, you simply go back to that one place in the model to make the change, and all subtypes in the model get updated instantly.

Abstraction is a powerful design principle; it conveys **substitutability**, meaning that a subtype will be accepted wherever its supertype is required. For example, Figure 3.22 shows that the *Flight Computer*

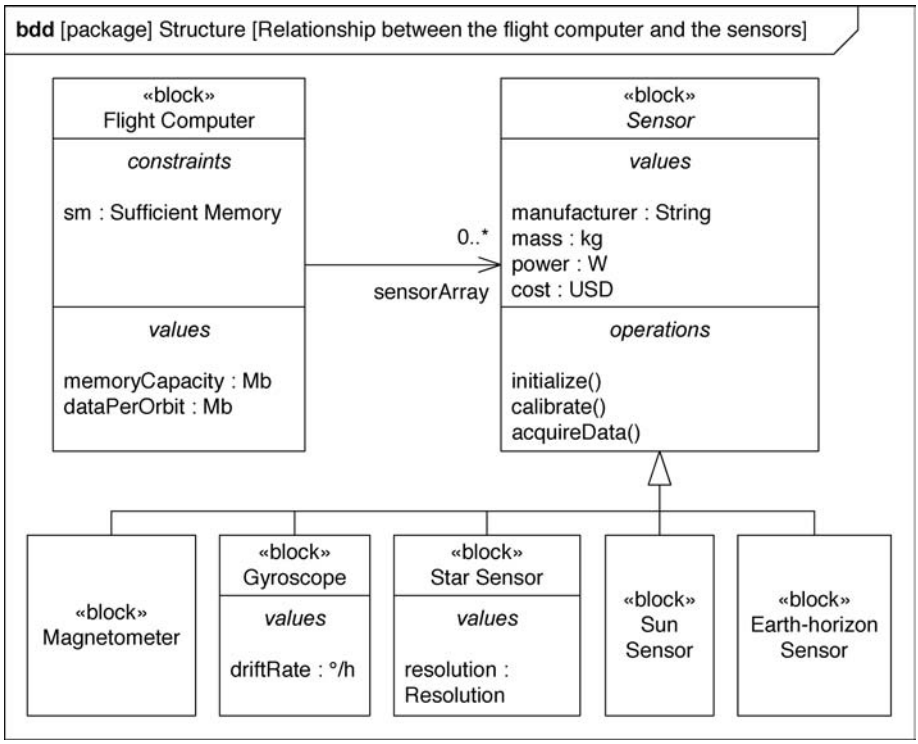

Figure 3.22 *Designing to an abstraction*

block has a reference property named *sensorArray* of type *Sensor*. This model conveys that a flight computer may need access to one or more of the features—structural or behavioral—that are common to all sensors. Therefore, any of the five subtypes of *Sensor* would be acceptable to a flight computer, because all of them inherit those common features from their supertype, *Sensor*.

This is an example of designing to an abstraction. This practice creates extensibility in your design. When the customers' requirements change later in the life cycle and you need to add a new type of sensor to the satellite design, you can simply define a new subtype of the *Sensor* block within the system model, and that addition will be transparent to all clients (such as *Flight Computer*) that reference the *Sensor* block. For all these reasons, building generalizations into your model can significantly reduce the time it takes to modify your system design as the life cycle progresses—and that capability directly translates into cost savings.

3.7 Dependencies

A **dependency** is the third kind of relationship you can display on BDDs. It means what it sounds like: One element in the model, the **client**, depends on another element in the model, the **supplier**. More precisely, a dependency conveys that when the supplier element changes, the client element *may* also have to change.

Most often, you create a dependency between two model elements solely to establish traceability between them. A dependency relationship lets you use your modeling tool to perform automated downstream impact analysis when you make changes to your design. When you make a change to one element, you can query your modeling tool to generate a list of the other elements in the model that may be impacted by the change; the modeling tool navigates the set of dependencies that you've created between elements to generate that list.

This is a practical reason to create dependencies in your model. However, you seldom have a reason to display them on BDDs. They are part of the structure of the model and not of the system that the model represents. And you will spend most of your time creating BDDs to convey system structure to your stakeholders.

