

AD-A274 691



2

DTIC
ELECT
JAN 14 1994

CONSORTIUM REQUIREMENTS ENGINEERING GUIDEBOOK

SPC-92060-CMC

VERSION 01.00.09

DECEMBER 1993

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

94-01392



94 1 11 180

CONSORTIUM REQUIREMENTS ENGINEERING GUIDEBOOK

SPC-92060-CMC

VERSION 01.00.09

DECEMBER 1993

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Produced by the
SOFTWARE PRODUCTIVITY CONSORTIUM SERVICES CORPORATION
under contract to the
VIRGINIA CENTER OF EXCELLENCE
FOR SOFTWARE REUSE AND TECHNOLOGY TRANSFER

SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

Copyright © 1992, 1993, Software Productivity Consortium Services Corporation, Herndon, Virginia. Permission to use, copy, modify, and distribute this material for any purpose and without fee is hereby granted consistent with 48 CFR 227 and 252, and provided that the above copyright notice appears in all copies and that both this copyright notice and this permission notice appear in supporting documentation. This material is based in part upon work sponsored by the Advanced Research Projects Agency under Grant #MDA972-92-J-1018. The content does not necessarily reflect the position or the policy of the U. S. Government, and no official endorsement should be inferred. The name Software Productivity Consortium shall not be used in advertising or publicity pertaining to this material or otherwise without the prior written permission of Software Productivity Consortium, Inc. SOFTWARE PRODUCTIVITY CONSORTIUM, INC. AND SOFTWARE PRODUCTIVITY CONSORTIUM SERVICES CORPORATION MAKE NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS MATERIAL FOR ANY PURPOSE OR ABOUT ANY OTHER MATTER, AND THIS MATERIAL IS PROVIDED WITHOUT EXPRESS OR IMPLIED WARRANTY OF ANY KIND.

CONTENTS

ACKNOWLEDGMENTS	xvii
1. INTRODUCTION	1-1
1.1 Purpose of This Guidebook	1-2
1.2 Intended Audience	1-2
1.3 Scope of the Method and Guidebook	1-3
1.4 Organization of the Guidebook	1-3
1.5 Using This Guidebook	1-4
1.6 Typographic Conventions	1-4
2. THE CoRE MODELS	2-1
2.1 CoRE Process Overview	2-2
2.2 Purpose of the CoRE Models	2-2
2.2.1 Rationale for the CoRE Approach	2-3
2.2.2 Purpose of the Behavioral and Class Models	2-3
2.3 The CoRE Behavioral Model	2-5
2.3.1 Environmental Variables	2-5
2.3.2 The CoRE Relations	2-5
2.3.3 Relations NAT and REQ	2-8
2.3.3.1 Relation NAT	2-8
2.3.3.2 Relation REQ	2-9
2.3.4 Relations IN and OUT	2-9
2.4 The CoRE Class Model	2-11
2.4.1 Objects and Classes	2-11

2.4.2 Packaging Relationships Among Classes	2-12
2.4.2.1 Encapsulates	2-12
2.4.2.2 Depends-on	2-13
2.4.2.3 Generalization/Specialization	2-14
2.4.3 Allocating the Behavioral Model to Classes	2-15
3. AN EXAMPLE: THE FUEL LEVEL MONITORING SYSTEM	3-1
4. REPRESENTING THE CoRE BEHAVIORAL MODEL	4-1
4.1 Representing the Functional View	4-3
4.1.1 Monitored and Controlled Variables	4-3
4.1.2 Conditions	4-3
4.1.3 Events and Event Expressions	4-4
4.1.3.1 Definitions	4-4
4.1.3.2 Implementation Considerations	4-5
4.1.4 Terms	4-5
4.1.5 Capturing State History	4-6
4.1.5.1 Modes and Mode Machines	4-6
4.1.5.2 Mode Transition Diagram	4-7
4.1.5.3 Mode Transition Tables	4-7
4.1.5.4 Properties of Mode Machines	4-9
4.1.6 Tabular Representation of Functions	4-9
4.1.6.1 Condition Table	4-10
4.1.6.2 Event Table	4-10
4.1.6.3 Selector Table	4-12
4.2 Representing the Dynamic View	4-12
4.2.1 Periodic Scheduling	4-13
4.2.2 Demand Scheduling	4-14
4.3 Specifying REQ, NAT, and Undesired Events	4-15

4.3.1 Specifying Controlled Variable Behavior	4-15
4.3.2 Specifying NAT Relations	4-16
4.3.3 Specifying Required Responses to Undesired Events	4-17
5. REPRESENTING THE CoRE CLASS MODEL	5-1
5.1 Information Model	5-1
5.1.1 Generalization/Specialization Relationship	5-3
5.1.2 Aggregation	5-3
5.1.3 Application-Specific Relationship	5-4
5.2 Class Definitions	5-4
5.3 Diagramming Conventions	5-6
5.3.1 Context Diagram	5-6
5.3.2 Dependency Graph	5-7
5.3.3 Leveling	5-8
5.4 Class Specification	5-9
5.4.1 Class Interface	5-9
5.4.2 Class Encapsulated Information	5-10
5.4.3 Objects	5-10
5.4.4 Inheritance	5-12
5.5 Class Model Notation Summary	5-14
6. CoRE PROCESS OVERVIEW	6-1
6.1 The Idealized CoRE Process	6-1
6.1.1 Identify Environmental Variables	6-3
6.1.2 Preliminary Behavior Specification	6-4
6.1.3 Class Structuring	6-4
6.1.4 Detailed Behavior Specification	6-5
6.1.5 Define Hardware Interface	6-6
6.2 CoRE in Practice	6-7

6.2.1 Specifying Required Behavior	6-7
6.2.2 Iteration Among CoRE Activities	6-11
6.2.3 Managing Requirements Development	6-12
7. IDENTIFY ENVIRONMENTAL VARIABLES	7-1
7.1 Goals	7-1
7.2 Entrance Criteria	7-2
7.3 Activities	7-2
7.3.1 Identify and Define Attributes	7-3
7.3.2 Identify Entities	7-5
7.3.3 Identify Generalization/Specialization Relation	7-5
7.3.4 Identify Aggregation Relation	7-6
7.3.5 Identify Application-Specific Relation	7-6
7.3.6 Identify Likely Requirements Changes and Associated Variables	7-9
7.4 Evaluation Criteria	7-10
7.5 Exit Criteria	7-10
8. PRELIMINARY BEHAVIOR SPECIFICATION	8-1
8.1 Goals	8-1
8.2 Entrance Criteria	8-2
8.3 Activities	8-2
8.3.1 Identify and Define Monitored and Controlled Variables	8-2
8.3.1.1 Identify Controlled Variables	8-2
8.3.1.2 Identify Monitored Variables	8-3
8.3.1.3 Define Monitored and Controlled Variables	8-5
8.3.1.4 Create the System Context Diagram	8-7
8.3.2 Establish Controlled Variable Function Domains	8-7
8.3.2.1 Identify Monitored Variables	8-8
8.3.2.2 Identify Modes	8-9

8.3.2.3 Identify Scheduling Requirements	8-9
8.3.3 Define Mode Machines	8-10
8.3.3.1 Identify Mode Machines	8-10
8.3.3.2 Identify Modes and Transitions	8-11
8.4 Evaluation Criteria	8-12
8.5 Exit Criteria	8-13
9. CLASS STRUCTURING	9-1
9.1 Goals	9-1
9.2 Entrance Criteria	9-2
9.3 Class Structuring Activities	9-2
9.3.1 Create Boundary Classes	9-3
9.3.1.1 Allocate Monitored and Controlled Variables	9-3
9.3.1.2 Define the Boundary Class Interface	9-4
9.3.2 Create Mode Classes	9-7
9.3.3 Create Term Classes	9-7
9.3.4 Define the Encapsulation Structure	9-8
9.3.5 Define the Generalization/Specialization Structure	9-9
9.3.6 Establish Dependencies	9-10
9.4 Evaluation Criteria	9-12
9.4.1 Evaluating Classes	9-12
9.4.2 Evaluating Class Dependencies	9-14
9.5 Exit Criteria	9-14
10. DETAILED BEHAVIOR SPECIFICATION	10-1
10.1 Goals	10-1
10.2 Entrance Criteria	10-1
10.3 Activities	10-2
10.4 Define Controlled Variable Behavior	10-2

10.4.1 Specify Initial Value	10-3
10.4.2 Define Sustaining Conditions	10-3
10.4.3 Specify Demand Behavior	10-4
10.4.3.1 Specify Demand Functions	10-4
10.4.3.2 Demand Scheduling and Timing Constraints	10-5
10.4.4 Specifying Periodic Behavior	10-6
10.4.4.1 Specify Periodic Functions	10-6
10.4.4.2 Specify Periodic Scheduling and Timing	10-8
10.4.5 Specify Tolerance Constraints	10-9
10.5 Refine Mode Classes	10-9
10.6 Refine Remaining Classes	10-10
10.7 Revisit Class Structuring	10-10
10.8 Evaluation Criteria	10-11
10.8.1 Completeness	10-11
10.8.2 Consistency	10-12
10.9 Exit Criteria	10-12
11. DEFINE HARDWARE INTERFACE	11-1
11.1 Goals	11-1
11.2 Entrance Criteria	11-2
11.3 Activities	11-2
11.3.1 Assign Input and Output Variables to Boundary Classes	11-2
11.3.2 Define Input and Output Variables	11-2
11.3.3 Define IN and OUT Relations	11-3
11.3.3.1 Define IN for a Monitored Variable	11-4
11.3.3.2 Define OUT for a Controlled Variable	11-5
11.4 Evaluation Criteria	11-6
11.5 Exit Criteria	11-7

12. ANALYZING A CoRE SPECIFICATION	12-1
12.1 Monitored and Controlled Variables	12-1
12.2 Controlled Variable Functions	12-1
12.3 Terms and Modes	12-2
12.4 IN and OUT Relations	12-3
12.5 Global Checks	12-3
APPENDIX A. SOFTWARE REQUIREMENTS FOR THE FUEL LEVEL MONITORING SYSTEM	A-1
A.1 Introduction	A-1
A.2 Requirements for the Fuel Level Monitoring System	A-1
APPENDIX B. CoRE SPECIFICATION OF THE SOFTWARE REQUIREMENTS FOR THE FUEL LEVEL MONITORING SYSTEM	B-1
B.1 System Context	B-5
B.2 Fuel Level Monitoring System Dependency Graph	B-6
B.3 mode_Class_In_Operation	B-7
B.3.1 Class Interface	B-7
B.3.2 Encapsulated Information	B-7
B.3.3 Traceability	B-8
B.4 class_Fuel_Tank	B-9
B.4.1 Class Interface	B-9
B.4.1.1 NAT Relation	B-10
B.4.2 Encapsulated Information	B-10
B.4.2.1 Input Variables	B-10
B.4.2.2 IN Relation	B-11
B.4.3 Traceability	B-12
B.5 class_Pump	B-13
B.5.1 Class Interface	B-13

B.5.2 Encapsulated Information	B-13
B.5.2.1 REQ Relation	B-13
B.5.2.2 Output Variables	B-14
B.5.2.3 OUT Relation	B-14
B.5.3 Traceability	B-14
B.6 class_Time	B-15
B.6.1 Class Interface	B-15
B.6.2 Encapsulated Information	B-15
B.6.2.1 Input Variables	B-15
B.6.2.2 IN Relation	B-15
B.6.3 Traceability	B-15
B.7 class_Operator	B-16
B.7.1 Class Interface	B-16
B.7.2 Encapsulated Information	B-16
B.7.3 Traceability	B-16
B.8 class_Operator_Communication	B-17
B.8.1 Class Interface	B-17
B.8.2 Encapsulated Information	B-17
B.8.2.1 REQ Relation	B-18
B.8.2.2 Output Variables	B-21
B.8.2.3 OUT Relation	B-21
B.8.3 Traceability	B-22
B.9 class_Switch	B-23
B.9.1 Class Interface	B-23
B.9.2 Encapsulated Information	B-23
B.9.2.1 Input Variables	B-23
B.9.2.2 IN Relation	B-24

B.9.3 Traceability	B-24
B.10 Safety Requirements	B-24
B.11 Security Requirements	B-25
B.12 Other Requirements	B-25
APPENDIX B INDEX	B-26
APPENDIX C. CoRE MAPPING TO DOD-STD-2167A	C-1
C.1 Introduction	C-1
C.2 Software Requirements Specification	C-3
C.2.1 SRS Paragraph 3.1: CSCI External Interface Requirements	C-3
C.2.2 SRS Paragraph 3.2: CSCI Capability Requirements	C-3
C.2.3 SRS Paragraph 3.2.x: (Capability Name and Project-Unique Identifier)	C-4
C.2.4 SRS Paragraph 3.3.: CSCI Internal Interfaces	C-4
C.2.5 SRS Paragraph 3.4.: CSCI Data Element Requirements	C-4
C.2.6 SRS Paragraph 3.5.: Adaptation Requirements	C-4
C.2.7 SRS Paragraph 3.5.1.: Installation-Dependent Data	C-4
C.2.8 SRS Paragraph 3.5.2.: Operational Parameters	C-4
LIST OF ABBREVIATIONS AND ACRONYMS	Abb-1
GLOSSARY	Glo-1
REFERENCES	Ref-1
BIBLIOGRAPHY	Bib-1
INDEX	Ind-1

FIGURES

Figure 2-1.	System Viewed as a Black Box	2-6
Figure 2-2.	System Viewed With Input and Output	2-6
Figure 2-3.	The Behavioral Model Relations	2-7
Figure 2-4.	Graphic Depiction of the Depends-on Relation	2-13
Figure 2-5.	Canonical Allocation of Behavioral Model to Classes	2-15
Figure 3-1.	Fuel Level Monitoring System Pump and Tank Configuration (Front View) .	3-2
Figure 4-1.	Representation of CoRE's Functional View	4-2
Figure 4-2.	Example of a Mode Transition Diagram	4-7
Figure 4-3.	The Semantics of <i>INMODE</i> , <i>EXITED</i> , and <i>ENTERED</i>	4-9
Figure 4-4.	Time Line for Periodic Controlled Variable Process	4-14
Figure 4-5.	Time Line for Demand Controlled Variable Process	4-15
Figure 5-1.	Pump Entity and Attributes Example	5-2
Figure 5-2.	Entity-Relationship Diagram Notation	5-3
Figure 5-3.	Generalization/Specialization Entity-Relationship Diagram Notation	5-4
Figure 5-4.	Aggregation Entity-Relationship Diagram Notation	5-4
Figure 5-5.	Application-Specific Relationship Notation	5-5
Figure 5-6.	Class Structuring Notation	5-7
Figure 5-7.	Representation of an Overview of a Specification Using a Context Diagram .	5-7
Figure 5-8.	Dependency Graph Notation	5-8
Figure 5-9.	Class Structuring Leveling Diagrams	5-8
Figure 5-10.	Class Interface Notation	5-10
Figure 5-11.	Encapsulation Structure Notation	5-11

Figure 5-12.	class_Engine Diagram	5-11
Figure 5-13.	class_Engine Diagram With Objects	5-12
Figure 5-14.	Inheritance Notation	5-14
Figure 5-15.	Class Model Notation Summary	5-15
Figure 6-1.	CoRE Activities and Products	6-2
Figure 6-2.	Results of Steps 1 and 2	6-8
Figure 6-3.	Results of Steps 3 and 4	6-9
Figure 6-4.	Result of Steps 5 and 6	6-10
Figure 6-5.	Results of Steps 7 and 8	6-10
Figure 7-1.	Information Model for the Fuel Level Monitoring System	7-7
Figure 7-2.	Weapon and Target Constraint	7-9
Figure 8-1.	Controlled Variable Overview for con_Low_Alarm	8-8
Figure 8-2.	Using a Mode Transition Diagram to Represent mode_class_In_Operation	8-13
Figure 9-1.	Partial Definitions of class_Fuel_Tank	9-5
Figure 9-2.	Mode Class Interface Example	9-7
Figure 9-3.	Dependency Graph for Operator Interface	9-9
Figure 9-4.	Generalization/Specialization Example	9-11
Figure 9-5.	Fuel Level Monitoring System Dependency Graph	9-13
Figure 11-1.	IN Relation for in_Diff_Press	11-4
Figure 11-2.	Accuracy Specification for in_Diff_Press	11-5
Figure B-1.	Fuel Level Monitoring System: Context Diagram	B-5
Figure B-2.	Fuel Level Monitoring System Dependency Graph	B-6
Figure B-3.	Fuel Level Monitoring System Pump and Tank Configuration (Front View)	B-9

TABLES

Table 2-1.	Specification Properties Versus CoRE Mechanism	2-4
Table 4-1.	Template for Monitored and Controlled Variable Definitions	4-3
Table 4-2.	Using a Table to Represent the Mode Machine In_Operation	4-8
Table 4-3.	Format of a Condition Table	4-10
Table 4-4.	Example of a Condition Table	4-11
Table 4-5.	Format of an Event Table	4-11
Table 4-6.	Example of an Event Table	4-12
Table 4-7.	Format of a Selector Table	4-12
Table 4-8.	Example of a Selector Table	4-12
Table 4-9.	Template for a Controlled Variable with Periodic Scheduling Constraints ...	4-15
Table 4-10.	Template for a Controlled Variable With Demand Scheduling Constraints ..	4-16
Table 5-1.	Entity Template	5-2
Table 5-2.	Partial Attribute Matrix for Fuel Level Monitoring System	5-3
Table 5-3.	Class Template	5-5
Table 5-4.	superclass_A Template	5-12
Table 5-5.	class_B Template	5-13
Table 5-6.	class_C Template	5-13
Table 5-7.	Class Template Summary	5-14
Table 7-1.	Sample Fuel Monitoring System Attributes	7-4
Table 7-2.	Sample Definition of an Enumerated Attribute	7-4
Table 7-3.	Sample Definition of a Numeric Attribute	7-5
Table 7-4.	Sample Fuel Level Monitoring System Entities and Attributes	7-6

Table 7-5.	Fuel Level Monitoring System Attribute Matrix	7-8
Table 8-1.	Definition of Enumerated Environmental Variable	8-6
Table 8-2.	Definition of Numeric Environmental Variable	8-6
Table 10-1.	Event Table Example (Incomplete)	10-5
Table 10-2.	Event Table Example (Completed)	10-5
Table 10-3.	Initial Condition Table Example (Incomplete)	10-7
Table 10-4.	Condition Table Example (Completed)	10-7
Table 10-5.	Initiation and Termination Events for con_Status	10-8
Table 11-1.	Input and Output Variable Template	11-3
Table 11-2.	Sample Definition of Input Variable Diff_Press	11-3
Table 11-3.	Sample Definition of Output Variable Shutdown Signal	11-3
Table 11-4.	Sample OUT Relation for Controlled Variable con_Shutdown_Relay	11-6
Table 11-5.	Sample Tolerance and Delay for Controlled Variable con_Shutdown_Relay .	11-6
Table C-1.	Relationship of CoRE Specification Elements to the Software Requirements Specification	C-1
Table C-2.	Example of CSCI System States Mapping to Capabilities	C-3

This page intentionally left blank.

ACKNOWLEDGMENTS

The following contributors have been instrumental in developing the CoRE method and this guidebook:

- **Stuart R. Faulk, James Kirby, Jr., Lisa Finneran, and Assad Moini** authored this version of the guidebook.
- **Paul Ward** has worked as both consultant and reviewer on the CoRE project. He has been instrumental in helping develop CoRE's class model.
- **John Brackett** assisted in identifying the problems CoRE addresses and has provided excellent reviews of this guidebook.
- **Guy Cox, Doug Smith, and Steve Wartik** contributed to the development of CoRE.
- **Much of what is good in this guidebook is due to excellent reviews provided by John Brackett, Paul Clements, Ron Damer, Howard Lykins, Vance Mall, David Parnas, and Paul Ward.**
- **The production quality is due to the technical editing of Mary Mallonee, word processing by Debbie Morgan and Deborah Tipeni, and clean proofing by Betty Leach and Tina Medina.**

This page intentionally left blank.

