



Model-Based Engineering with AADL

An Introduction to the SAE
Architecture Analysis and Design Language



SEI SERIES IN SOFTWARE ENGINEERING

Peter H. Feiler

David P. Gluch

FREE SAMPLE CHAPTER



SHARE WITH OTHERS



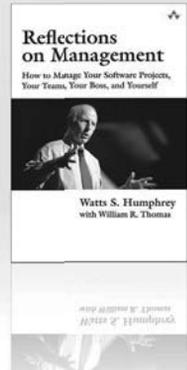
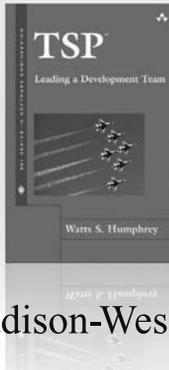
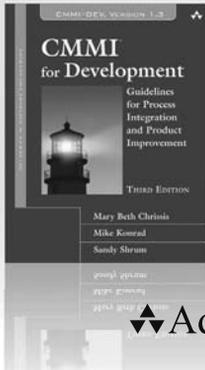
Model-Based Engineering with AADL

The SEI Series in Software Engineering



Software Engineering Institute

Carnegie Mellon



Addison-Wesley

Visit informit.com/sei for a complete list of available products.

The **SEI Series in Software Engineering** represents a collaborative undertaking of the Carnegie Mellon Software Engineering Institute (SEI) and Addison-Wesley to develop and publish books on software engineering and related topics. The common goal of the SEI and Addison-Wesley is to provide the most current information on these topics in a form that is easily usable by practitioners and students.

Books in the series describe frameworks, tools, methods, and technologies designed to help organizations, teams, and individuals improve their technical or management capabilities. Some books describe processes and practices for developing higher-quality software, acquiring programs for complex systems, or delivering services more effectively. Other books focus on software and system architecture and product-line development. Still others, from the SEI's CERT Program, describe technologies and practices needed to manage software and network security risk. These and all books in the series address critical problems in software engineering for which practical solutions are available.

PEARSON

Addison-Wesley

Cisco Press

EXAM/CRAM

IBM Press

que

PRENTICE HALL

SAMS

Safari Books Online

Model-Based Engineering with AADL

An Introduction to the SAE
Architecture Analysis &
Design Language

Peter H. Feiler

David P. Gluch

◆◆ Addison-Wesley

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



The SEI Series in Software Engineering

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

CMM, CMMI, Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, and CERT Coordination Center are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ATAM; Architecture Tradeoff Analysis Method; CMM Integration; COTS Usage-Risk Evaluation; CURE; EPIC; Evolutionary Process for Integrating COTS Based Systems; Framework for Software Product Line Practice; IDEAL; Interim Profile; OAR; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Options Analysis for Reengineering; Personal Software Process; PLTP; Product Line Technical Probe; PSP; SCAMPI; SCAMPI Lead Appraiser; SCAMPI Lead Assessor; SCE; SEI; SEPG; Team Software Process; and TSP are service marks of Carnegie Mellon University.

Special permission to reproduce in this book two figures from Feiler, Peter; Hansson, Jörgen; de Niz, Dionisio; & Wrage, Lutz. System Architecture Virtual Integration: An Industrial Case Study (CMU/SEI-2009-TR-017), Copyright © 2009 by Carnegie Mellon University; a variant of a figure from an article and conference presentation: Peter H. Feiler, "Model-based Validation of Safety-Critical Embedded Systems," Proceedings of IEEE Aerospace Conference, March 2010; and three figures from the SEI course: "Modeling System Architectures using the Architecture Analysis and Design Language (AADL)" is granted by the Software Engineering Institute.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Cataloging-in-Publication data is on file with the Library of Congress.

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-88894-5

ISBN-10: 0-321-88894-4

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, September 2012

Contents

Preface	xv
Introduction	1
PART I Model-Based Engineering and the AADL	3
Chapter 1 Model-Based Software Systems Engineering	5
1.1 MBE and Software System Engineering	6
1.1.1 MBE for Embedded Real-Time Systems	6
1.1.2 Analyzable Models and MBE	8
1.1.3 MBE and the AADL	10
1.2 AADL and Other Modeling Languages	12
1.2.1 AADL, MDA, and UML	14
1.2.2 AADL and SysML	15
Chapter 2 Working with the SAE AADL	17
2.1 AADL Models	19
2.1.1 Component Categories	19
2.1.2 Language Syntax	20
2.1.3 AADL Classifiers	21
2.1.4 Summary of AADL Declarations	22
2.1.5 Structure of AADL Models	25
2.2 System Specification and System Instances	26
2.2.1 Creating System Instance Models	26
2.2.2 AADL Textual and Graphical Representation	27
2.2.3 Analyzing Models	30

Chapter 3 Modeling and Analysis with the AADL: The Basics	31
3.1 Developing a Simple Model	31
3.1.1 Defining Components for a Model	32
3.1.2 Developing a Top-Level Model	36
3.1.3 Detailing the Control Software	38
3.1.4 Adding Hardware Components	40
3.1.5 Declaring Physical Connections	41
3.1.6 Binding Software to Hardware	43
3.1.7 Conducting Scheduling Analyses	45
3.1.8 Summary	47
3.2 Representing Code Artifacts	47
3.2.1 Documenting Source Code and Binary Files	48
3.2.2 Documenting Variable Names	49
3.2.3 Modeling the Source Code Structure	50
3.3 Modeling Dynamic Reconfigurations	51
3.3.1 Expanded PBA Model	51
3.3.2 Specifying Modes	53
3.4 Modeling and Analyzing Abstract Flows	55
3.4.1 Specifying a Flow Model	55
3.4.2 Specifying an End-to-End Flow	57
3.4.3 Analyzing a Flow	57
3.5 Developing a Conceptual Model	58
3.5.1 Employing Abstract Components in a PBA Model	58
3.5.2 Detailing Abstract Implementations	61
3.5.3 Transforming into a Runtime Representation	63
3.5.4 Adding Runtime Properties	65
3.5.5 Completing the Specification	67
3.6 Working with Component Patterns	69
3.6.1 Component Libraries and Reference Architectures	69
3.6.2 Establishing a Component Library	70
3.6.3 Defining a Reference Architecture	72
3.6.4 Utilizing a Reference Architecture	74

Chapter 4 Applying AADL Capabilities	77
4.1 Specifying System Composition	77
4.1.1 Component Hierarchy	77
4.1.2 Modeling Execution Platform Resources	78
4.1.3 Execution Platform Support of Communication	80
4.1.4 System Hierarchy	81
4.1.5 Creating a System Instance Model	81
4.1.6 Working with Connections in System Instance Models	82
4.1.7 Working with System Instance Models	83
4.2 Component Interactions	84
4.2.1 Modeling Directional Exchange of Data and Control	85
4.2.2 Modeling Shared Data Exchange	86
4.2.3 Modeling Local Service Requests or Function Invocation	87
4.2.4 Modeling Remote Service Requests and Function Invocations	90
4.2.5 Modeling Object-Oriented Method Calls	92
4.2.6 Modeling Subprogram Parameters	95
4.2.7 Interfacing to the External World	97
4.3 Modeling Data and Its Use	97
4.3.1 Defining a Simple Data Type	98
4.3.2 Representing Variants of a Data Type	99
4.3.3 Detailing a Data Type	100
4.4 Organizing a Design	101
4.4.1 Using Packages	102
4.4.2 Developing Alternative Implementations	104
4.4.3 Defining Multiple Extensions	105

PART II Elements of the AADL	109
Chapter 5 Defining AADL Components	113
5.1 Component Names	113
5.2 Component Categories	114
5.3 Declaring Component Types	114
5.4 Declaring a Component's External Interfaces	118
5.5 Declaring Component Implementations	121
5.6 Summary	125
Chapter 6 Software Components	127
6.1 Thread	128
6.1.1 Representation	130
6.1.2 Properties	131
6.1.3 Constraints	132
6.2 Thread Group	133
6.2.1 Representations	133
6.2.2 Properties	134
6.2.3 Constraints	134
6.3 Process	135
6.3.1 Representations	136
6.3.2 Properties	137
6.3.3 Constraints	137
6.4 Data	138
6.4.1 Representations	138
6.4.2 Properties	140
6.4.3 Constraints	140
6.5 Subprogram	141
6.5.1 Representations	143
6.5.2 Properties	143
6.5.3 Constraints	144
6.6 Subprogram Group	144
6.6.1 Representations	145

6.6.2 Properties	146
6.6.3 Constraints	146
Chapter 7 Execution Platform Components	147
7.1 Processor	148
7.1.1 Representations	148
7.1.2 Properties	150
7.1.3 Constraints	150
7.2 Virtual Processor	151
7.2.1 Representations	151
7.2.2 Properties	152
7.2.3 Constraints	152
7.3 Memory	153
7.3.1 Representations	153
7.3.2 Properties	154
7.3.3 Constraints	155
7.4 Bus	156
7.4.1 Representations	156
7.4.2 Properties	157
7.4.3 Constraints	157
7.5 Virtual Bus	158
7.5.1 Representations	158
7.5.2 Properties	159
7.5.3 Constraints	159
7.6 Device	160
7.6.1 Representations	160
7.6.2 Properties	161
7.6.3 Constraints	161
Chapter 8 Composite and Generic Components	163
8.1 System	163
8.1.1 Representations	164
8.1.2 Properties	165
8.1.3 Constraints	165

8.2 Abstract	165
8.2.1 Representations	166
8.2.2 Properties	168
8.2.3 Constraints	168
Chapter 9 Static and Dynamic Architecture	169
9.1 Subcomponents	169
9.1.1 Declaring Subcomponents	170
9.1.2 Using Subcomponent Declarations	170
9.1.3 Declaring Subcomponents as Arrays	172
9.2 Modes	173
9.2.1 Declaring Modes and Mode Transitions	174
9.2.2 Declaring Modal Component Types and Implementations	175
9.2.3 Using Modes for Alternative Component Configurations	177
9.2.4 Inheriting Modes	180
9.2.5 Mode-Specific Properties	181
9.2.6 Modal Configurations of Call Sequences	182
Chapter 10 Component Interactions	185
10.1 Ports and Connections	186
10.1.1 Declaring Ports	186
10.1.2 Declaring Port to Port Connections	189
10.1.3 Using Port to Port Connections	189
10.1.4 Constraints on Port to Port Connections	193
10.1.5 Port Communication Timing	196
10.1.6 Sampled Processing of Data Streams	198
10.1.7 Deterministic Sampling	199
10.1.8 Mixed Port-Based and Shared Data Communication	203
10.1.9 Port and Port Connection Properties	207
10.1.10 Aggregate Data Communication	207

10.2	Data Access and Connections	210
10.3	Bus Access and Connections	213
10.4	Feature Groups and Connections	217
10.4.1	Declaring Feature Group Types	218
10.4.2	Declaring a Feature Group as a Feature of a Component	220
10.4.3	Declaring Feature Group Connections	221
10.5	Abstract Features and Connections	225
10.5.1	Declaring Abstract Features	226
10.5.2	Refining Abstract Features	226
10.6	Arrays and Connections	227
10.6.1	Explicitly Specified Array Connections	228
10.6.2	Array Connection Patterns	229
10.6.3	Using Array Connection Properties	230
10.7	Subprogram Calls, Access, and Instances	232
10.7.1	Declaring Calls and Call Sequences	233
10.7.2	Declaring Remote Subprogram Calls as Bindings	234
10.7.3	Declaring Remote Subprogram Calls as Access Connections	236
10.7.4	Modeling Subprogram Instances	237
10.8	Parameter Connections	240
10.8.1	Declaring Parameters	240
10.8.2	Declaring Parameter Connections	241
Chapter 11	System Flows and Software Deployment	245
11.1	Flows	245
11.1.1	Declaring Flow Specifications	246
11.1.2	Declaring Flow Implementations	249
11.1.3	Declaring End-to-End Flows	253
11.1.4	Working with End-to-End Flows	256
11.2	Binding Software to Hardware	256
11.2.1	Declaring Bindings with Properties	257
11.2.2	Processor Bindings	259

11.2.3	Memory Bindings	259
11.2.4	Connection Bindings	260
11.2.5	Binding Remote Subprogram Calls	260
Chapter 12	Organizing Models	263
12.1	Naming and Referencing Model Elements	263
12.1.1	Naming and Referencing with Packages	263
12.1.2	Naming and Referencing Classifiers	264
12.1.3	References to Model Elements	265
12.1.4	Naming and Referencing with Property Sets	266
12.2	Organizing Models with Packages	266
12.2.1	Declaring Packages	267
12.2.2	Referencing Elements in Packages	269
12.2.3	Aliases for Packages and Type References	271
12.3	Evolving Models by Classifier Refinement	273
12.3.1	Declaring Classifier Extensions	274
12.3.2	Declaring Model Element Refinements	275
12.3.3	Classifier Substitution Rules for Refinements	277
12.3.4	Refining the Category	280
12.4	Prototypes as Classifier Parameters	281
12.4.1	Declaring Prototypes	281
12.4.2	Using Prototypes	283
12.4.3	Providing Prototype Actuals	284
12.4.4	Properties	287
Chapter 13	Annotating Models	289
13.1	Documenting Model Elements	289
13.1.1	Comments and Description Properties	289
13.1.2	Empty Component Sections	290
13.2	Using Properties	291
13.2.1	Assigning Property Values	292
13.2.2	AADL Property Types and Values	294
13.2.3	Determining a Property Value	297

13.2.4	Contained Property Associations	299
13.2.5	Determining the Property Value: An Example	300
Chapter 14	Extending the Language	303
14.1	Property Sets	303
14.1.1	Declaring Property Sets	304
14.1.2	Property Type Declarations	305
14.1.3	Property Definitions	309
14.1.4	Property Constant Declarations	311
14.2	Annex Sublanguages	312
14.2.1	Declaring Annex Concepts in Libraries	313
14.2.2	Using Annex Concepts in Subclauses	314
Chapter 15	Creating and Validating Models	317
15.1	Model Creation	317
15.2	Model Creation Tools	319
15.3	System Validation and Generation	321
15.4	System Validation and Generation Tools	322
Appendixes		325
Appendix A	Syntax and Property Summary	327
A.1	AADL Syntax	327
A.2	Component Type and Implementation Elements	342
A.3	Basic Property Types and Type Constructors	347
A.4	AADL Reserved Words	348
A.5	AADL Properties	349
A.5.1	Deployment Properties	350
A.5.2	Thread-Related Properties	364
A.5.3	Timing Properties	371
A.5.4	Communication Properties	384
A.5.5	Memory-Related Properties	391
A.5.6	Programming Properties	398

A.5.7 Modeling Properties	408
A.5.8 Project-Specific Constants and Property Types	410
A.6 Runtime Services	418
A.6.1 Application Runtime Services	418
A.6.2 Runtime Executive Services	421
A.7 Powerboat Autopilot System	425
A.7.1 Description	425
A.7.2 Enhanced Versions of the PBA System	426
A.7.3 AADL Components of the PBA System	427
A.7.4 An Alternative AADL Representation	428
Appendix B Additional Resources	429
B.1 Modeling System Architectures	429
B.2 Cases Studies	431
Appendix C References	435
Index	441

Preface

In this book, we introduce readers to the concepts, structure, and use of the SAE Architecture Analysis & Design Language (AADL) and demonstrate how AADL is an effective tool for Model-Based Engineering (MBE) of software system architectures. If you are just learning about AADL, we provide sufficient detail to enable you to develop and analyze basic system models. The core skills acquired by mastering the material in this book will provide you a foundation upon which to build your AADL and MBE expertise. Even when you are an accomplished AADL user, we anticipate you will find this book to be a valuable reference.

What and Why: MBE and AADL

Model-based engineering is the creation and analysis of models of your system such that you can predict and understand its capabilities and operational quality attributes (e.g., its performance, reliability, or security). By doing so throughout the lifecycle, you can discover system-level problems—those usually not found until system integration and acceptance testing—and avoid costly rework late in development and maintenance. In the past, separate models have been created for various system components and for each of the different analyses. A systematic and less fragmented approach is an architecture-centric one. Architecture-centric approaches address system-level issues and maintain a self-consistent set of analytical views of a system such that individual analyses retain their validity amidst architectural changes within the set.

The Architecture Analysis & Design Language (AADL) is an SAE International (formerly known as the Society of Automotive Engineers)

standard [AS5506A¹]. The AADL is a unifying framework for model-based software systems engineering that you use to capture the static modular software architecture, the runtime architecture in terms of communicating tasks, the computer platform architecture on which the software is deployed, and any physical system or environment with which the system interacts. You capture both the static structure and the dynamics in a single architecture model and annotate it with information that is relevant to the analysis of various operational characteristics. The concepts provided by AADL, such as threads, processes, or devices, have well-defined execution semantics that allow you to conduct both lightweight and formal analyses of systems. In addition, using its extensibility constructs, you as well as tool developers can blend custom analysis and specification techniques with core AADL capabilities to create a complete engineering environment for architectural modeling and analysis.

In developing an AADL model, you represent the architecture of your system as a hierarchy of interacting components. You organize interface specifications and implementation blueprints of software, hardware, and physical components into packages to support large-scale and team-based development.

As a standard, AADL provides you with the stability often not found in proprietary technologies and allows you to participate in defining enhancements to the language. Additional elements of the standard suite that extend the AADL framework are found in the *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1* [AS5506/1] and *SAE Architecture Analysis and Design Language (AADL) Annex Volume 2* [AS5506/2]. Released as a standard in June 2006, SAE AS-5506/1 defines annexes for the AADL graphical Notation, AADL Meta-Model and Interchange Formats, Language Compliance and Application Program Interface, and Error Model Language. Released as a standard in January 2011, SAE AS-5506/2 defines annexes for Behavior Modeling, for guidance on incorporating Data Modeling with AADL, and for ARINC653 Partitioned Architecture modeling.²

1. The standard AS5506A was originally published in November 2004. The book covers its revision published in January 2009, as well as errata corrections approved in 2012. For more information on the AADL, go to the Web site www.aadl.info. To purchase a copy of the standard, go to the Web site www.sae.org/technical/standards/AS5506A.

2. Additional annexes are in development for ballot in late 2012: a revision of the Error Model Annex standard, a Requirements Definition and Analysis Annex, and a Code Generation Annex.

Who Will Benefit from Reading This Book

You benefit from this book if you are a developer of software-reliant systems, whether a system or software architect, a system engineer, or an embedded software system developer. This book provides a foundation to enable you to apply the AADL and model-based engineering directly in your work. If you are a technical leader or project manager, the core principles and examples discussed in this book provide you with the knowledge required to guide technical personnel in the application of the AADL.

For graduate and advanced undergraduate software engineering students, this book offers a basis to understand and apply the AADL and MBE in your learning experiences. This book can be used as part of the material for a course on software architecture or software systems engineering of embedded real-time applications.

What You Need to Know to Get the Most Value from This Book

A basic knowledge of core software engineering practices (e.g., software architecture, software design), real-time systems (e.g., concurrency, scheduling, communications), and knowledge of computer runtime concepts (e.g., threads, execution semantics) will help you benefit most from this book. As a minimum, the level of expertise you should have in these areas is that commensurate with an advanced undergraduate student in computer science or software engineering. If you are a software developer with a degree in a technical discipline with two to three years' experience in developing embedded real-time software systems, you will find this book especially valuable in modeling software system architectures.

Structure of the Book

We have organized the material in the book into two parts, plus three appendixes. Part I is an overview of both the AADL language and MBE practices. It presents basic software systems modeling and analysis using the AADL in the context of an example system, including

guidelines for effectively applying the AADL. Part II describes the characteristics of the elements of the AADL including representations, applicability, and constraints on their use. The appendixes include comprehensive listings of AADL language elements, properties that are defined as part of the AADL standard, a description of the example system used in the book, a list of references, and an index.

Terminology

AADL is a component-based modeling language that distinguishes between component interface specifications (component type declarations), component implementation blueprints (component implementation declarations), and component instances (subcomponent declarations). Component types and implementations are referred to as component classifiers. AADL also distinguishes between component categories with specific semantics to model the application software (e.g., thread, process, data), the execution platform (e.g., processor, bus, device), and composite components (system). The AADL standard document uses terms such as *system type declaration* or *system implementation declaration*. In this book, we use abbreviated terms such as *system type* or *system* where the context makes the meaning clear.