When a dependency appears on a BDD, the notation is a dashed line with an open arrowhead, which is drawn *from* the client *to* the sup-

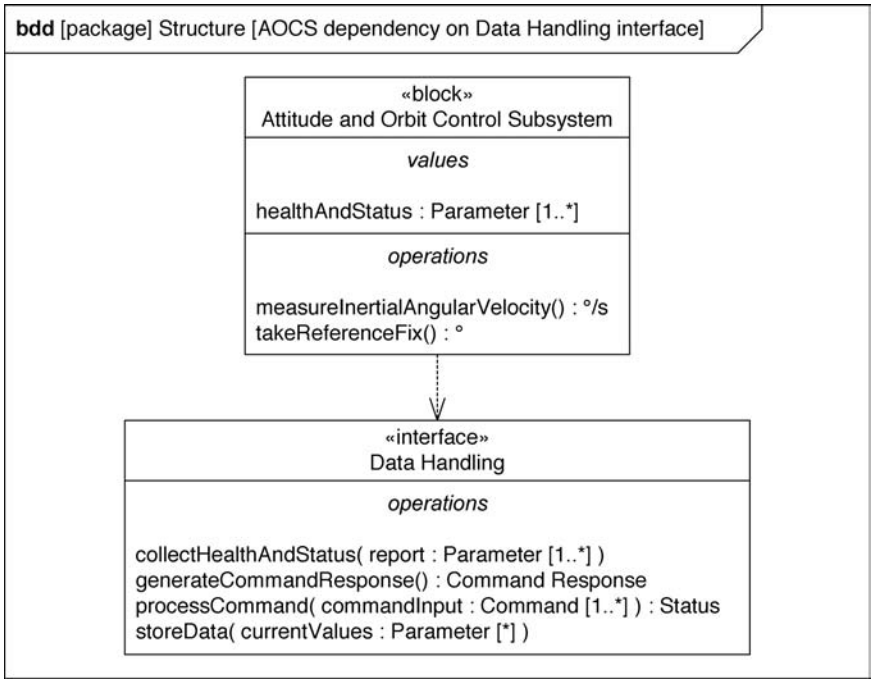

Figure 3.23 *A dependency relationship between two named elements*

plier. In Figure 3.23, for example, the *Attitude and Orbit Control Subsystem* block is the client, and the *Data Handling* interface is the supplier. This model conveys that the block depends on the interface; if the interface changes, the block may need to change, too.

Note that SysML defines specialized kinds of dependency relationships (e.g., package import, viewpoint conformance, and several kinds of requirements relationships). Although you rarely display dependencies on BDDs, you often display these specialized kinds of dependencies on package diagrams and requirements diagrams. I discuss these topics in detail in Chapter 10, "Package Diagrams," and Chapter 11, "Requirements Diagrams."

3.8 Actors

An **actor** represents someone or something that has an external interface with your system. The name of an actor conveys a **role** played by

a person, an organization, or another system when it interacts with your system.

SysML defines two notations for an actor: a stick figure and a rectangle with the keyword «actor» preceding the name. Figure 3.24 shows examples of both notations. It's legal to use either notation for any type of actor—person or system. However, modelers often adopt the convention of using the stick figure notation to represent a person and the rectangle notation to represent a system, although the language doesn't require it.

You will occasionally display actors on BDDs to express the generalizations between actors and the associations between actors and blocks (as shown in Figure 3.24). It's far more common, though, to display actors on use case diagrams, where you express which use cases each actor participates in. I cover these topics in detail in Chapter 5, "Use Case Diagrams."