1. INTRODUCTION

The Consortium Requirements Engineering (CoRE) method is a method for capturing, specifying, and analyzing software requirements. The Consortium has worked with industrial developers of real-time and embedded systems to identify their key problems and to provide a method that addresses their needs. CoRE supports the development of precise, testable specifications that are demonstrably complete and consistent. CoRE also supports key process issues, such as managing changing requirements and reuse. CoRE is a single coherent requirements method that:

- ***Integrates Object-Oriented and Formal Models.*** Behavioral¹ requirements in CoRE are written in terms of two underlying models: the behavioral model and the class mode. The behavioral model provides a standard structure for analyzing and capturing behavioral requirements (i.e., what the software must do) in a form that is precise, analyzable, and testable. The class model provides facilities for organizing a CoRE specification into parts; it provides facilities supporting change management, reuse, and concurrent development. These models are integrated in a single CoRE specification.
- ***Integrates Graphical and Rigorous Specifications.*** A key goal of the method is to improve communication among the parties involved in requirements engineering. CoRE provides a graphic representation that helps all parties, customers, engineers, designers, and programmers grasp essential relationships among system components. CoRE also provides a rigorous underlying model for capturing detailed behavioral, timing, and accuracy constraints. This rigorous model allows you to develop requirements that are precise, unambiguous, testable, and demonstrably complete and consistent. CoRE provides a consistent interpretation of both graphical and rigorous notations so they combine smoothly in a single specification.
- ***Uses Existing Skills and Notations.*** The language used to specify requirements in CoRE is based on familiar concepts and existing notations. You can apply CoRE using basic concepts familiar to programmers and others writing requirements, e.g., events, Boolean expressions, and state machines. Although CoRE is based on an underlying mathematical mode, just as programming languages are based on formal models, CoRE can be applied without a detailed understanding of formalisms.
- ***Avoids Premature Design Decisions.*** CoRE allows you to specify requirements without prematurely specifying design or implementation details. CoRE describes required behavior in terms of relations that the software must maintain between quantities that the software monitors and those it controls. This allows you to specify what the software must do without having to provide an algorithm or detailed design.

1. Sometimes called functional requirements. This guidebook reserves the use of the term function to refer to mathematical functions and uses behavior to describe the software's visible effects.

- ***Provides Guidance.*** The CoRE behavioral model provides practical guidance in eliciting software requirements, developing a requirements specification, and analyzing the specification for completeness and consistency. The behavioral model defines exactly what requirements information must be captured, and in what form, to develop a complete and consistent specification.

1.1 PURPOSE OF THIS GUIDEBOOK

This guidebook provides a detailed guide to the practice of CoRE. It is intended as an engineering handbook that systems and software engineers can use as a reference when applying the CoRE method. It describes the following:

- The goals and benefits of the CoRE method
- The underlying concepts and principles used to develop rigorous requirements specifications in CoRE
- The set of notations and specification techniques needed to write a CoRE specification
- A complete requirements development process starting with system-level requirements as input and ending with a software requirements specification as output
- Heuristics for applying specific techniques to accomplish your specification goals, such as reuse or change management
- Criteria and a process for checking a CoRE specification for completeness and consistency
- Illustration of the techniques and heuristics through a common example—the Fuel Level Monitoring System (FLMS).

This guidebook is arranged for self-study. Read front to back, this guidebook presents the CoRE method as a logical progression of topics beginning with basic concepts and notations.

1.2 INTENDED AUDIENCE

This guidebook is intended for engineers developing the requirements for production-quality software.

Experience in the development of real-time systems, particularly real-time system requirements, is needed to fully understand the details of CoRE. All of the notations specific to CoRE and their use are described in this guidebook. This guidebook assumes only a basic knowledge of the following concepts:

- Finite state machines
- Sets
- Boolean expressions

How these concepts are used to represent requirements in CoRE is fully described in this guidebook.

1.3 SCOPE OF THE METHOD AND GUIDEBOOK

The version of CoRE described in this guidebook supports specification of requirements for real-time embedded systems. CoRE provides a behavioral model of embedded system behavior; this is described in detail in Section 2. The behavioral model addresses behavioral requirements, including those for precision, timing, and accuracy.

Software requirements that are most easily captured using CoRE are those with properties that are consistent with the CoRE behavioral model. CoRE expresses requirements by defining the relationships the software must maintain between changes in the environment that the software monitors and the required effect on the devices, displays, and other observable quantities that the software controls. These are properties typical of embedded control systems, such as avionics and process control systems.

The CoRE guidebook currently does not provide explicit guidance for modeling complex data relationships and data transactions. Thus, this guidebook does not directly address parts of applications whose primary purpose is maintaining complex databases and implementing data transactions. These include some data-intensive command, control, communications, and intelligence (C³I) and information management systems. Subsequent Consortium products will provide methods and guidance for applying CoRE to C³I systems.

The scope of this guidebook includes all of the activities and products necessary for developing a detailed software requirements specification from system requirements. This guidebook addresses behavioral requirements (sometimes called functional requirements), timing constraints (including performance requirements), accuracy constraints, and software and hardware interfaces. It does not directly address some nonfunctional requirements, such as maintainability or capacity requirements. However, your current approach to these requirements can be used with CoRE.

1.4 ORGANIZATION OF THE GUIDEBOOK

This guidebook is structured to support self-study and for use as a reference. In particular, it is organized so it presents a logical progression of topics and a logical sequence of activities when read from front to back as follows:

- *Introduction and Background.* Sections 1, 2, 4, and 5 provide the introductory material necessary to understand CoRE's purpose, to gain an overall perspective of the way CoRE models requirements, and to understand the notational conventions used throughout the guidebook.
- *Example.* Section 3 introduces the sample problem, the FLMS, that will be used throughout the guidebook to illustrate CoRE concepts. A more detailed prose specification is given in Appendix A, and a complete specification of the FLMS is provided in Appendix B. The detailed process sections (Sections 7 through 12) discuss pieces of the specification in detail and provide rationale for their construction.
- *CoRE Concepts and Notation.* Sections 4 and 5 describe the underlying concepts, syntax, and semantics for representing the CoRE behavioral model and CoRE class model.
- *CoRE Process and Activities.* Sections 6 through 11 describe the CoRE process and give detailed guidance for each of the CoRE activities. Section 6 gives an overview of the CoRE process

from both an idealized and a practical perspective. Each of the subsequent sections describes one of the CoRE activities in detail.

- **Analyzing and Using a CoRE Specification.** Section 12 of the guidebook describes an overall process for analyzing completeness and consistency of a CoRE specification. This section discusses the detailed analyses provided in Sections 7 through 11 in the context of the overall process.

1.5 USING THIS GUIDEBOOK

This guidebook is primarily intended as a reference for practitioners of the CoRE method, but it can also be used, with supporting documentation, as a tutorial.

- **Reading for an Overview.** Sections 1, 2, and 6 provide a complete overview of the CoRE method. Section 1 describes the intended use of CoRE and the role of this guidebook. Section 2 gives an overview of the key ideas behind the CoRE method, including the standard models for capturing requirements. Section 6 gives an overview of the CoRE process, including the inputs, outputs, and key decisions in each CoRE activity.
- **Using the Guidebook as a Reference.** Sections 4 and 5 provide detailed discussions of the notations used to represent the behavioral and class models, respectively. Sections 6 through 11 provide a detailed guide to applying the CoRE method. Use Section 6 to get an overview of the process and understand which detailed process section applies. Use the detailed process section to understand a particular CoRE activity. Each of the detailed sections describes (for a given CoRE activity) the inputs needed, the work products produced, the detailed procedures, use of CoRE notation, and applicable heuristics.
- **Using the Guidebook for a Tutorial.** As a tutorial on the CoRE method, this guidebook is designed to be read from front to back. Section 2 introduces all of the important CoRE concepts, including the standard model for specifying CoRE requirements. Section 3 describes an example of an application; this example is used throughout the text to illustrate the method. Sections 4 and 5 describe the CoRE notations and their use. Section 6 gives an overview of the CoRE process and summarizes the key inputs, outputs, and decisions made in each CoRE activity. The subsequent sections then describe the activities in detail.

1.6 TYPOGRAPHIC CONVENTIONS

This guidebook uses the following typographic conventions:

Serif font General presentation of information.

Italicized serif font Mathematical expressions and publication titles.

Boldfaced serif font Section headings and emphasis. Section headings of a CoRE template.

Boldfaced italicized serif font Run-in headings in bulleted lists and low-level titles in the process section.

Sans serif font Variable names, expressions, or mode names from a CoRE example in the text. Specific parameters of a CoRE specification in the text.

Italicized sans serif font Within commands, generic values to be supplied by the user, in a CoRE example in the text, and in a CoRE term.

This page intentionally left blank.

2. THE CoRE MODELS

CoRE is a method for capturing, specifying, and analyzing real-time software requirements. CoRE provides a step-by-step approach, including principles and guidelines, for proceeding from system-level requirements to a precise, testable software requirements specification. A CoRE specification captures requirements in terms familiar to the customer (i.e., physical quantities monitored and controlled by the software) so issues of customer validation are addressed early in the development process. A CoRE requirements specification provides a precise description of the range of acceptable software behaviors; thus, a CoRE specification provides a common vehicle for communicating requirements among developers or contractors, acquisition managers, and users. A CoRE specification serves as both the test-to and design-to specification, ensuring that designers and testers are working to the same requirements. A CoRE specification is also sufficiently rigorous that the requirements can be analyzed for completeness and consistency. This helps the developer ensure that requirements errors are identified and corrected early in development.

This guidebook provides a description of all the CoRE activities and the order in which you are most likely to pursue those activities. CoRE provides evaluation criteria for deciding when an activity is complete. Each CoRE activity is defined by its entrance criteria, subactivities, evaluation criteria, and exit criteria.

A CoRE specification is written in terms of two underlying models: the behavioral model and the class model:

- The behavioral model provides a standard structure for analyzing and capturing the behavioral requirements of an embedded system; i.e., you specify what the software must do in terms of the behavioral model. The behavioral model provides the mechanisms you need to specify requirements that are precise, testable, complete, and consistent.
- The class model provides a set of facilities for packaging the information in a CoRE specification; i.e., you organize the specification as a set of classes. The class model allows you to divide the specification into relatively independent parts and control the relationships between the parts. The class model provides the mechanisms to manage requirements changes, create reusable requirements, and develop parts of the software system in parallel.

The two models are integrated in a single work product, a software requirements specification. The behavioral model defines the required behavior, i.e., what the software must do. The class model guides the basic organization of the specification.

This section describes the underlying CoRE models and how they address specific requirements issues, such as change management and the development of complete and consistent specifications. After reading this section, you should understand:

- The major steps of the CoRE process (Section 2.1)
- The purpose and objectives of the CoRE models (Section 2.2)
- The CoRE behavioral model for real-time software behavior, how behavioral requirements are captured in terms of the model, and how the behavioral model supports analysis of completeness and consistency (Section 2.3)
- The relationship between the CoRE behavioral model and the class model, how classes are used to manage complexity and change, and how the behavioral model guides you in choosing and defining classes (Section 2.4)
- How the behavioral and class models are integrated in a CoRE specification (Section 2.4.2)

2.1 CoRE PROCESS OVERVIEW

The CoRE process describes a sequence of activities that you follow to develop a CoRE requirements specification. The CoRE process is driven by two concerns. The first concern is the step-by-step construction of a required behavior specification in terms of the CoRE behavioral model. The goal is to develop a complete and consistent description of the required behavior. The second concern is the packaging of the specification in elements of the class model. This aspect of the process satisfies packaging goals, such as change management and reuse. Because packaging and specification activities overlap in time, the threads of these activities are intertwined in the CoRE process.

The input to the CoRE process is some form of system requirements. The output of the CoRE process is a complete specification of the software requirements (i.e., suitable input for a software design process). A complete overview of the intervening sequence of CoRE activities and products is given in Section 6; in brief, these activities are:

- Identify the environmental quantities with which the software interacts and the constraints among such quantities.
- Identify the software boundary by specifying the environmental quantities that the software must track or affect.
- Package the environmental quantities among a set of CoRE classes and relationships.
- Define the software behavior, timing, and accuracy constraints.
- Define the software inputs and outputs.

The first two steps initiate definition of the behavioral model by establishing which environmental quantities the software monitors and controls and the basic relationships the software must implement among them. The third step determines how the elements of the behavioral model should be allocated to CoRE classes and the relationships among the classes. The final steps complete the class definitions by filling in the details of the parts of the behavioral model allocated to each class.

2.2 PURPOSE OF THE CoRE MODELS

The CoRE method, including the use of the underlying models, has been developed to address specific problems of industrial developers of embedded software. This section describes the issues CoRE has been developed to address and how the CoRE models help address these issues.

2.2.1 RATIONALE FOR THE CoRE APPROACH

In creating CoRE, the Consortium met with developers to identify their problems with current requirements methods and tools. These representatives helped define a set of requirements that the CoRE method should meet. These requirements are summarized below:

- **Critical Applications.** The method must support the development of precise, testable specifications for real-time embedded systems.
- **Changing Requirements.** The method must support developing requirements specifications that are easy to change throughout the software life cycle. When requirements change, it must be easy to tell which parts of the requirements specification and other work products are affected.
- **Audience.** The method must support the development of requirements specifications that are understandable and useful to the specification's entire audience, including systems engineers, hardware engineers, and software engineers. It must support the ability to derive customer-oriented views of software requirements.
- **System Interface.** The method must support the delineation of system boundaries and the precise specification of system interfaces that concern the software. It must support descriptions of both the system and the environment in which it operates, including other systems under development.
- **Separation of Concerns.** The method must support the definition of requirements as a set of distinct and relatively independent parts. It must support localizing requirements that are fuzzy, incomplete, or defined at different levels of detail and allow work to proceed independently on distinct parts.
- **Derivation From System Requirements.** The method must include guidelines and examples of required inputs for the software requirements process and the form such inputs from systems engineering must take.
- **Nonalgorithmic Specification.** The method must allow nonalgorithmic specification of requirements when a specific algorithm is not actually required.
- **Consistent Requirements.** The method must define what makes a set of requirements consistent (unambiguous). It must include principles, guidelines, and techniques for determining whether requirements are internally consistent and for keeping these requirements internally consistent after they have been changed.
- **Complete Requirements.** It must be possible to determine where the requirements specification is internally incomplete. The method must permit definition and use of incomplete requirements. For example, the method must allow detailing and checking of one part of the requirements before another is complete.

The CoRE models have been developed so that these and other requirements concerns can be addressed in a CoRE specification.

2.2.2 PURPOSE OF THE BEHAVIORAL AND CLASS MODELS

The CoRE behavioral model provides a standard structure for analyzing and capturing the behavioral requirements for real-time embedded systems. The CoRE behavioral model is standardized in the

sense that every CoRE specification captures behavioral requirements using the same basic structures and relationships, i.e., as a set of relationships between the quantities the software system monitors and controls. This standardized approach allows CoRE to provide a well-defined sequence of activities and checks that proceeds systematically from the early analysis of what the software controls to a complete specification of required behavior. The behavioral model also provides mechanisms for expressing the behavioral requirements rigorously so the resulting specification is precise and testable.

Because a CoRE specification is written in a rigorous language, there are systematic procedures for determining the consistency and completeness. The use of mathematical expressions also helps avoid overspecifying the requirements. Defining required behavior in terms of operations or algorithms often necessitates introducing decisions that are not actually requirements (e.g., the order of operations and how one operation interacts with another). The designer cannot determine which parts of the model actually reflect the customer's requirements and which are artifacts of the construction. CoRE specifications capture only what the software must do, not how to do it.

A CoRE specification is structured, using object-oriented terminology, as a set of class definitions in which a class is a template for an object. The CoRE class is the basic mechanism for dividing the specification into parts and controlling the relationships among different parts of the specification. The set of classes and their relationships in a CoRE specification are collectively called the class model.

While the behavioral model addresses properties of the behavioral requirements, such as completeness, consistency, precision, and testability, the class model addresses packaging concerns. Packaging concerns are properties of a specification that result from the way information is partitioned. Packaging concerns include ease of change, ease of use, encapsulation of fuzzy requirements, and reusability. For example, if a requirement that is likely to change is encapsulated in a CoRE class definition, only that class definition will change if the requirement changes. Table 2-1 summarizes the properties supported by each of the CoRE models.

Table 2-1. Specification Properties Versus CoRE Mechanism

Specification Properties	Supporting CoRE Mechanism
Desired Functional Requirements Properties • Complete • Consistent • Precise • Unambiguous • Testable	• Behavioral model (Four-variable model)
Desired Packaging Properties • Easy to change • Encapsulates fuzzy requirements • Allows asynchronous development • Readable • Reusable	• Class model

The class and behavioral models are integrated into a single requirements specification in which the information in the behavioral model is partitioned among a set of CoRE classes. The classes make up the organization. You determine such characteristics as the number of classes, what information is hidden, and which parts of the model are allocated to the same class based on your overall goals for the requirements structure. You can address many of the issues of readability, reuse, and change management by judicious packaging.

Separating the behavioral and class models is one of the key features of CoRE. The separation allows you to make and subsequently change decisions about packaging issues with limited and controlled effect on the meaning of the specification. The CoRE class model allows you to write the specification to have desirable properties like ease of change without designing the software.

2.3 THE CoRE BEHAVIORAL MODEL

The CoRE behavioral model is based on a four-variable model developed by Parnas and Madey (1990) and Van Schouwen (1990). Portions of the discussion in this section are taken from Parnas and Madey (1990). The four-variable approach extends previous work on embedded system requirements as described by Heninger (1980) and Alspaugh et al. (1992). Many of the notations used to represent the behavioral model are derived from this work. The four variables are monitored, controlled, input, and output. The monitored and controlled variables are collectively called the environmental variables.

2.3.1 ENVIRONMENTAL VARIABLES

CoRE views a software system as existing within and interacting with an environment. An automotive engine control system, for example, exists within an environment that includes the engine parts, atmosphere, external load, driver-controlled devices, and so on. Only certain quantities in the environment are relevant to this particular system. For example, an automotive engine-control system needs information about air pressure but not altitude. An aircraft-control system needs information about altitude; air pressure is a means to measure that quantity.

CoRE represents each environmental quantity of interest with a mathematical variable (called an environmental variable) so that there is a clearly defined correspondence between the variable and the environmental quantity it models. There are two classes of environmental variables. Monitored variables represent environmental quantities that the software system must track, e.g., the ambient air pressure. Controlled variables represent quantities that the software system sets, e.g., the fuel flow to the cylinders. If there is feedback in the system, a variable can be both monitored and controlled. For example, the automotive engine system developer might define a monitored variable called `Air_Pressure` measured in pounds per square inch.

2.3.2 THE CoRE RELATIONS

CoRE captures the required, externally visible software behavior as a set of relations among the values of monitored and controlled variables. A CoRE specification maps possible values of the monitored variables to acceptable values of the controlled variables (not computer inputs to computer outputs). This corresponds to an intuitive notion of required behavior in that it relates specific, observable changes in the environment (e.g., the ambient air pressure decreases) to observable actions (e.g., the fuel flow is decreased).

The specification of required behavior in response to undesired events (i.e., failures of system components or of the system itself) is integrated with the specification of “normal” behavior. Engineers define **monitored variables** to denote undesired events. This allows you to abstract from the sources of undesired events, such as specific device failures.

The behavioral model defines the required, externally visible behavior in terms of two relations from monitored variables to controlled variables called NAT (for nature) and REQ (for required). The NAT relation describes those constraints placed on the software system by the external environment, e.g., physical laws and the properties of physical systems. These are properties of the environment that affect the software but exist whether the software exists or not. For software requirements, NAT includes the properties of the monitored or controlled hardware, e.g., the possible states of physical devices, such as the maximum and minimum degree of a flap’s elevation. The REQ relation describes the additional constraints on the controlled variables that must be enforced by the software. In writing REQ, you view the software system as a black box (Figure 2-1). REQ describes properties that the software system (i.e., the black box) is required to maintain between the monitored and controlled variables.

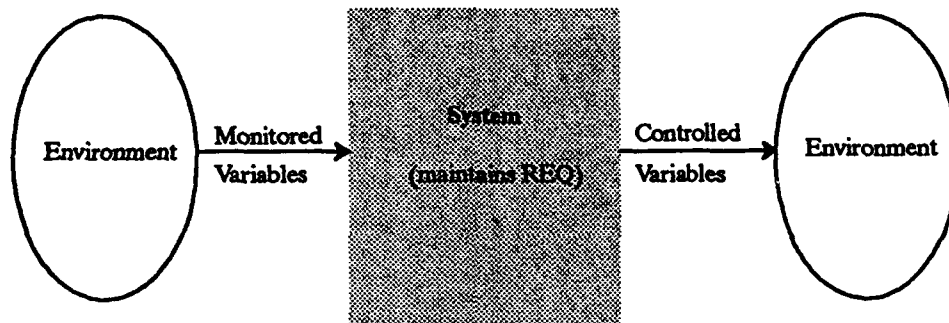


Figure 2-1. System Viewed as a Black Box

CoRE treats the software system’s actual inputs and outputs as resources available to the software to determine the values of monitored and controlled quantities. You complete a CoRE specification by describing the values provided by the system’s hardware devices and interfacing software systems or, if necessary, suitable abstractions of those values called the input variables and output variables (Figure 2-2). The relationship between the software system input and output and the environmental variables is expressed in two additional relations called IN (for input) and OUT (for output).