Example Application System

We use a powerboat autopilot (PBA) control system as the basis for most of the examples throughout this book. The PBA is an embedded real-time system for the speed, navigational, and guidance control of a maritime vessel. However, the PBA is an invention created to provide a backdrop for demonstrating the AADL and does not represent any specific commercial, military, or research system. While the PBA is a maritime application, it represents key elements of vehicle control for a wide range of applications including aircraft, spacecraft, and automotive and other land vehicles.³

3. Details of the PBA system are provided in Appendix A.

About the Authors

Dr. Peter Feiler is a Senior Member of Technical Staff in the Research Technology and Systems Solutions (RTSS) program at the Software Engineering Institute (SEI). He is a 27-year veteran of the SEI. His interests include architecture-centric engineering of safety-critical embedded real-time systems. He is collaborating with researchers at Carnegie Mellon University and other research institutions to develop model-based architecture technology and is investigating its practicality with commercial industry. He is the author and editor of the SAE International (formerly known as Society of Automotive Engineers) Architecture Analysis & Design Language (AADL) standard. Peter has a Ph.D. in computer science from Carnegie Mellon University and is a senior member and member of ACM, IEEE, and SAE International. He recently received the Carnegie Science Award for Information Technology.

Dr. David P. Gluch is a professor in the Department of Electrical, Computer, Software, and Systems Engineering at Embry-Riddle Aeronautical University and a visiting scientist at the Software Engineering Institute (SEI). His research interests are technologies and practices for model-based software engineering of complex systems, with a focus on software verification. Prior to joining the faculty at Embry-Riddle, he was a senior member of the technical staff at the SEI where he participated in the development and transition of innovative software engineering practices and technologies. His industrial research and development experience has included fault-tolerant computer, fly-by-wire aircraft control, Space Shuttle software modeling, and automated process control systems. He has co-authored a book on real-time UNIX systems and authored numerous technical reports and professional articles. Dave has a Ph.D. in physics from Florida State University and is a senior member of IEEE.

Acknowledgments

We would like to thank a number of people for helping make this book a reality.

We would like to thank Bruce Lewis as the chair of the SAE AADL committee in making AADL a reality. The quarterly standards meetings provided a forum for user feedback on the use of AADL. We would

also like to thank the members of the committee, especially those from industry, in helping to shape AADL into a language that meets a practical need. It was in this setting that the idea for a book on the use of AADL in model-based engineering came about.

We also appreciate the efforts of the research and advanced technology community from various universities and industry in using AADL as a platform for a wide range of formal software-reliant system analysis and demonstrating the feasibility of model-based engineering with AADL. Their tools and technology to drive the analysis of architectures allows AADL to show off its strength.

At the Software Engineering Institute (SEI), we are thankful to Tricia Oberndorf and Linda Northrop, our program managers for allowing us to invest time and effort into the endeavor of writing this book and encouraging us to bring it to completion. The other SEI AADL team members, Lutz Wrage, Aaron Greenhouse, John Hudak, Joseph Seibel, Dio DeNiz, and Craig Meyers, contributed in various form to the body of knowledge on the use of AADL, a small portion of which is reflected in this book. They led and contributed to the creation and use of the OSATE tool set, the development and presentation of tutorials and two courses on AADL, and the use of AADL on customer projects. We also received feedback on various drafts of the book.

We appreciate the feedback from external reviewers of book drafts, in particular Bruce Lewis, Jérôme Hugues, and Oleg Sokolsky. Finally, we want to thank Peter Gordon and Kim Boedigheimer from Addison-Wesley for the production of the book.

Modeling and Analysis with the AADL: The Basics

In this chapter, we illustrate the development of basic AADL models and present general guidance on the use of some of the AADL's core capabilities. With this, we hope to provide a basic understanding of architectural modeling and analysis and start you on your way in applying the AADL to more complex software-dependent systems.

While reading the first part of this chapter, you may want to use an AADL development tool to create the specifications and conduct the analyses described. OSATE supports all of the modeling and analyses discussed in this chapter.

3.1 Developing a Simple Model

In this section, we present a step-by-step development and analysis of an AADL model. Specifically, we model a control system that provides a single dimension of speed control and demonstrate some of the analyses that can be conducted on this architectural model. The speed control functionality is part of a powerboat autopilot (PBA) system that is

detailed in Appendix A. While specialized to a powerboat, this model exemplifies the use of the AADL for similar control applications such as aeronautical, automotive, or land vehicle speed control systems.

The approach we use is introductory, demonstrating the use of some of the core elements and capabilities of the AADL. We do not include many of the broader engineering capabilities of the language. For example, we do not address packages, prototypes, or component extensions in developing this simple model. These are discussed later in this chapter. Instead, we proceed through the generation of a basic declarative model and its instance and show a scheduling analysis of the system instance. During your reading of this section, you may want to reference Part II for details on specific AADL elements or analyses used in the example.

Initially we create a high-level system representation using AADL system, process, and device components. Building on this initial representation, we detail the runtime composition of all of the elements; allocate software to hardware resources; and assign values to properties of elements to a level that is required for analysis and for the creation of an instance of the system. In these steps, we assume that requirements are sufficiently detailed to provide a sound basis for the architectural design decisions and trade-offs illustrated in the example. In addition, while we reference specific architectural development and design approaches that put the various steps into a broader context, we do not advocate one approach over another.

3.1.1 Defining Components for a Model

A first step is to define the components that comprise the system and place their specification in packages. The process of defining and capturing components is similar to identifying objects in an object-oriented methodology. It is important to realize that components may include abstract encapsulations of functionality as well as representations of tangible things in the system and its environment. The definition of components is generally an iterative and incremental process, in that the first set of components may not represent a complete set and some components may need to be modified, decomposed, or merged with others.

First, we review the description of the speed controller for the PBA and define a simplified speed control model. In this model, we include a pilot interface unit for input of relevant PBA information, a speed

sensor that sends speed data to the PBA, the PBA controller, and a throttle actuator that responds to PBA commands.

For each of the components identified, we develop type definitions, specifically defining the component's name, runtime category, and interfaces. Since we are initially developing a high-level (conceptual) model, we limit the component categories to system, process, and device.

The initial set of components is shown in Table 3-1, where both the AADL text and corresponding graphical representations are included. For this example, the textual specifications of all of the components required for the model are contained in a single package and no references to classifiers outside the package are required. Thus, a package name is not needed when referencing classifiers. For the graphical representations, the implementation relationship is shown explicitly. Note that the icon for an implementation has a bold border when compared to the border of its corresponding type icon.

The speed sensor, pilot interface, and throttle actuator are modeled as devices and the PBA control functions are represented as a process. We use the devices category for components that are interfaces to the external environment that we do not expect to decompose extensively (e.g., a device can only have a bus as a subcomponent).

Devices in AADL can represent abstractions of complex components that may contain an embedded processor and software. With a device component, you represent only those characteristics necessary for analysis and an unambiguous representation of a component. For example, in modeling a handheld GPS receiver, we may only be interested in the fact that position data is available at a communication port. The fact that the GPS receiver has an embedded processor, memory, touch screen user interface, and associated software is not required for analysis or modeling of the system. Alternatively, a device can represent relatively simple external components, such as a speed sensor, whose only output is a series of pulses whose frequency is proportional to the speed being sensed. If you require a complex interface to the external environment, you can use a system component. In this case, you can detail its composition and as needed include an uncomplicated device subcomponent to represent the interface to the environment.

The use of a process component for the control functions reflects the decision that the core control processing of the PBA is to be implemented in software. The software runtime components will be contained within an implementation of this process type. The implementation declarations in Table 3-1 do not include any details. As

Table 3-1: Component Type and Implementations for the Speed Control Example

<pre> classDiagram class sensor { <> sensor_data } class sensor_speed { <> sensor_data } class interface { <> set_speed <> disengage } class interface_pilot { <> set_speed <> disengage } class control { <> command_data <> sensor_data <> set_speed <> disengage } class control_speed { <> command_data <> sensor_data <> set_speed <> disengage } class actuator { <> cmd } class actuator_speed { <> cmd } sensor < -- sensor_speed interface < -- interface_pilot control < -- control_speed actuator < -- actuator_speed </pre>	<pre> device sensor features sensor_data: out data port; end sensor; device implementation sensor.speed end sensor.speed; device interface features set_speed: out data port; disengage: out event port; end interface; device implementation interface.pilot end interface.pilot; process control features command_data: out data port; sensor_data: in data port; set_speed: in data port; disengage: in event port; end control; process implementation control.speed end control.speed; device actuator features cmd: in data port; end actuator; device implementation actuator.speed end actuator.speed; </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

the design progresses, we will add to these declarations (i.e., adding subcomponents and properties as appropriate).

The interfaces for the PBA components are port features declared within a component type and are reflected in each implementation of that type. For example, the type *sensor* outputs a value of the speed via a data port *sensor_data*. The pilot interface type *interface* provides a value for the set speed via a data port *set_speed* and generates a signal to disengage the speed control via an event port *disengage*.

Notice that we have used explicit as well as abbreviated naming for the ports and other elements of the model (e.g., *command_data* and *cmd* for the command data at input and output ports). The specificity of names is up to you, provided they comply with AADL naming constraints for identifiers (e.g., the initial character cannot be a numeral). Note that naming is case insensitive and *Control* is the same name as *control*.

In the PBA example, we have chosen to assign specific runtime component categories to each of the components (e.g., the speed sensor is a device). However, in real-world development as a design matures, the definition of these components may change (e.g., a component that computes the PBA speed control laws may initially be represented as a system and later modified to a process or thread). Using the approach we outline here, these changes are done manually within the AADL model (i.e., changing a system declaration to a process category declaration). An alternative approach is to use the generic abstract component category (i.e., not defining a specific runtime essence). Then later in the development, converting this abstract category into a specific runtime category employing the AADL **extends** capability (e.g., converting an abstract component to a thread). We have chosen to use the former approach to simplify the presentation and focus on decisions and issues related to representations of the system as concrete runtime components. A discussion of the use of the abstract component category is provided in Section 3.5.

For each of the component types we define a single implementation. These declarations are partial, in that we omit substantial details needed for a complete specification of the architecture. For example, we do not define the type of data that is associated with the ports. We will address these omissions as required in later steps. However, we can conduct a number of analyses for our simple example without including many of these details.

3.1.2 Developing a Top-Level Model

In the next step, we integrate the individual component implementations into a system by declaring subcomponents instances and their connections. We do this by defining an enclosing system type and implementation as shown in Listing 3-1, where we define a system type *Complete* and its implementation *Complete.PBA_speed_control*. There is nothing special about our choice of naming for this enclosing system. Another naming scheme, such as a type of *PBA* and an implementation of *PBA.speed*, would work as well.

Within the implementation, we declare four subcomponents. The three device subcomponents represent the speed sensor, throttle, and the pilot interface unit. The process subcomponent *speed_control* represents the software that provides the speed control for the PBA. Notice that there are no external interfaces for the system type *Complete*. All of the interactions among the system's subcomponents are internal to the implementation *Complete.PBA_speed_control*, with the devices that comprise the system providing the interfaces to the external environment (e.g., sensors determining speed information from the vehicle).

Within the implementation, we define connections for each of the ports of the subcomponents. For example, connection *DC2* is the data connection between the *command_data* port on the process *speed_control* and the *cmd* data port on the device *throttle*. Each of the connections is labeled in the graphical representation shown in Listing 3-1 by the nature of the connection.¹ For example, connection *EC4* between the event port *disengage* on the *interface_unit* device and the event port *disengage* on the *speed_control* process is labeled as <<Event>>. It is our choice to match most of the port names. It is not required that connected ports have the same name. However, they must have matching data classifiers if specified (they are omitted in this initial representation).

Listing 3-1: Subcomponents of the Complete PBA System

```

system Complete
end Complete;

system implementation Complete.PBA_speed_control
  subcomponents
    speed_sensor: device sensor.speed;

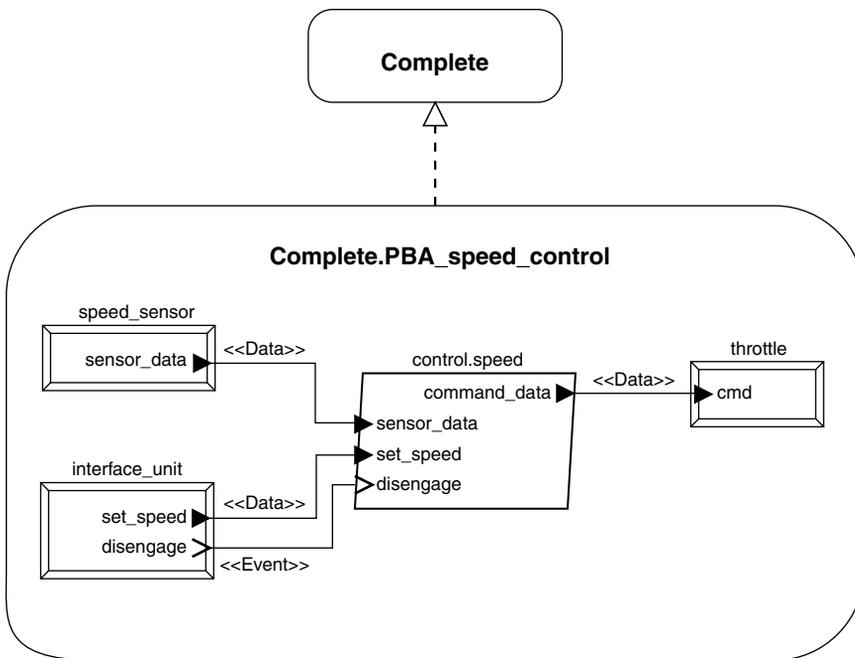
```

1. The detailed graphical representation of the implementation *Complete.PBA_speed_control* is taken from the OSATE environment.

```

throttle: device actuator.speed;
speed_control: process control.speed;
interface_unit: device interface.pilot;
connections
DC1: port speed_sensor.sensor_data ->
    speed_control.sensor_data;
DC2: port speed_control.command_data -> throttle.cmd;
DC3: port interface_unit.set_speed ->
    speed_control.set_speed;
EC4: port interface_unit.disengage ->
    speed_control.disengage;
end Complete.PBA_speed_control;

```



Depending upon your development environment the graphical portrayals may differ from those shown in Listing 3-1. For example, within OSATE you cannot display the containment explicitly. Rather, the internal structure of an implementation is presented in a separate diagram that can be accessed hierarchically through the graphical icon representing the implementation *Complete.PBA_speed_control*.

3.1.3 Detailing the Control Software

At this point, we begin to detail the composition of the process *speed_control*. This involves decisions relating to partitioning the functionality and responsibilities required of the PBA system to provide speed control. Since we have treated the speed control as an autonomous capability, we have implicitly assumed that there are no interactions between the directional or other elements of the PBA and the speed control system. This may not be the case in advanced control systems. In addition, for the purposes of this example, we partition the functions of the speed control process into two subcomponents. The first is a thread that receives input from speed sensor; scales and filters that data; and delivers the processed data to the second thread. The second is a thread that executes the PBA speed control laws and outputs commands to the throttle actuator. Again, this simplification may not be adequate for a realistic speed control system (e.g., the control laws may involve extensive computations that for efficiency must be separated into multiple threads or may involve complex mode switches that are triggered by various speed or directional conditions)².

Since the interfaces for the two threads are different, we define a type and implementation for each, as shown in Listing 3-2. We have used property associations to assign execution characteristics to the threads. Each is a periodic thread (assigned using the *Dispatch Protocol* property association) with a period of 50ms (assigned using the *Period* property association).

The assignment of periodic execution and the values for the period of the threads reflect design decisions. Generally, these are based upon the input of application domain and/or control engineers. The assignments we use here are not necessarily optimal but are chosen to provide specific values to enable analysis of system performance. They do not reflect the values for any specific control system.

Listing 3-2: PBA Control Threads Declarations

```
thread read_data
  features
    sensor_data: in data port;
    proc_data: out data port;
```

2. In the next section we will demonstrate the addition of operational modes.

```

properties
    Dispatch_Protocol => Periodic;
    Period => 50 ms;
end read_data;

thread implementation read_data.speed
end read_data.speed;

thread control_laws
    features
        proc_data: in data port;
        cmd: out data port;
        disengage: in event port;
        set_speed: in data port;
    properties
        Dispatch_Protocol => Periodic;
        Period => 50 ms;
end Control_laws;

thread implementation control_laws.speed
end control_laws.speed;

```

We detail the declaration of the process implementation *control.speed* that is presented in Table 3-1 to include the two thread subcomponents and their interactions (**connections**), as shown in Listing 3-3. There are five connections declared. Four of these connect ports on the boundary of the process with ports on the threads (i.e., *DC1*, *DC3*, *DC4*, and *EC1*). The fifth connects the out data port *proc_data* on the thread *scale_speed_data* to the in data port *proc_data* on the thread *speed_control_laws*.

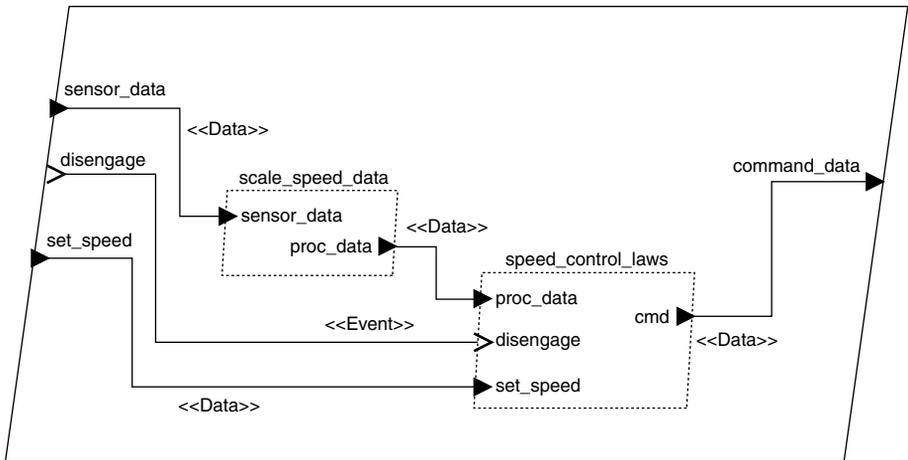
Listing 3-3: *The Process Implementation control.speed*

```

process implementation control.speed
    subcomponents
        scale_speed_data: thread read_data.speed;
        speed_control_laws: thread control_laws.speed;
    connections
        DC1: port sensor_data -> scale_speed_data.sensor_data;
        DC2: port scale_speed_data.proc_data ->
            speed_control_laws.proc_data;
        DC3: port speed_control_laws.cmd -> command_data;
        EC1: port disengage -> speed_control_laws.disengage;
        DC4: port set_speed -> speed_control_laws.set_speed;
end control.speed;

```

continues



3.1.4 Adding Hardware Components

At this point, we have defined the software components of the speed control system. We now define the execution hardware required to support the software. In modeling the hardware and binding the control software to that hardware, we can analyze the execution timing and scheduling aspects of the system.

In Listing 3-4, we define a processor, memory, and bus. The processor will execute the PBA control code (threads) and the memory will store the executable code (process) for the system. In addition, we have declared that the processor type *Real_Time* and the memory type *RAM* require access to an instance of the bus implementation *Marine.Standard*. This bus will provide the physical pathway for the system. We will add properties to these declarations later in the modeling process.

Listing 3-4: *Execution Platform Declarations*

```

processor Real_Time
  features
    BA1: requires bus access Marine.Standard;
  end Real_Time;
processor implementation Real_Time.one_GHz
end Real_Time.one_GHz;

memory RAM
  features
    BA1: requires bus access Marine.Standard;
  end RAM;

```

```

memory implementation RAM.Standard
end RAM.Standard;

bus Marine
end Marine;

bus implementation Marine.Standard
end Marine.Standard;

```

3.1.5 Declaring Physical Connections

To continue the integration of the system, we add instances of the required execution platform components into the system implementation *Complete.PBA_speed_control* by declaring subcomponents for the implementation. In addition, we declare that these components are attached to the bus. This is done by connecting the **requires** interfaces on the processor and memory components to the bus component.