All the key ideas about generalizations, reference associations, and composite associations also apply when actors are involved in these relationships. There are two constraints:

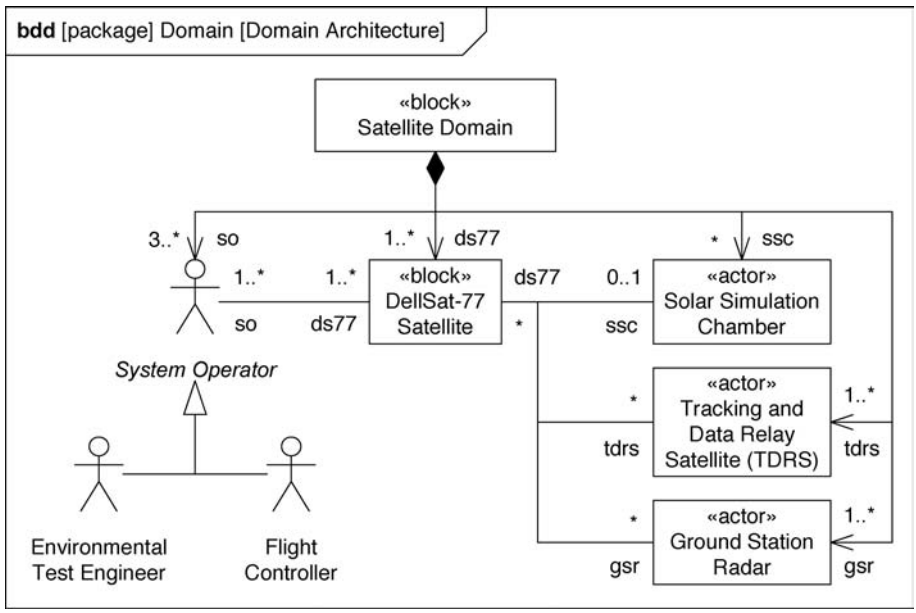

Figure 3.24 *Actors on a BDD*

- You cannot define a generalization between an actor and a block.
- An actor cannot have parts; that is, it cannot appear at the composite end of a composite association. (We always regard an actor as a "black box.")

3.9 Value Types

Like a block, a **value type** is an element of definition—one that generally defines a type of quantity. I say "generally" because there are two value types in SysML—*Boolean* and *String*—that arguably are not quantities.

You can use a value type in many places throughout your model. Most often, it appears as the type of a **value property**, which is a kind of structural feature of blocks. (Section 3.4.1.3, "Value Properties," has more details.) But that's not the only place where value types make an appearance; they're actually ubiquitous in system models. They can also appear as the types of the following:

- Atomic flow ports on blocks and actors
- Flow properties in flow specifications
- Constraint parameters in constraint blocks
- Item flows and item properties on connectors
- Return types of operations
- Parameters of operations and receptions
- Object nodes, pins, and activity parameters within activities

There are three kinds of value types—primitive, structured, and enumerated—that you typically define in your system model. A **primitive** value type has no internal structure (it doesn't own any value properties). Its notation is a rectangle with the stereotype «valueType» preceding the name.

SysML defines four primitive value types: *String, Boolean, Integer,* and *Real*. You can, of course, define your own primitive value types as specializations (subtypes) of these four. For example, Figure 3.25 shows three value types (°, *V*, and °*C*) that are subtypes of *Real*.

As its name implies, a **structured** value type has an internal structure—generally two or more value properties. As with a primitive value type, the notation for a structured value type is a rectangle with