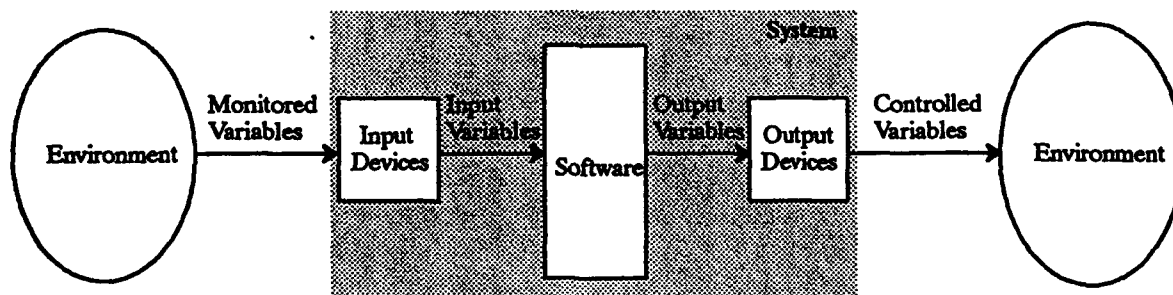


Figure 2-2. System Viewed With Input and Output

A CoRE specification is made precise, testable, and analyzable by describing the required relationships as mathematical relations. A relation maps the elements of one set to the elements of another. Each element of the first set can be mapped to one or more elements of the second². The first set is called the domain of the relation; the second set is called the range of the relation. In CoRE, you typically represent these relations by mapping the possible values of the monitored quantities to the acceptable values of the controlled quantities. This provides a nonalgorithmic description of the required behavior. The specification is complete when the mappings are mathematically complete; i.e., the relation defines the acceptable values of the controlled variables for all possible values of the monitored variables. Sections 2.3.3 and 2.3.4 describe the CoRE relations and their mathematical model in more detail.

Figure 2-3 shows the sets of variables and relations between the sets of variables; constraints within a set are also possible, such as environmental constraints on the possible values of individual environmental quantities. The NAT relation also includes constraints on monitored or controlled variables and between variables of the same type; e.g., temperature and pressure in a sealed reaction vessel are related quantities and change together.

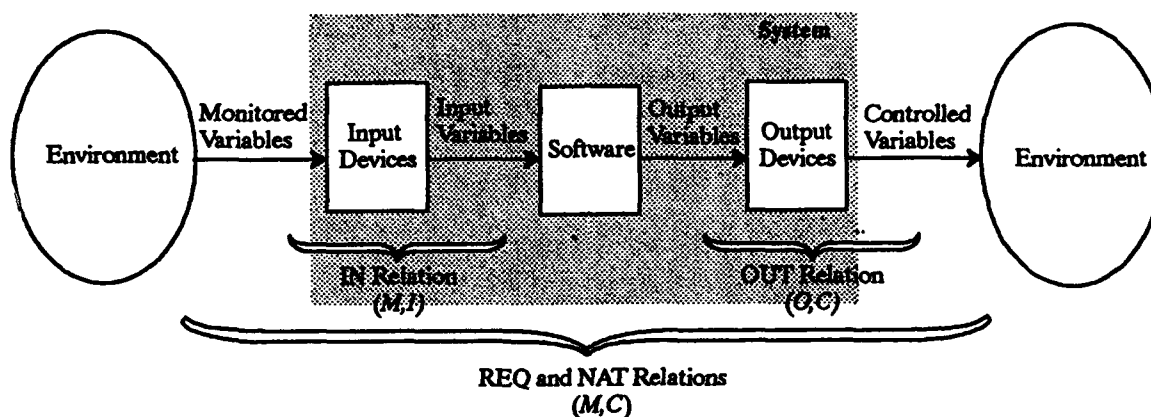


Figure 2-3. The Behavioral Model Relations

The relations of the behavioral model allow you to separate concerns for what the software system must do, viewed as a black box, from how the software system uses available hardware resources to do its job. The REQ and NAT relations describe requirements that will change if the purpose of the system changes but will not change if a hardware device is modified or replaced without changing the function of the system. The IN and OUT relations change if assumptions about the hardware or the hardware itself change but not if the same hardware is used to accomplish a slightly different purpose.

CoRE describes behavioral requirements as a relation from monitored quantities to controlled quantities rather than from inputs to outputs because the relationship that the software system must maintain between the monitored and controlled quantities is usually simpler, easier to write, and more intuitive than the relationship between inputs and outputs. This also allows the analyst to focus on the behavioral requirements that the customers and users are concerned with independently of the hardware issues. For example, an avionics system must typically track and report the aircraft's current position to a certain accuracy. This requirement can be expressed in terms of monitored and controlled quantities by stating the relationship that the displayed position (controlled) must have

2. A function is a relation in which each element of the first set maps to exactly one element of the second set. The CoRE relations are expressed by defining the ideal behavior as a function, then describing the allowed deviation from ideal.

relative to the actual position (monitored); e.g., the displayed latitude and longitude must equal the actual latitude and longitude plus or minus one tenth of a degree. The actual inputs used to calculate the position might be incremental accelerations on an inertial platform. Specifying the behavioral requirements in terms of incremental accelerations would be more difficult for both the writer and the reader and does nothing to improve the precision of the specification.

2.3.3 RELATIONS NAT AND REQ

In the following discussion, monitored quantities are denoted as $m_1, m_2, \dots, m_p$, and controlled quantities are denoted as $c_1, c_2, \dots, c_q$. Because certain quantities may be both monitored and controlled by the software system, these lists may have elements in common.

Each environmental quantity has a value that can be recorded as a function of time. For example, the engine temperature has a particular value at some given time t . For a given environmental variable v , you write the function, giving its value over time as v^t . The value of function v^t at a particular time t is written $v^t(t)$. It also makes sense to talk about the vector of all monitored variable functions ($m_1^t, m_2^t, \dots, m_p^t$) or all controlled variable functions ($c_1^t, c_2^t, \dots, c_q^t$). For the monitored variables, this is called the monitored state function and is written M^t . Similarly, the controlled state function is written C^t .

2.3.3.1 Relation NAT

The NAT relation describes all of the external constraints on the values that the environmental variables can assume. Physical laws, the properties of physical systems, and, where software is concerned, the properties of the interfacing hardware all constrain the possible values that the monitored and controlled variables can assume and the possible relations between them. For example, the NAT relation for flight program software might specify the aircraft's maximum rate of climb, maximum altitude, and other constraints typically described by the aircraft's flight envelope.

Relation NAT is defined as follows:

- The domain of NAT is the set of vectors containing exactly the values of M^t allowed by the environmental constraints.
- The range of NAT is the set of vectors containing exactly the values of C^t allowed by the environmental constraints.
- (M^t, C^t) is in NAT only if environmental constraints allow the controlled variables to take the values described by C^t when the monitored variables have the values given by M^t .

NAT is a relation rather than a function because there are typically many possible values of the controlled variables for a given set of values of the monitored variables.

The specification of NAT is important because it explicitly captures the limits of required behavior. For a specification to be complete, it needs to cover the set of possibilities allowed by the NAT relation. In particular, it must define all the possible values that the monitored variables can assume, all the possible values that the controlled variables can assume, and any constraints between their possible values. Conversely, you need not specify the REQ relation for any state not included in the NAT relation because it cannot occur. Therefore, the NAT relation makes explicit the environmental constraints that are implicit in most specifications. This allows a well-defined notion of completeness

constraints that are implicit in most specifications. This allows a well-defined notion of completeness because the analyst can ensure that the specification describes the required behavior for all possible values the software will encounter.

2.3.3.2 Relation REQ

The software system imposes additional constraints on the values of the controlled variables. These constraints are what you typically think of as the behavioral requirements. For example, the heater is required to be on if the temperature in the reaction vessel falls below 500 degrees. Equivalently, the controlled variable that is the heater state is constrained to have the value "on" whenever the state of the environment is such that the monitored variable corresponding to the vessel temperature has a value of less than 500 degrees.

Relation REQ is defined as follows:

- The domain of REQ is the set of vectors containing the values of M^t allowed by the environmental constraints.
- The range of REQ is the set of vectors containing the values of C^t allowed by the environmental constraints.
- (M^t, C^t) is in REQ only if the software may permit the controlled variables to take the values described by C^t when the monitored variables have the values given by M^t .

REQ is typically a relation rather than a function because there is tolerance in the required behavior in time, value, or both. This means that each monitored variable value is associated with more than one possible value of the controlled variables. For example, an aircraft's operational flight program is expected to show the current altitude plus or minus some number of feet. This is another way of saying that, for a given externally measurable altitude, there are a number of acceptable values that the program can display and still satisfy the requirements (i.e., any of those within the range of the actual altitude plus or minus the allowed tolerance). Where discrete outputs permit no tolerance in value, there is typically tolerance in time given by the maximum delay between the change in the monitored value and the corresponding output.

While REQ is a relation, the requirements often will be easier to write and use if the relation is specified by giving the ideal behavior as a function, then specifying the allowed tolerances in value or time separately. This follows standard engineering practice because the ideal function is usually much simpler than the complete relation; it is usually easier to understand the ideal behavior first, then understand how the behavior is allowed to deviate from the ideal. The approach also provides an appropriate separation of concerns because requirements for tolerances can change independently of requirements for ideal behavior.

2.3.4 RELATIONS IN AND OUT

Ultimately, the system development process must identify the resources available to the software to determine the values of the monitored variables and to affect the values of controlled variables. Early in the process, you may represent these resources by abstractions of the ultimate inputs and outputs. When the hardware becomes defined, the interface devices provide these values. In any case, a complete specification must define the resources available to the software. The behavioral model captures

monitored variables to software system inputs (input variables). Relation OUT gives the correspondence from software system outputs (output variables) to the controlled variables.

Relation IN describes what the software will see in terms of the available inputs for possible values of the monitored variables. This specifies the accuracy with which the environmental values of interest can be measured. Let I^t denote the vector $(i_1^t, i_2^t, \dots, i_q^t)$ of inputs to the system.

Relation IN is defined as follows:

- The domain of IN is the set of vectors containing the possible values of M^t .
- The range of IN is the set of vectors containing the possible values of I^t .
- (M^t, I^t) is in IN only if I^t describes the possible values of the inputs when M^t describes the monitored values.

Similarly, relation OUT specifies (mathematically) how the controlled variables are affected by sending particular values to the output devices. Let O^t denote the vector $(o_1^t, o_2^t, \dots, o_q^t)$ for each output of the system.

Relation OUT is defined as follows:

- The domain of OUT is the set of vectors containing the possible values of O^t .
- The range of OUT is the set of vectors containing the possible values of C^t .
- (O^t, C^t) is in OUT only if C^t describes the possible values of the controlled variables when O^t describes the output values.

In the FLMS example described in Section 3, the actual device used to determine the fuel level senses differential pressure and represents that pressure as an 8-bit unsigned integer with a particular scale, offset, and time delay. For this case, the IN relation must specify the exact relationship from the actual fuel level in the tank to the values that the program receives from the device. Note that both the accuracy and delay associated with the device are necessary parts of the specification. Like REQ, IN and OUT are relations because input and output devices have limited accuracy. Both input and output devices have associated delays.

By including the IN and OUT relations in the behavioral model, it is not the intention of CoRE to say that a given software requirements document necessarily describes the details of the input and output devices. It is understood that there are different conventions for allocating information to different types of documents (e.g., requirements versus design). The exact characteristics of the hardware may be unknown before detailed design. Finally, where you must specify a system as layers of abstract machines, values that are represented as monitored and controlled variables at one level may be represented as inputs and outputs in the specification of the layer below.

For a requirements specification, the IN and OUT relations serve two roles. First, where the actual hardware providing the inputs and outputs are part of the requirements (i.e., their use is not at the discretion of the designer), these requirements may be documented by the input and output variables and the IN and OUT relations. Second, these relations are useful for verifying certain properties of the requirements. You can use the IN and OUT relations to show that the software can, in practice,

the requirements. You can use the IN and OUT relations to show that the software can, in practice, monitor the quantities it is supposed to track and control the quantities it is supposed to affect through the available inputs and outputs. You can also show that the hardware characteristics are adequate to support the required precision of the monitored and controlled quantities.

2.4 THE CoRE CLASS MODEL

The CoRE class model provides a set of facilities for packaging the behavioral model as a set of objects, classes, and superclasses. The set of classes and the relationships for a given CoRE specification are called its class structure. The class structure is constructed using the facilities provided by the class model. The class model provides CoRE's packaging mechanisms.

CoRE classes provide facilities for abstraction and encapsulation. We define abstraction as a view of a problem that extracts the essential details relevant to a particular purpose. Encapsulation is the process of hiding details that are not relevant to your purpose by providing an abstract interface.

CoRE applies these basic principles to requirements. For CoRE, the "purpose" of abstracting and encapsulating requirements is to address specific packaging issues, such as ease of change, ease of use, encapsulation of fuzzy requirements, and reusability. For example, if you expect that certain requirements associated with controlling an aircraft engine will be the same across several systems (e.g., the same engine will be used for more than one aircraft), you can package those requirements in an engine-control class. The class would encapsulate exactly those requirements that were common to each use of the engine. The resulting class would then be reusable across the systems using the same engine.

CoRE differs from most object-oriented approaches in its separation between the behavioral model and class model. The CoRE class model is not intended to express requirements or constrain subsequent design³. For this reason, CoRE classes are defined entirely in terms of the behavioral model. Both the information defined on the interface of a CoRE class and the encapsulated information must be part of the definitions of the behavioral model (e.g., part of the definition of REQ, NAT, IN, or OUT). The definitions and relations of the behavioral model are packaged as a set of class definitions. The definitions and relations of the behavioral model determine the software's required behavior. The class model determines how that information is structured, which definitions are shared (can be used commonly), and which definitions cannot be shared.

2.4.1 OBJECTS AND CLASSES

The packaging mechanism in CoRE is the class, a template for an object. A class represents a piece of the requirements specification written in terms of the behavioral model. The requirements allocated to a class may apply in the same way to several components of the software application (e.g., a class may specify the requirements associated with controlling each of the engines of a four-engine aircraft). A piece of a requirements specification as applied to a specific component of the software application is an object. Thus, the requirements for controlling the left inboard engine, for controlling the left outboard engine, and so on are objects corresponding to the class of engine control requirements.

3. If an object-oriented design method is used, the CoRE class structure provides guidance in determining what requirements have common properties (e.g., are likely to change together). However, the designer is free to refine or alter the class structure if necessary since it does not define a requirement on the structure of the design.

Rather than create a separate piece of the requirements specification, i.e., a separate object, for each piece of the software application to which the requirements may be applied, CoRE groups the requirements that apply identically to a group of components and specifies them in terms of a class. The basic distinctions are as follows:

- **Class:** A class is a template for an object or a set of related objects. A class defines a set of requirements or terms common to one or more objects. A class definition has an interface and a set of encapsulated information. The encapsulated information cannot be used outside of the class definition. Information defined on the interface can be used in the definition of other classes.
- **Object:** An object is an instance of a class as applied to a single component of the software application. The object specifies a subset of the definition of REQ, NAT, IN, and OUT relations (including definitions of the variables and terms) for a component.
- **Interface:** A class interface is constrained to be defined using only **terms**. A term must be an expression written in terms of the monitored variables; e.g., monitored, conditions, events, and predicates on the modes are all terms. Terms provide a mechanism for abstracting details of the behavioral model.

Because the requirements associated with each object of a class are the same, writing out the object definitions individually would introduce redundant information wherever there is more than one object in a class. Redundancies in the requirements unnecessarily increase the size of the specification and make it harder to manage changes. To keep the specification concise and to avoid redundancy, a CoRE specification is written entirely as class (not object) definitions even where there is only one potential object in a class.

EXAMPLE: An avionics system monitors (e.g., oil pressure, oil temperature) and controls (e.g., propeller pitch) each of the engines of a four-engine aircraft. The requirements implemented by the software are exactly the same for each of the four engines (i.e., relations specifying the required behavior are the same except for the engine number). You would then define the engine control requirements as a class. Each object in the class would correspond to the requirements for one of the engines. The specification then needs to contain only the definition of the class.

2.4.2 PACKAGING RELATIONSHIPS AMONG CLASSES

The packaging properties of a CoRE specification are determined by three relationships among CoRE classes: encapsulates, depends-on, and generalization/specialization. The specific packaging properties of your specification are determined primarily by these relationships.

2.4.2.1 Encapsulates

A CoRE class may encapsulate the definition of other CoRE classes. A class encapsulates some subset of the behavioral specification. One benefit of encapsulation is that it limits the impact of change. Where the encapsulated information is complex or parts of it are likely to change independently, additional packaging may be useful. To address such issues, CoRE allows you to define a class in terms of

other classes. For example, a class `Valve_Interface` might define the behavior for the controlled variable `Valve` and an indicator light `Valve_Indicator` that the software must light when the valve is open. These are packaged as part of the same class because their values change together, but they also represent pieces of the specification that can change independently; e.g., the hardware used for each may change, so the outputs and OUT relations may change. You can ensure that such a change affects only one relation by defining the encapsulated part of `Valve_Interface` as two classes: one encapsulating the requirements for `Valve`, the other containing the requirements for `Valve_Indicator`.

The encapsulates relation induces a hierarchy on the set of classes called the encapsulation structure, i.e., class `Valve_Interface` encapsulates classes `Valve` and `Valve_Indicator` and is one level above them in the encapsulation hierarchy. CoRE constrains the encapsulates relation so that all requirements are defined by the classes at the leaves of the hierarchy. In particular, the leaf classes, such as `Valve` and `Valve_Indicator`, must contain all of the definitions of the REQ, NAT, IN, and OUT relations. Classes above the leaves may export terms defined by their encapsulated classes and may define additional terms that are functions of the terms or variables defined by their encapsulated classes. However, they may not define other parts of the behavioral variables or relations.

2.4.2.2 Depends-on

The depends-on relation denotes exactly which classes use what information provided by other classes. Classes may use each other only by using the terms defined on the class interface. A class `X` uses a term `T` provided by class `Y` only if `X` employs term `T` in its definition. In this case, we say that the definition of class `X` depends on class `Y`. For example, in Figure 2-4, the arrow labeled `valve_open`

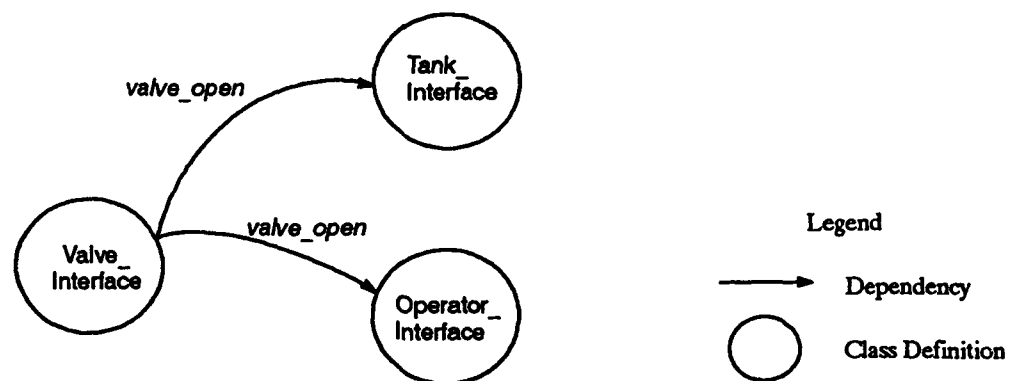


Figure 2-4. Graphic Depiction of the Depends-on Relation

denotes that the term `valve_open` is defined on the interface of class `Valve_Interface` and used in the definitions of classes `Tank_Interface` and `Operator_Interface`.

Managing the depends-on relation is critical to accomplishing many packaging goals. For example, how classes depend on each other determines which requirements changes will be difficult to make and which will be relatively easy. If many classes depend on information that changes, then all those class definitions may need to change as well.

2.4.2.3 Generalization/Specialization

The generalization/specialization structure is used to denote the inheritance relation. Inheritance is a mechanism for specifying common properties among a set of classes. A superclass defines a set of common properties and acts as a template for a class, much as the classes serve as templates for objects. You may define one or more subclasses of a superclass. Objects of a subclass inherit the properties of the superclass plus additional properties defined by the subclass. These constructs are defined as follows:

- **Superclass:** A superclass in CoRE defines a set of requirements or terms that is common to two or more CoRE classes. A superclass is defined in exactly the same way as a class.
- **Inheritance:** Inheritance denotes the requirements or terms that are defined by a superclass and shared among its subclasses. In CoRE, a class inherits the contents of its superclass by encapsulating the definition of the superclass and using the superclass interface (i.e., the superclass definition acts like an encapsulated class for each of its subclasses).
- **Subclass:** A subclass is a class that is defined as an instance of a superclass. In CoRE, a subclass specializes the definition of its superclass by adding or constraining requirements. A class may be a subclass of only one superclass.