Since the PBA control software executing on the processor must receive data from the sensors and pilot interface unit as well as send commands to the throttle actuator, we declare that these sensing and actuator devices are connected to the bus as well. To do this, we add **requires bus access** declarations in the type declarations for these three devices and connect them to the bus. The updated declarations for the three devices are shown in Listing 3-5 and the graphical representation of the system with the declaration of the physical (**bus access**) connections is shown in Listing 3-6.

Listing 3-5: Updated Device Declarations

```

device interface
  features
    set_speed: out data port;
    disengage: out event port;
    BA1: requires bus access Marine.Standard;
end interface;

device sensor
  features
    sensor_data: out data port;
    BA1: requires bus access Marine.Standard;
end sensor;

```

continues

```

device actuator
  features
    cmd: in data port;
    BA1: requires bus access Marine.Standard;
end actuator;

```

In Listing 3-6, we have defined a processor *RT_1GHz*³, bus *Standard_Marine_Bus*, and memory *Stand_Memory* as subcomponents. In addition, we have declared the connections for the bus *Standard_Marine_Bus* to the **requires bus access** features of each of the physical components (e.g., from *Standard_Marine_Bus* to *RT_GHz.BA1* and to the **requires bus access** feature on the processor *RT_1GHz.BA1*). The **requires access** features and the bus access connections are shown in the graphical representation in the lower portion of Listing 3-6.

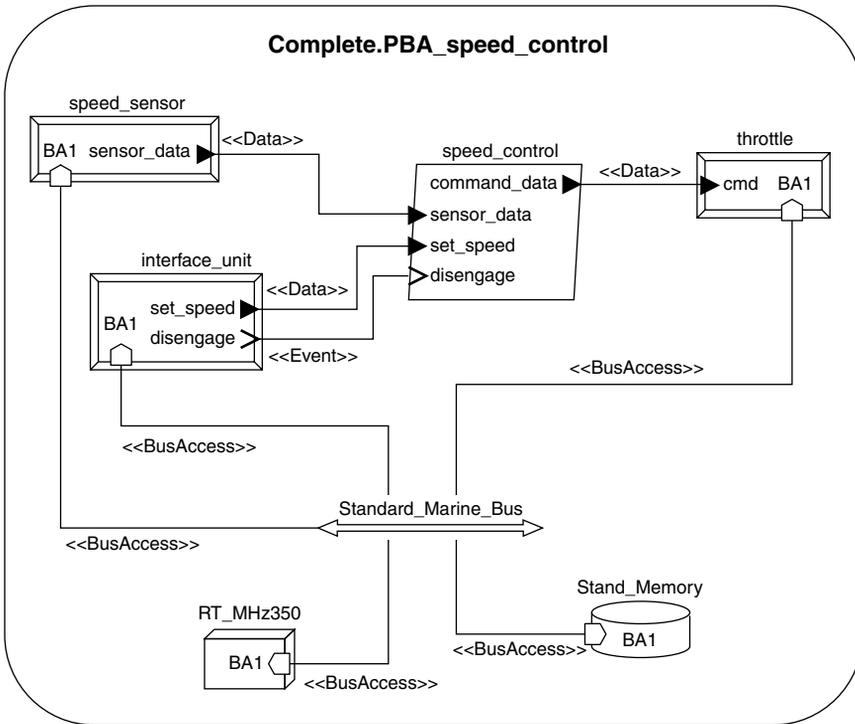
Listing 3-6: *Integrated Software and Hardware System*

```

system implementation Complete.PBA_speed_control
  subcomponents
    speed_sensor: device sensor.speed;
    throttle: device actuator.speed;
    speed_control: process control.speed;
    interface_unit: device interface.pilot;
    RT_1GHz: processor Real_Time.one_GHz;
    Standard_Marine_Bus: bus Marine.Standard;
    Stand_Memory: memory RAM.Standard;
  connections
    DC1: port speed_sensor.sensor_data ->
        speed_control.sensor_data;
    DC2: port speed_control.command_data -> throttle.cmd;
    DC3: port interface_unit.set_speed ->
        speed_control.set_speed;
    EC4: port interface_unit.disengage ->
        speed_control.disengage;
    BAC1: bus access Standard_Marine_Bus <-> speed_sensor.BA1;
    BAC2: bus access Standard_Marine_Bus <-> RT_1GHz.BA1;
    BAC3: bus access Standard_Marine_Bus <-> throttle.BA1;
    BAC4: bus access Standard_Marine_Bus <-> interface_unit.BA1;
    BAC5: bus access Standard_Marine_Bus <-> Stand_Memory.BA1;
end Complete.PBA_speed_control;

```

3. Since it is not the first character in the name, the numeric 1 can be used within the processor subcomponent name *RT_1GHz*. However, an implementation name *Real_Time.1_GHz* is not legal, since the numeric is the first character in the implementation identifier.



3.1.6 Binding Software to Hardware

In addition to specifying the physical connections, we bind software components to the appropriate physical hardware using contained property associations, as shown in Listing 3-7. These property associations are added to the system implementation declaration *Complete.PBA_speed_control*. The first two declarations allow the threads *speed_control_laws* and *scale_speed_data* to be bound to the processor *rt_mhz500*. The reference part of the property association identifies the specific processor instance *rt_mhz500* and the **applies to** identifies the specific thread in the hierarchy (e.g., **applies to** *speed_control.scale_speed_data* identifies the thread *scale_speed_data* that is located in the process *speed_control*). In this notation, a period separates the elements in the hierarchy.

We could have specified a specific binding using the *Actual_Processor_Binding* property. However, the *Allowed_Processor_Binding* property permits scheduling tools to assign the threads to processors. For example, the resourced allocation and scheduling analysis plug-in

that is available in the OSATE environment⁴ binds threads to processors taking into consideration the threads' period, deadline, and execution time; processor(s) speed and scheduling policies; and the constraints imposed by the actual and allowed binding properties⁵. Specifically, if only allowed processor bindings are defined (i.e., the *Allowed_Processor_Binding* property), the plug-in schedules the thread onto processors and reports back the actual thread to processor bindings and the resulting processor utilizations. If actual processor bindings are defined (i.e., the *Actual_Processor_Binding* property) the plug-in reports processor utilization based upon those bindings; allocates threads to processors; and runs a scheduling analysis to determine whether the bindings are acceptable.

Generally, a scheduling analysis or scheduling tool does scheduling analysis such that given a set of threads and their binding to processors (allowed or actual bindings), requisite attributes of the threads and processors (e.g., period, worst case execution time, processor cycle time, etc.), and defined scheduling policy, it determines if the set of threads meets the system's timing requirements. Typical scheduling policies include round-robin (RR), shortest job first (SJF), earliest deadline first (EDF), and rate monotonic (RM). The specific information required by and output from scheduling analysis tools vary.

The OSATE resource allocation and scheduling analysis plug-in makes binding decisions and in that process runs a scheduling analysis determining whether the binding is acceptable. This can be based on earliest deadline first (EDF) and rate monotonic scheduling (RMS) for periodic threads. In addition, it can conduct a rate monotonic analysis for periodic threads. This is useful for control system applications where all tasks are periodic, such as the PBA speed control example. In cases where threads are already bound to processors (i.e., using the *Actual_Processor_Binding* property), the plug-in determines schedulability for that specific deployment configuration.

If priority is assigned by hand and rate monotonic scheduling is used, another OSATE plug-in (priority inversion checker) enables the determination of whether the system has potential priority inversion. More sophisticated schedulability analysis tools are available for analyzing AADL models. A listing of these is available at <https://wiki.sei.cmu.edu/aadl>.

4. The resource allocation and scheduling analysis OSATE plug-in combines a bin-packing algorithm with scheduling algorithms. The OSATE tool is available for download from www.aadl.info.

5. Some scheduling policies may require additional properties, such as explicit priority assignment. The scheduling tool in OSATE assumes all periodic tasks without shared logical resources; other scheduling tools, such as Cheddar, accommodate the full set of tasks in AADL including tasks with shared data components.

The third entry in Listing 3-7, binds the code and data within the process *speed_control* to the memory component *Standard_Memory*. We chose to use the actual rather than the allowed memory binding property, since there is only one memory component in the system and, while an additional processor might be added, we do not anticipate additional memory components to be added.

Listing 3-7: *Binding Property Associations*

properties

```

Allowed_Processor_Binding => (reference(RT_1GHz))
    applies to speed_control.speed_control_laws;
Allowed_Processor_Binding => (reference(RT_1GHz))
    applies to speed_control.scale_speed_data;
Actual_Memory_Binding => (reference(Stand_Memory))
    applies to speed_control;

```

3.1.7 Conducting Scheduling Analyses

Having defined the threads and established their allowed bindings to processors, we can begin to assess processor loading and analyze the schedulability of the system.

Before we proceed with a scheduling analysis, we define the requisite execution characteristics for the threads as they relate to the capabilities of the processors to which they may be bound. We specify this information through properties of the threads and processors. In this case, there is only one processor with an execution speed of 1GHz, as shown in Listing 3-8. Both threads are declared as *Periodic* with a period of 50ms. The default value in the AADL standard for the *Deadline* is the value of the *Period*. This value can be overridden by assigning a value to the *Deadline* using a property association. The execution time of the *read_data* thread ranges from 1 millisecond (ms) to 2 milliseconds (ms), whereas the *control_laws* thread's execution time ranges from 3ms to 5ms (as assigned using the *Compute_Execution_Time* property associations). These execution times are relative to the processor *Real_Time.one_GHz* declared in the model⁶.

6. If an AADL model has a single type of processor (i.e., only one processor speed) then the execution time is with respect to that processor. If there are multiple processors with different speeds, you can specify an execution time for each processor type (using in binding) or specify an execution time with respect to one of the other processors (the reference processor) using a scaling factor that is associated with each processor type. There is a *Reference_Processor* property and a *Scaling_Factor* property for this purpose.

Execution time estimates for the threads can be based upon timing measurements from prototype code or historical data for similar systems (e.g., systems with the same or comparable processors). By conducting the analysis early in the development process, you can get quantitative predictions of a system's performance. This information can be updated and re-evaluated as the design progresses. These early and continuing predictions can help to avoid last minute problems during code implementation and system integration (e.g., during testing when deadlines are not met because the processor loading exceeds the capability of the processor).

Listing 3-8: *Updated Declarations for Analysis*

```

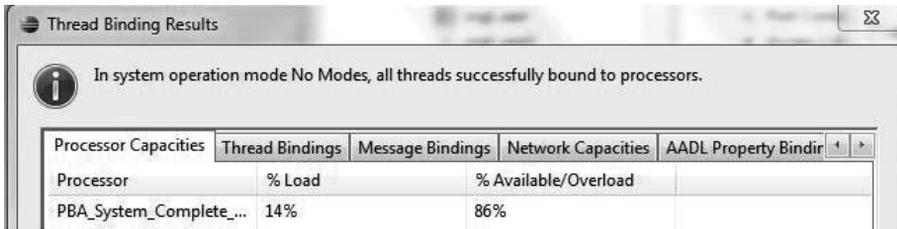
thread read_data
  features
    sensor_data: in data port;
    proc_data: out data port;
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 1 ms .. 2 ms;
    Period => 50 ms;
end read_data;

thread control_laws
  features
    proc_data: in data port;
    cmd: out data port;
    disengage: in event port;
    set_speed: in data port;
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 3 ms .. 5 ms;
    Period => 50 ms;
end Control_laws;

```

At this point, we have defined a declarative model for a simple speed control system including all of the components, properties, and bindings to describing a deployment configuration. From the top-level system implementation of this declarative model you create a system instance model and analyze it with the OSATE scheduler and scheduling analysis plug-in.

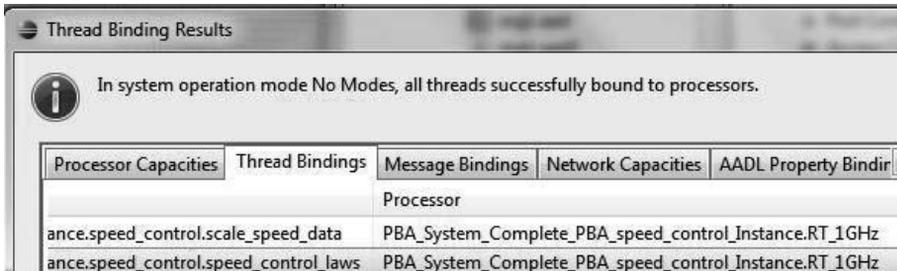
In Figure 3-1, we show the results of that analysis. It shows that the two threads in the system only use 14% of the processor capabilities. The worst case execution time for the two threads is 7ms, which is 14% of their 50 millisecond period.



The screenshot shows a window titled "Thread Binding Results" with a status message: "In system operation mode No Modes, all threads successfully bound to processors." Below this is a table with tabs for "Processor Capacities", "Thread Bindings", "Message Bindings", "Network Capacities", and "AADL Property Bindir". The "Processor Capacities" tab is active, showing a table with columns "Processor", "% Load", and "% Available/Overload".

Processor	% Load	% Available/Overload
PBA_System_Complete_...	14%	86%

Figure 3-1: *Processor Capacities of the Speed Control System Instance*



The screenshot shows the same "Thread Binding Results" window, but with the "Thread Bindings" tab selected. The table shows the binding of threads to processors.

Thread	Processor
ance.speed_control.scale_speed_data	PBA_System_Complete_PBA_speed_control_Instance.RT_1GHz
ance.speed_control.speed_control_laws	PBA_System_Complete_PBA_speed_control_Instance.RT_1GHz

Figure 3-2: *Bindings from the OSATE Scheduler and Scheduling Analysis Plug-in*

Notice the information provided by the plug-in, including the actual binding for the threads determined by the plug-in (as shown in the cropped output of Figure 3-2).

3.1.8 Summary

At this point, we have developed a basic architectural model of the PBA speed control system. In so doing, we have demonstrated some of the core capabilities of the AADL. For this relatively simple model, we analyzed the execution environment and made predictions of schedulability of the system. In subsequent sections, we describe additional capabilities of the AADL and discuss alternative modeling approaches that can be applied in this simple example.

3.2 Representing Code Artifacts

Within a comprehensive AADL architectural specification, source code files and related information needed for specifying and developing the

software within a system are documented using standard properties. These properties capture information for documenting architectural views such as code views [Hofmeister 00] and implementation views of allocation view types [Clements 10].

In this section, we document information relating to the PBA application software contained within the process *control.speed*. This excludes software that may be resident in the sensors, actuators, and interface devices as well as the operating system within the processor. First, we assume the application software has been written in a programming language, such as C or Java or in a modeling language, such as Simulink. We also assume that the software has been organized using the capabilities of the source language (e.g., by organizing Java classes and methods into packages with public and private elements). In this case, we can focus on specifying a mapping of the source files into the processes and threads of the application runtime architecture. Section 3.2.1 illustrates this mapping, which can be used to generate build scripts from the AADL model. Section 3.2.2 discusses how you can map identifier names used in AADL to identifier names that are acceptable in the source language. For larger systems, we may want to reflect not only the application runtime architecture in AADL, but also the modular source code structure. Section 3.2.3 illustrates how we utilize AADL packages for that purpose.

3.2.1 Documenting Source Code and Binary Files

A modified excerpt of the PBA specification is shown in Listing 3-9. This includes a properties section within the implementation *control.speed*, where the property association for the property *Source_Language* declares that the source code language for the implementation is C. This property is of type *Supported_Source_Languages*, which is defined in the property set *AADL_Project* and has the enumeration values (Ada95, C, Simulink_6_5 are some examples). Property types and constants in the *AADL_Project* property set can be tailored for specific projects. For example, languages such as Java can be added to the *Supported_Source_Languages* property type.

Using a property association for the newly defined property *Source_Language*, the C language is declared as the programming language for all of the source code involved in the process *control.speed*. Two contained property associations for the property *Source_Text* identify the source and object code files for the threads *speed_control_laws* and *scale_speed_data*. Two other contained property associations for the property

Source_Code_Size define the size of the compiled, linked, bound, and loaded code used in the final system.

In Listing 3-9, the data type *sampled_speed_data* is declared with a property association for the property *Source_Data_Size*. This property specifies the maximum size required for an instance of the data type. This data type is the classifier for the ports associated with the data that originates at the speed sensor.

Listing 3-9: PBA Specification with Code Properties

```

process implementation control.speed
  subcomponents
    scale_speed_data: thread read_data.speed;
    speed_control_laws: thread control_laws.speed;
  connections
    DC1: port sensor_data -> scale_speed_data.sensor_data;
    DC2: port scale_speed_data.proc_data ->
      speed_control_laws.proc_data;
    DC3: port speed_control_laws.cmd -> command_data;
    EC1: port disengage -> speed_control_laws.disengage;
    DC4: port set_speed -> speed_control_laws.set_speed;
  properties
    Source_Language => (C);
    Source_Text => ("ControlLaws.cc", "ControlLaws.obj")
      applies to speed_control_laws;
    Source_Text => ("ScaleData.cc", "ScaleData.obj")
      applies to scale_speed_data;
    Source_Code_Size => 4 KByte applies to scale_speed_data;
    Source_Code_Size => 10 KByte applies to speed_control_laws;
end control.speed;

```

3.2.2 Documenting Variable Names

A data port maps to a single variable in the application code. For example, the variable name for a data port can be specified using the *Source_Name* property. This is shown in Listing 3-10, for the in data port *set_speed* whose data classifier is the data type *set_speed_value*. The variable name for this port in the source code is *SetValue*.

We can use this mechanism to map data type and other component identifiers in an AADL model into the corresponding name in the source code. This is useful if the syntax of the source language allows characters in identifiers that are not allowed in AADL. We may also use this if we want to introduce more meaningful names in the AADL model for cryptic source code names.

Listing 3-10: *Example of Documenting Variable Names*

```
thread control_laws
  features
    proc_data: in data port;
    cmd: out data port;
    disengage: in event port;
    set_speed: in data port set_speed_value
      {Source_Name => "SetValue"};
  properties
    Dispatch_Protocol => Periodic;
    Period => 50 ms;
end control_laws;

data set_speed_value
end set_speed_value;
```

3.2.3 Modeling the Source Code Structure

Source code expressed in programming languages typically consists of data types and functions. They may take the form of subprograms and functions, classes and methods, or operations on objects. These source code elements are typically organized into libraries and modular packages. Some of the library or package content is considered public (i.e., it can be used by others), whereas other parts are considered private (i.e., can only be used locally). In the case of modeling languages such as Simulink, block libraries play a similar role.

Sometimes it is desirable to represent this modular source code structure in the AADL model. We can do so by making use of the AADL package concept. For example, we can model the functions making up a Math library by placing the subprogram type declarations representing the function signatures into an AADL package together with the subprogram group declaration representing the library itself, as shown in Listing 3-11.

We can place data component types that represent classes within the same source code package, into one AADL package. We can place the data component type and the subprogram types representing the operations on the source code data type in the same package. The methods of classes can be recorded as subprogram access features of the data component type (see Section 4.2.5). Any module hierarchy in the source code can be reflected in the AADL package naming hierarchy. For more on the use of AADL packages to organize component declarations into packages, see Section 4.4.1.

Listing 3-11: Example of Modular Structure

```

package MathLib
  public
  with Base_Types;
  subprogram group Math_Library
  features
    sqrt: provides subprogram access SquareRoot;
    log: provides subprogram access;
    pow: provides subprogram access;
  end Math_Library;

  subprogram SquareRoot
  features
    input: in parameter Base_Types::float;
    result: out parameter Base_Types::float;
  end SquareRoot;

end MathLib;

```

3.3 Modeling Dynamic Reconfigurations

Modes can be used to model various operational states and the dynamic reconfiguration of a system or component. In this section, we present the use of modes to represent the operation of the PBA speed control system. In this section, we develop another, slightly expanded model of the PBA speed control system.

3.3.1 Expanded PBA Model

We modify the PBA speed control model to include a *display_unit*. In addition, we add an out event port *control_on* to the *interface_unit*. Figure 3-3 shows the implementation of the expanded system including its subcomponents and their interconnections.