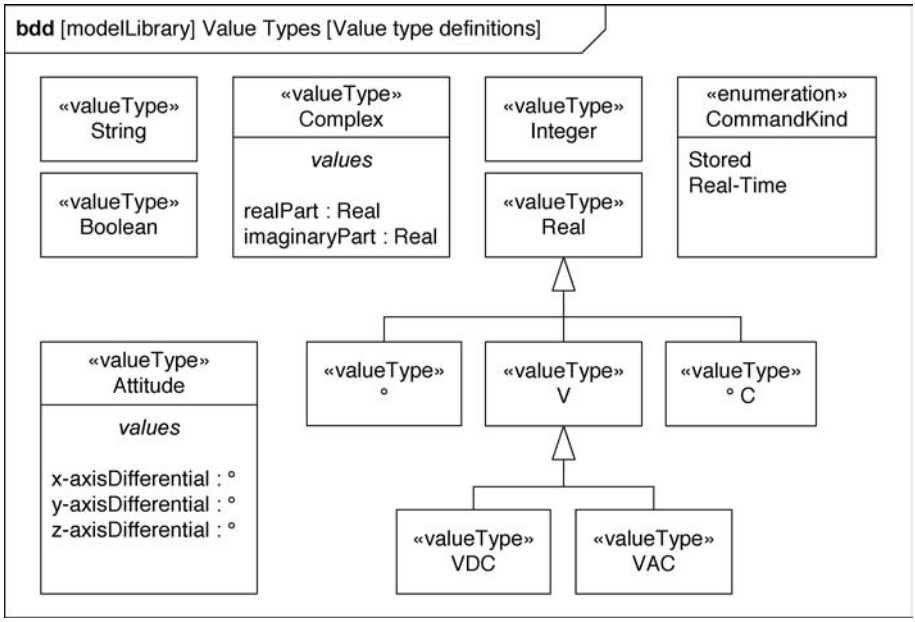

Figure 3.25 *Value types*

the stereotype «valueType» preceding the name. SysML defines one structured value type: *Complex*. Its structure consists of two value properties—*realPart* and *imaginaryPart*—that are both of type *Real*. One structured value type may, in turn, be the type of a value property within another structured value type. In this way, you can create arbitrarily complex systems of value types.

An **enumerated** value type—colloquially called an **enumeration**—simply defines a set of **literals** (legal values). If a parameter of an operation (or some other kind of element shown in the earlier bulleted list) is typed by an enumeration, then the value it holds at any moment must be one of the literals in that enumeration. The BDD in Figure 3.25 shows an enumeration named *CommandKind,* which defines two literals: *Stored* and *Real-Time.* I could use this enumeration, for example, to type an input parameter named *kind* in an operation named *buildCommand*. When a client calls this operation (within a running system), the only legal values it can pass are *Stored* and *Real-Time.*

I mentioned earlier that value types can be related to one another by using generalizations. A value type hierarchy can be arbitrarily deep, and generalizations—as you may recall—are transitive. For ex-

ample, Figure 3.25 conveys that the value types *VDC* and *VAC* are (indirectly) subtypes of *Real*. The principle of substitutability applies here just as it does in the case of generalizations between blocks: Values of type *VDC* and *VAC* will be accepted wherever their supertypes (*V* and *Real*) are required. These supertypes are abstractions. And the principle of designing to an abstraction—and its consequent extensibility—also applies to this practice of creating a value type hierarchy. This is a widely used and powerful modeling practice.

3.10 Constraint Blocks

Like a block, a **constraint block** is an element of definition—one that defines a Boolean **constraint expression** (an expression that must evaluate to either *true* or *false*). Most often, the constraint expression you define in a constraint block is an equation or an inequality: a mathematical relationship that you use to constrain value properties of blocks. You would do this for two reasons:

- To specify assertions about valid system values in an operational system
- To perform engineering analyses during the design stage of the life cycle

The variables in a constraint expression are called **constraint parameters**. Generally, they represent quantities, and so they're typed most often by value types. For example, Figure 3.26 shows a constraint block named *Transfer Orbit Size*, which defines a constraint expression that contains three constraint parameters: *semimajorAxis*, *initialOrbitRadius*, and *finalOrbitRadius*. These three constraint parameters are typed by the value type *km*.

Constraint parameters receive their values from the value properties they're bound to—that is, the value properties that are being constrained. At any given moment, those values either satisfy the constraint expression, or they don't; the system is either operating nominally, or it isn't. Note, however, that a BDD by itself can't convey which constraint parameters and value properties are bound to one another. You would express this piece of information on a parametric diagram. (I discuss this in detail in Chapter 9.)

The notation for a constraint block on a BDD is a rectangle with the stereotype «constraint» preceding the name. The constraint expression

always appears between curly brackets ({}) in the constraints compartment. The constraint parameters in the constraint expression are listed individually in the parameters compartment.

You sometimes build a more complex constraint block from a set of simpler constraint blocks. You would do this to create a more complex mathematical relationship from simpler equations and inequalities. The more complex constraint block can display its constituent parts as a list of **constraint properties** in the constraints compartment. Recall from Section 3.4.1.4 that a constraint property has a name and a type in the format *name : type.* The type, as mentioned earlier, must be the name of a constraint block.