A subclass inherits the properties of a superclass by using the terms and variables defined on the superclass interface. A subclass of a superclass is constrained to use all the information defined on the interface of the superclass in its definition but may add or constrain information.

You use CoRE's generalization/specialization structure both to reduce the redundancy in a specification and to make commonality in requirements explicit. Where there are two or more classes with many requirements in common, representing these requirements in a superclass reduces the amount of specification that must be repeated. In addition, the use of a superclass makes explicit the commonality between the subclasses, giving subsequent developers the opportunity to simplify the design.

EXAMPLE: Part of an avionics system monitors and controls the aircraft's communications radios. The system contains two types of data entry terminals: one Radio Tune Control Panel (RTCP) and two Command and Display Units (CDUs). Both units show the current state of each of the aircraft's six communication radios. The crew can also use either type of terminal to select a given radio and enter a new frequency. Frequencies can be entered numerically by giving an integer from 1 to 28 corresponding to a preset frequency or by entering a three-letter preset mnemonic. The types of terminals differ in that only the CDUs can be used to enter a mnemonic because only the CDUs have letters on their keypads. Otherwise, the requirements for getting frequencies, selecting a radio, and displaying radio state are the same for the two types of terminals.

The commonality in requirements for the terminals is an essential part of the problem in the sense that the commonality is driven by common underlying requirements for controlling the radios rather than being an incidental artifact of similar hardware. You can make such commonality explicit and reduce redundancy in the specification by creating a Radio Terminal superclass that encapsulates all the common requirements. You would then define a CDU subclass that refines the Radio Terminal superclass by adding the requirements associated with entering frequencies as mnemonics.

2.4.3 ALLOCATING THE BEHAVIORAL MODEL TO CLASSES

The class definitions, including the encapsulated information and the class interface, are constrained by the behavioral model. In particular, all of the interface information provided by the classes must be expressed in terms of the environmental variables. Thus, the class interfaces only provide information in terms of the state of environmental variables, changes in their state, or the history of such changes. This keeps the model consistent with CoRE's goal of providing nonalgorithmic specification.

A representative allocation of elements of the behavioral model to a class structure is shown in Figure 2-5. While an actual specification will be much more complicated in terms of the number of classes, levels in the class hierarchies, and use of information, the complex structure is composed of many such simple structures that are used repeatedly. When the basic structure is understood, it is not difficult to read a CoRE specification.

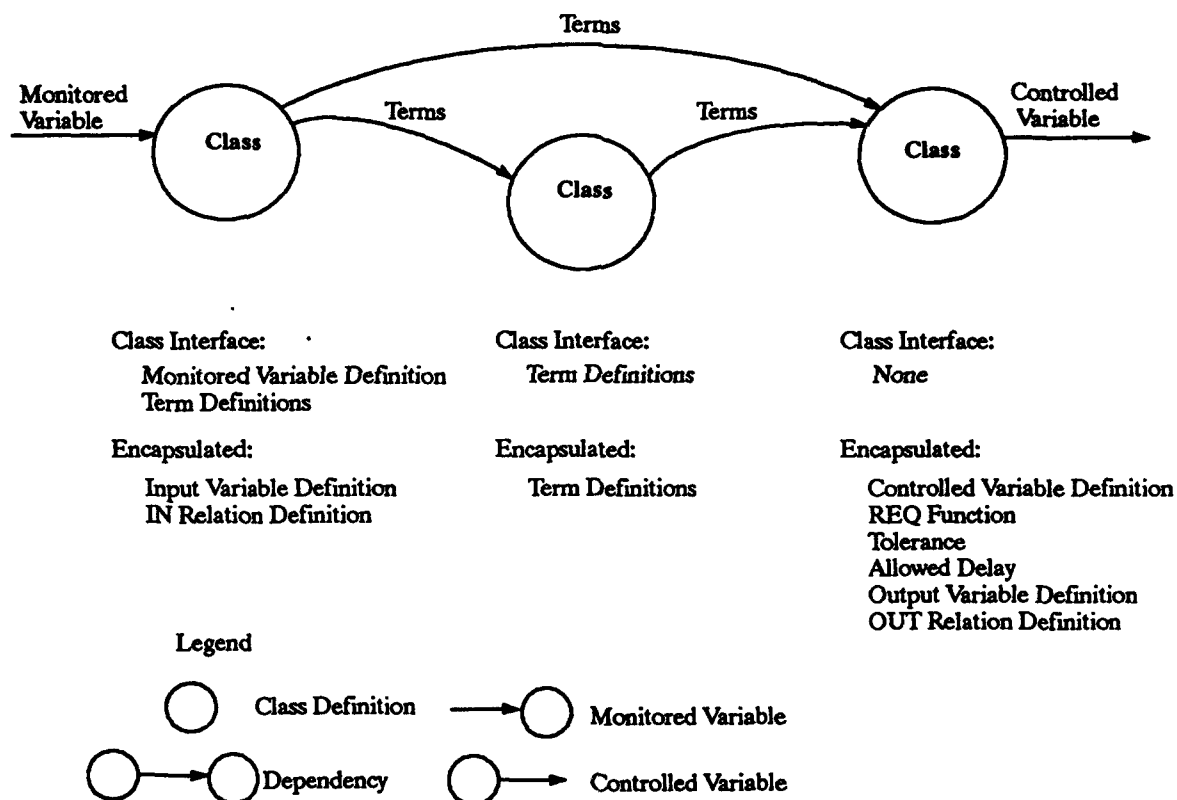


Figure 2-5. Canonical Allocation of Behavioral Model to Classes

In general, a class that defines a controlled variable defines the REQ relation for the variable. The REQ relation is defined using terms defined by other classes in the requirements specification. Other classes can define finite state machines (mode machine) or terms; both the modes and terms are themselves required to be functions of other modes, terms, or monitored variables. Thus, the internal classes also use modes and terms defined by other classes. Classes that define monitored variables provide the variable or terms to other classes.

The CoRE class model differs from others in that class interfaces are not defined in terms of operations (also called methods, access programs, or functions) and object interactions are not

defined in terms of messages. Because CoRE is intended to specify only required behavior, not design, CoRE uses specification techniques based on the behavioral model rather than programming (i.e., sets and relations rather than operations or procedures). You specify what information one class may use about another but not the mechanism by which it will be used (i.e., the protocol or transfer mechanism).

3. AN EXAMPLE: THE FUEL LEVEL MONITORING SYSTEM

CoRE principles and heuristics are illustrated by applying them to a small real-time example called the FLMS Specification. This section describes the FLMS system mission and requirements.

The FLMS is a simplified version of a safety shutdown system that is part of a shipboard fuel level monitoring and control system. The overall system provides fuel to the engines and moves fuel between the shipboard tanks to ensure a constant supply and to help maintain trim. The safety shutdown system is a separate component of the overall system that shuts down the fuel pumps under unsafe conditions, such as too low or too high a fuel level in a tank. The problem is simplified by allocating a single tank and pair of pumps to the software (a similar system would monitor the other tank or engine pairs). The example also assumes a relatively simple method of measuring the fuel level based on differential pressure in the tank (i.e., you do not address issues like extreme roll or pitch that complicate the measure of fuel level in a real shipboard system). The FLMS problem is based on a similar problem by Van Shouwen (1990). Figure 3-1 shows a front view of an FLMS pump and tank. The prose description of the FLMS problem is as follows.

The design of a fuel control system typically comprises automatic or manual control mechanisms (engine and fuel-level control) and safety monitoring devices. The safety monitoring devices include: fuel gauges and gauge cocks that convey the fuel level in the tank, fusible plugs or fuse alarms that alert the operator when the fuel level is too low or too high, and fuel flow rate gauges and other gauges showing the engine's operating conditions. The FLMS is intended to replace or complement the above-mentioned devices. It monitors and displays the fuel level in the tank and provides visible and audible alarms for high and low fuel levels. With the currently selected hardware configuration, fuel level is displayed in a window on a cathode-ray tube (CRT) display, two "annunciation" windows on the CRT provide visible indication of exceeded fuel-level limits, and the computer's speaker provides the audible alarm.

In addition to annunciation windows and the alarm, the pumps are shut down under the following conditions: (1) when the fuel level is too high because an overly high fuel level can cause pipeline rupture; (2) when the fuel level is too low because an overly low fuel level may result in the engine running dry and being damaged; and (3) when the monitoring system detects that it is unable to determine the fuel level due to the failure of a sensor. It is assumed that the shutdown mechanism is relay operated. Hence, the FLMS outputs a single signal when the pumps are to be shut down.

The FLMS provides two push buttons that are used for the following purposes: (1) the button labeled SELF TEST allows the operator to check the FLMS's output hardware while the system is shut down; and (2) the button labeled RESET allows the system to be brought back into normal operation following a shutdown or testing as long as the fuel level is within a specified range.

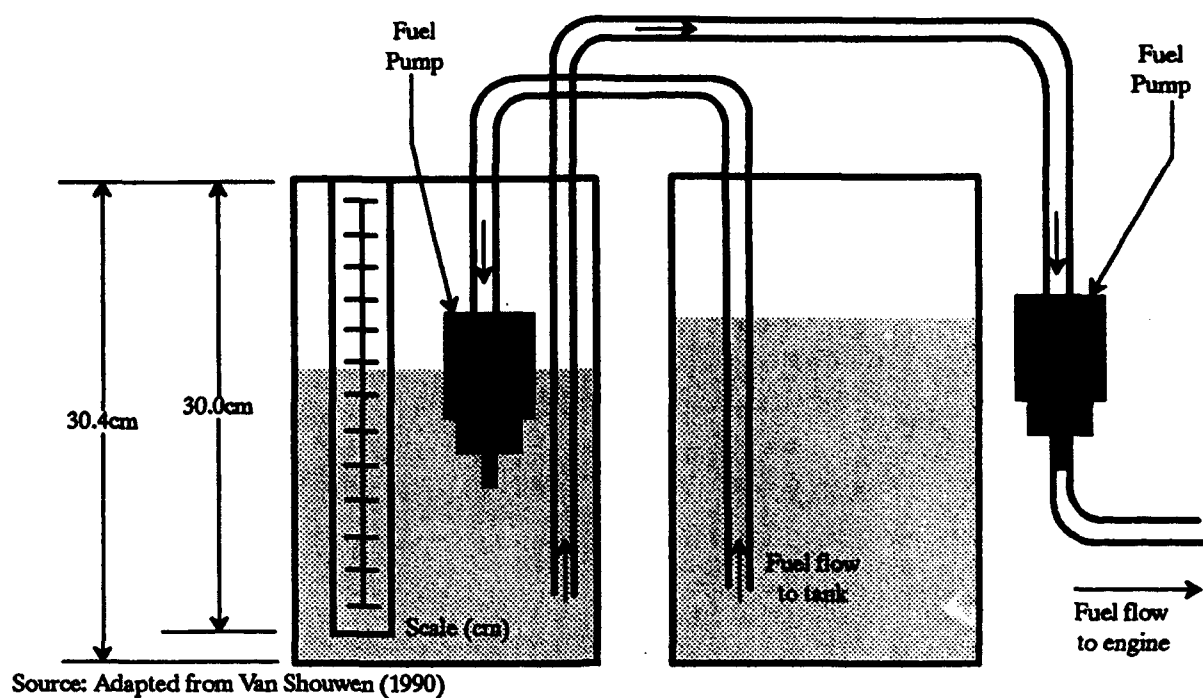


Figure 3-1. Fuel Level Monitoring System Pump and Tank Configuration (Front View)

4. REPRESENTING THE CoRE BEHAVIORAL MODEL

This section describes the underlying concepts of the CoRE behavioral model and presents the syntax and semantics of the notation used to represent the model. It also defines CoRE's approach to specifying timing and accuracy constraints. Chapters 8 and 10 describe how and when the concepts and notations are applied to develop a CoRE specification.

The behavioral model captures the software's required behavior, including the required values of the controlled variables, the timing constraints, the modes of the system, and the conditions and events that cause modes to change. As discussed in Section 2.3, the CoRE behavioral model captures required behavior in terms of the relationship between monitored and controlled quantities over time. The behavioral model captures the value and timing requirements using two complementary views: (1) the functional view captures the software behavioral requirements as a set of functions, i.e., what values the software must produce; and (2) the dynamic view captures required timing behavior and scheduling characteristics, i.e., when the software must initiate and complete the required behavior.

CoRE's functional view is based on the four-variable model (Parnas and Madey 1990). To capture the relations of the four-variable model, CoRE uses the following:

- Conditions characterize the state of the monitored or controlled variables.
- Events characterize changes in the state of environmental variables.
- Mode machines (a form of finite state machine) characterize the history of events.
- Functions define the required values of controlled variables in terms of conditions, events, modes, and terms (expressions of monitored variables).
- Tolerances define the range of acceptable behaviors.

The relation REQ maps every possible value of the monitored variables to acceptable values of the controlled variables. To make the specification of REQ readable, CoRE decomposes the relation into parts. In particular, there is a set of functions associated with each controlled variable:

- The ideal value function maps each value of the monitored variables in its domain to an ideal value of the controlled variable.
- The tolerance function defines the acceptable range of behaviors for every possible value of the monitored variables.
- The timing constraints define the acceptable range of behavior in time (e.g., acceptable delay) for all possible values of the monitored variables. This is captured as part of the dynamic view.

Where the value of a controlled variable depends on the history of events, the controlled variable function is written in terms of modes. A given set of modes may be used to define many controlled variable functions. Thus, the functions and modes associated with one controlled variable define one part of the REQ relation for all the values of the monitored variables that determine the value of that controlled variable. The union of all of the controlled variable functions and modes then defines the entire REQ relation. For convenience in the following sections, the part of the REQ relation associated with a single controlled variable is described as the REQ relation for that variable.

The CoRE functional view is illustrated in Figure 4-1. The arrows show where one part of the representation uses another. The required values of the controlled variables are written as a set of functions in terms of the modes, monitored variables, and terms. The mode machines change mode based on events defined as changes in the values of the monitored variables.

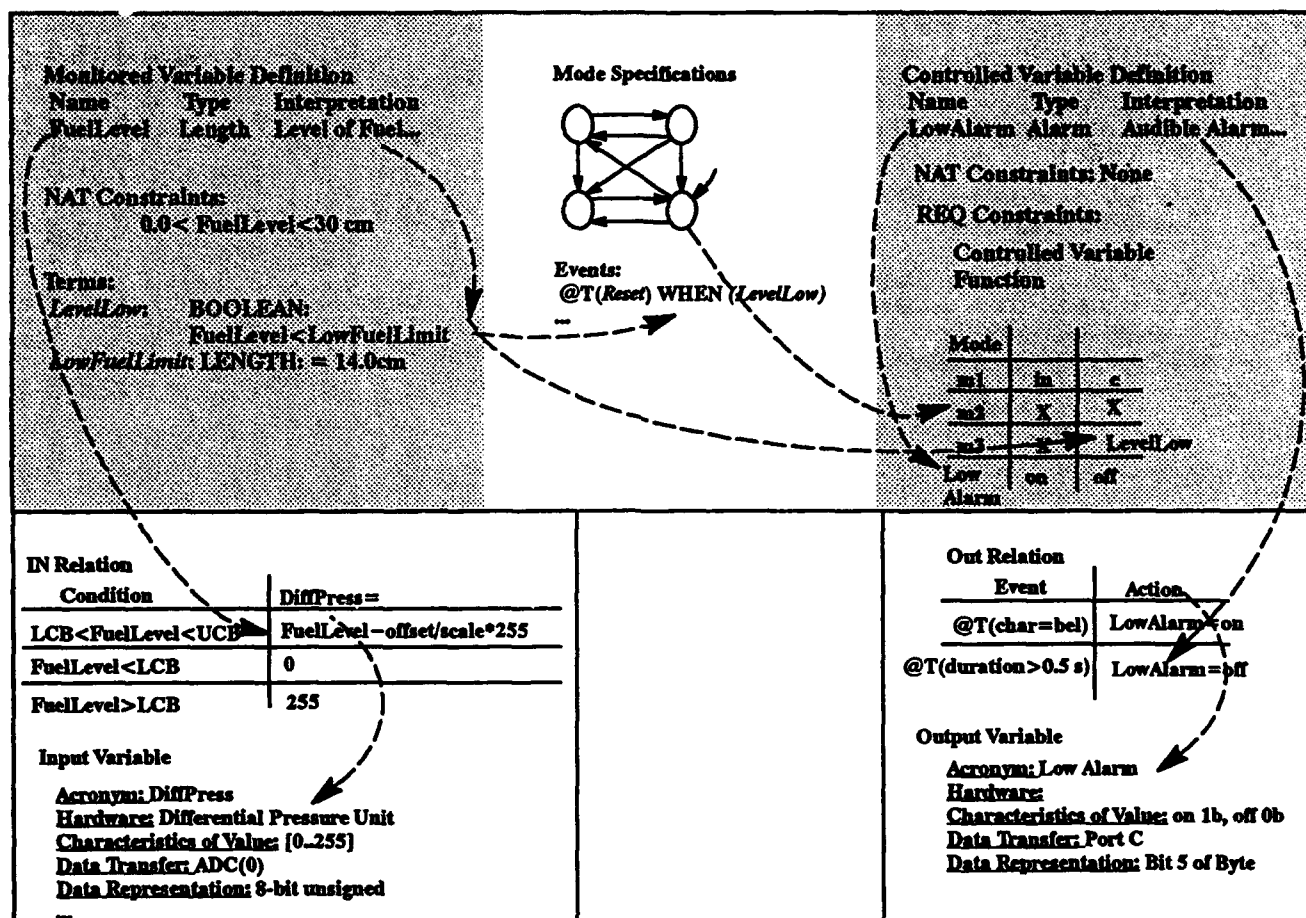


Figure 4-1. Representation of CoRE's Functional View

As for REQ, the remaining relations of the four-variable model, NAT, IN, and OUT are decomposed for readability and specified in parts. The NAT relation is decomposed and specified along with the relevant controlled variable functions or environmental variables. The IN and OUT relations are decomposed so that the part of the IN relation associated with a particular monitored variable is defined with that variable. Similarly, the part of the OUT relation for a controlled variable is defined with that variable. This is discussed further in Section 11.

CoRE's dynamic view defines the required behavior in time. You capture the dynamic requirements by specifying the scheduling and timing behavior relative to controlled variables. In particular:

- Define the scheduling requirement for each controlled variable function. Specify whether the value must be set periodically or on demand.
- Define the tolerance in time, such as the maximum delay allowed to set or update a controlled variable.

CoRE provides notations for representing each part, both the functional and dynamic views. The following sections introduce the notation and semantics for each of these components in turn.

4.1 REPRESENTING THE FUNCTIONAL VIEW

The functional view specifies the required behavior of the controlled variables as functions of the monitored variables. This section describes the notation and semantics of each of the elements used to represent the functional view, including monitored and controlled variables, conditions, events, and terms. It also provides a set of tabular representations for CoRE functions: condition tables, event tables, and selector tables.

4.1.1 MONITORED AND CONTROLLED VARIABLES

CoRE models the environmental quantities of interest with monitored and controlled variables. The template for defining a monitored or controlled variable is shown in Table 4-1.

Table 4-1. Template for Monitored and Controlled Variable Definitions

Name	Type	Values	Physical Interpretation
Variable name	Variable type	Possible values of the variable.	Description of the quantity modeled by the variable.

- **Type.** Specification of the units of measurement for the variable (e.g., feet, pounds per square inch, degrees). For enumerated types, use **ENUMERATED**. For Boolean types, use **BOOLEAN**. You may also define types as needed.
- **Values.** The range and precision of the values that environmental variables can assume in value. For enumerated variables, list the values. For numeric variables, record the lowest and highest values the variable can assume. The precision can be given as a decimal with the value (e.g., 0.02) or separately. This specifies the precision and the range of values over which the software is required to be able to track a monitored variable or set a controlled variable.
- **Physical Interpretation.** A description of the relationship between the monitored or controlled variable and the quantity that the variable models. The physical interpretation relates the quantities used to write the specification to externally visible phenomena.

4.1.2 CONDITIONS

In CoRE, the information that characterizes the environmental state and state changes is recorded using a language of events and conditions. A **condition** is a predicate (i.e., a statement that is true or

false) about the values of environmental variables that holds for a continuous, measurable period of time. A condition characterizes an aspect of the environmental or system state. For example, if altitude is a monitored variable, `mon_Altitude > 500 feet` is a condition that is true or false for an aircraft at any given time.