The type classifiers used in the expanded PBA model are shown in Listing 3-12. In this table, we define a process type *control_ex* that includes the additional features required for interfacing to the *display_unit* and *interface_unit*. We could have declared this process type as an extension of the process type *control*, as shown in the comment. We also define the device type *interface_unit* as the device type *interface* with an additional port. Finally, we have added a new device type *display_unit*. The event port *control_on* is the trigger for a mode transition from *monitoring* to *controlling* and the event port *disengage* is the trigger for the reverse transition.

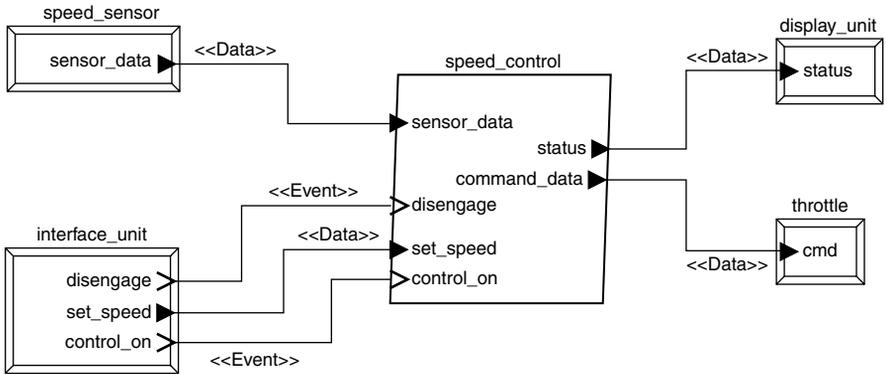


Figure 3-3: Expanded PBA Control System

Listing 3-12: Type Classifiers for the Expanded PBA Model

```

-- Type classifiers for the expanded PBA control system model --
process control_ex
  features
    sensor_data: in data port;
    command_data: out data port;
    status: out data port;      -- added port
    disengage: in event port;
    set_speed: in data port;
    control_on: in event port;  -- added port
  properties
    Period => 50 Ms;
end control_ex;

device interface_unit
  features
    disengage: out event port;
    set_speed: out data port;
    control_on: out event port;  -- added port
end interface_unit;

device display_unit          -- new device
  features
    status: in data port;
end display_unit;

thread monitor              -- new thread
  features
    sensor_data: in data port;
    status: out data port;
end monitor;

```

```

thread control_laws_ex
  features
    proc_data: in data port;
    set_speed: in data port;
    disengage: in event port;
    control_on: in event port;    -- added port
    status: out data port;       -- added port
    cmd: out data port;
end control_laws_ex;
    
```

Listing 3-12 also includes the new thread type *monitor*, and the modified thread type *control_laws* with an extra port, now called *control_laws_ex*. These are the thread types of the subcomponents of the process implementation *control_ex.speed*. A graphical representation of the subcomponents and connections for the process implementation *control_ex.speed* is shown in Figure 3-4. The process *speed_control*, shown in Figure 3-3, is an instance of *control_ex.speed*.

3.3.2 Specifying Modes

The textual specification for the implementation *control_ex.speed* is shown in Listing 3-13. In this implementation, two modes *monitoring* and *controlling* are declared in the **modes** section of the implementation. In the declarations for the subcomponents, the **in modes** declarations

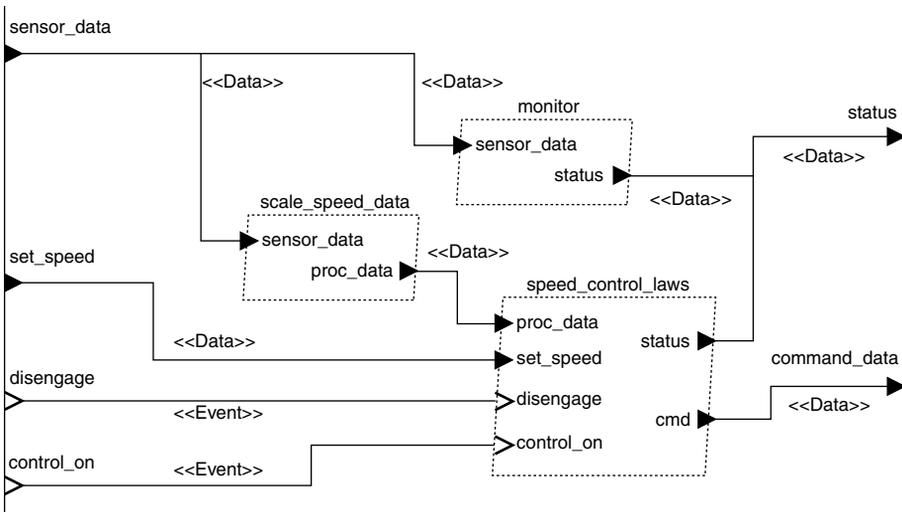


Figure 3-4: Process Implementation *control_ex.speed*

constrain the thread *monitor* to execute only in the *monitoring* mode and the threads *scale_speed_data* and *speed_control_laws* execute only in the *controlling* mode. Similarly, in the connection declarations are mode dependent such that the connections to the *monitor* thread are only active in the monitoring mode. The transitions between modes are triggered by the in event ports *control_on* and *disengage*. These are declared in the **modes** section of the implementation. If the modes are observable or are controlled from outside a component, then you may want to declare the modes in the component type.

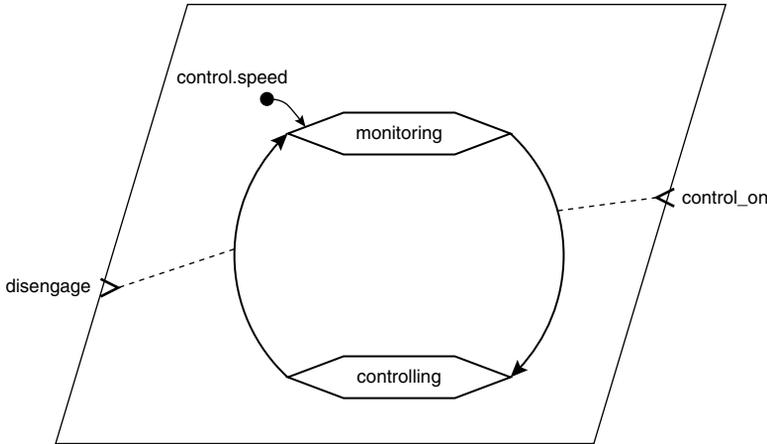
A graphical representation for the mode transitions is shown in the lower portion of Listing 3-13. Modes are represented as dotted hexagons. The short arrow terminating at the monitoring mode denotes that the initial state is *monitoring*. The arrows connecting modes represent transitions. The input events are associated with the transitions that they trigger with a dotted line.

Listing 3-13: *Process Implementation of control_ex.speed with Modes*

```

process implementation control_ex.speed
  subcomponents
    scale_speed_data: thread read_data in modes (controlling);
    speed_control_laws: thread control_laws_ex
      in modes (controlling);
    monitor: thread monitor in modes (monitoring);
  connections
    DC1: port sensor_data -> scale_speed_data.sensor_data
      in modes (controlling);
    DC2: port scale_speed_data.proc_data ->
      speed_control_laws.proc_data in modes (controlling);
    DC3: port speed_control_laws.cmd -> command_data
      in modes (controlling);
    DC4: port set_speed -> speed_control_laws.set_speed
      in modes (controlling);
    DC5: port monitor.status -> status in modes (monitoring);
    DC6: port sensor_data -> monitor.sensor_data
      in modes (monitoring);
    DC8: port speed_control_laws.status -> status
      in modes (controlling);
    EC1: port disengage -> speed_control_laws.disengage
      in modes (controlling);
  modes
    monitoring: initial mode ;
    controlling: mode ;
    monitoring -[ control_on ]-> controlling;
    controlling -[ disengage ]-> monitoring;
end control_ex.speed;

```



3.4 Modeling and Analyzing Abstract Flows

One of the important capabilities of the AADL is the ability to model and analyze flow paths through a system. For example within the PBA system, it is possible to analyze the time required for a signal to travel from the interface unit, through the control system, to the throttle actuator, and result in a throttle action.

3.4.1 Specifying a Flow Model

For this section, we add flow specifications to the expanded PBA speed control system shown in Figure 3-3. We investigate a flow path (the end-to-end flow) involving a change of speed via *set_speed* that extends from the pilot's interface unit to the throttle. In specifying the flow, we define each element of the flow (i.e., a flow source, flow path(s), flow sink), using *on_flow* as a common prefix for each flow specification name. In addition, we allocate transport latencies for each of these elements. Listing 3-14 presents an abbreviated version of the specification for the PBA speed control system that includes the requisite flow declarations.

The flow source is named *on_flow_src* that exits the interface unit through the port *set_speed*. The flow proceeds through the *speed_control* process via the flow path *on_flow_path* that enters through the port *set_speed* and exits through the port *command_data*. The flow sink occurs

through the data port *cmd* in the *throttle* component. Note that a flow path can go from any kind of incoming port to a port of a different kind, for example from an event port to a data port.

Each flow specification is assigned a latency value. For example, the worst case time for the new speed to emerge from the interface unit after the pilot initiates the *set_point* request is 5ms. The worst case transit time through the *speed_control* process is 20ms and the time for the throttle to initiate an action is 8ms.⁷

Listing 3-14: *Flow Specifications for the Expanded PBA*

```
-- flow specifications are added to type declarations for this analysis ---

device interface_unit
  features
    set_speed: out data port;
    disengage: out event port;
    control_on: out event port;
  flows
    on_flow_src: flow source set_speed {latency => 5 ms .. 5 ms};
end interface_unit;

process control_ex
  features
    sensor_data: in data port;
    command_data: out data port;
    status: out data port;
    disengage: in event port;
    set_speed: in data port;
    control_on: in event port;
  flows
    on_flow_path: flow path set_speed -> command_data
      {latency => 10 ms .. 20 ms};
  properties
    Period => 50 Ms;
end control_ex;

device actuator
  features
    cmd: in data port;
    BA1: requires bus access Marine.Standard;
  flows
    on_flow_snk: flow sink cmd {latency => 8 ms .. 8 ms};
end actuator;
```

7. These latency values are illustrative and do not reflect the performance of any particular device or speed control system.

3.4.2 Specifying an End-to-End Flow

The complete path, the end-to-end flow, for this example runs from the source component *interface_unit* through to the component *throttle*. This is declared in the system implementation *PBA.expanded*, as shown in Listing 3-15. The declaration originates at the source component and its source flow *interface_unit.on_flow_src* and the connection *EC4*. It continues through *speed_control.on_flow_path*, the connection *DC2*, and terminates at the sink *throttle.on_flow_snk*. In addition, we have specified a latency of 35ms for the flow. This value is drawn from the requirements for the system.

Listing 3-15: *An End-to-End Flow Declaration*

```

system implementation PBA.expanded
  subcomponents
    speed_sensor: device sensor.speed;
    throttle: device actuator.speed;
    speed_control: process control_ex.speed;
    interface_unit: device interface_unit;
    display_unit: device display_unit;
  connections
    DC1: port speed_sensor.sensor_data ->
      speed_control.sensor_data;
    DC2: port speed_control.command_data -> throttle.cmd;
    DC3: port interface_unit.set_speed ->
      speed_control.set_speed;
    EC4: port interface_unit.disengage ->
      speed_control.disengage;
    EC5: port interface_unit.control_on->
      speed_control.control_on;
    DC6: port speed_control.status -> display_unit.status;
  flows
    on_end_to_end: end to end flow
      interface_unit.on_flow_src -> EC5 ->
      speed_control.on_flow_path -> DC2 ->
      throttle.on_flow_snk {Latency => 35 ms .. 35 ms;};
end PBA.expanded;

```

3.4.3 Analyzing a Flow

At this point, we have defined a top-level end-to-end flow, assigned an expected (required) latency value to this flow, and defined latencies for each of the elements that comprise the flow. OSATE includes a flow latency analysis tool that automatically checks to see if end-to-end latency requirements are satisfied. For example, the tool will trace the path and total the latencies for the individual elements of the flow.

Infos (2 items)	
i	End-to-end flow on_end_to_end calculated latency 33 ms is less than expected latency 35 ms
i	Expected end-to-end flow on_end_to_end latency is 35 ms

Figure 3-5: *Top Level End-to-End Flow Analysis Results*

This total is compared to the latency expected for an end-to-end flow. Figure 3-5 presents the results of this analysis for the PBA system we have specified. The total latency for the three elements of the flow at 33ms is less than the expected 35ms.

We could have manually determined, through a separate calculation, that the cumulative latency for the end-to-end flow would not violate the 35ms latency requirement. However, for very large systems, these calculations are difficult to do manually and it is difficult to ensure that latency values are correctly connected through the elements that comprise an architecture. The automated capabilities that can be included within an AADL tool facilitate easy calculation and re-calculation of these values. Moreover, having this data integral to the architecture provides a reliable way to manage the information and ensure consistency with updates to the data and the architecture.

3.5 Developing a Conceptual Model

It is possible to defer identifying the runtime nature of components until late in the development process. As noted earlier, you can do this by using system components and later manually changing the category in the relevant type, implementation, and subcomponent declarations for these components. In the next few sections, we present an alternative approach where you declare components as **abstract** and build an architectural component hierarchy. Then you use component extensions to create multi-view architectural representations. For example, in using the Siemens architecture approach [Hofmeister 00], you can include abstract components in a conceptual (runtime neutral) view and later extend these into runtime specific components, creating an execution view of the architecture.

3.5.1 Employing Abstract Components in a PBA Model

In declaring components for the PBA system, rather than using the device category for the pilot interface and the system category for the

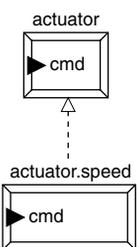
control components as we did in Table 3-1, we declare them as **abstract**. For this example, we assume there is a potential for decomposing the pilot interface into a complex interface unit. We could have made the sensor and actuator components abstract as well. However, to simplify the example and to demonstrate that you can mix abstract with runtime-specific categories, we maintain these components as devices. The declarations for this approach are shown in Table 3-2, where we have used the same partitioning and naming convention that is used in Table 3-1. Abstract components are represented graphically by dashed rectangles.

Table 3-2: *Abstract Component Declarations for the PBA*

<pre> classDiagram class sensor { <<abstract>> sensor_data : out data port } class sensor_speed { sensor_data : out data port } class interface { <<abstract>> set_speed : out data port disengage : out event port } class interface_pilot { set_speed : out data port disengage : out event port } class control { <<abstract>> sensor_data : in data port set_speed : in data port disengage : in event port command_data : out data port } class control_speed { sensor_data : in data port set_speed : in data port disengage : in event port control_speed : in data port command_data : out data port } sensor < -- sensor_speed interface < -- interface_pilot control < -- control_speed </pre>	<pre> device sensor features sensor_data: out data port; end sensor; device implementation sensor.speed end sensor.speed; abstract interface features set_speed: out data port; disengage: out event port; end interface; abstract implementation interface.pilot end interface.pilot; abstract control features command_data: out data port; sensor_data: in data port; set_speed: in data port; disengage: in event port; end control; abstract implementation control.speed end control.speed; </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

continues

Table 3-2: *Abstract Component Declarations for the PBA (continued)*

 <pre> classDiagram class actuator { cmd } class actuator_speed { cmd } actuator .. > actuator_speed </pre>	<pre> device actuator features cmd: in data port; end actuator; device implementation actuator.speed end actuator.speed; </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

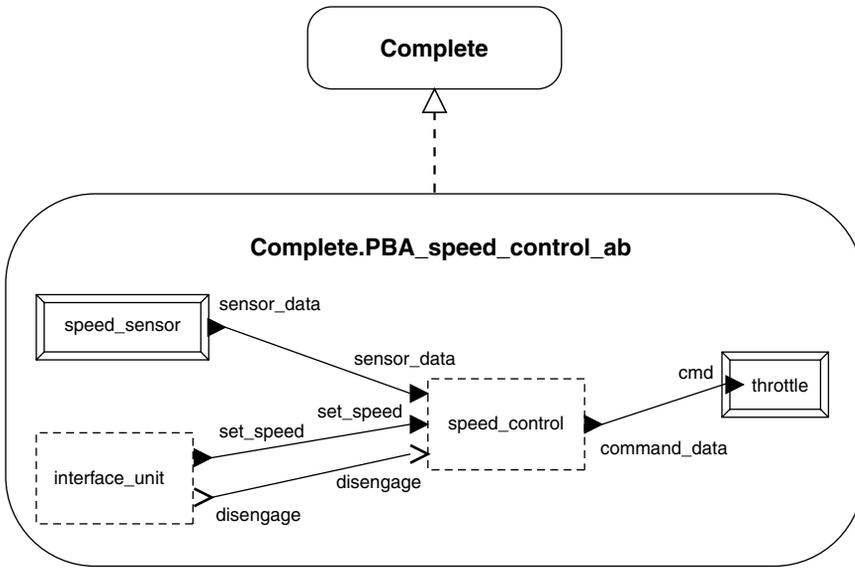
A complete system implementation using abstract components is shown in Listing 3-16. We have used an enclosing system, since we plan on instantiating it. However, we could have modeled the enclosing system as abstract as well, converting it to a system model later for instantiation. We have not included the hardware components or their relevant connections in this specification. We add these later in this discussion.

Listing 3-16: *Complete PBA System Using Abstract Components*

```

system Complete
end Complete;

system implementation Complete.PBA_speed_control_ab
  subcomponents
    speed_sensor: device sensor.speed;
    throttle: device actuator.speed;
    speed_control: abstract control.speed;
    interface_unit: abstract interface.pilot;
  connections
    DC1: port speed_sensor.sensor_data ->
      speed_control.sensor_data;
    DC2: port speed_control.command_data -> throttle.cmd;
    DC3: port interface_unit.set_speed ->
      speed_control.set_speed;
    EC4: port interface_unit.disengage ->
      speed_control.disengage;
  end Complete.PBA_speed_control_ab;
  
```



3.5.2 Detailing Abstract Implementations

In this section, we define the implementation *control.speed* for the *speed_control* subcomponent. This is shown in Listing 3-17. We detail this component by partitioning it into two subcomponents, as we did earlier in Listing 3-3. We declare the components *read_data* and *control* as abstract and include them as abstract subcomponents in the abstract implementation *control.speed*. In these declarations we have not included any property associations, specifically no runtime related properties. We will defer these until we generate the execution (runtime) representation. The interfaces and connections for data and control flow are included, since this information is nominally included in a conceptual (runtime neutral) representation. As before, no data types are defined for these interfaces⁸.

8. Although we do not demonstrate this in the PBA example, the specific data types and data implementations can be added when the runtime categories are defined or even later in the process.

Listing 3-17: *Abstract Subcomponents for the control.speed Implementation*

```

abstract read_data
  features
    sensor_data: in data port;
    proc_data: out data port;
end read_data;

abstract implementation read_data.speed
end read_data.speed;

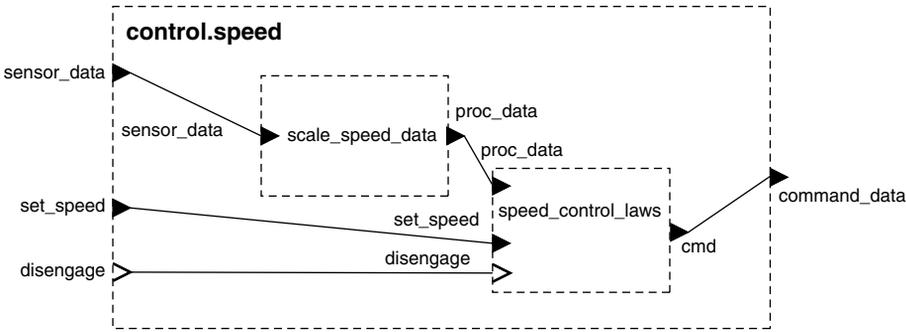
abstract control_laws
  features
    proc_data: in data port;
    cmd: out data port;
    disengage: in event port;
    set_speed: in data port;
end control_laws;

abstract implementation control_laws.speed
end control_laws.speed;

abstract control
  features
    command_data: out data port;
    sensor_data: in data port;
    set_speed: in data port;
    disengage: in event port;
end control;

abstract implementation control.speed
  subcomponents
    scale_speed_data: abstract read_data.speed;
    speed_control_laws: abstract control_laws.speed;
  connections
    DC1: port sensor_data -> scale_speed_data.sensor_data;
    DC2: port scale_speed_data.proc_data ->
        speed_control_laws.proc_data;
    DC3: port speed_control_laws.cmd -> command_data;
    EC1: port disengage -> speed_control_laws.disengage;
    DC4: port set_speed -> speed_control_laws.set_speed;
end control.speed;

```



3.5.3 Transforming into a Runtime Representation

We transform abstract representations into runtime representations by extending implementations. In so doing, we change the category of an implementation and its corresponding type and transform the categories of the subcomponents that reference those classifiers. We start at the lowest level of the component hierarchy, extending implementations that have subcomponents. We progress upward until we reach the complete system. For this example, we use the same runtime categories as those in the previous section (i.e., those shown in Table 3-1).