For example, Figure 3.26 shows that the constraint block *Hohmann Transfer* is composed of two constraint properties—*ttof* and *tos*—which

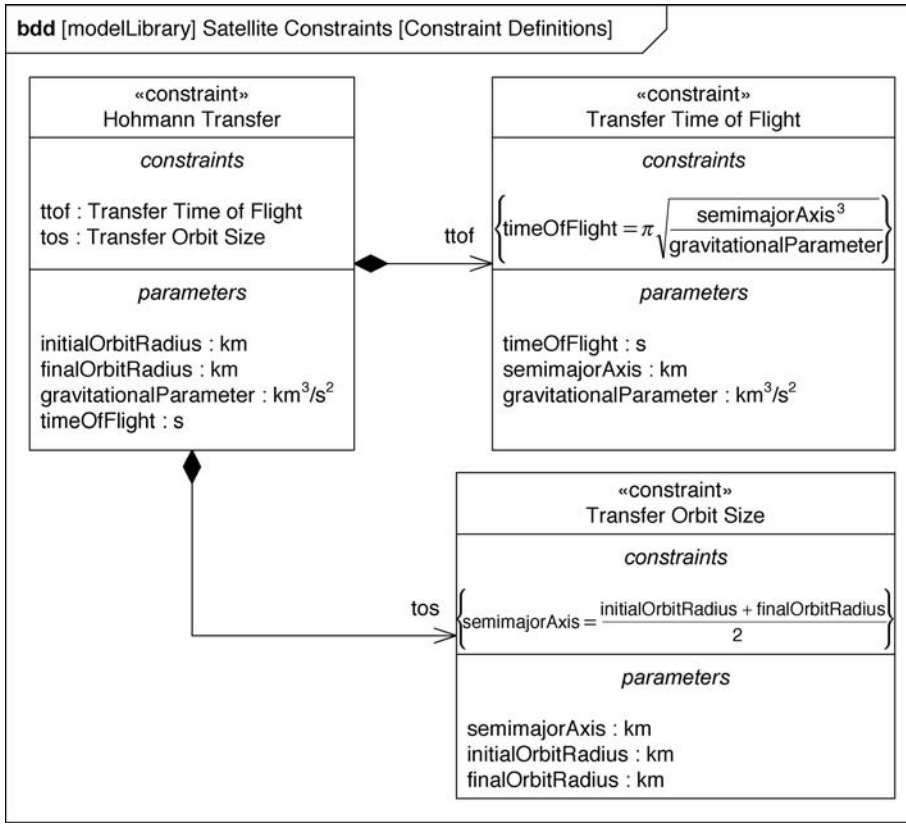

Figure 3.26 *Relationships between constraint blocks*

represent usages of the constraint blocks *Transfer Time of Flight* and *Transfer Orbit Size*, respectively. This model conveys that *Hohmann Transfer* defines a constraint expression that is a composite of two simpler constraint expressions—in effect, defining a more complex mathematical relationship.

Note, though, what this BDD doesn't (and can't) convey: *where* those two simpler constraint expressions are specifically connected to each other to create the composite constraint expression. A parametric diagram would convey this additional piece of information (more on this in Chapter 9).

As an alternative to the constraints compartment notation, you can use composite associations to convey that one constraint block is composed of other, simpler ones (as shown in Figure 3.26). Note that the role names shown on the part ends of the two composite associations correspond to the names of the constraint properties in the *Hohmann Transfer* constraint block. These are equivalent notations. You use composite associations when you need to expose the details of the simpler constraint blocks; in contrast, you use the constraints compartment notation to hide those details when they're not the focus of the diagram.

3.11 Comments

SysML has a lot of rules (and they all exist to serve the very useful purpose of giving your design unambiguous meaning from one reader to the next). However, you sometimes need to express information on a diagram in an unconstrained way as a block of text. You can do this with a comment.

A **comment** is, in fact, a model element. It consists of a single attribute: a string of text called the **body**. You can convey any information you need to in the body of a comment, and you can optionally attach a comment to other elements on a diagram to provide additional information about them. You can use comments on any of the nine kinds of SysML diagrams.

The notation for a comment is commonly referred to as a **note symbol**: a rectangle whose upper-right corner is bent. You use a dashed line to attach a comment to other elements (as shown at the bottom of the BDD in Figure 3.27). If you need to, you can attach a comment to several model elements simultaneously by using a separate dashed line for each one.