A condition is represented by a Boolean expression. Compound conditions are formed by connecting two or more conditions using the logical operators AND, OR, and NOT. For example, given the conditions C , C_1 , and C_2 , the following are compound conditions:

Compound condition:	Is true when:
NOT C	C is not true
C_1 AND C_2	Both C_1 and C_2 are true
C_1 OR C_2	C_1 or C_2 or both are true

The operations are listed in the descending order of precedence. You can use parentheses to alter the evaluation order. By definition, each compound condition is also a condition.

4.1.3 EVENTS AND EVENT EXPRESSIONS

4.1.3.1 Definitions

An event occurs when a condition changes value. Hence, any condition has two kinds of events associated with it; those events that occur precisely when the condition changes from false to true and those that occur when the condition changes from true to false. An event occurrence is a moment in time when a condition's value changes. Each event occurrence is instantaneous (takes zero time) and atomic (all or none occurs).

Event expressions are used to represent the set of events associated with a particular condition. CoRE uses the notations $@T(C)$ and $@F(C)$ to represent event expressions denoting changes in the state of a condition C . An event denoted by $@T(C)$ occurs at any moment in time when the condition C transitions from false to true (Boolean expression evaluates to true). Similarly, $@F(C)$ signifies any event of the condition C becoming false.

For example, consider the monitored variable `mon_Push_Button`, representing the state of a push button. At any moment in time, the button is in one of the two possible states: pressed or released. The following event expressions denote the events corresponding to changes in the state of the button:

Event expression:	Represents:
$@T(\text{mon_Push_Button} = \text{pressed})$	The event class corresponding to a change in the state of the button from released to pressed
$@F(\text{mon_Push_Button} = \text{pressed})$	The event class corresponding to a change in the state of the button from pressed to released

Where the occurrence of an event depends on the truth or falsehood of other conditions, CoRE provides the notation $@T(C_1) \text{ WHEN } C_2$. This event occurs at any instant in time when C_1 transitions from false to true while (given that) C_2 is true at the same time.

For example, the expression $@T(\text{mon_Reset_Switch} = \text{pressed}) \text{ WHEN } \text{term_Fuel_Level_Range} = \text{withinlimits}$ represents the event of the variable `mon_Reset_Switch` changing value to pressed while the variable `term_Fuel_Level_Range` has the value `withinlimits`.

Compound event expressions are formed by connecting two or more event expressions with the operator OR. Note the difference between $@T(C_1 \text{ OR } C_2)$ and $@T(C_1) \text{ OR } @T(C_2)$:

- The event $@T(C_1 \text{ OR } C_2)$ occurs when the disjunction $C_1 \text{ OR } C_2$ changes value from false to true. This occurs when one condition becomes true at a time when both were false. It does not occur if one condition becomes true while the other condition is already true.
- The event $@T(C_1) \text{ OR } @T(C_2)$ occurs when either event occurs.

4.1.3.2 Implementation Considerations

The above definitions of conditions and event expressions represent an ideal view in the sense that the definitions abstract from the practical issues involved with evaluating conditions and detecting events occurring in real-time. For example, it is possible for two or more occurrences of an event to happen so closely together in time that the software cannot distinguish them as discrete events. Similarly, a condition may be true for such a short interval that this is missed by the software.

Where these concerns are an issue, the specification must describe the tolerances in behavior; i.e., the minimum interval over which events will be detected. You use the NAT relation to record properties of the environment, such as the minimum possible interval between events. CoRE does not provide any additional notation to specify the tolerance in evaluating conditions or event expressions so any such specifications must be provided in the commentary. For example, you should specify the order of evaluation in the following cases.

1. Consider the evaluation of $@T(C_1)$ WHEN C_2 where the same external stimulus causes $@F(C_2)$ to occur shortly after $@T(C_1)$; i.e., the same external stimulus causes the value of condition C_2 to change from true to false shortly after $@T(C_1)$ takes place. In this case, the implementation must use the value of C_2 , obtained most recently before the occurrence of the event $@T(C_1)$.
2. Consider the case where the condition C_2 is only defined (becomes accessible) after the event $@T(C_1)$ occurs. In this case, the specification must clearly state that C_2 be tested only after the event $@T(C_1)$ occurs.
3. Consider the case of two external stimuli S_1 and S_2 both causing the event $@T(C_1)$ to occur so that S_2 changes the value of C_2 from true to false or from false to true while S_1 has no impact on C_2 . If both stimuli should arrive nearly simultaneously and there is a requirement for handling S_1 before S_2 , then condition C_2 must be checked both before and after evaluating $@T(C_1)$.

In general, the specification should state the sequencing and timing of event detection and specify the order in which the conditions in an event expression must be evaluated if the correct behavior depends on it.

4.1.4 TERMS

The definition of several controlled variable functions may depend on expressions of monitored variables. Rewriting the same expression throughout the specification can be tedious and error prone. To simplify the specification and to note such dependencies explicitly, CoRE provides terms. A term

is a named expression of one or more monitored variables, i.e., a formula that defines the computation of a value using one or more monitored variables to which you have assigned a name. Each term has a value and a type that is determined by the type of its constituent monitored variables and the operators applied.

Using terms, you can abbreviate or replace lengthy expressions with names. The notion of terms in CoRE is analogous to the concept of language macros (textual replacements) in some programming languages. Here are the most common reasons for defining terms:

- To shorten a complex and lengthy event expression used in one or more event tables or a compound condition used in one or more condition tables. The use of properly defined terms reduces errors of inconsistency and improves the clarity of the representations.
- To abstract a complex expression and hide its details. The reason may be that you have not yet finalized the details or you might want to change them later.

4.1.5 CAPTURING STATE HISTORY

For most real-time embedded systems, the software's behavior depends not only on the current values of the environmental variables but on how those values have changed over time (i.e., the history of events). For example, to release a weapon, the pilot must select the weapon, arm the weapon, aim the weapon, pull the trigger, in sequence, before the software actually releases the weapon. In CoRE, the *term state* is used to refer to the set of values of the environmental variables at a given time. The state history refers to how those values have changed over time. Thus, the required behavior of a system is usually a function of both the current state and the state history. CoRE captures such environmental variable state history using a form of finite state machine called a mode machine.

The following discussion assumes that the reader is familiar with the basic concepts of finite state automata and their representation as state transition diagrams and decision tables.

4.1.5.1 Modes and Mode Machines

A mode machine definition consists of:

- A finite set of states called modes.
- A distinguished initial mode.
- A set of transition events—events that cause transitions from one mode to another. Mode transitions are atomic and instantaneous (i.e., completed in zero time).
- A set of mode transitions. Each mode transition maps a mode and an event to a new mode.

A mode machine specializes the notion of a state machine, first, in that the behavior of the machine must be defined entirely in terms of the CoRE behavioral model and, second, in that a mode machine

does not define "actions"⁴. The modes of the mode machine are defined in terms of the states of CoRE environmental variables, and the transition events are defined in terms of changes to the environmental variables.

You define a mode machine using a form of state transition diagram called a mode transition diagram or a form of state transition table called a mode transition table.

4.1.5.2 Mode Transition Diagram

A mode transition diagram provides a graphic representation of a mode machine from the following components:

- Modes are represented by rectangular boxes containing the names of the modes they represent.
- Mode transitions are represented by lines with arrowheads showing the direction of the transition.
- Transition events are represented by event expressions labeling the transition arcs where the event causes a transition.

The initial mode is distinguished by a mode transition terminating in the initial mode with no source. Figure 4-2 shows a simple mode transition diagram. The initial mode is mode_Shutdown. There is a transition from mode_Shutdown to mode_Operating on occurrence of event_Reset. The mode machine transitions from mode_Operating to mode_Hazard when the condition term_Fuel_Level_Range = withinlimits becomes false. It transitions back to mode_Operating from mode_Hazard if the event @T(term_Fuel_Level_Range = withinlimits) WHEN (NOT term_Selftest) occurs.

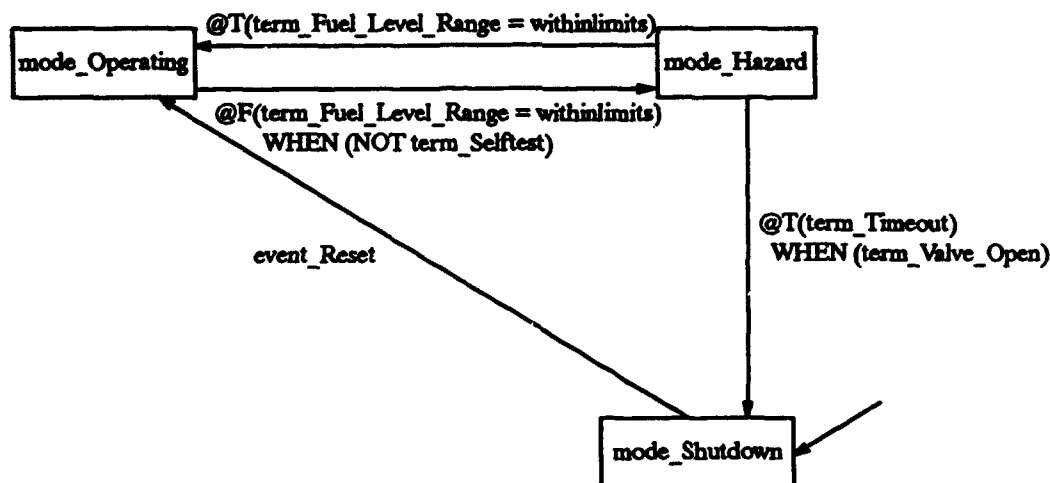


Figure 4-2. Example of a Mode Transition Diagram

4.1.5.3 Mode Transition Tables

A mode transition table provides a tabular representation of a mode machine. Table 4-2 represents the same mode machine as shown in the mode transition diagram in Figure 4-2. A mode transition table has the following form:

4. In many methods, finite state machines are used as finite controls where the machine performs specific actions or executes functions; e.g., the form of finite state machine used Real-Time Structured Analysis as described in Hatley and Pirbhai (1988). In CoRE, the machines are used only to capture state information. That state information is then used to specify the CoRE relations.

- The current mode is listed on the left side under the column heading Current Mode. Every mode of the machine must appear once.
- The set of possible conditions and events causing a mode transition is listed, one per column in the center of the table.
- The set of modes the machine can transition to from each current mode is listed on the right side of the table under the heading New Mode.

Table 4-2. Using a Table to Represent the Mode Machine In_Operation

Current Mode	event_Reset	term_Fuel_Level_Range = withinlimits	term_Timeout	term_Selftest	term_Valve_Open	New Mode
mode_Operating		@F				mode_Hazard
mode_Hazard		@T		f		mode_Operating
			@T		t	mode_Shutdown
mode_Shutdown	@T					mode_Operating

Each row in the table specifies an event that will trigger a transition from the current mode to the new mode:

- A table entry of @T under a condition heading C denotes @T(C) and implies that the triggering event occurs when the condition C changes from false to true.
- An entry of @F denotes @F(C) and implies that the triggering event occurs when the condition C changes from true to false.
- The lower-case entries t and f correspond to the WHEN clause and signify that the condition must have a particular value when the event occurs. For example, an entry of t with a column heading Y means WHEN(Y), and an entry of f means WHEN(NOT Y). There may be two or more t or f entries on each row; in this case, the WHEN clause condition is the conjunction of the corresponding conditions. If a condition does not affect a particular mode transition, then the corresponding box is blank.

For example, there is a transition from mode_Hazard to mode_Operating when the event @T(Fuel_Level_Range = withinlimits) WHEN (NOT term_Selftest) occurs as shown by the @T under the condition (Fuel_Level_Range = withinlimits and the t under term_Selftest in the row linking mode_Hazard and mode_Operating.

4.1.5.4 Properties of Mode Machines

A mode machine must be defined so that the machine is in only one mode at any given time. This means that it will always make sense to talk about the current mode of the machine. CoRE provides standard notation for referring to the current mode of a machine and the events associated with entering and exiting a given mode.

For each mode M of a given mode machine C , there are two related events, denoted by $ENTERED(C, M)$ and $EXITED(C, M)$, occurring exactly when the mode machine C enters and exits mode M , respectively. You can use these predefined events as event expressions.

CoRE also provides the predefined condition $INMODE(C, M)$ to refer to the current mode. The condition $INMODE(C, M)$ evaluates to true if mode machine C is currently in mode M ; it evaluates to false, otherwise. Where there is only one mode machine or where the context is well defined (e.g. in condition or event tables), the parameters of $INMODE$ may be omitted. Figure 4-3 illustrates the relationship between the predefined events $ENTERED(C, M)$ and $EXITED(C, M)$ and the predefined condition $INMODE(C, M)$ for a given machine C :

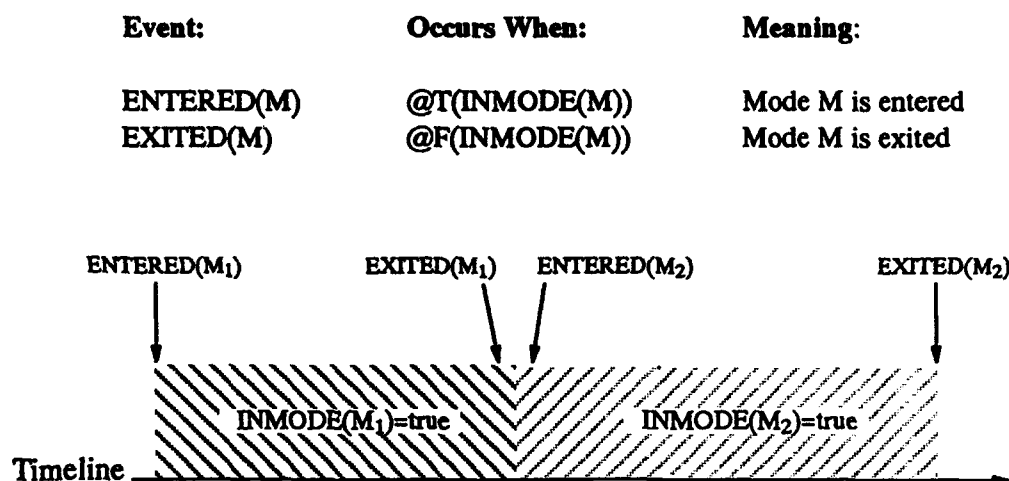


Figure 4-3. The Semantics of $INMODE$, $EXITED$, and $ENTERED$

Any number of mode machines may be used to capture distinct aspects of the software behavior. A CoRE specification, therefore, may include a set of concurrent mode machines. Mode transitions in one mode machine may be used as transition events in another mode machine.

4.1.6 TABULAR REPRESENTATION OF FUNCTIONS

CoRE provides three types of tables for recording the information required to characterize the controlled variable functions: condition tables, event tables, and selector tables. The CoRE tables provide a uniform, standardized organization for recording information and for promoting conciseness and completeness in writing specifications.

4.1.6.1 Condition Table

A condition table is used to represent a function where the value of a variable is a function of the modes and a set of mutually exclusive conditions. For example, use a condition table to specify the value function for a periodic controlled variable (see Section 4.2.1).

Each row in the table specifies a mode or a group of modes and all the relevant conditions that affect the value of the variable while in the mode. Table 4-3 illustrates the general format of a condition table. The *Expression* in the bottom row specifies the value of the variable V in mode M_i , given that the condition *Condition* holds true. Thus, to determine the value of the variable for a given mode and condition, (1) locate the row corresponding to the mode, (2) within the row, find the column corresponding to the condition, and (3) follow the column to the bottom of the table to find the value of the controlled variable. An X entry in the body of a condition table indicates that the expression at the bottom of the column is not used to set the value of the controlled variable in the mode on that row.

Table 4-3. Format of a Condition Table

Mode	Condition		
Mode M_1	<i>Condition_{1,1}</i>	..	<i>Condition_{1,n}</i>
..	..	..	
Mode M_m	<i>Condition_{m,1}</i>	..	<i>Condition_{m,n}</i>
Variable V	<i>Expression₁</i>	..	<i>Expression_n</i>

Each condition table must satisfy the following criteria:

- The modes of a condition table must belong to the same mode machine; every mode in the mode machine must appear only once in the table.
- The conditions in each row must be mutually exclusive.
- The disjunction of the conditions on each row must be true.
- The expressions specifying the value of the controlled variable must be of the appropriate type.

EXAMPLE: The periodic controlled variable *con_Altitude* (Table 4-4) is set to *Value₁* if condition C_1 is true; it is set to *Value₂* if C_2 is true whenever the system is in mode M_1 . Whenever the system is in mode M_2 or M_3 , the controlled variable is always set to *Value₁*. In mode M_4 , *con_Altitude* is set to *Value₁* if condition C_1 ; otherwise, it is set to *Value₂*. Note that, in mode M_1 , (C_1 OR C_2) must be always true and (C_1 AND C_2) must be false. This is the case because the conditions in each row of a condition table must be mutually exclusive and their disjunction must be always true.

4.1.6.2 Event Table

An event table is used to represent a function where the value of the variable must be set based on the occurrence of an event. For example, the value function for a demand controlled variable is written as an event table (see Section 4.2.2).

Table 4-4. Example of a Condition Table

Mode	Conditions	
M ₁	C ₁	C ₂
M ₂		
M ₃	INMODE	X
M ₄	C	NOT C
con_Altitude =	Value ₁	Value ₂

Each row in the table specifies a mode or a group of modes and all the events that cause the variable to change value while in the mode. The entries in the body of the table are event expressions. At the bottom of the table, there is an entry corresponding to each mode and event combination, defining the value of the variable. An X entry in the body of the table indicates that the expression at the bottom of that column is not used to set the value of the controlled variable in that mode. The table is arranged so that all the events with a common variable expression are in the same column.

Table 4-5 illustrates the general format of an event table. The *Expression* in the bottom row specifies what the value of the variable *V* is set to for a given mode *M_i* and a triggering event *EventExpression_{i,j}*. Thus, to determine the value of the variable for a given mode and event, (1) locate the row corresponding to the mode, (2) within the row, find the column with the corresponding event, and (3) follow the column to the bottom of the table to read the expression defining the variable value.

Table 4-5. Format of an Event Table

Mode	Event		
Mode <i>M₁</i>	<i>EventExpression_{1,1}</i>	..	<i>EventExpression_{1,n}</i>
..	..	..	..
Mode <i>M_m</i>	<i>EventExpression_{m,1}</i>	..	<i>EventExpression_{m,n}</i>
Variable <i>V</i> =	<i>Expression₁</i>	..	<i>Expression_n</i>

Each event table must satisfy the following criteria:

- The modes of a condition table must belong to the same mode machine; every mode in the mode machine must appear only once in the table.
- The triggering events in each row must be mutually exclusive; i.e., only one can occur at a time.
- The expressions specifying the value of the variable must be of the appropriate type.

EXAMPLE: In Table 4-6, the controlled variable con_Switch is set to closed if, while in mode M₁, condition C changes from true to false. con_Switch is set to open if C₁ changes from true to false while in M₂; it is set to closed at entry to M₂ if condition C₁ is true. The third row defines a grouping of modes M₃ and M₄. In this case, the controlled variable con_Switch is set to open at entry to either of the two modes; however, exiting M₃ and immediately entering M₄ is not counted as a new event occurrence; hence, con_Switch is not set to open again. Similarly, it is set to closed when leaving either M₃ or M₄ but not entering the other one. The controlled variable con_Switch is set to open on exiting M₅ and is set to closed on entering M₅.

if C_2 is true at the entry to this mode. Notice that the condition $@F(INMODE(M_5))$ is equivalent to $EXITED(M_5)$.

Table 4-6. Example of an Event Table

Mode	Events	
M_1	X	$@F(C)$
M_2	$@F(C_1)$	$@T(INMODE \text{ AND } C_1)$
(M_3, M_4)	ENTERED	$@F(INMODE)$
M_5	$@F(INMODE)$	ENTERED when (C_2)
con_Switch =	open	closed

4.1.6.3 Selector Table

A selector table is a tabular representation of strictly mode-dependent information. Each row of the table corresponds to the modes of a mode machine that completely determines the information. The columns provide the information that is relevant to each mode. Each column heading names a term or controlled variable whose value is specified in that column. Each mode of the mode machine must appear only once in the table. Each entry in the body of the table must depend entirely on the corresponding mode. An X entry in the table indicates that the item named at the column heading is not defined in that mode. Table 4-7 illustrates the general format of a selector table.

Use a selector table to specify a term, controlled variable function, or any quantity whose definition is completely determined by an active mode.

Table 4-7. Format of a Selector Table

Mode	Name or Description of Mode-Dependent Entity	..	Name or Description of Mode-Dependent Entity
Mode M_1	$Expression_1$	..	$Expression_1$
..	..	..	..
Mode M_n	$Expression_n$	..	$Expression_n$

EXAMPLE: In the following example of a selector table (Table 4-8), the value of the term `term_Max_Level` is 50 in `mode_Operating` and `mode_Shutdown` and 12 in `mode_Failure`.