First, we extend the implementation `control.speed`, since it is the lowest level abstract implementation with subcomponents.⁹ This is shown in Listing 3-18, where a process type `control_rt` and a process implementation `control_rt.speed` are declared. The declaration of the type `control_rt` simply extends the type `control`, changing the category from abstract to process. There are no other refinements to the type. For the implementation `control_rt.speed`, the declaration changes the category of the implementation to process and refines (**refined to**) the category of both subcomponent to threads. Note that all of the characteristics (e.g., features, properties) of their ancestors are inherited by the components defined in an extension declaration (**extends**). Therefore, only modified elements of an implementation are included in an extension

9. We could have developed an abstract implementation for the pilot interface component `interface_pilot` that included subcomponents and a complex internal structure. In that case, we would refine it to a system or other runtime category.

declaration of that implementation. It is not necessary to extend the type or implementation declarations for the abstract implementations *read_data.speed* and *control_laws.speed*, since there are no subcomponents in either of these implementations.¹⁰

In changing a category, it is important that the features, subcomponents, modes, properties, etc. declared for an abstract component are consistent with the semantics of the new category. For example, an abstract component with a processor subcomponent cannot be extended into a thread component.

Next, we extend the enclosing system type *Complete* to create *Complete_rt* and its implementation *Complete.PBA_speed_control_ab* to create *Complete_rt.PBA_speed_control_ab*, as shown in Listing 3-18. In the declaration of *Complete_rt.PBA_speed_control_ab*, we also refine the subcomponents *speed_control* and *interface_unit*, changing their category from abstract to a runtime specific category. These choices parallel the categories of the simplified model developed earlier.

Listing 3-18: *Transforming the Generic PBA System into a Runtime Representation*

```

process control_rt extends control
end control_rt;

process implementation control_rt.speed extends control.speed
  subcomponents
    scale_speed_data: refined to thread read_data.speed;
    speed_control_laws: refined to thread control_laws.speed;
end control_rt.speed;

device interface_rt extends interface
end interface_rt;

device implementation interface_rt.pilot extends interface.pilot
end interface_rt.pilot;

system Complete_rt extends Complete
end Complete_rt;

```

10. In some cases, you may use abstract components that you decide should become processes, leaving them incomplete and detailing their implementation later (e.g., by adding subcomponents).

```

system implementation Complete_rt.PBA_speed_control_ab extends
Complete.PBA_speed_control_ab
  subcomponents
    speed_control: refined to process control_rt.speed;
    interface_unit: refined to device interface_rt.pilot;
end Complete_rt.PBA_speed_control_ab;

```

3.5.4 Adding Runtime Properties

At this point, we have refined the PBA model to include runtime components and subcomponents. However, we have not included runtime properties. For example, values for the timing properties required for a scheduling analysis are not assigned (e.g., the execution time for threads). We can do this in a number of ways. We could add local property associations to the individual abstract declarations, as shown in Listing 3-19 for the abstract types that are refined into threads. For properties that are declared as inheritable, we could modify components that are higher in the component hierarchy, relying on the inheritance of values to subcomponents (e.g., putting the values in the declarations for the abstract component type *control*).

Listing 3-19: *Modifying Declarations with Local Property Associations*

```

abstract read_data
  features
    sensor_data: in data port;
    proc_data: out data port;
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 1 ms .. 2 ms;
    Period => 50 ms;
end read_data;

abstract control_laws
  features
    proc_data: in data port;
    cmd: out data port;
    disengage: in event port;
    set_speed: in data port;
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 3 ms .. 5 ms;
    Period => 50 ms;
end Control_laws;

```

We could adopt a policy where we assign relevant properties by extending an abstract component (**extends**) and adding the property associations into the extension. This allows us to create multiple variants of the component parameterized with different property values. We can also parameterize individual subcomponents with different property values as part of a subcomponent refinement (**refined to**). An example of adorning the subcomponent refinements is shown in Listing 3-20.

Listing 3-20: *Property Associations Adorning Subcomponent Refinements*

```

process implementation control_rt.speed extends control.speed
  subcomponents
    scale_speed_data: refined to thread read_data.speed
      {Dispatch_Protocol => Periodic;
       Compute_Execution_Time => 1 ms .. 2 ms;
       Period => 50ms;};
    speed_control_laws: refined to thread control_laws.speed
      {Dispatch_Protocol => Periodic;
       Compute_Execution_Time => 3 ms .. 5 ms;
       Period => 50ms;};
end control_rt.speed;

```

Another approach to centralizing property associations is to include all property declarations for the extended system in the highest system implementation declaration or for a very large system in a limited number of system implementations. To do this we use contained property associations. This is useful when different instances of the same component need to have different property values. We effectively configure an instance of the model through properties and place this configuration information (property associations) in one place instead of modifying different parts of the model. An example is shown in Listing 3-21. We assign values to the *Period* and *Compute_Execution_Time* properties for the thread subcomponents using individual property associations. We use a single property association to apply the value *Periodic* to the *Dispatch_Protocol* property for both threads.

Listing 3-21: *Contained Property Associations within a System Implementation*

```

system implementation Complete_rt.PBA_speed_control_ab extends
  Complete.PBA_speed_control_ab
  subcomponents
    speed_control: refined to process control_rt.speed;
    interface_unit: refined to device interface_rt.pilot;

```

```

properties
  Period => 50ms applies to speed_control.scale_speed_data;
  Compute_Execution_Time => 1 ms .. 2 ms
    applies to speed_control.scale_speed_data;

  Period => 50ms applies to speed_control.speed_control_laws;
  Compute_Execution_Time => 3 ms .. 5 ms
    applies to speed_control.speed_control_laws;

  Dispatch_Protocol => Periodic
    applies to speed_control.scale_speed_data,
              speed_control.speed_control_laws;
end Complete_rt.PBA_speed_control_ab;

```

3.5.5 Completing the Specification

In order to complete the specification for the PBA system to the level of the model we developed in the previous section, we need to include hardware components, their relevant interfaces, and their interconnections. For this purpose, we simply use the updated hardware component declarations as shown in Listing 3-4.

In addition, we need to add a bus access feature to the abstract component *interface.pilot*. A completed PBA speed control system implementation is shown in Listing 3-22. In the table, we have highlighted the portions of *Complete.PBA_speed_control_ab* that were modified in the extension to *Complete_rt.PBA_speed_control_ab*.

By adding the hardware subcomponents into the system implementation *Complete.PBA_speed_control_ab*, we have a system implementation *Complete_rt.PBA_speed_control_ab* that is comparable to the one we generated in the previous section. That is, with this representation, we can add binding properties and conduct a scheduling analysis as we did in Section 0.

Listing 3-22: A Complete PBA System Implementation

```

abstract interface
  features
    set_speed: out data port;
    disengage: out event port;
    BA1: requires bus access Marine.Standard;
end interface;

device sensor
  features
    sensor_data: out data port;

```

continues

```

    BA1: requires bus access Marine.Standard;
end sensor;

device actuator
  features
    cmd: in data port;
    BA1: requires bus access Marine.Standard;
end actuator;

system implementation Complete.PBA_speed_control_ab
  subcomponents
    speed_sensor: device sensor.speed;
    throttle: device actuator.speed;
    speed_control: abstract control.speed;
    interface_unit: abstract interface.pilot;
    RT_1GHz: processor Real_Time.one_GHz;
    Standard_Marine_Bus: bus Marine.Standard;
    Stand_Memory: memory RAM.Standard;
  connections
    DC1: port speed_sensor.sensor_data ->
      speed_control.sensor_data;
    DC2: port speed_control.command_data -> throttle.cmd;
    DC3: port interface_unit.set_speed ->
      speed_control.set_speed;
    EC4: port interface_unit.disengage ->
      speed_control.disengage;
    BAC1: bus access Standard_Marine_Bus <-> speed_sensor.BA1;
    BAC2: bus access Standard_Marine_Bus <-> RT_1GHz.BA1;
    BAC3: bus access Standard_Marine_Bus <-> throttle.BA1;
    BAC4: bus access Standard_Marine_Bus <-> interface_unit.BA1;
    BAC5: bus access Standard_Marine_Bus <-> Stand_Memory.BA1;
end Complete.PBA_speed_control_ab;

system implementation Complete_rt.PBA_speed_control_ab extends
Complete.PBA_speed_control_ab
  subcomponents
    speed_control: refined to process control_rt.speed;
    interface_unit: refined to device interface_rt.pilot;
  properties
    Period => 50ms applies to speed_control.scale_speed_data;
    Compute_Execution_Time => 1 ms .. 2 ms
      applies to speed_control.scale_speed_data;

    Period => 50ms applies to speed_control.speed_control_laws;
    Compute_Execution_Time => 3 ms .. 5 ms
      applies to speed_control.speed_control_laws;

    Dispatch_Protocol => Periodic
      applies to speed_control.scale_speed_data,
        speed_control.speed_control_laws;
end Complete_rt.PBA_speed_control_ab;

```

3.6 Working with Component Patterns

As you use the AADL for multiple projects, you will find it convenient to reuse such things as data sensors, processors, buses, control software, and layered control architecture that have been successfully used in other projects. This is especially true if you are working in a product-line development environment.

In previous examples, we have seen how AADL can be used to define component templates (i.e., component descriptions that are completed and refined later through extension). In some cases, it is desirable to explicitly specify the placeholders (i.e., parameters) and what must be provided within a template. For example, we may have a template that is an abstract component defining a dual redundancy pattern. In that case, a user is expected to supply a single classifier that is used for both redundant instances in the pattern.

In this section, we discuss the use of parameterized component templates patterns. In so doing, we declare incomplete component types and implementations; explicitly specify what is needed to complete a pattern by declaring a prototype as a pattern parameter; and illustrate how such parameterized templates are used.

3.6.1 Component Libraries and Reference Architectures

With the AADL, it is possible to archive components and proven system solutions and reuse them through extension declarations. For this purpose, we suggest partitioning archival elements into two sets: a component library and reference architecture archive. The partitioning separates concerns such that individual, relatively simple elements are archived separately from elements involving a complex component hierarchy.

A component library is a collection of component types and component implementations with limited subcomponents that represent individual elements of a system architecture. These may be generic or runtime specific. For example, in a component library you may have a processor type *marine_certified* and a collection of implementations that have different processor speeds, manufacturers, and internal memory sizes. Similarly, you may have an abstract type *PID_controller* and its implementations that represent proportional-integral-derivative control with varying capabilities. The abstract components can be extended into runtime specific components such as a process or thread. For

software components, this is the most flexible category for archiving in a library.

In your work, you may have identified a number of proven architecture solutions that have been useful. You can compile these solutions (reference architectures) into an archive that can be used in other projects. These reference architectures define common building blocks and reflect a common topology and are common throughout embedded systems development. Examples include layered control and triple modular redundant reference architectures that can be used for high dependability control avionics as well as space systems. Reference architectures can be defined at different levels of abstraction. Reference architectures can be defined using runtime-specific categories or abstract components and prototypes.

As a third approach to modeling the PBA speed control system, we use a generic component library, a reference architecture archive, prototypes, extensions, refinements, and multiple packages as demonstrations of reusing generic patterns for components and system architectures. We refine the generic components into runtime specific components in developing the PBA-specific architecture.

A library and archive can be developed without using prototypes (i.e., using only extensions and refinements). However, using prototypes makes explicit the elements (e.g., port and subcomponent classifiers) that are being refined.

3.6.2 Establishing a Component Library

Listing 3-23 shows an example generic component library that consists of two packages: *interfaces_library* and *controller_library*. In these packages, we define generic application components as **abstract** components. The packages are partitioned based upon a separation of concerns (e.g., the *interfaces_library* package has generic representations for sensors, actuators, and user interfaces). Another generic package could include only execution hardware with standard processors, memory, and bus components.

For this example, only the *generic_control* type has an implementation with subcomponents. In this implementation, prototypes are used in defining the subcomponents. Note that the property *Prototype_Substitution_Rule* is assigned the value *Type_Extension*. This allows within refinements, the substitution of classifiers for prototypes that are of the same type or are an extension of the original type used for the prototype. Although most of the components in this library are abstract,

runtime-specific categories can be used. For example, in the abstract type declaration for *generic_interface*, we define a data prototype *out_data* that is used to define the data type in the declaration of the out data port *output*.

Listing 3-23: *Generic Component Library*

```

--- generic component library ---

package interfaces_library
public

abstract generic_sensor
  features
    output: out data port;
end generic_sensor;

abstract generic_interface
  prototypes
    out_data: data;
  features
    output: out data port out_data;
    disengage: out event port;
end generic_interface;

abstract generic_actuator
  features
    input: in data port;
end generic_actuator;

end interfaces_library;

package controller_library
public

abstract generic_control
  features
    input: in data port;
    output: out data port;
    set_value: in data port;
    disengage: in event port;
end generic_control;

abstract implementation generic_control.partitioned
  prototypes
    rd: abstract generic_read_data;
    cl: abstract generic_control_laws;
  subcomponents
    r_data: abstract rd;
    c_laws: abstract cl;

```

continues

```

connections
  DC1: port input -> r_data.input;
  DC2: port r_data.output -> c_laws.input;
  DC3: port c_laws.output-> output;
  EC1: port disengage -> c_laws.disengage;
  DC4: port set_value -> c_laws.set_value;
properties
  Prototype_Substitution_Rule => Type_Extension;
end generic_control.partitioned;

abstract generic_read_data
features
input: in data port;
output: out data port;
end generic_read_data;

abstract implementation generic_read_data.impl
end generic_read_data.impl;

abstract generic_control_laws
features
  input: in data port;
  set_value: in data port;
  disengage: in event port;
  output: out data port;
end generic_control_laws;

abstract implementation generic_control_laws.impl
end generic_control_laws.impl;

end controller_library;

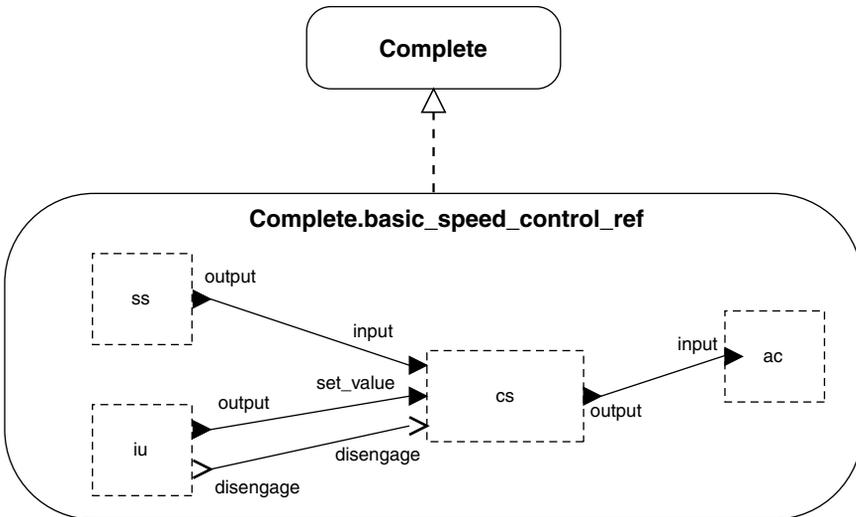
```

3.6.3 Defining a Reference Architecture

A sample reference architecture archive is shown in Listing 3-24, in which we have defined a generic speed control implementation *Complete.basic_speed_control_ref*. This implementation uses prototypes. The prototypes used here are abstract. However, prototypes can be runtime specific. In this reference architecture, we use the prototypes as classifier placeholders for the subcomponent classifiers of the implementation. For example, the prototype *ssg* represents the *generic_sensor* type that is defined in the package *interfaces_library*. This prototype is used in the declaration of the subcomponent *ss*. In using the reference architecture for the PBA, we refine the prototype into a specific runtime implementation. In this case, since we have assigned the value *Type_Extension* to the *Prototype_Substitution_Rule* property, we can substitute implementations of extensions of the component type declared in the prototype bindings.

Listing 3-24: Reference Architectures

```
--- reference architecture archive ---  
package reference_arch  
public  
with interfaces_library, controller_library;  
system Complete  
end Complete;  
  
system implementation Complete.basic_speed_control_ref  
  prototypes  
    ssg: abstract interfaces_library::generic_sensor;  
    csg: abstract controller_library::generic_control;  
    iug: abstract interfaces_library::generic_interface;  
    acg: abstract interfaces_library::generic_actuator;  
  subcomponents  
    ss: abstract ssg;  
    ac: abstract acg;  
    cs: abstract csg;  
    iu: abstract iug;  
  connections  
    DC1: port ss.output -> cs.input;  
    DC2: port cs.output -> ac.input;  
    DC3: port iu.output -> cs.set_value;  
    EC4: port iu.disengage-> cs.disengage;  
  properties  
    Prototype_Substitution_Rule => Type_Extension;  
end Complete.basic_speed_control_ref;  
  
end reference_arch;
```



3.6.4 Utilizing a Reference Architecture

We use the reference architecture described in the previous section to define a PBA architecture. This is shown in Listing 3-25, where the first declaration extends the abstract type *Complete* found in the package *reference_arch*. In this extension, the system category is substituted for abstract. Similarly, the abstract implementation *Complete.basic_speed_control_ref* is extended creating the system *Complete.PBA_speed_control*. In this extension, the prototypes for the subcomponents are bound to an actual classifier using a prototype binding (e.g., *acg => device actuator.speed*). In our example, we have fixed the classifier to be a device called *actuator.speed*, which will be used in the subcomponent declaration that refers to the prototype.

In the second part of Listing 3-25, each of the type and some implementation classifiers used in the prototype refinements are extended from the component library. In these extensions, PBA specific refinements can be made. For example, the data classifier *speed_data* is added to the out port of the sensor in the *sensor* type and to the input of the process type *control*. In addition, property associations are added in the *control.speed* implementation. One is the period for the threads in the process *control.speed* and the other is a contained property association, assigning a compute execution time to the control laws thread *cl* within the process *control.speed*.