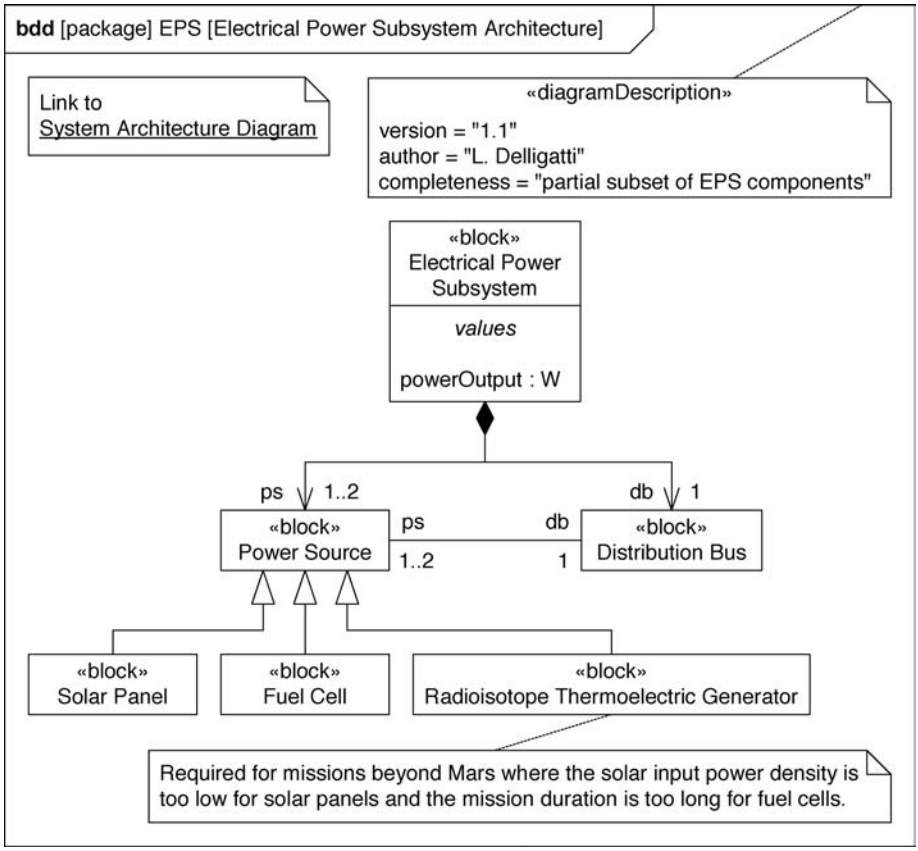

Figure 3.27 *Comments on a BDD*

Modelers sometimes put freestanding comments with hyperlinks on a diagram to enable readers to quickly navigate to a related diagram in the model (or to an external document). An example of this is shown in the upper-left corner of the BDD in Figure 3.27. To be clear, though, this capability is a function of the modeling tool you use; not all tools do this. And SysML itself says nothing about this capability.

SysML defines some specialized kinds of comments: rationale, problem, and diagram description. These appear as a note symbol with the respective stereotype preceding the body of the comment. Figure 3.27 shows an example of a diagram description comment in the upper-right corner of the BDD. Modelers often use rationale comments in conjunction with requirements relationships and allocations. I discuss these topics in detail in Chapters 11 and 12.

Summary

The BDD is the primary kind of diagram you create to communicate structural information about a system. A BDD enables you to express the types of structures that can exist internally within a system and externally in a system's environment. You can also use BDDs to express the types of services each structure provides and requires, the types of constraints each structure must conform to, and the types of values that can exist within an operational system.

Generalization relationships between elements let you define type hierarchies and design to abstractions. This is a powerful design technique—one that creates extensibility in your system design by decoupling the clients of services from any specific implementation of a provider of those services. As your stakeholders' requirements evolve over time, you can modify existing providers or add new ones with minimal impact on the rest of the system design.

This page intentionally left blank

Index

B

C

D

Q

R

S

T

U

V

W

X

This page intentionally left blank