Table 4-8. Example of a Selector Table

Mode	term_Max_Level
mode_Operating	50
mode_Shutdown	
mode_Failure	12

4.2 REPRESENTING THE DYNAMIC VIEW

The dynamic view of the CoRE behavioral model describes the timing characteristics and scheduling requirements. It specifies when the activities described in the functional view must be initiated or

completed. It also captures the scheduling characteristics (periodic or demand) of the system's required behavior. These timing requirements are defined in terms of the externally visible behavior of the system. For example, the deadline for a controlled variable function is defined in terms of the time from the occurrence of the external event to which the software responds by setting the value of the controlled variable.

Scheduling requirements are classified as periodic or demand. Where a controlled variable has a periodic scheduling requirement, its value must be set at regular fixed time intervals; i.e., the initiating event for setting its value is the passing of a certain amount of clock time. Where a controlled variable has a demand scheduling requirement, its value must be set upon the occurrence of a sporadic event (e.g., button pressed, mode changed, etc.).

4.2.1 PERIODIC SCHEDULING

A controlled variable is considered periodic if it must be set or updated at fixed, regular real-time intervals. For example, a value that must be supplied as part of a feedback control loop every 200 milliseconds has a periodic scheduling constraint.

To express the timing characteristics and scheduling requirements of controlled variables, you must define the following parameters:

- **Period.** The period parameter (P) specifies a constant time interval at which the controlled variable value is set or updated. This represents the time between two consecutive periodic cycles for setting the controlled variable.
- **Initiation Delay.** The initiation delay parameter (I) specifies the length of the time between the start of any periodic cycle and the earliest time the system is allowed to update the controlled variable. Only after this period is elapsed may the software set the controlled variable.

This is an optional parameter with a default value of zero. Where the initiation delay is zero, the controlled variable may be set immediately.

- **Completion Deadline.** The completion deadline parameter (d) specifies the time by which the controlled variable must be set or updated during any periodic cycle.

This is an optional parameter with a default deadline at the end of the period. Where the deadline is earlier, you must specify the deadline parameter.

- **Initiation and Termination Events.** Some periodic controlled variables are set only under certain conditions, e.g., when the system enters or is in a particular mode. Use the initiation and termination parameters to specify the events that signal when the controlled variable must be periodically updated. Where the initiation and termination events are specified, the requirement is that the controlled variable must be updated periodically at fixed time intervals during the period between the arrivals of the initiating and terminating events, respectively. For a controlled variable process that runs continuously, you only provide the initiating event (e.g., system initialization).

The timing characteristics of a periodic controlled variable may vary as a function of the mode, in which case its scheduling parameters will also be functions of the mode. Usually, these parameters are constant quantities.

Figure 4-4 depicts the relationship between the periodic scheduling parameters. Note that it must be the case that $I \leq d \leq P$.

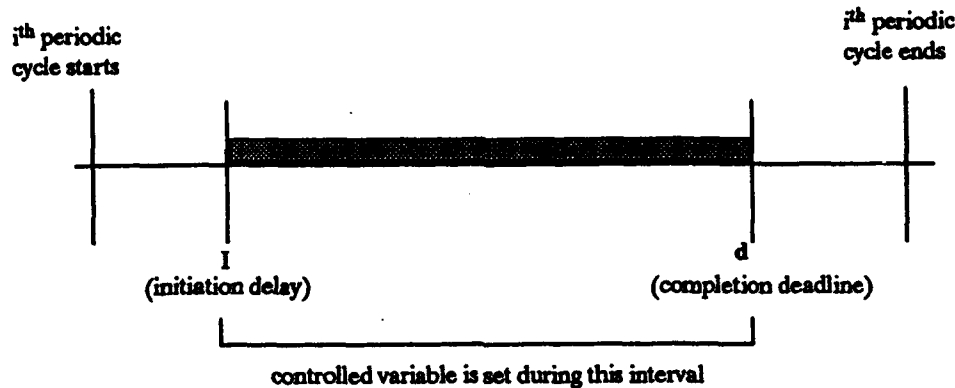


Figure 4-4. Time Line for Periodic Controlled Variable Process

4.2.2 DEMAND SCHEDULING

A controlled variable is considered demand if it must be set in response to sporadic events. For example, a controlled variable that must be set when a button is pressed or a mode change occurs has a demand scheduling constraint.

To express the timing characteristics and scheduling requirements of demand REQ relations, you specify the following parameters if applicable:

- **Initiation Delay.** The initiation delay parameter (I) specifies the length of the time between the detection of the initiating event and the earliest time the software is allowed to update the controlled variable. Only after this period is elapsed may the software set the controlled variable.

This is an optional parameter with a default value of zero. Where the initiation delay is zero, the controlled variable process can be set immediately upon detecting the initiating event.

- **Completion Deadline.** The completion deadline parameter (d) specifies the time, measured from the occurrence of an initiating event, by which the controlled variable must be set. It represents the maximum acceptable time delay between the detection of an initiating event and generation of the required system response (i.e., the worst-case response time).

The ability of the software to respond to initiating events depends on how closely together such events can occur in time. Where events can occur closely enough for this to be an issue, the NAT relation should specify the minimum interval between events. The specification of the controlled variable must discuss the acceptable tolerance in behavior (i.e., if events can be missed).

Figure 4-5 depicts the relationship among the demand scheduling parameters.

Occasionally, the timing characteristics of a demand controlled variable may vary as a function of the mode, in which case its scheduling parameters will also be functions of the mode. Usually, the scheduling parameters are constant quantities.

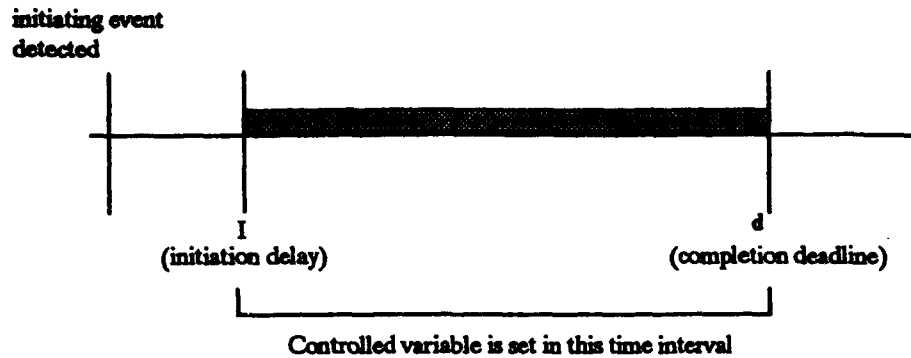


Figure 4-5. Time Line for Demand Controlled Variable Process

4.3 SPECIFYING REQ, NAT, AND UNDESIREED EVENTS

This section gives an overview of how the components discussed in this section are combined to specify the behavior of a controlled variable.

4.3.1 SPECIFYING CONTROLLED VARIABLE BEHAVIOR

Table 4-9 shows the template for specifying a controlled variable with periodic scheduling constraints. Table 4-10 shows the template for specifying a controlled variable with demand scheduling constraints.

Table 4-9. Template for a Controlled Variable with Periodic Scheduling Constraints

Section Title	Brief Description
Controlled Variable	Name of the controlled variable
Initial Value	Expression or table giving the initial value of the controlled variable
Mode Class	Names of the mode classes in the domain of the controlled variable
Sustaining Conditions	Conditions that must hold for the value function to be defined
Value Function	Function specifying the ideal values of the controlled variable
Tolerance	Expression or table giving the allowed deviation from the ideal values defined by the value function
NAT Constraints	Description of any NAT constraints on the controlled variable behavior
Period	Expression giving the period of the controlled variable function
Initiation Delay	Expression giving the allowed initiation delay
Completion Deadline	Expression giving the allowed deadline (worst-case response time)
Initiation and Termination	Event expressions or table giving the initiating and terminating events for the periodic function
Comments	Additional comments or descriptive material

For each controlled variable, you must define a complete mapping from the possible states of the monitored variables to the possible states of the controlled variable. You must also define the timing

and scheduling constraints. The templates in Tables 4-9 and 4-10 show all of the components of such a specification for controlled variables with periodic and demand timing constraints, respectively.

For both types of variables, you must define the initial value and identify the modes for which the variable is a function. The ideal value of a periodic variable is defined using a conditional expression if simple enough or as a condition table if the variable is a function of modes. A demand variable is defined using an event expression or an event table if it is a function of the modes. The sustaining conditions identify any conditions that must be true for the function to be evaluated; i.e., it identifies any conditions that must be true for the function to be defined and the conditions to be evaluated.

You define the range of acceptable values by defining the tolerance. The tolerance is specified only for numeric values. You use a constant, expression, or condition table to define the acceptable range of values.

To specify any relevant parts of NAT, you use the same components used to specify parts of the REQ relation. The remaining parameters are used to specify the scheduling constraints and tolerance in time. Here, as well, you may use condition tables, event tables, expressions, and so on to describe the required behavior. For example, if the initiating and terminating events of a periodic variable depend on the mode, use an event table to identify the events that initiate or terminate the periodic behavior in each mode.

Table 4-10. Template for a Controlled Variable With Demand Scheduling Constraints

Section Title	Brief Description
Controlled Variable	Name of the controlled variable
Initial Value	Expression or table giving the initial value of the controlled variable
Mode Class	Names of the mode classes in the domain of the controlled variable
Sustaining Conditions	Conditions that must hold for the value function to be defined
Value Function	Function specifying the ideal values of the controlled variable; typically given as an event table
Tolerance	Expression or table giving the allowed deviation from the ideal values defined by the value function
NAT Constraints	Description of any NAT constraints on the controlled variable behavior
Initiation Delay	Expression giving the allowed initiation delay
Completion Deadline	Expression giving the allowed deadline (worst-case response time)
Comments	Additional comments or descriptive material

4.3.2 SPECIFYING NAT RELATIONS

Use the NAT relation to derive the types and ranges of the monitored variables and the bounds on the REQ relation. The NAT relation defines all the values that the monitored and controlled variables can assume as well as any constraints imposed on them by the physical world; e.g., it may not be possible to close a valve (the controlled quantity) at certain tank pressures (the monitored quantity).

As you identify and define each of the monitored and controlled variables, you will specify its type, precision, minimum and maximum values, maximum rate of change, and so on depending on how the quantity will be used. These values are typically bounded due to environmental constraints that are totally outside the control of the software. For example, if the system must monitor an aircraft's altitude, the aircraft's operational ceiling determines its maximum value, the earth's surface determines its least value, and so on. These environmental constraints determine the useful bounds on the environmental quantities; the software may require less. For example, there are altitudes that an aircraft will never reach.

Use the NAT relation to specify the bounding assumptions on the monitored and controlled variables. You then use this information to define the ranges over which the software must monitor the quantities it tracks or must affect those it controls. Thus, NAT is typically used to derive the characteristics of the monitored and controlled variables, e.g., the type, maximum and minimum values, or maximum rate of change.

NAT plays a similar role in the definition of the REQ relation when the function describing REQ must cover all the possible values of the monitored and controlled variables. Use the NAT relation to capture any constraints on the possible relationships between a controlled variable and the monitored variables that determine its values. For example, when the possible relationships between the monitored and controlled variables are constrained, e.g., when there are some values of the controlled variable that cannot occur for certain monitored values, this must be captured in the NAT relation. This allows the reader to determine that the REQ relation covers all the possible values of the monitored and controlled quantities (i.e., it is complete).

You use the same basic components, i.e., mathematical expressions, conditions, events, condition tables and so on, to specify NAT as you do to specify REQ. You may also use illustrations or even prose if the constraints cannot be captured more rigorously.

4.3.3 SPECIFYING REQUIRED RESPONSES TO UNDESIRED EVENTS

Over its operational life, any real-time embedded system will experience undesired events, the failure of components of the system or of the system itself. Sensors fail to provide inputs or are used in operating conditions that affect the accuracy of the inputs they provide. Software databases become corrupted, leading them to provide inconsistent data. Actuators fail to respond to commands or their performance degrades. The communication lines connecting devices fail. These failures may be temporary or permanent.

When specifying a system, engineers must consider and account for undesired events that the system is likely to encounter. The requirements specification must describe the required behavior of the software when it encounters these undesired events.

CoRE treats the response to undesired events just as it treats other requirements; i.e., the software's response to undesired events must be specified by the REQ. However, you will occasionally create additional monitored variables specifically to denote failure conditions; in particular, you may create them to model the inability of the software to get the value of a monitored variable or set a controlled one.

These additional monitored variables are used in the REQ relation to specify required system behavior on the occurrence of undesired events, e.g., to trigger mode changes or determine the value

of a controlled variable. These monitored variables abstract from the system components, i.e., they do not unnecessarily assume or preclude particular system components or a particular system design. They are defined in terms of the inability of the system to measure a monitored variable or measure it to the desired precision or in terms of the inability of the system to affect the controlled variable or to affect it to the desired tolerance.

5. REPRESENTING THE CoRE CLASS MODEL

This section describes the underlying concepts, syntax, and semantics for representing the CoRE class model. The class model provides the set of facilities for packaging the behavioral model as a set of classes and dependencies. You will use the concepts and notations described in this section to develop and document a class structure for your software requirements. A class structure represents the set of classes and relationships for a given CoRE specification. The process, guidelines, and heuristics for applying the class notation to package the behavioral requirements are described in Section 9.

The goal in creating a class structure is to package the behavioral model to facilitate ease of change, create reusable requirements, and develop parts of the software in parallel. To achieve these goals, you use the concepts of abstraction and encapsulation in decomposing the requirements into a class structure. You decompose the behavioral model into a set of classes, each of which may be refined independently. The process of decomposing requirements allows you to analyze and understand large, complex systems. With CoRE, decomposition is based not on algorithmic decomposition (e.g., steps taken to process an input) but rather on the behavioral model (e.g., monitored and controlled quantities, software modes, and terms).

The class model contains classes with well-defined interfaces and dependencies on these interfaces. The class model provides the mechanisms to:

- Manage requirements changes (via abstraction, encapsulation, and inheritance).
- Create reusable requirements (via abstraction, encapsulation, and inheritance).
- Develop parts of the software system in parallel (via classes and their dependencies).

The CoRE class model provides the mechanisms to package the behavioral model as a set of classes, objects, and dependencies. The sections that follow tell you :

- The mechanism for gaining a preliminary understanding of the software and its environment (recorded in an information model)
- What information to record about each class (class definitions, interfaces, encapsulated information)
- The diagramming notation for showing a class structure (context diagram and dependency graphs)
- The information and diagramming notation for each component of a class definition

5.1 INFORMATION MODEL

The CoRE information model serves a different purpose than a traditional data model (e.g., those used to specify complex data requirements as outlined by Chen [1976]) in that you use the information

model to identify, collect, and organize the information you will need for subsequent CoRE activities. You use an information model to obtain a preliminary understanding of the software's environment. This includes entities whose behavior can affect the software behavior, characteristics of these entities (physical quantities that the software monitors and controls), and the relationships among instances of these entities.

You can use the information model in building both the CoRE behavioral model and class model. First, the physical quantities that you identify help in determining the environmental variables and the associations that must be maintained by the REQ or NAT relation in the behavioral model. Second, identifying entities and capturing similarities among entities aid in identifying classes and the inheritance relation in the class model.

An entity is a representation of any aspect of the system environment that is of interest to the system. Entities in the information model describe physical things, roles played by persons or organizations, incidents, and interactions that are significant to the software. For each entity, you record the information found in Table 5-1.

Table 5-1. Entity Template

Section Title	Brief Description
Entity Name	The name of the entity
Entity Description	Brief prose description of what the entity represents
Attributes	List of attributes that characterize the entity
Instances	List of number of instances that you have identified

An attribute characterizes some important aspect or fact about an entity. Consider physical quantities that may be relevant to the software. The software may monitor or control these quantities, or their values may influence quantities that the software monitors or controls. Provide a definition for each attribute that describes the association between the physical quantity and the attribute that denotes it. Figure 5-1 shows an example of the entity Pump and its attributes from the FLMS.

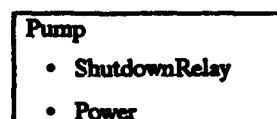


Figure 5-1. Pump Entity and Attributes Example

You impose organization on the entities by creating relationships between entities. The relationships you create are the generalization/specialization (see Section 5.1.1), aggregation (see Section 5.1.2), and application-specific (see Section 5.1.3) relationships.

There are a variety of notations you can use to represent a CoRE information model, e.g., an entity-relationship diagram (ERD) (see Figure 5-2), a text-based list of candidate environmental variables, or an attribute matrix (see Table 5-2).

- **ERD:** Use the ERD to create a graphic representation of entities and relations. The ERD uses rectangles for entities and diamonds for relationships with lines connecting related entities. Attributes of entities are shown as text within the entity symbols.

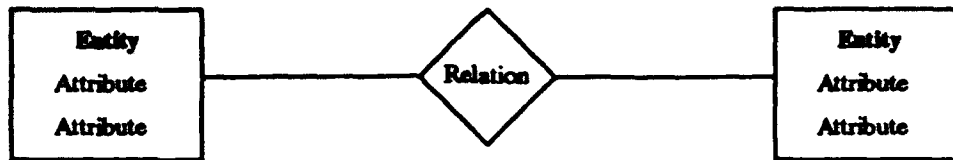


Figure 5-2. Entity-Relationship Diagram Notation

- **Attribute Matrix:** You can represent relations between entities using a table that maps attributes to attributes. Place the attribute names in the first column and first row. Place an X in the intersection for pairs of attributes in which the value of one determines, prescribes, or constrains the value of the other.

Table 5-2. Partial Attribute Matrix for Fuel Level Monitoring System

	ShutdownRelay	Power	..	LevelDisplay
ShutdownRelay		X		
Power				
...				
LevelDisplay				

5.1.1 GENERALIZATION/SPECIALIZATION RELATIONSHIP

The generalization/specialization relation recognizes that an entity may be related to a set of entities by being a generalization of the entities in the set. Attributes that characterize the more general entity also characterize the members of the set. The members of the set may specialize the general entity by adding or constraining attributes and may inherit its attributes.

Use the generalization/specialization relation to record which entities inherit similar characteristics. Recording this information will help you in subsequent class structuring activities to identify common requirements and help you understand which changes are likely to occur together.

Represent the generalization/specialization relationship with an *is_a* relationship in an ERD. Capture those attributes that are common among a set of entities with the *is_a* relationship and associate those attributes with the generalization entity. An entity can have, at most, one generalization parent entity. For example, consider an air traffic control system that tracks potential target aircraft in which host and target aircraft are specializations of the aircraft entity (see Figure 5-3). The attribute Location applies to both the Host and Target Aircraft entities; therefore, it is associated with the Aircraft entity (generalization entity). However, the speed of the target aircraft (attribute Relative Speed) relative to the host aircraft only applies to the Target Aircraft entity attribute; therefore, it is associated with the Target Aircraft entity only.

5.1.2 AGGREGATION

The aggregation relation indicates that one or more entities are part of another entity. This relationship allows you to represent entities as components of an entire assembly.

Use the aggregation relationship to help in understanding the software's environment and also to provide structure to the information model. There may be information or relationships that the software must maintain that belong to the aggregate but not to any of the parts individually.

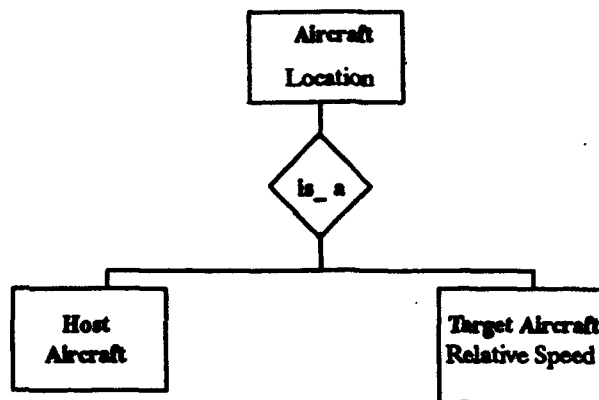


Figure 5-3. Generalization/Specialization Entity-Relationship Diagram Notation

Represent the aggregation relationship with an is_part_of relation. For example, an operator console that contains an alarm and display may need to ensure that, when an alarm is sounding, the display must reflect the condition that the alarm is signaling (see Figure 5-4).