Listing 3-25: *Using a Reference Architecture*

```

package mysystem
public
with reference_arch, interfaces_library, controller_library;
system Complete extends reference_arch::Complete
end Complete;
system implementation Complete.PBA_speed_control
  extends reference_arch::Complete.basic_speed_control_ref
    (acg => device actuator.speed,
     ssg => device sensor.speed,
     csg => process control.speed,
     iug => device interface.pilot )
end Complete.PBA_speed_control;

-- defining subcomponent substitutions ---

device sensor extends interfaces_library::generic_sensor
features
output: refined to out data port speed_data;
end sensor;
data speed_data
end speed_data;
```

```

device implementation sensor.speed
end sensor.speed;

device actuator extends interfaces_library::generic_actuator
features
input: refined to in data port cmd_data;
end actuator;

data cmd_data
end cmd_data;

device implementation actuator.speed
end actuator.speed;

device interface extends interfaces_library::generic_interface
end interface;

device implementation interface.pilot
end interface.pilot;

process control extends controller_library::generic_control
features
input: refined to in data port speed_data;
end control;

process implementation control.speed extends controller_
library::generic_control.partitioned
(
  rd => thread read_speed_data.impl,
  cl => thread speed_control_laws.impl )
properties
Period => 20 ms;
Compute_Execution_Time => 2ms..5ms applies to cl;
end control.speed;

thread read_speed_data
extends controller_library::generic_read_data
end read_speed_data;

thread implementation read_speed_data.impl extends controller_
library::generic_read_data.impl
end read_speed_data.impl;

thread speed_control_laws
  extends controller_library::generic_control_laws
end speed_control_laws;

thread implementation speed_control_laws.impl
  extends controller_library::generic_control_laws.impl
end speed_control_laws.impl;

end mysystem;

```

This page intentionally left blank

Index

Symbols

- . (period)
 - use in component names, 114, 264
 - use in feature identifiers, 265
- : (colon), use in declarations, 170, 174, 186, 309
- :: (double colon)
 - in package names, 268
 - in property set names, 266
- _ (underscore), in identifier syntax, 263
- (double hyphen), in comment syntax, 289
- > (hyphen and angle bracket),
 - directional connection, 189
- <-> (hyphen between angle brackets),
 - bidirectional connections, 189
- => (equal sign and angle bracket),
 - property association, 292
- +> (plus sign, equal sign, and angle bracket), property additive association, 293

A

- AADL (Architecture Analysis & Design Language)
 - benefits of, 3–4
 - binding software to hardware. *see* Software deployment
 - components. *see* Components
 - data modeling. *see* data modeling
 - declarative model, 26
 - design organization. *see* Design organization
 - elements of, 110–111

- as foundation for model-based engineering, 1–2
- MBE (model-based engineering) and, 10–12
- models. *see* Models
- modes. *see* Modes
- other modeling languages and, 14–15
- powerboat autopilot system example.
 - see* PBA (powerboat autopilot) system
- properties. *see* Properties
- reserved words, 348
- runtime services. *see* Runtime services
- SAE AADL. *see* SAE AADL
- syntax of, 20–21, 327–342
- SysML used with, 15–16
- system composition. *see* System composition
- system flows. *see* Flows
- AADL extensions. *see also* Annex sublanguages
 - declaring property constants, 311–312
 - declaring property sets, 304–305
 - declaring property types, 305–309
 - defining properties, 309–311
 - overview of, 303
 - property sets, 303–304
- AADL Inspector, 320
- AADL Web, 435
- aadlboolean** property type, 308–310
- aadlinteger** property type, 308–309
- AADLSimulink. *see* Simulink
- Abort_Process*, runtime executive service, 424

- Abort_Processor*, runtime executive service, 425
- Abort_System*, runtime executive service, 425
- Abort_Virtual_Processor*, runtime executive service, 425
- Abstract components
 - constraints on, 168
 - declaring, 58–59
 - employing in PBA system model, 58–61, 428
 - overview of, 166–167
 - properties of, 167–168
 - representation of, 167
- Abstract features
 - for component interactions, 185
 - connections and, 225–226
 - declaring, 226
 - refining, 226–227
- Abstract flows
 - analyzing, 57–58
 - overview of, 54
 - specifying, 55–57
- Abstract implementation, in PBA system model, 61–63
- Abstraction
 - conceptual modeling. *see* Conceptual modeling
 - principles for managing complexity in software, 6
 - transforming abstract representation into runtime representation, 63–65
- Acceptable_Array_Size*, modeling property, 409
- Access
 - component feature category, 118–121
 - data, 210–213
 - subprograms, 231–233
- Access connections
 - bus access, 156
 - data access, 210–212
 - declaring remote calls as, 236–237
 - overview of, 210–213
- Access_Right*, memory-related property, 391–392
- Access_Time*, memory-related property, 392
- Actions, thread, 129
- Activate_Deadline*, timing property, 372
- Activate_Entrypoint*, programming property, 398
- Activate_Entrypoint_Call_Sequence*, programming property, 398–399
- Activate_Entrypoint_Source_Text*, programming property, 399
- Activate_Execution_Time*, timing property, 372
- Active_Thread_Handling_Protocol*, thread-related property, 369
- Active_Thread_Queue_Handling_Protocol*, thread-related property, 370
- Actual_Connection_Binding*, connection binding property, 260
- Actual_Latency*, communication property, 390
- Actual_Memory_Binding*, deployment property, 259, 352
- Actual_Processor_Binding*, deployment property, 257–259, 353
- Actual_Subprogram_Call*, deployment property, 355
- Actual_Subprogram_Call_Binding*, deployment property, 261, 356
- Ada language, 152
- Aggregate data communication, 207–209
- Aliases, package, 271–273
- Allowed_Connection_Binding*, deployment property, 351
- Allowed_Connection_Binding_Class*, deployment property, 351
- Allowed_Connection_Type*, deployment property, 359–360
- Allowed_Dispatch_Protocol*, deployment property, 360
- Allowed_Memory_Binding*, deployment property, 352
- Allowed_Memory_Binding_Class*, deployment property, 353
- Allowed_Message_Size*, memory-related property, 393

- Allowed_Period* property, deployment property, 360–361
- Allowed_Physical_Access*, deployment property, 361
- Allowed_Physical_Access_Class*, deployment property, 361
- Allowed_Processor_Binding*, deployment property, 258, 354
- Allowed_Processor_Binding_Class*, deployment property, 354–355
- Allowed_Subprogram_Call*, deployment property, 355
- Allowed_Subprogram_Call_Binding*, deployment property, 261, 356
- Analysis
 - in AADL model, 30
 - benefits of model-based approach to, 1
- Analyzable architectural models, benefits of MBE, 8–10
- Annex libraries
 - declaring annex concepts in, 313–314
 - declaring classifiers, 312
- annex** reserved word, in component implementation declaration, 122
- Annex subclauses
 - in AADL component declaration, 23–24
 - referencing annex classifiers, 312
 - using annex concepts in, 314–315
- Annex sublanguages
 - declaring annex concepts in libraries, 313–314
 - overview of, 312–313
 - using annex concepts in subclauses, 314–315
- Annotation, of models
 - AADL semantics and, 321
 - assigning property values, 292–294
 - comments and description properties, 289–290
 - contained property associations, 299–300
 - creating models and, 318
 - determining property values, 297–299, 300–302
 - empty component sections and, 290
 - overview of, 289
 - properties in, 291–292
 - property types and values, 294–297
- Aperiodic threads, 129
- Application runtime architecture, in embedded software systems, 7
- Application runtime services
 - Get_Count*, 420
 - Get_Value*, 420
 - Next_Value*, 420
 - Put_Value*, 419
 - Receive_Input*, 419
 - Send_Output*, 418
 - Updated*, 421
- Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs (ASIIST), 323, 436
- Applications. *see also* Software components
 - application software component category, 19
 - case studies in modeling application systems, 430–433
 - control applications, 31–32
- applies to** statement, for assigning property values, 265
- Architecture
 - abstract component and, 166
 - analyzable architectural models as benefit of MBE, 8–10
 - application runtime architecture, 7
 - component use in system architecture, 6
 - modeling system architectures, 429–430
 - reference architectures. *see* Reference architectures
 - static vs. dynamic, 169
 - validating system architecture, 321–322
 - working with architectural descriptions in AADL, 17–18
- Architecture-centric approach, to model-based engineering, 9–10

Ariane 5 Flight 501, 435

ARINC653, 136, 151

Arrays

connection patterns, 229–230

connection properties, 230–231

connections and, 227–228

declaring subcomponents as, 172–173

explicitly specified connections, 228

AS5506, 436

ASIIST (Application Specific I/O

Integration Support Tool for

Real-Time Bus Architecture

Designs), 323, 436

Assign_Time, memory-related property,
393

Assignment operator, assigning prop-
erty values, 292–293

Await_Dispatch, runtime executive
service, 422–423

Await_Result, runtime executive service,
424

B

Background threads, 129

BAnnex (Behavior Annex)

describing thread behavior, 232

extensions to AADL, 11

modeling subprogram calls, 88

reference to, 436

for representing functional behavior

of application components, 174

as sublanguage, 313

system validation and generation
tools, 323

Base_Address, memory-related property,
394

Base_Types package, DAnnex (Data
Modeling Annex), 100

Behavioral Language for Embedded
Systems with Software (BLESS),
323, 436

Bell LaPadula, 324

Bidirectional access connections, 210

Binary file, documenting, 48–49

Bindings

for component interactions, 185–186

connection bindings, 260

mapping software to execution
platform, 81

memory bindings, 259–260

processor bindings, 259

properties in declaration of, 257–259

remote calls declared as, 234–236

remote subprogram call bindings,
260–262

software to hardware, 256–257

Binpacker tool, 322

BLESS (Behavioral Language for
Embedded Systems with Soft-
ware), 323, 436

Buses

access features, 156

adding hardware components for
PBA system, 40–42

communication support in execution
platform, 80–81

connections and, 213–217

constraints on, 157–158

description of, 147

modeling execution platform
resources, 78–80

overview of, 156

properties of, 157

representation of, 156–157

working with hardware in AADL, 18

Byte_Count, memory-related property,
396

C

C language, 48–49

Call sequences

declaring for subprogram, 233–234

modes for specifying, 174, 182–183

symbol representing, 242

Calls

- for component interactions, 185–186
- declaring remote calls as access connections, 236–237
- declaring remote calls as bindings, 234–236
- declaring subprogram calls, 233–234
- subprogram, 231–233
- symbol representing, 242

calls reserved word, in component implementation declaration, 122

CAN bus, 199

Case studies, in modeling application systems, 430–433

Casteres 09, 437

CAT (Consumption Analysis Toolbox), 436

Categories

- component categories, 19–20, 114–115
- component feature categories, 118–121
- refining, 280–281

Cheddar, 437

Chinese Wall, 324

Classifier_Matching_Rule, modeling property, 224–225, 409

Classifier_Substitution_Rule, modeling property, 277, 409–410

Classifiers

- aliases, 272
- constraints on data classifier declarations, 140–141
- constraints on process classifier declarations, 138
- constraints on subprogram classifier declarations, 144
- constraints on subprogram group classifier declarations, 146
- constraints on thread classifier declarations, 132
- constraints on thread group classifier declarations, 135
- declaring classifier extensions, 274
- declaring subcomponents by omitting, 171–172

- defining components and, 113
- defining data types and, 97–98
- in expanded PBA model, 51–52
- naming and referencing, 264
- organizing component classifiers into packages, 114

overview of, 21–22

prototypes as classifier parameters, 281

providing prototype actuals, 284–287

referencing package elements, 269–270

for refining models, 273–274

substitution rules, 277–279

using prototypes, 283–284

Client_Subprogram_Execution_Time, timing property, 374

Clock_Jitter, timing property, 380

Clock_Period, timing property, 380–381

Clock_Period_Range, timing property, 381

Code artifacts

documenting source code and binary files, 48–49

documenting variable names, 49–50

modeling source code structure, 50–51

representing, 47–48

Collocated, deployment property, 357

Comments, in annotation of models, 289–290

Communication properties

Actual_Latency, 390

Connection_Pattern, 384

Connection_Set, 385

Fan_Out_Policy, 384

Input_Rate, 387–388

Input_Time, 388

Latency, 391

Output_Rate, 388–389

Output_Time, 389

Overflow_Handling_Protocol, 385

Queue_Processing_Protocol, 385–386

Queue_Size, 386

Required_Connection, 386

- Communication properties (*continued*)
 - Subprogram_Call_Rate*, 389
 - Timing*, 386–387
 - Transmission_Time*, 390
 - Transmission_Type*, 387
- Communication protocols
 - bus support for, 156
 - processors supporting, 149
 - virtual bus representing, 158
- COMPASS (Correctness, Modeling and Performance of Aerospace Systems), 324, 437
- Component libraries
 - establishing, 70–72
 - organizing classifiers into packages, 114
 - overview of, 69–70
- Components
 - categories, 19–20, 114–115
 - composite, 20, 163
 - constraints on application software components, 342–347
 - declaring, 23–24
 - declaring abstract components, 58–59
 - declaring abstract components in PBA system model, 58–61
 - declaring component types, 114–118
 - declaring external interfaces, 118–121
 - declaring implementations, 121–125
 - defining for PBA system, 32–35
 - detailing abstract implementations for PBA system, 61–63
 - dynamic reconfiguration of, 51–54
 - empty component sections, 290
 - execution platform. *see* Execution platform components
 - flow specification for, 245
 - generic, 20, 163
 - hierarchy of, 77–78
 - mapping to source code, 48–49
 - modeling, 13–14
 - modes for applying alternative component configurations, 177–179
 - names, 113–114, 264
 - overview of, 113
 - port communication timing and, 196
 - properties differing from mode to mode, 173
 - software. *see* Software components
 - subcomponents. *see* Subcomponents
 - summary, 125–126
 - transforming abstract representation into runtime representation, 63–65
- Components, interactions
 - abstract features and connections, 225–226
 - aggregate data communication, 207–209
 - array connection patterns, 229–230
 - array connection properties, 230–231
 - arrays and connections, 227–228
 - bus access and connections, 213–217
 - combining port-based communication with shared data communication, 203–206
 - constraints on port to port connections, 193–196
 - data access and connections, 210–213
 - declaring abstract features, 226
 - declaring calls and call sequences, 233–234
 - declaring feature group connections, 221–225
 - declaring feature group types, 218–220
 - declaring feature groups, 220–221
 - declaring parameter connections, 241–243
 - declaring parameters, 240–241
 - declaring port to port connections, 189
 - declaring ports, 186–187
 - declaring remote subprogram calls as access connections, 236–237
 - declaring remote subprogram calls as bindings, 234–236
 - deterministic sampling of data streams between ports, 199–203

- explicitly specified array connections, 228
- feature groups and connections, 217–218
- interfacing to external world, 97
- modeling directional exchange of data and control, 85–86
- modeling local service requests or function invocation, 87–89
- modeling object-oriented method calls, 92–95
- modeling remote service requests or function invocation, 90–92
- modeling shared data exchange, 86–87
- modeling subprogram instances, 237–240
- modeling subprogram parameters, 95–97
- overview of, 84–85, 185–186
- parameter connections, 240
- ports as interfaces, 188
- properties of port to port connections, 207
- refining abstract features, 226–227
- sampled processing of data streams between ports, 198–199
- subprogram calls, access, and instances, 231–233
- timing port to port connections, 196–198
- using port to port connections, 189–193
- Components, pattern reuse
 - component libraries, 69–72
 - overview of, 69
 - reference architectures, 72–75
- Components, type declaration
 - aliases in, 271–273
 - flow declarations, 246, 248
 - interface declarations, 118–121
 - modal types and, 175
 - overview of, 114, 116–118
- Composite components
 - component categories in AADL, 20
 - overview of, 163
 - system. *see* System component
- Compute_Deadline*, timing property, 372–373
- Compute_Entrypoint*, programming property, 399–400
- Compute_Entrypoint_Call_Sequence*, programming property, 400
- Compute_Entrypoint_Source_Text*, programming property, 400–401
- Compute_Execution_Time*, timing property, 373
- Computer Architectures: Readings and Examples* (Bell and Newell), 12
- Conceptual modeling
 - adding hardware components to specification, 67–68
 - adding runtime properties, 65–67
 - detailing abstract implementations, 61–63
 - employing abstract components, 58–61
 - overview of, 58
 - transforming abstract representation into runtime representation, 63–65
- Concurrency control, for shared data access, 210
- Concurrency_Control_Protocol* property, thread-related property, 366
- Configuration, modes for alternative component configuration, 173, 177–179
- Connection_Pattern*, communication property, 229–230, 384
- Connection_Set*, communication property, 385
- Connections
 - array, 227–231
 - bindings, 260
 - bus access and, 213–217
 - component interactions and, 185

- Connections (*continued*)
 - data access and, 210–213
 - feature group, 221–225
 - parameters, 240–243
 - working with connection instances in
 - system instance models, 82–83
 - Connections, port to port
 - constraints on, 193–196
 - declaring, 189
 - deterministic sampling of data
 - streams, 199–203
 - properties of, 207
 - sampled processing of data streams,
 - 198–199
 - timing, 196–198
 - using, 189–193
 - connections** reserved word, in component implementation declaration, 122
 - Consistency checking, AADL standard and, 321
 - constant* keyword, assigning property values with, 292
 - Constants
 - assigning property values, 292
 - declaring property constants, 311–312
 - project-specific, 410
 - Constructors, property type, 307–308
 - Consumption Analysis Toolbox (CAT), 436
 - Context-free syntax, in AADL language, 20–21
 - Control data, persistent data store for, 133
 - Controls
 - detailing control software for PBA System, 38–40
 - directional control system, 171
 - exchange of. *see* Exchange of control and data
 - modeling directional exchange of, 85–86
 - using AADL for, 31–32
 - Correctness, Modeling and Performance of Aerospace Systems (COM-PASS), 324, 437
 - Criticality* property, thread-related property, 366
 - Current_System_Mode*, runtime executive service, 424
- ## D
- DAnnex (Data Modeling Annex)
 - defining data component properties, 291–292
 - reference to, 437
 - standard, 195
 - Data
 - access, 210–213
 - classifiers, 97–98
 - component types, 99
 - constraints on, 140–141
 - description of, 127
 - exchange. *see* Exchange of control and data
 - overview of, 138
 - properties of, 140
 - representation of, 138–140
 - sampled processing of data streams
 - between ports, 198–199
 - working with runtime software
 - abstractions in AADL, 18
 - Data Distribution Service (DDS), 208, 437
 - Data modeling
 - defining simple data types, 98–99
 - detailing data types, 100–101
 - overview of, 97–98
 - representing variants on data types, 99
 - Data Modeling Annex. *see* DAnnex (Data Modeling Annex)
 - data port** reserved word, 186
 - Data ports
 - aggregate data communication and, 208–209

- combining port-based communication with shared data communication, 203–206
 - description of, 188
 - port communication timing and, 197–198
 - port to port connections and, 194
 - properties of port to port connections, 207
- Data types
 - defining simple, 97–99
 - detailing, 100–101
 - mapping to source code, 48–49
 - modeling source code structure, 50–51
 - representing variants on, 99
- Data_Representation*, DAnnex property, 100
- Data_Volume*, project property, 415
- DDS (Data Distribution Service), 208, 437
- Deactivate_Deadline*, timing property, 374
- Deactivate_Entrypoint*, programming property, 401
- Deactivate_Entrypoint_Call_Sequence*, programming property, 401
- Deactivate_Entrypoint_Source_Text*, programming property, 402
- Deactivate_Execution_Time*, timing property, 374–375
- Deactivation_Policy* property, thread-related property, 370
- Deadline*, timing property, 375
- Declarations, summary of AADL declarations, 22–24
- Declarative models
 - for component hierarchy, 77–78
 - for system composition, 77
- Delayed connections, deterministic sampling and, 199–201
- DeNiz, 437–438
- Deploying software on hardware. *see* Software deployment
- Deployment properties
 - Actual_Memory_Binding* property, 352
 - Actual_Processor_Binding* property, 353
 - Actual_Subprogram_Call* property, 355
 - Actual_Subprogram_Call_Binding* property, 356
 - Allowed_Connection_Binding* property, 351
 - Allowed_Connection_Binding_Class* property, 351
 - Allowed_Connection_Type* property, 359–360
 - Allowed_Dispatch_Protocol* property, 360
 - Allowed_Memory_Binding* property, 352
 - Allowed_Memory_Binding_Class* property, 353
 - Allowed_Period* property, 360–361
 - Allowed_Physical_Access* property, 361
 - Allowed_Physical_Access_Class* property, 361
 - Allowed_Processor_Binding* property, 354
 - Allowed_Processor_Binding_Class* property, 354–355
 - Allowed_Subprogram_Call* property, 355
 - Allowed_Subprogram_Call_Binding* property, 356
 - Collocated* property, 357
 - Memory_Protocol* property, 362
 - Not_Collocated* property, 356–357
 - Preemptive_Scheduler* property, 362
 - Priority_Map* property, 363
 - Priority_Range* property, 363
 - Provided_Connection_Quality_Of_Service* property, 358–359
 - Provided_Virtual_Bus_Class* property, 358
 - Required_Connection_Quality_Of_Service* property, 359
 - Required_Virtual_Bus_Class* property, 358
 - Scheduling_Protocol* property, 362
 - Thread_Limit* property, 363
 - Dequeue_Protocol*, thread-related property, 367–368