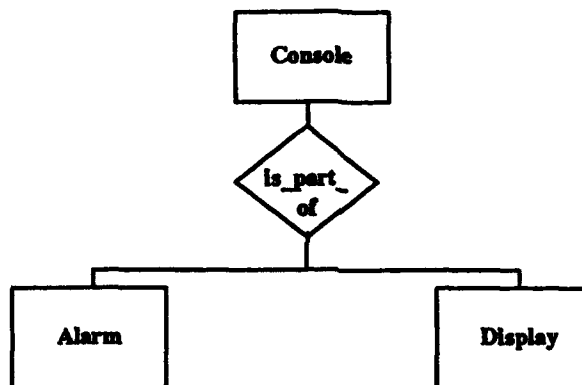


Figure 5-4. Aggregation Entity-Relationship Diagram Notation

5.1.3 APPLICATION-SPECIFIC RELATIONSHIP

An application-specific relationship represents a logical association between entities. A logical association can be a physical or conceptual connection between entities. For example, a target aircraft flies relative to a host aircraft. "Flies relative to" is the logical association between the two entities.

You use application-specific relationships to capture an early understanding of what information you will need to write the REQ and NAT relations. The attributes of one entity may be needed to set the attributes of another entity. These entities may be connected via a logical association.

You name the application-specific relationship using the terminology consistent with the software application. You can represent the application-specific relationships using an ERD (see Figure 5-5) or an attribute matrix (see Table 5-2). These associations should be named with a verb phrase.

5.2 CLASS DEFINITIONS

The CoRE class model provides a set of facilities for packaging the behavioral model into a structure consisting of classes and their dependencies and the inheritance relation. A class serves as a template

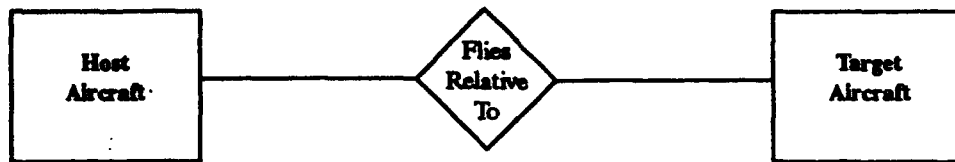


Figure 5-5. Application-Specific Relationship Notation

for an object or a set of related objects. A class defines a set of requirements or terms common to one or more objects and provides facilities for abstraction and encapsulation. An object is an instance of a class that specifies a subset of the definition of REQ, NAT, IN, and OUT relations for a given specification. A CoRE specification is written in terms of classes (not objects) to keep the specification concise and to avoid redundancy.

To define a class, you need to identify its name, description, class interface, encapsulated information, the name and number of objects defined by each class, and traceability information (see Table 5-3). The definition partitions the information into what can be used by other classes (the class interface) and what cannot be used by other classes (encapsulated information). How you represent each of these characteristics is discussed in later sections.

To complete a class structure, you must also define and represent the relationships among classes. These relationships are the encapsulates relation, depends-on relation, and the inheritance relation. Section 2.4 showed that:

- The encapsulates relation allows you to define a class in terms of other classes.
- The depends-on relation denotes exactly which classes use information provided by other classes.
- The inheritance relation denotes a class containing common properties to be shared among a set of classes.

The sections that follow show you how to represent a class structure graphically, including the set of classes and the relationships among them.

Table 5-3. Class Template

Section Title	Brief Description
Class Name	Use a descriptive name (e.g., one recognizable by the customer) to communicate the class abstraction preceded with "class_" or "superclass_" (e.g., class_Alarm).
Class Description	Brief prose description of what the class encapsulates and the abstraction it provides.
Class Interface	Information that can be used by other classes in their definitions. 1. Names and definitions of monitored or controlled variables that the class defines. Begin the monitored variable name with "mon_" (e.g., mon_Fuel_Level). 2. Names and definitions of terms that the class defines. Use a descriptive name preceded with "term_" (e.g., term_Inside_Hys_Range). 3. Provide names and definitions of any events that the class defines. Use a descriptive name preceded with "event_" (e.g., event_Reset). 4. Provide names and definitions of any modes that the class defines. Use a descriptive name preceded with "mode_" (e.g., mode_Operating).

Table 5-3, continued

Section Title	Brief Description
Encapsulated Information	Information that cannot be used by any other class: - Names and definitions of monitored variables. - Names and definitions of controlled variables. - Names and definitions of events. - Names and definitions of terms. - Names and definitions of input and output variables. - Definitions of REQ relations. - Definitions of NAT relations. - Definitions of IN relations. - Definitions of OUT relations. — or — - Names and definitions of classes and their dependencies if not a leaf class.
Objects Defined	List the name of each object derived from this class (e.g., Engine 1, Engine 2, Engine 3, and Engine 4 would be objects derived from class <u>Engine</u>). Objects are derived only from classes at the leaves of the hierarchy, so this section can be empty if the class is a higher level class.
Subclasses Defined	If this template describes a superclass, list the subclasses derived from the superclass here.
Traceability	Specify which requirements are defined by the class. This section is filled only for leaf classes.

5.3 DIAGRAMING CONVENTIONS

Figure 5-6 illustrates the diagraming notation you use to show a class structure. The following sections describe how the graphic elements are used to illustrate the relationship between the software and its environment and the dependency relationships among classes.

5.3.1 CONTEXT DIAGRAM

The CoRE context diagram illustrates the boundaries between the software and the external environment. Specifically, the CoRE context diagram shows the monitored and controlled variables in and out of the software system unlike the information model, which implies no directional information. The diagram identifies the entities and the environmental quantities needed by the software.

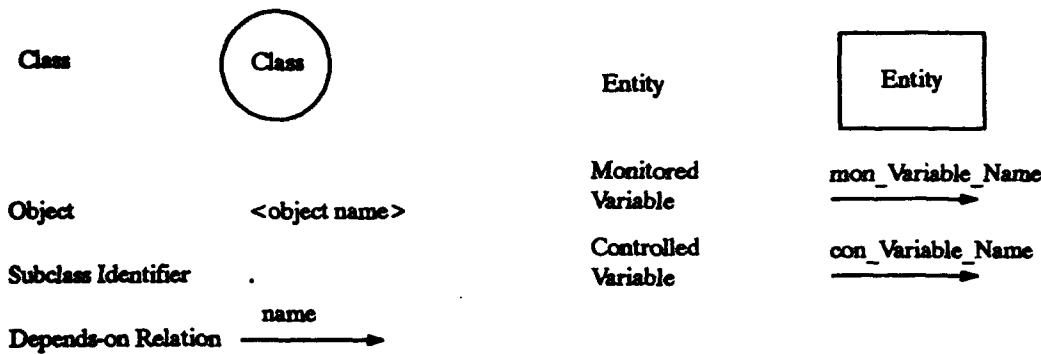


Figure 5-6. Class Structuring Notation

The context diagram (see Figure 5-7) contains a circle that represents the CoRE specification as a whole. You further decompose the software system class. Each rectangle represents an entity from the information model. An arrow connecting a rectangle to the circle represents an environmental variable. Label the arrow with the name of the environmental variable that it represents (e.g., `mon_Variable_Name`, `con_Variable_Name`).



Figure 5-7. Representation of an Overview of a Specification Using a Context Diagram

Arrows representing monitored variables can originate only from an entity on the context diagram. Likewise, the arrows for controlled variables can terminate only at an entity on the context diagram. An arrow from a rectangle to the circle represents a monitored variable, and an arrow from the circle to the rectangle represents a controlled variable. To reduce the number of arrows contained in a context diagram, you can aggregate monitored variables originating from the same entity into a single arrow. Similarly, you can aggregate controlled variables.

5.3.2 DEPENDENCY GRAPH

The depends-on relation denotes which classes use what information provided by other class interfaces. Formally, the depends-on relation is defined as:

`class_X` uses `T`, where `T` can be a monitored variable, term, mode, or event, provided by `class_Y` only if `class_X` employs `T` in its definition.

You show these relationships in a dependency graph. An arrow between two classes represents an interface provided by the class at the tail of the arrow, which is needed to define the class at the head of the arrow. Label each arrow with the name of the environmental variable, event, or term that it represents (see Figure 5-8).

In establishing the dependencies, it is important to remember that a dependency shows only that the definition of a term defined by one class is used by another. It implies nothing about how the data may appear in an implementation or how it actually moves from one place to another. In particular, it does not imply that there is some form of “request” for the data followed by some form of “send.” CoRE classes are not operational and do not perform actions like requesting and sending.

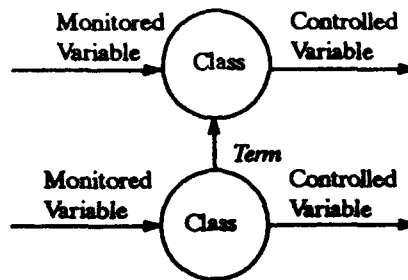


Figure 5-8. Dependency Graph Notation

5.3.3 LEVELING

You represent the encapsulates and depends-on relations as a set of leveled diagrams. The root of the hierarchy is the CoRE context diagram. Leveling allows you to decompose a class into more detailed classes (encapsulates structure), providing a means to manage complexity in the detailed requirements. You could represent the leaves of the hierarchy as a single diagram; however, your ability to understand the specification decreases due to the amount of information shown. The higher level diagrams are abstractions of the lower level diagrams (see Figure 5-9).

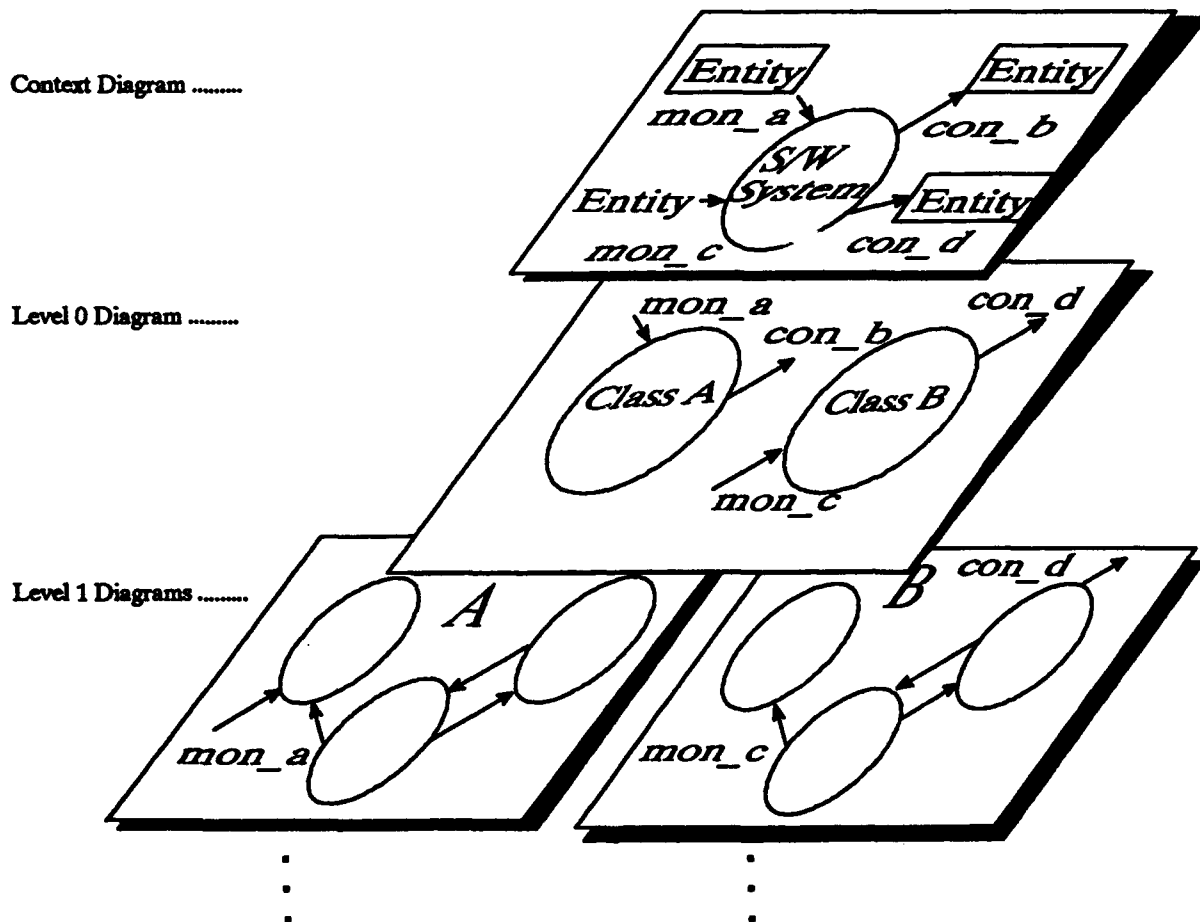


Figure 5-9. Class Structuring Leveling Diagrams

You also show the dependencies among classes at each level of the encapsulation structure. Notationally, this means that dependencies into and out of a class on a parent diagram must be equivalent to the dependencies into and out of a child diagram. This is referred to as balancing. In other words, all dependencies into a child diagram must be drawn coming into the parent class. Similarly, all dependencies out of a child diagram must be drawn going out of the parent class (see Figure 5-9).

The leveled set of diagrams represents the decomposition of the software system class on the context diagram. All arrows representing the monitored variables come into the Level 0 diagram. Similarly, all arrows representing the controlled variables exit the Level 0 diagram (see Figure 5-9).

5.4 CLASS SPECIFICATION

Table 5-3 defined the components of an individual class. This section expands on the definition of each component as well as shows you how to graphically represent them. The definition partitions the information into what can be used by other classes (the class interface) and what cannot be used by other classes (encapsulated information). The inheritance relation is also discussed.

5.4.1 CLASS INTERFACE

A class interface defines information (e.g., monitored variables or expressions of monitored variables) that can be used in the definition of other classes. A class interface is an abstraction in the sense that it hides extraneous details and shows only essential aspects of the information defined by the class. Equivalently, there is a one-to-many relationship between the interface and the possible internal structures. For a CoRE class, the abstract interface comprises the information defined by the class that other classes can use.

In developing the class interface, you must seek a balance between sometimes conflicting goals: providing a useful interface, preserving the encapsulated information of the class, and maintaining ease of use:

- ***Provide a useful interface.*** Your first objective is to provide sufficient information about each monitored variable so that you can fully specify the REQ relations that depend on those variables.
- ***Preserve the encapsulated information.*** The class encapsulated information characterizes information that other classes should not depend on. You must define an interface that does not provide information that is part of the class encapsulated information.
- ***Maintain ease of use.*** In developing the class interface, you create information to be used in other parts of the requirements. Your objective is to define the interface terms so their meaning is clear, precise, as simple as possible, and unambiguous.

The actual information provided on a given interface depends on the information defined in the class, what information is needed by the classes defining the REQ relations, and the packaging goals. However, CoRE restricts the interface specification to contain only monitored variables, modes, terms, and events (expressions of the monitored variables).

To illustrate a class interface, label an arrow from the class with the name of the environmental variable or term that is defined by the interface. For example, consider Figure 5-10: `mon_A` is on the

interface of class_B, mon_D and term_C are on the interface of class_G, and class_H has no interface. The controlled variables and the REQ relations are defined in the encapsulated information for the classes (see Section 5.4.2).

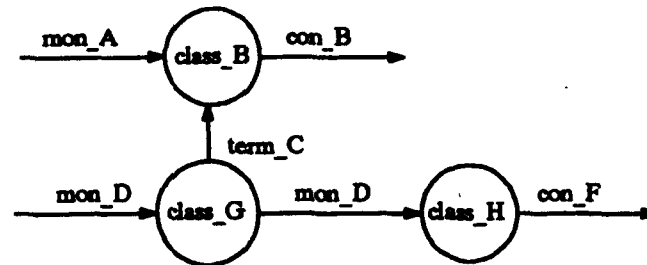


Figure 5-10. Class Interface Notation

The arrows between classes represent depends-on relations. The class at the head of the arrow uses the definition that appears on the interface of the class at the tail of the arrow. Arrows representing monitored variables originating from an entity on the context diagram are defined by the class at the head of the arrow. Arrows representing controlled variables terminating at an entity on the context diagram are defined by the class at the tail of the arrow. In Figure 5-10, class_B defines mon_A and con_B, class_G defines mon_D, and class_H defines con_F.

5.4.2 CLASS ENCAPSULATED INFORMATION

The encapsulated information for a class is information that cannot be used by other classes. The encapsulated information for a given class must either decompose into a more detailed class structure or provide definitions of part of the behavioral model. Thus, the encapsulation structure is a hierarchy of classes and subclasses, the leaves of which contain only definitions of environmental variables, input and output variables, and REQ, NAT, IN, and OUT relations. Because traceability information is tied to the behavioral model (e.g., environmental variables, REQ, NAT, IN, OUT), you trace requirements only in the leaf classes.

When a class becomes too complex (i.e., its definition becomes unmanageable) or parts are likely to change independently, additional packaging may be useful. You provide this packaging by defining a class in terms of other classes, i.e., by encapsulating the definition of a class in a parent class. This relationship induces a hierarchy on the set of classes by using leveling. This is constrained so that all requirements are defined by the classes at the leaves of the hierarchy (e.g., the encapsulated information would be written in terms of monitored variables, controlled variables, REQ, NAT, IN, or OUT). Classes above the leaves may export terms defined by their encapsulated classes, and they may define additional terms that are functions of the terms or variables defined by their encapsulated classes.

Figure 5-11 illustrates the notation for the encapsulated information. The class class_B encapsulates the definition of class_I and class_J in its encapsulated information. The interface of class_I defines term_X in its interface, and class_J encapsulates the definition of the controlled variable con_B and the REQ relation for con_B. The classes class_I and class_J are the leaves of the hierarchy for class_B. Traceability information would be found in class_I (associated with IN relation for the mon_A) and class_J (associated with REQ relation for con_B).

5.4.3 OBJECTS

An object is an instance of a class that specifies a subset of the definition of REQ, NAT, IN, and OUT relations, including the definitions of the variables and terms, for a given specification. There is no

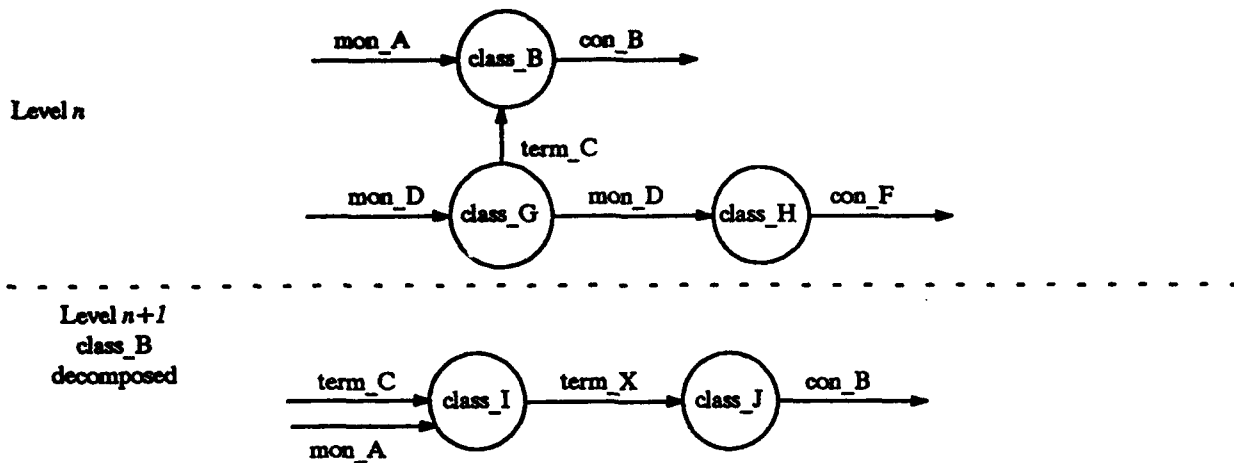


Figure 5-11. Encapsulation Structure Notation

graphical icon for a CoRE object; however, you do need to identify, by name, the objects specified by each class if these objects will be referred to individually. If you need to specify more than one object of a CoRE class, use the following conventions:

- Names of monitored and controlled variables and terms should be suffixed by a pair of angle brackets ($< >$) containing the identifier of the object or a variable ranging over the possible identifiers; e.g., a variable called $<location>$ could denote a range of objects for pilot and copilot.
- Function definitions for classes whose objects use terms or monitored variables from multiple objects of other classes may need to be written in terms of specific objects.

Consider a control system for a four-engine aircraft whose CoRE specification contains the definition of class_Engine with a monitored variable $mon_Engine_Temperature$ and a term $term_Engine_Fail$. The class definition would contain the names of the four objects:

Number of Objects Specified: 4

Object Names:

left_out (left outboard engine)

left_in (left inboard engine)

right_in (right inboard engine)

right_out (right outboard engine)

The definition of the monitored variable $mon_Engine_Temperature$ and the term $term_Engine_Fail$ would be the same for all four objects. Graphically, class_Engine would be depicted by Figure 5-12.