- Dequeued_Items*, thread-related property, 368
 - Description properties, 290
 - Design organization
 - defining multiple extensions, 105–107
 - developing alternative implementations, 104–105
 - overview of, 101
 - packages in, 102–104
 - Deterministic sampling, of data streams
 - between ports, 199–203
 - Device_Register_Address*, memory-related property, 394
 - Devices
 - bus access and, 213
 - communication support in execution platform, 80–81
 - constraints on, 161–162
 - defining components for PBA system, 33
 - defining connections for PBA system, 41–42
 - defining subcomponents for PBA system, 36
 - description of, 147
 - deterministic sampling of data streams between ports, 199–203
 - as interface to external world, 97
 - modeling execution platform resources, 78–80
 - overview of, 160
 - port communication timing and, 196
 - properties of, 161
 - representation of, 160–161
 - sampled processing of data streams between ports, 198–199
 - working with hardware in AADL, 18
 - Dimensions, array specification and, 172
 - Directional access connections, 210
 - Directional control system, 171
 - Directional transfer, ports for, 186
 - Dispatch protocol
 - port communication timing and, 196–197
 - for threads, 129
 - Dispatch_Able*, thread-related property, 367
 - Dispatch_Jitter*, timing property, 375–376
 - Dispatch_Offset*, timing property, 376
 - Dispatch_Protocol*, thread-related property, 364
 - Dispatch_Trigger*, thread-related property, 364
 - Dynamic architecture, 169. *see also* Modes
 - Dynamic reconfiguration, of systems or components, 51–54
- ## E
- EAnnex (Error Model Annex)
 - as example of sublanguage, 312
 - extensions to AADL, 11
 - reference to, 438
 - as standard for sublanguages, 303
 - system validation and generation tools, 323
 - Eclipse, 315, 317, 319, 438
 - EDF (earliest deadline first), scheduling policy, 44
 - Embedded systems
 - applications of, 6
 - co-engineering with system engineering, 13
 - as engineering challenge, 1
 - key elements of, 7–8
 - Encapsulation, managing complexity in software, 6
 - end** reserved word, in component implementation declaration, 122
 - End-to-end flows, 253–255
 - Equivalence/complement rule, using with feature groups, 224
 - Error Model Annex. *see* EAnnex (Error Model Annex)
 - ErrorData* subcomponent, in modeling object-oriented method calls, 93
 - ESA (European Space Agency), 320, 324
 - event data port** reserved word, 186

- Event data ports
 - combining port-based communication with shared data communication, 203
 - description of, 188
 - port communication timing and, 197
 - port to port connections and, 194
 - properties of port to port connections, 207
 - event port** reserved word, 186
 - Event ports
 - description of, 188
 - port communication timing and, 197
 - port to port connections and, 193
 - properties of port to port connections, 207
 - Exchange of control and data
 - consistency checks on port connections, 195
 - logical interactions between application components, 84
 - modeling directional exchange, 85–86
 - modeling shared data exchange, 86–87
 - Execution platform. *see also* Hardware communication support in, 80–81
 - component categories in AADL, 20
 - mapping software to, 81
 - modifying resources, 78–80
 - system abstraction for composite that includes, 163
 - Execution platform components. *see also* Hardware components
 - bus, 156–158
 - device, 160–162
 - memory, 153–156
 - overview of, 147–148
 - processors, 148–150
 - virtual bus, 158–159
 - virtual processors, 150–153
 - Execution_Time*, timing property, 376
 - extends** clause
 - declaring classifiers, 274
 - declaring components, 22–24
 - defining extension declaration, 63–64
 - prototype use as classifier parameters, 281
 - refining component categories, 280
 - extends** reserved word
 - adding runtime properties, 66
 - in component implementation declaration, 122
 - Extensions. *see also* AADL extensions
 - declaring classifier extensions, 274
 - declaring model refinements, 275–277
 - defining multiple in design organization, 105–107
- ## F
- Failure Mode and Effect Analysis (FMEA), 323
 - Fan_Out_Policy*, communication property, 384
 - Feature groups
 - aggregate data communication and, 208–209
 - in component declaration, 23–24
 - for component interactions, 185
 - declaring component features, 220–221
 - declaring connections, 221–225
 - declaring model refinements, 275
 - declaring types of, 218–220
 - empty component sections and, 290
 - names, 264
 - overview of, 217–218
 - prototypes for, 283–284
 - Features section
 - in AADL component declaration, 22–24
 - declaring component interfaces, 118–121
 - Feiler 07 and 07A, 438
 - FHA (Functional Hazard Assessment), 323
 - FIACRE, 323, 438
 - Finalize_Deadline*, timing property, 376
 - Finalize_Entrypoint*, programming property, 402

Finalize_Entrypoint_Call_Sequence, programming property, 402–403
Finalize_Entrypoint_Source_Text, programming property, 403
Finalize_Execution_Time:Time, timing property, 377
First_Dispatch_Time, timing property, 375
 Flow latency analysis tool, OSATE, 57
 Flow path, 247
 Flow sink, 247
 Flow source, 247
 Flows

- analyzing, 57–58
- declaring end-to-end, 253–255
- declaring implementations, 249–253
- declaring specifications, 246–248
- overview of, 54, 245–246
- specifying end-to-end, 57
- specifying flow model, 55–56
- working with end-to-end, 255

flows reserved word, in component implementation declaration, 122
flows section

- in AADL component declaration, 22–24
- component type declaration, 246
- declaring end-to-end flows, 254–255
- declaring flow implementations, 249–253

 FMEA (Failure Mode and Effect Analysis), 323
Frame_Period, timing property, 383
 Function invocation

- modeling local, 87–89
- modeling remote, 90–92

 Functional Hazard Assessment (FHA), 323
 Functions, modeling source code structure and, 50–51

G

Generic components

- abstract. *see* Abstract component
- component categories in AADL, 20
- overview of, 163

Generic features. *see* Abstract features
Get_Count, application runtime service, 420
Get_Error_Code, runtime executive service, 423
Get_Resource, runtime executive service, 421
Get_Value, application runtime service, 420
 Global variables, 96–97
 Graphical languages

- AADL as, 11
- overview of, 17
- SysML as, 15
- UML as, 14–15

 Graphical representation

- in AADL, 27–29
- of abstract components, 167
- of bus, 156–157
- comments included in, 289
- of data, 138–140
- of devices, 160–161
- of feature groups, 220
- of memory, 153–154
- of models, 317
- of modes, 176–177
- of parameters, 243
- of processes, 136–137
- of processors, 148–149
- of subprogram calls, 233–234
- of subprogram groups, 145–146
- of subprograms, 143
- of system components, 164
- of threads, 130
- of virtual bus, 158–159
- of virtual processor, 151–152

H

Hansson 08, 438
 Hardware, deploying software on hardware. *see* Software deployment
 Hardware components. *see also* Execution platform components
 adding to PBA system model, 40–41

- binding to software in PBA system
 - model, 43–45
 - bus access and, 213
 - bus and, 156
 - modeling execution platform
 - resources, 78–80
 - working with in AADL, 17–18
 - Hardware_Description_Source_Text*,
 - programming property, 407–408
 - Hardware_Source_Language*, programming property, 408
 - Hofmeister 00, 439
 - Hybrid threads, 129
- I**
- Identifiers. *see also* Names
 - classifier, 264
 - component, 113–114
 - property set, 304
 - subcomponent, 170
 - syntax of, 263
 - IEEE (Institute of Electrical and Electronics Engineers)
 - Systems and Software Engineering-Architecture Descriptions* (42010), 5, 439
 - VHDL (VHSIC Hardware Description Language), 12
 - IMA (integrated modular avionics), 10
 - Immediate connections, deterministic sampling and, 202–203
 - Implementations
 - declaring component implementations, 121–125
 - declaring flow implementations, 249–253
 - declaring modal implementations, 175–177
 - detailing abstract implementations, 61–63
 - developing alternative implementations in design organization, 104–105
 - top-level model for PBA system, 36–37
 - Implemented_As*, modeling property, 410
 - in binding** statement, 293
 - in modes** statement
 - assigning property values, 293
 - declaring subcomponents, 170
 - In ports
 - description of, 188
 - port to port connections and, 195
 - inherit** reserved word, 310
 - Inheritance
 - modes and, 180–181
 - property inheritance, 310
 - Initialize_Deadline*, timing property, 377
 - Initialize_Entrypoint*, programming property, 403
 - Initialize_Entrypoint_Call_Sequence*, programming property, 403–404
 - Initialize_Entrypoint_Source_Text*, programming property, 404
 - Initialize_Execution_Time*, timing property, 377
 - Input_Rate*, communication property, 387–388
 - Input_Time*, communication property, 388
 - Instance models
 - modeling subprogram instances, 237–240
 - modeling system instances. *see* System instance models
 - Instances, subprogram, 231–233
 - Institute of Electrical and Electronics Engineers. *see* IEEE (Institute of Electrical and Electronics Engineers)
 - Integers, in property value summary, 296
 - Integrated modular avionics (IMA), 10
 - Interfaces
 - for components of PBA system, 34
 - declaring external component interfaces, 118–121

Interfaces (*continued*)

to outside application system, 97
ports as, 188

Inverse of statement, in declaring
feature group types, 219

J

Java VM (virtual machine), 151

L

Languages

Ada language, 152

annex sublanguages. *see* Annex
sublanguages

C language, 48–49

graphical. *see* Graphical languages

MDA. *see* MDA (model-driven
architecture)

modeling languages. *see* Modeling
languages

SysML. *see* SysML

textual, 11

UML. *see* UML (Unified Modeling
Language)

Latency, communication property, 391

Libraries

annex libraries. *see* Annex libraries

component libraries, 69–72

modeling source code structure

and, 50

Load_Deadline, timing property, 378

Load_Time, timing property, 378

Logical flows. *see* Flows

Logical interface, between embedded
application software and physical
systems, 7–8

M

MARTE (Modeling and Analysis of
Real-time and Embedded Sys-
tems), 12–13, 439

Mathworks Simulink. *see* Simulink

Max_Aadlinteger, project property, 415

Max_Base_Address, project property, 415

Max_Byte_Count, project property, 417

Max_Memory_Size, project property, 416

Max_Queue_Size, project property, 416

Max_Target_Integer, project property, 415

Max_Thread_Limit, project property, 416

Max_Time, project property, 416

Max_Urgency, project property, 416

Max_Word_Space, project property, 417

MBE (model-based engineering)

AADL and, 10–12

AADL used with MDA and UML,
14–15

AADL used with SysML, 15–16

analyzable models and, 8–10

benefits of, 1–2

for embedded real-time systems,
6–8

modeling languages and, 12–14
overview of, 5

MCD (model-centered development), 5

MDA (model-driven architecture)

AADL used with MDA and UML,
14–15

OMG initiatives in model-based
engineering, 12

reference to, 439

software applications of model-based
engineering, 5

MDD (model-driven development), 5

Mean Time To Failure (MTTF), 323

Memory

adding hardware components for
PBA system, 40–42

bindings, 259–260

constraints on, 155

description of, 147

modeling execution platform
resources, 78–80

overview of, 153

processor access to, 149

properties, 154–155, 391–397

- representation of, 153–154
 - working with hardware in AADL, 18
- Memory_Protocol* property, deployment property, 362
- Memory-related properties
 - Access_Right*, 391–392
 - Access_Time*, 392
 - Allowed_Message_Size*, 393
 - Assign_Time*, 393
 - Base_Address*, 394
 - Byte_Count*, 396
 - Device_Register_Address*, 394
 - overview of, 154–155
 - Read_Time*, 394–395
 - Source_Code_Size*, 395
 - Source_Data_Size*, 395
 - Source_Heap_Size*, 395–396
 - Source_Stack_Size*, 396
 - Word_Size*, 396–397
 - Word_Space*, 397
 - Write_Time*, 397
- Meta models
 - annex sublanguages and, 312
 - limiting property ownership via
 - Meta model class, 310
 - standardization of, 317
- META toolset. *see* Rockwell Collins
 - META toolset
- MetaH, AADL patterned after, 10
- Method calls, modeling object-oriented, 92–95
- MIL-STD 1553 bus, 199
- Mission Data System reference architecture, NASA, 319
- Mode_Transition_Response* property,
 - thread-related property, 368–369
- Model-based engineering. *see* MBE (model-based engineering)
- Model-centered development (MCD), 5
- Model-driven development (MDD), 5
- Modelica, component models in, 14
- Modeling and Analysis of Real-time and Embedded Systems (MARTE), 12–13, 439
- Modeling languages
 - AADL used with MDA and UML, 14–15
 - AADL used with SysML, 15–16
 - overview of, 12–14
 - state-based languages for representing functional behavior of application components, 174
- Modeling properties
 - Acceptable_Array_Size*, 409
 - Classifier_Matching_Rule*, 409
 - Classifier_Substitution_Rule*, 409–410
 - Implemented_As*, 410
 - Prototype_Substitution_Rule*, 410
- Modeling system architectures,
 - resources related to, 429–430
- Models
 - aliases for package and type references, 271–273
 - analysis of, 30
 - annotating. *see* Annotation, of models
 - case studies in modeling application systems, 430–433
 - category refinement, 280–281
 - classifier substitution rules for refining, 277–279
 - classifiers for naming and referencing elements in, 264
 - classifiers for refining, 273–274
 - component categories of, 19–20
 - creating, 317–319
 - declaring classifier extensions, 274
 - declaring packages, 267–269
 - declaring prototypes, 281–283
 - declaring refinements, 275–277
 - naming and referencing elements
 - with packages, 263–264
 - naming and referencing elements
 - with property sets, 266
 - overview of, 19, 263
 - packages in, 266–267
 - property substitution and, 287
 - prototypes as classifier parameters, 280–281

Models (*continued*)

- providing prototype actuals, 284–287
- referencing model elements, 265–266
- referencing package elements, 269–271
- structure of, 25–26
- subprogram instances, 237–240
- system instances, 26–27
- system validation and generation tools, 322–324
- tools for creating, 319–320
- using prototypes, 283–284
- validation of system architecture, 321–322

Modes

- for alternative call sequences, 182–183
- for alternative component configurations, 177–179
- declaring component types and implementations, 175–177
- declaring modes and mode transitions, 174–175
- dynamic reconfiguration of PBA system with, 51–53
- inheriting, 180–181
- overview of, 173–174
- properties associated with, 181–182
- specifying, 53–54

modes reserved word, in component implementation declaration, 122

Modes section

- in AADL component declaration, 22–24
- mode transition declaration in, 175

MTTF (Mean Time To Failure), 323

N**Names**

- aliases for package and type references, 271–273
- classifier, 264
- component, 113–114
- model elements, 263–264
- package, 263–264, 268

- property set, 266, 303–304
- subcomponent, 170

NASA, 319, 439

Next_Value, application runtime service, 420

none statement, empty component sections and, 290

Not_Collocated property, deployment property, 356–357

N-Version redundancy, 282

O

Object Constraint Language (OCL), 313

Object-orientation, modeling object-oriented method calls, 92–95

Ocarina, 323–324, 439

OCL (Object Constraint Language), 313

OMG (Object Management Group)
DDS (Data Distribution Service), 208, 437

model-based engineering and, 12–13

One-dimensional arrays, 227–228

Open Source AADL Tool Environment.
see OSATE (Open Source AADL Tool Environment)

Operational states, modes for representing, 173

OSATE (Open Source AADL Tool Environment)

downloading, 28

flow latency analysis tool, 57

generating instance models with, 27

graphical and textual representation with, 29

handling of packages and property sets by, 26

model creation tools, 319–320

reference to, 439

resource allocation and scheduling plug-in, 44, 47

system validation and generation tools, 322

user interface, 317–318

out feature command, 227

Out ports

description of, 188

port to port connections and, 195

Output_Rate, communication property,
388–389

Output_Time, communication property,
389

Overflow_Handling_Protocol, communica-
tion property, 385

P

Packages

aliases, 271–273

in component declaration, 23–24

declaring, 267–269

in design organization, 102–104

empty component sections and, 290

in model organization, 266–267

modeling source code structure and,
50

for naming and referencing model
elements, 263–264

organizing component classifiers
into, 114

referencing elements in, 269–271

structure of AADL models and, 25–26

Parameters

component feature categories,
118–121

for component interactions, 185

connections, 240

declaring, 240–241

declaring connections for, 241–243

modeling subprogram parameters,
95–97

Passing by reference, pseudocode, 95–97

Patterns

abstract component and, 166

array connection patterns, 229–230

component libraries and, 69–72

redundancy patterns, 282

reference architectures, 72–75

PBA (powerboat autopilot) system

AADL components of, 427–428

with abstract components, 58–61, 428

abstract flows and, 54

adding hardware components, 40–41,
67–68

adding runtime properties, 65–67

analyzing flows, 57–58

binding software to hardware, 43–45

component libraries and, 69–72

conceptual modeling, 58

conducting scheduling analyses,
45–47

defining components, 32–35

defining physical connections, 41–43

description of, 425–426

detailing abstract implementations,
61–63

detailing control software, 38–40

developing simple model, 31–32

developing top-level model, 36–37

documenting source code and binary
files, 48–49

documenting variable names, 49–50

dynamic reconfiguration of, 51–54

enhanced versions of, 426–427

modeling source code structure,
50–51

overview of, 425

reference architectures and, 72–75

representing code artifacts, 47–48

specifying flows, 55–57

summary, 47

transforming abstract representation
into runtime representation, 63–65

PCI bus, 213

Period, timing property, 128, 378

Periodic threads, 129

Permanent storage, memory compo-
nents and, 153

Persistent data store, for control data,
133

Physical connections, defining for PBA
system, 41–43

Physical environment, logical interface
with embedded application
software, 7–8

- Physical system components, modeling, 78–80
- PIMs (platform independent models), 14–15
- Platform independent models (PIMs), 14–15
- Platform specific models (PSMs), 14–15
- PMS (Processor Memory Switch), 12
- Ports
 - aggregate data communication, 207–209
 - combining port-based communication with shared data communication, 203–206
 - component feature categories, 118–121
 - component interactions and, 185
 - constraints on port to port connections, 193–196
 - declaring, 186–187
 - declaring port to port connections, 189
 - deterministic sampling of data streams between, 199–203
 - as interface, 188
 - properties of port to port connections, 207
 - sampled processing of data streams between, 198–199
 - timing port to port connections, 196–198
 - using port to port connections, 189–193
- POSIX_Scheduling_Policy* property, thread-related property, 365
- Powerboat autopilot system. *see* PBA (powerboat autopilot) system
- Predeclared properties, 291
- Preemptive_Scheduler*, deployment property, 362
- Priority*, thread-related property, 365
- Priority_Map*, deployment property, 363
- Priority_Range*, deployment property, 363
- Private section, of packages, 266, 268
- Process_Swap_Execution_Time*, timing property, 381
- Processes
 - in AADL, 11
 - constraints on, 137–138
 - defining components for PBA system, 33–34
 - description of, 127
 - detailing control software for PBA system, 39–40
 - overview of, 135–136
 - properties of, 137
 - representation of, 136–137
 - working with runtime software
 - abstractions in AADL, 18
- Processor Memory Switch (PMS), 12
- Processors
 - adding for PBA system, 40–42
 - binding to software, 44, 259
 - communication support in execution platform, 80–81
 - constraints on, 150
 - defining execution characteristics, 45–46
 - description of, 147
 - modeling execution platform
 - resources, 78–80
 - overview of, 148
 - properties of, 150
 - representation of, 148–149
 - threads assigned to, 128
 - working with hardware in AADL, 18
- Programming properties
 - Activate_Entrypoint*, 398
 - Activate_Entrypoint_Call_Sequence*, 398–399
 - Activate_Entrypoint_Source_Text*, 399
 - Compute_Entrypoint*, 399–400
 - Compute_Entrypoint_Call_Sequence*, 400
 - Compute_Entrypoint_Source_Text*, 400–401
 - Deactivate_Entrypoint*, 401

- Deactivate_Entrypoint_Call_Sequence*, 401
- Deactivate_Entrypoint_Source_Text*, 402
- Finalize_Entrypoint*, 402
- Finalize_Entrypoint_Call_Sequence*, 402–403
- Finalize_Entrypoint_Source_Text*, 403
- Hardware_Description_Source_Text*, 407–408
- Hardware_Source_Language*, 408
- Initialize_Entrypoint*, 403
- Initialize_Entrypoint_Call_Sequence*, 403–404
- Initialize_Entrypoint_Source_Text*, 404
- Recover_Entrypoint*, 404
- Recover_Entrypoint_Call_Sequence*, 405
- Recover_Entrypoint_Source_Text*, 405
- Source_Language*, 405–406
- Source_Name*, 406
- Source_Text*, 406–407
- Supported_Source_Language*, 407
- Project properties
 - Data_Volume*, 415
 - Max_Aadlinteger*, 415
 - Max_Base_Address*, 415
 - Max_Byte_Count*, 417
 - Max_Memory_Size*, 416
 - Max_Queue_Size*, 416
 - Max_Target_Integer*, 415
 - Max_Thread_Limit*, 416
 - Max_Time*, 416
 - Max_Urgency*, 416
 - Max_Word_Space*, 417
 - Size_Units*, 417
 - Supported_Active_Thread_Handling_Protocols*, 411
 - Supported_Classifier_Substitutions*, 414
 - Supported_Concurrency_Control_Protocols*, 412
 - Supported_Connection_Patterns*, 411
 - Supported_Connection_QoS*, 413
 - Supported_Dispatch_Protocols*, 412
 - Supported_Distributions*, 414
 - Supported_Hardware_Source_Languages*, 413
 - Supported_Queue_Processing_Protocols*, 412–413
 - Supported_Scheduling_Protocols*, 413–414
 - Supported_Source_Languages*, 414
 - Time_Units*, 417
- Properties
 - abstract component, 167–168
 - adding runtime properties, 65–67
 - in annotation of models, 289–292
 - array connection, 230–231
 - assigning property values, 292–294
 - built-in property types, 306
 - bus properties, 157
 - communication properties. *see* communication properties
 - contained property associations, 299–300
 - of data components, 140
 - declaring bindings with, 257–259
 - declaring property constants, 311–312
 - declaring property sets, 304–305
 - declaring property types, 305–309
 - defining, 309–311
 - deployment properties. *see* deployment properties
 - description properties, 290
 - determining property values, 297–299
 - device, 161
 - example of determining property value, 300–302
 - list of AADL property types, 347–348
 - memory-related. *see* Memory-related properties
 - modeling properties. *see* Modeling properties
 - mode-specific, 181–182
 - predeclared, 291
 - of processes, 137
 - of processors, 150
 - programming properties. *see* Programming properties

- Properties (*continued*)
- project-specific constants and
 - property types. *see* Project properties
 - for source code documentation, 48
 - of subprogram groups, 146
 - of subprograms, 143
 - substitution in model organization, 287
 - system component, 165
 - of thread groups, 134
 - thread-related. *see* Thread-related properties
 - timing properties. *see* Timing properties
 - types and values, 294–297
 - of virtual bus, 159
 - of virtual processor, 152
- properties** reserved word, in component implementation declaration, 122
- Properties sections, in AADL component declaration, 22–24
- Property associations
- assigning property values, 292–293
 - contained, 299–300
- Property sets
- in AADL component declaration, 23–24
 - declaring, 304–305
 - for naming and referencing elements, 266
 - overview of, 303–304
 - structure of AADL models and, 25–26
- prototype** reserved word, in component implementation declaration, 122
- Prototype_Substitution_Rule*, modeling property, 287, 410
- Prototypes
- as classifier parameters, 280–281
 - declaring, 281–283
 - libraries and archives and, 70
 - providing prototype actuals, 284–287
 - substitution rules, 287
 - using, 283–284
- Prototypes section, in AADL component declaration, 22–24
- Provided_Connection_Quality_Of_Service*, deployment property, 358–359
- Provided_Virtual_Bus_Class*, deployment property, 358
- provides bus access** feature, 81
- provides data access** feature, 137
- provides subprogram access** feature, 131, 232
- Pseudocode, passing by reference, 95–97
- PSMs (platform specific models), 14–15
- Public section, of packages, 266, 268
- Put_Value*, application runtime service, 419
- Q**
- Quantitative analysis, 8–10
- Queue_Processing_Protocol*, communication property, 385–386
- Queue_Size*, communication property, 386
- R**
- Raise_Error*, runtime executive service, 423
- RAM (random access memory), 153. *see also* Memory
- Rate monotonic (RM), types of scheduling policies, 44
- RC META. *see* Rockwell Collins META toolset
- Read_Time*, memory-related property, 394–395
- Read-only memory (ROM), 153. *see also* Memory
- Real numeric values, property values, 296
- Receive_Input*, application runtime service, 419
- Recover_Deadline*, timing property, 379
- Recover_Entrypoint*, programming property, 404

- Recover_Entrypoint_Call_Sequence*, programming property, 405
- Recover_Entrypoint_Source_Text*, programming property, 405
- Recover_Execution_Time*, timing property, 379
- Redundancy patterns, 282
- Reference architectures
 - abstract component and, 166
 - defining, 72–73
 - utilizing, 74–75
- Reference_Processor*, timing property, 381–382
- References
 - aliases for package and type references, 271–273
 - classifier, 264
 - model element, 263–266
 - package, 263–264, 269–271
 - property set, 266, 304
- refined to
 - adding runtime properties, 66
 - declaring model refinements, 275–277
 - refining abstract features, 227
 - refining component categories, 280
 - refining declarations, 63–64
- Refinements, model
 - abstract feature refinement, 227
 - adding runtime properties, 66
 - category refinement, 280–281
 - classifier substitution rules for, 277–279
 - declaration refinement, 63–64
 - declaring, 275–277
- Reflective memory, 153
- Release_Resource*, runtime executive service, 422
- Remote calls
 - declaring as access connections, 236–237
 - declaring as bindings, 234–236
 - remote subprogram call bindings, 260–262
- renames** statement
 - declaring aliases, 271–273
 - for visibility declarations, 268
- Required_Connection*, communication property, 386
- Required_Connection_Quality_Of_Service*, deployment property, 359
- Required_Virtual_Bus_Class*, deployment property, 358
- requires bus access** feature, 215
- requires data access** (this) option, 93, 131
- Requires modes section, mode inheritance and, 175
- requires modes** statement
 - mode inheritance and, 180
 - using modes for alternative component configurations, 178
- requires subprogram access feature, 233
- Reserved words, list of, 348–349
- Resources, supporting this book
 - case studies, 430–433
 - modeling system architectures, 429–430
- Resumption_Policy* property, thread-related property, 368–369
- RM (rate monotonic), types of scheduling policies, 44
- Rockwell Collins META toolset
 - extending OSATE, 320
 - reference to, 439
 - system validation and generation tools, 324
- ROM (read-only memory), 153. *see also* Memory
- Round-robin (RR), types of scheduling policies, 44
- RR (round-robin), types of scheduling policies, 44
- Runtime components
 - adding runtime properties, 65–67
 - defining components for PBA system, 34
 - transforming abstract representation into runtime representation, 63–65

Runtime executive services

- Abort_Process*, 424
- Abort_Processor*, 425
- Abort_System*, 425
- Abort_Virtual_Processor*, 425
- Await_Dispatch*, 422–423
- Await_Result*, 424
- Current_System_Mode*, 424
- Get_Error_Code*, 423
- Get_Resource*, 421
- Raise_Error*, 423
- Release_Resource*, 422
- Set_System_Mode*, 424
- Stop_Process*, 424
- Stop_Processor*, 425
- Stop_System*, 425
- Stop_Virtual_Processor*, 425

Runtime services

- application runtime services. *see*
Application runtime services
- overview of, 418
- runtime executive services. *see*
Runtime executive services

Runtime states, of threads, 129–130

- Runtime_Protection*, thread-related
property, 370

S

SAE AADL

- classifiers, 21–22
- component categories, 19–20
- language syntax, 20–21
- model analysis, 30
- model structure, 25–26
- models, 19
- software, hardware, and architectural
descriptions and operations, 17–18
- summary of AADL declarations,
22–24
- system instance models, 26–27
- textual and graphical representation,
27–29

Sampling communication

- deterministic sampling of data
streams between ports, 199–203
- processing data streams between
ports, 198–199

SAVI (System Architecture Virtual
Integration), 320–323, 438*Scaling_Factor*, timing property, 382*Scheduler_Quantum*, timing property, 382

Scheduling analyses

- conducting for PBA system, 45–47
- types of scheduling policies, 44

Scheduling threads, 150

Scheduling_Protocol property, deploy-
ment property, 362*Send_Output*, application runtime
service, 418

Service requests

- modeling local, 87–89
- modeling remote, 90–92

Set_System_Mode, runtime executive
service, 424

Shared data exchange

- combining port-based communica-
tion with, 203–206
- modeling, 86–87

Shortest job first (SJF), types of schedul-
ing policies, 44

Simulink

- component models in, 14
- component use in system architec-
ture, 6
- documenting source code and binary
files, 48–49
- extracting AADL models from
Simulink models, 320
- modeling source code structure and,
50
- modeling subprogram calls, 88
- overview of, 12
- reference to, 435
- system validation and generation
tools, 323

- Size_Units*, project property, 417
- SJF (shortest job first), types of scheduling policies, 44
- Slot_Time*, timing property, 383
- Software components
 - constraints on, 342–347
 - constraints on data components, 140–141
 - constraints on processes, 137–138
 - constraints on subprogram groups, 146
 - constraints on subprograms, 144
 - constraints on thread groups, 134–135
 - constraints on threads, 132–133
 - data component instances and, 138
 - mapping to memory, 154
 - overview of, 127–128
 - processes, 135–136
 - properties of data components, 140
 - properties of processes, 137
 - properties of subprogram groups, 146
 - properties of subprograms, 143
 - properties of thread groups, 134
 - properties of threads, 131–132
 - representation of data components, 138–140
 - representation of processes, 136–137
 - representation of subprogram groups, 145–146
 - representation of subprograms, 143
 - representation of thread groups, 133–134
 - representation of threads, 130–131
 - scheduling and executing with virtual processors, 151
 - subprogram groups, 144–145
 - subprograms, 141–142
 - system abstraction for composite that includes, 163
 - thread groups, 133
 - threads, 128–130
 - working with in AADL, 17–18
- Software deployment
 - binding to hardware in PBA system model, 43–45
 - connection bindings, 260
 - declaring bindings with properties, 257–259
 - memory bindings, 259–260
 - overview of, 256–257
 - processor bindings, 259
 - remote subprogram call bindings, 260–262
- Source code, documenting, 48–51
- Source_Code_Size*, memory-related property, 395
- Source_Data_Size*, memory-related property, 395
- Source_Heap_Size*, memory-related property, 395–396
- Source_Language*, programming property, 405–406
- Source_Name*, programming property, 406
- Source_Stack_Size*, memory-related property, 396
- Source_Text*, programming property, 406–407
- Sporadic threads, 129
- Startup_Deadline*, timing property, 379–380
- Startup_Execution_Time*, timing property, 380
- State, modes and, 174
- State-based modeling language, 174
- Static architecture, 169. *see also*
 - Subcomponents
- STOOD, 320, 438
- Stop_Process*, runtime executive service, 424
- Stop_Processor*, runtime executive service, 425
- Stop_System*, runtime executive service, 425
- Stop_Virtual_Processor*, runtime executive service, 425

- Structure, AADL model, 25–26
- Subclauses. *see* Annex subclauses
- Subcomponents
 - in component hierarchy, 77
 - declaring, 170
 - declaring as arrays, 172–173
 - modes for defining alternative configurations of, 177–179
 - overview of, 169
 - port to port connections and, 194
 - using subcomponent declarations, 170–172
- subcomponents** reserved word, in component implementation declaration, 122
- Sublanguages. *see* Annex sublanguages
- Subprogram calls, modeling, 142
- Subprogram groups
 - constraints on, 146
 - description of, 127
 - overview of, 144–145
 - properties of, 146
 - representation of, 145–146
- Subprogram_Call_Rate*, communication property, 389
- Subprogram_Call_Type*, thread-related property, 371
- Subprograms
 - calls, access, and instances, 231–233
 - for component interactions, 186
 - constraints on, 144
 - declaring calls and call sequences, 233–234
 - declaring remote calls as access connections, 236–237
 - declaring remote calls as bindings, 234–236
 - description of, 127
 - modeling local service requests or function invocation, 87–89
 - modeling object-oriented method calls, 92–95
 - modeling remote service requests or function invocation, 90–92
 - modeling subprogram instances, 237–240
 - modeling subprogram parameters, 95–97
 - overview of, 141–142
 - properties of, 143
 - remote subprogram call bindings, 260–262
 - representation of, 143
 - symbol representing subprogram calls, 242
- Subset rule, using with feature groups, 224–225
- Substitution rules
 - classifiers, 277–279
 - prototypes, 287
- Supported_Active_Thread_Handling_Protocols*, project property, 411
- Supported_Classifier_Substitutions*, project property, 414
- Supported_Concurrency_Control_Protocols*, project property, 412
- Supported_Connection_Patterns*, project property, 411
- Supported_Connection_QoS*, project property, 413
- Supported_Dispatch_Protocols*, project property, 412
- Supported_Distributions*, project property, 414
- Supported_Hardware_Source_Languages*, project property, 413
- Supported_Queue_Processing_Protocols*, project property, 412–413
- Supported_Scheduling_Protocols*, project property, 413–414
- Supported_Source_Language*, programming property, 407
- Supported_Source_Languages*, project property, 414
- Synchronized_Component*, thread-related property, 371
- Syntax, AADL language, 20–21, 327–342

- SysML (System Modeling Language)
 - AADL used with, 15–16
 - component use in system architecture, 6
 - model creation tools, 320
 - OMG initiatives in model-based engineering, 12–14
 - reference to, 438
 - System architecture
 - component use in, 6
 - modeling, 429–430
 - validating, 321–322
 - System Architecture Virtual Integration (SAVI), 320, 321–323, 438
 - System components
 - constraints on, 165–166
 - overview of, 163
 - properties of, 165
 - representation of, 164
 - system abstraction for composite that includes, 163
 - System composition
 - communication support in execution platform, 80–81
 - component hierarchy in, 77–78
 - creating system instance model, 81
 - modeling execution platform resources, 78–80
 - overview of, 77
 - system hierarchy in, 81
 - working with connections in system instance model, 82–83
 - working with system instance model, 83–84
 - System engineering, 13
 - System flows. *see* Flows
 - System hierarchy, 81
 - System instance models
 - connections in, 82–83
 - creating, 26–27, 81
 - overview of, 26
 - working with, 83–84
 - System Modeling Language. *see* SysML (System Modeling Language)
 - Systems, dynamic reconfiguration of, 51–54
 - Systems and Software Engineering-Architecture Descriptions* (IEEE 42010), 5, 439
- ## T
- TASTE (The ASSERT Set of Tools for Engineering), 320, 324, 438
 - Templates, abstract component and, 166
 - Textual languages, AADL as, 11
 - Textual representation
 - in AADL, 27–29
 - of abstract components, 167
 - of bus, 156–157
 - comments included in, 289
 - of data, 138–140
 - of devices, 160–161
 - of memory, 153–154
 - of models, 317
 - of modes, 176–177
 - of parameters, 243
 - of processes, 136–137
 - of processors, 148–149
 - of subprogram groups, 145–146
 - of subprograms, 143
 - of system components, 164
 - of threads, 130
 - of virtual bus, 158–159
 - of virtual processors, 151–152
 - The ASSERT Set of Tools for Engineering (TASTE), 320, 324, 438
 - Thread groups
 - constraints on, 134–135
 - description of, 127
 - overview of, 133
 - properties of, 134
 - representation of, 133–134
 - Thread_Limit* property, deployment property, 363
 - Thread_Swap_Execution_Time*, timing property, 382–383

Thread-related properties

- Active_Thread_Handling_Protocol* property, 369
- Active_Thread_Queue_Handling_Protocol* property, 370
- Concurrency_Control_Protocol* property, 366
- Criticality* property, 366
- Deactivation_Policy* property, 370
- Dequeue_Protocol* property, 367–368
- Dequeued_Items* property, 368
- Dispatch_Able* property, 367
- Dispatch_Protocol* property, 364
- Dispatch_Trigger* property, 364
- Mode_Transition_Response* property, 368–369
- overview of, 131–132
- POSIX_Scheduling_Policy* property, 365
- Priority* property, 365
- Resumption_Policy* property, 368–369
- Runtime_Protection* property, 370
- Subprogram_Call_Type* property, 371
- Synchronized_Component* property, 371
- Time_Slot* property, 366
- Urgency* property, 367

Threads

- in AADL, 11
- binding software to hardware in PBA system model, 44
- constraints on, 132–133
- defining execution characteristics, 45–46
- describing behavior using BAnnex, 232
- description of, 127
- detailing control software for PBA system, 38–39
- deterministic sampling of data streams between ports, 199–203
- mapping software to execution platform, 81

modeling directional exchange of

- data and control, 85–86
 - overview of, 128–130
 - port communication timing and, 196–197
 - port to port connections and, 192–193
 - properties of, 131–132, 364–371
 - representation of, 130–131
 - runtime states of, 129–130
 - sampled processing of data streams between ports, 198–199
 - scheduling, 150
 - virtual processors representing, 152
 - working with connections in system instance model, 82–83
 - working with runtime software abstractions in AADL, 18
- Time_Slot* property, thread-related property, 366
- Time_Units*, project property, 417
- Timed threads, 129
- Time-deterministic data exchange, 85
- Timing*, communication property, 386–387

Timing properties

- Activate_Deadline*, 372
- Activate_Execution_Time*, 372
- Client_Subprogram_Execution_Time*, 374
- Clock_Jitter*, 380
- Clock_Period*, 380–381
- Clock_Period_Range*, 381
- Compute_Deadline*, 372–373
- Compute_Execution_Time*, 373
- Deactivate_Deadline*, 374
- Deactivate_Execution_Time*, 374–375
- Deadline*, 375
- Dispatch_Jitter*, 375–376
- Dispatch_Offset*, 376
- Execution_Time*, 376
- Finalize_Deadline*, 376
- Finalize_Execution_Time:Time*, 377
- First_Dispatch_Time*, 375

Frame_Period, 383
Initialize_Deadline, 377
Initialize_Execution_Time, 377
Load_Deadline, 378
Load_Time, 378
Period, 378
Process_Swap_Execution_Time, 381
Recover_Deadline, 379
Recover_Execution_Time, 379
Reference_Processor, 381–382
Scaling_Factor, 382
Scheduler_Quantum, 382
Slot_Time, 383
Startup_Deadline, 379–380
Startup_Execution_Time, 380
Thread_Swap_Execution_Time, 382–383
 TOPCASED, 321, 438
Transmission_Time, communication property, 390
Transmission_Type, communication property, 387

U

UML (Unified Modeling Language)
 AADL used with MDA and UML, 14–15
 component use in system architecture, 6
 features of, 14–15
 MARTE (Modeling and Analysis of Real-time and Embedded Systems), 12
 model creation tools, 320
 OMG initiatives in model-based engineering, 12–13
 reference to, 438
Updated, application runtime service, 421
Urgency, thread-related property, 367
 USB bus, communicating with camera via, 157
 User interface, OSATE, 317–318

V

Validation
 of system architecture, 321–322
 tools for, 322–324
 Values
 assigning property values, 292–294
 example of determining property value, 300–302
 rules for determining property values, 297–299
 summary of property values, 294–296
 Variables
 documenting names of, 49–50
 global, 96–97
 VHDL (VHSIC Hardware Description Language), 12, 14
 VHSICs (Very High Speed Integrated Circuits), 12
 Virtual bus
 constraints on, 159
 description of, 147
 overview of, 158
 properties of, 159
 representation of, 158–159
 Virtual channels, virtual bus representing, 158
 Virtual processors
 constraints on, 152–153
 description of, 147
 overview of, 151
 properties of, 152
 representation of, 151–152
 threads assigned to, 128

W

W3C 04, 438
with statement
 accessing property sets, 304
 for visibility declarations, 268
Word_Size, memory-related property, 396–397

Word_Space, memory-related property,
397
Write_Time, memory-related property,
397

X

XMI (XML interchange)
format specification for AADL, 317
included in AADL standard, 12