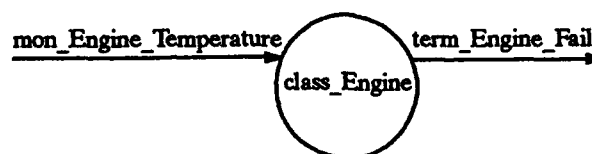


Figure 5-12. class_Engine Diagram

However, suppose that there is a mode machine class definition `modeclass_In_Emergency`, whose single object monitors all four engines and initiates emergency actions as appropriate. If a special action needs to be taken when both left engines fail, the associated condition is defined as:

`term_Engine_Fail<left_out>=True and term_Engine_Fail<left_in>=True`

Graphically, the mode machine class and associated terms can be represented as shown in Figure 5-13.

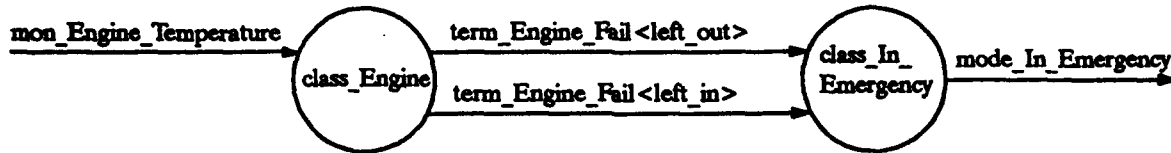


Figure 5-13. class_Engine Diagram With Objects

5.4.4 INHERITANCE

The generalization/specialization structure is used to denote the inheritance relation. Inheritance is a mechanism for specifying in a superclass the common requirements and properties that are shared among a set of subclasses. A superclass defines a set of common properties or requirements for subclasses. Each subclass of the superclass inherits the requirements or terms defined by the superclass. The subclass inherits the interface of its superclass by encapsulating the definition of the superclass (i.e., the superclass definition acts like an encapsulated class).

When defining functions of a subclass where you need to include terms or monitored variables from the superclass, separate the class names with a period (e.g., `mon_Var_Name.subclass_Name`, `term_Name.subclass_Name`).

EXAMPLE: You have defined the following superclass (Table 5-4) and subclasses (Tables 5-5 and 5-6). Graphically, represent the superclass as encapsulated information of a subclass. Figure 5-14 illustrates both the naming convention and the graphical representation for `superclass_A`, `class_B`, and `class_C` (in Figure 5-14, `class_D` and `class_E` represent classes encapsulated by `class_B` and `class_C`, respectively). The variable `mon_A.class_B` represents the `class_B` monitored variable `mon_A` (defined by `superclass_A`).

Table 5-4. superclass_A Template

Class Name	superclass_A
Class Description	Superclass capturing the common requirements for . . .
Class Interface	Defines: mon_A : definition
Encapsulated Information	Encapsulates: con_B : definition
Subclasses Derived	class_B and class_C
Traceability	This class encapsulates the following requirements . . .

Table 5-5. class_B Template

Class Name	class_B
Class Description	Class extending the requirements of superclass_A . . .
Class Interface	Defines: mon_X : definition term_Y : definition
Encapsulated Information	Encapsulates: superclass_A
Objects Derived	List object names
Traceability	This class encapsulates the following requirements . . .

Table 5-6. class_C Template

Class Name	class_C
Class Description	Class constraining the requirements of superclass_A . . .
Class Interface	Defines: term_Z: definition
Encapsulated Information	Encapsulates: superclass_A
Objects Derived	List object names
Traceability	This class encapsulates the following requirements . . .

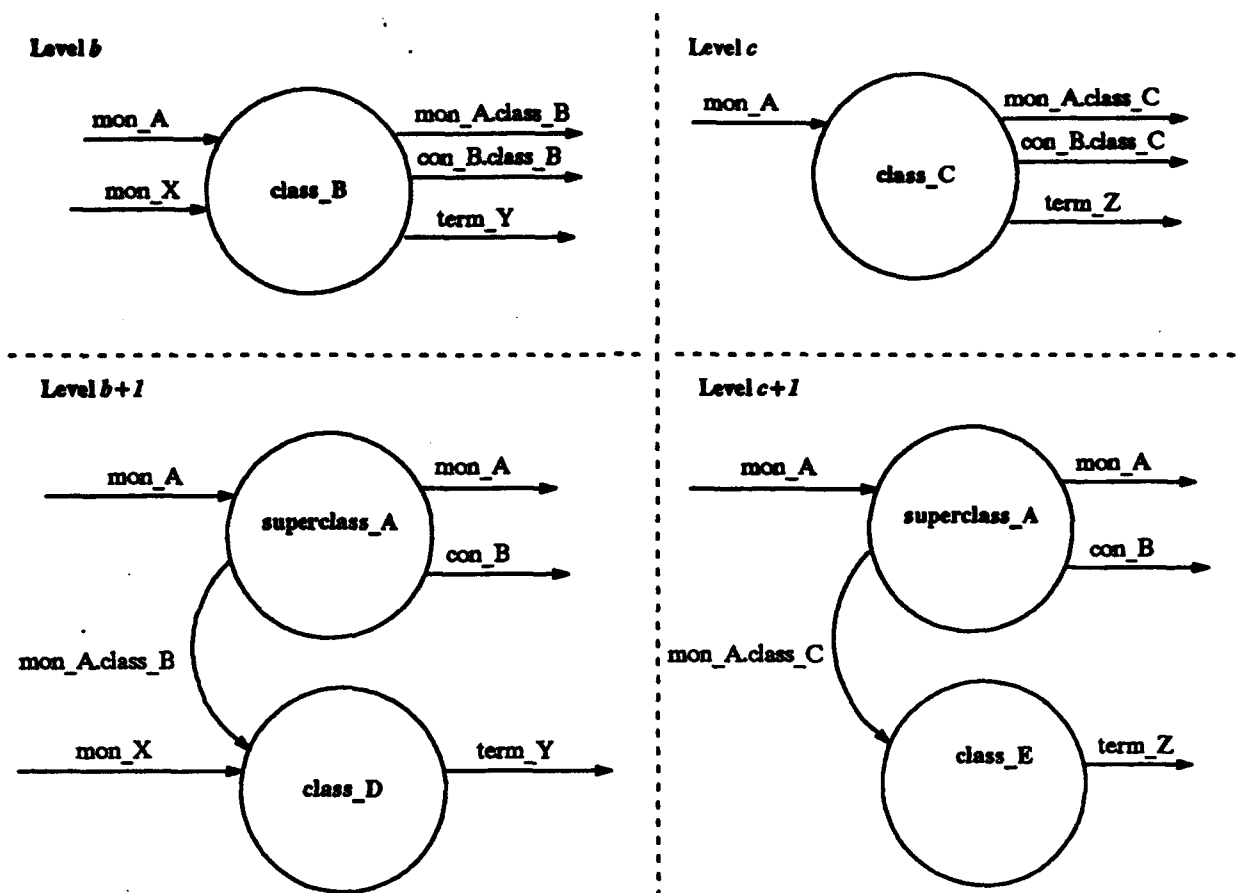


Figure 5-14. Inheritance Notation

5.5 CLASS MODEL NOTATION SUMMARY

This section provides a sample class definition (Table 5-7) and a summary of the notation (see Figure 5-15).

Table 5-7. Class Template Summary

Class Name	Provide class name preceded by "class_"
Class Description	Provide a description of the class.
Class Interface	Defines: List monitored variables, terms, or events.
Encapsulated Information	Encapsulates: List monitored variables, controlled variables, terms, modes, events, IN, OUT, REQ, NAT, and input and output variables or classes.
Objects Derived	List objects derived from this class.
Traceability	Provide traceability information.

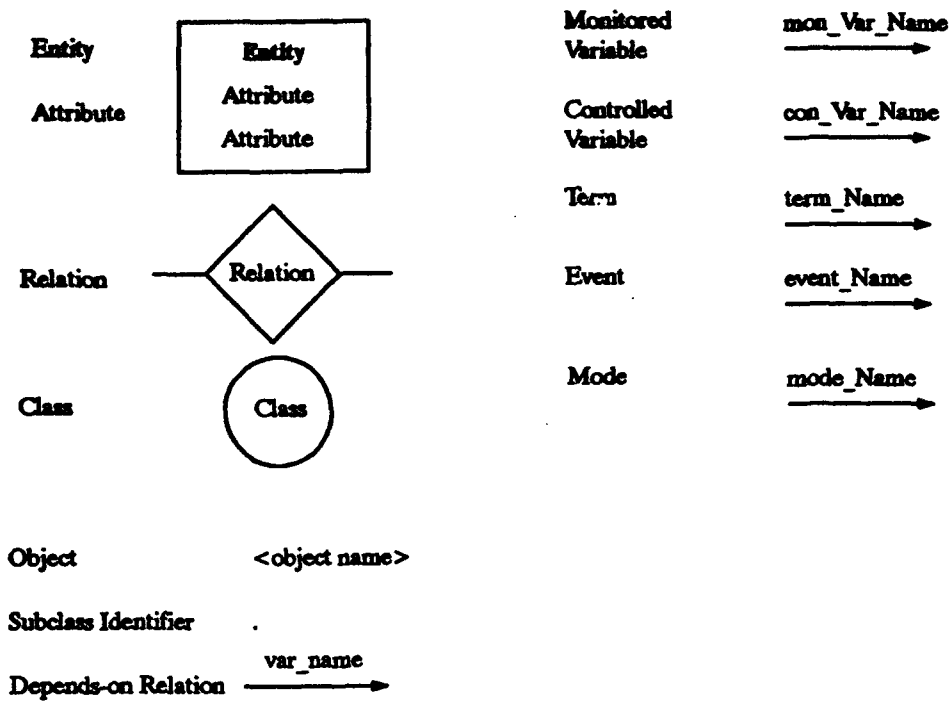


Figure 5-15. Class Model Notation Summary

This page intentionally left blank.

6. CoRE PROCESS OVERVIEW

This section has three purposes. First, it introduces you to the CoRE process. It outlines each of the key process activities and describes what information you need for each activity and what product you produce. After finishing this section, you should understand the overall purpose of each of the activities in CoRE and how they work together to produce a complete software requirements specification.

Second, this section describes the relationship between the idealized CoRE process and CoRE in practice. An idealized process assumes a perfectly rational developer who always makes the right choice in the right order. This ideal process produces a rational progression of products (top down) beginning with the system specification and ending with the software requirements specification. A description of the ideal process is useful because it provides an external standard to guide development and it serves as a yardstick for measuring progress.

An idealized process description tells you where you want to go, but it is not always the best guide on how to get there. The typical development process is iterative: more inside out than top down. The maturity and level of detail for different parts of the specification are uneven, and the captured requirements are constantly changing. In addition to the ideal process, CoRE provides heuristics and guidelines to guide you through the more uncertain stages of actual development. Upon finishing this section, you should understand how CoRE helps produce ideal products from a real process.

Finally, this section serves as a reference for the CoRE process. It provides an overview of each of the CoRE activities, including the inputs to the activity, the goals, and the products produced.

6.1 THE IDEALIZED CoRE PROCESS

The ideal CoRE process is a sequence of activities that begins with a system requirements (i.e., allocation of system requirements to hardware and software components is complete) and ends with a software requirements specification. The sequence of activities is organized around the behavioral and class models. In other words, each step gathers some part of the behavioral requirements in terms of the behavioral model, then captures that information in the class definitions. The process is idealized in that it does not account for errors, requirements changes, unknown requirements, or other factors requiring additional iteration, experimentation, or backtracking. A schematic showing the activities and products of the ideal CoRE process is shown in Figure 6-1.

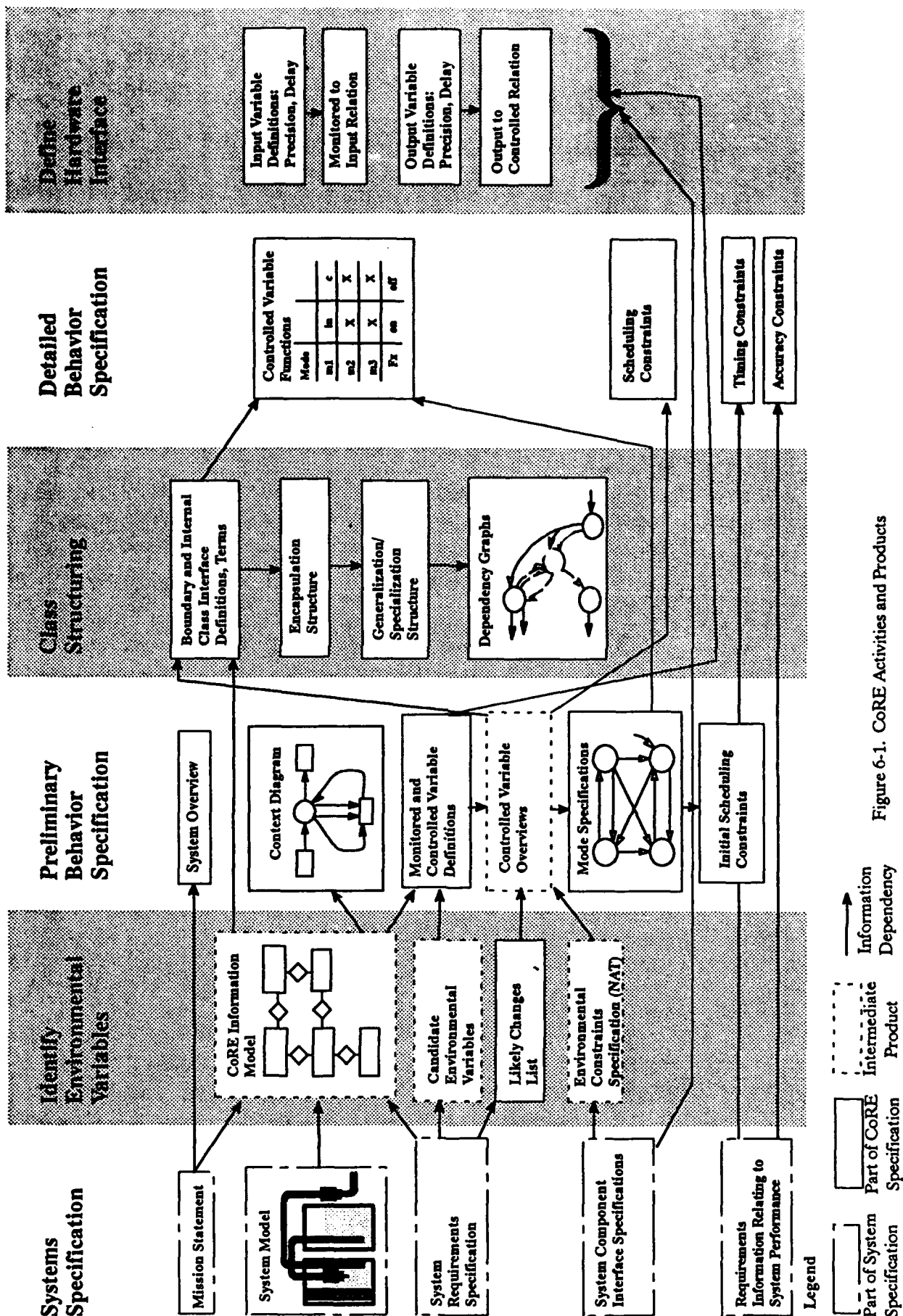


Figure 6-1. CoRE Activities and Products

6.1.1 IDENTIFY ENVIRONMENTAL VARIABLES

Goals

Identify candidate environmental variables and the relations among them. The overall goal is to identify environmental quantities that denote the monitored and controlled variables, relationships that will become parts of the REQ and NAT relations, and relationships that will become part of the generalization/specialization structure. Identify likely changes and their impacts on these environmental variables.

Inputs

- The system requirements
- System interface specifications
- Device interface specifications
- Other sources (including people) that define or constrain the role of the software

Purpose

The purpose of this activity is to identify the environmental quantities with which the software interacts and the constraints among such quantities. You use this information in building both the CoRE behavioral model and class model. First, the physical quantities that you identify help in determining the environmental variables and the associations that must be maintained by the REQ or NAT relation in the behavioral model. Second, identifying entities and capturing similarities among entities aid in identifying classes and the inheritance relation in the class model.

Products

- Information model (ERD, attribute matrix)
- Preliminary NAT relations
- Likely changes list

Decisions

Decisions about required behavior:

- Which entities and relationships described in the system specification and other sources imply software requirements
- Which entities and relationships in the environment represent constraints on the required behavior

Decisions about packaging the software specification:

- Which requirements might change and how likely is the change
- Which physical quantities are related and should be thought of as attributes of the same entity or which entities are inherently related and should be thought of as instances of some higher level entity
- Which requirements are poorly understood or represent high risk

6.1.2 PRELIMINARY BEHAVIOR SPECIFICATION

Goals	Identify and specify the monitored and controlled variables. Identify undesired events to which the software system must respond, and define monitored variables to denote them. Identify the domain and scheduling type for each controlled variable. Identify modes.
Inputs	• Information model • System requirements
Purpose	The overall purpose of this activity is to identify enough of the elements of the behavioral model to make appropriate packaging decisions in the subsequent activity. In this activity, you determine and capture the relationship between the software and its environment using the CoRE behavioral model. You decide which environmental quantities are monitored, controlled, or both. You then denote the quantities by monitored and controlled variables. You determine what information about the monitored variables and modes you will need to write each of the controlled variable functions. You decide how many mode machines are needed and the modes and possible transitions for each.
Products	• Monitored and controlled variable definitions • System context diagram • Partial REQ definition for each controlled variable • Initial mode machine definitions
Decisions	Decisions about required behavior: • Which quantities you consider monitored, controlled, or both • Which undesired events the software system must respond to • Which monitored variables, modes, or other information is needed to write the controlled variable functions • Whether each controlled variable has a periodic or demand scheduling requirement • How many mode machines are needed, the modes in each, and the allowed transitions

6.1.3 CLASS STRUCTURING

Goals	Create a class structure to address your packaging goals. Decide how the parts of the behavioral model will be allocated among CoRE classes. Create
--------------	---

boundary, mode, and term classes based on your packaging goals. Define the class interfaces and identify class dependencies.

Inputs

- Monitored and controlled variable definitions
- CoRE information model
- Partial REQ definition for each controlled variable
- Likely changes list
- Initial mode machine definitions

Purpose

In Class Structuring, you make the packaging decisions for your specification. You define a set of classes that will contain all the elements of the CoRE behavioral model. You determine which classes will be further decomposed into additional classes. You determine which common parts of the specification will be represented using superclasses and subclasses. You define the interface for each class applying the principles of abstraction and information hiding. You define interface terms to provide the information necessary to define the modes and controlled variable functions.

Products

- Boundary, mode, and term class interface definitions, including the encapsulation structure and generalization/specialization structure
- Term definitions
- Dependency graphs

Decisions

Decisions about required behavior:

- Which terms are needed
- What mode information is needed
- Which classes depend on what information defined on the interfaces of other classes.

Decisions about packaging the specification:

- Which boundary classes are required and how monitored or controlled variables are allocated to the boundary classes
- Which term classes are needed

6.1.4 DETAILED BEHAVIOR SPECIFICATION

Goals

Complete the class definitions by completing the specification of the controlled variable functions and timing constraints for each controlled variable. Refine the class structure to be consistent with the behavioral model needs.

Inputs

- Preliminary NAT relations
- Monitored and controlled variable definitions
- Class interface specifications, including modes and terms
- Controlled variable overviews
- System requirements

Purpose

In this activity, you complete the specification of the required behavior of the software by completing the definition functions and timing constraints for each controlled variable. You capture the behavioral requirements that the software must meet by defining the values each controlled variable is allowed to assume for each possible state of the system. You specify any timing and accuracy constraints on the variables, including period and deadlines.

Products

Complete controlled variable and mode machine specifications, including:

- Controlled variable function definitions
- Accuracy constraints
- Timing constraints
- NAT relations
- Detailed mode machine definitions
- Refined dependency graphs

Decisions

Decisions about required behavior. For each controlled variable:

- For which modes is the value of the controlled variable defined
- What determines the required value in each mode
- What is the tolerance in accuracy for each possible mode
- What are the detailed timing constraints governing when the controlled variable must be set

6.1.5 DEFINE HARDWARE INTERFACE

Goals

Define the software system inputs and outputs. Define the IN and OUT relations.

Inputs

- Boundary class definitions
- Device interface specifications

Purpose	Identify how the software system inputs and outputs can be used to establish the value of monitored variables and how to set the values of controlled variables. You complete the definitions of the boundary classes by specifying the input and output variables and defining the IN and OUT relations. Define any additional constraints on the presentation of monitored and controlled quantities (e.g., specify the requirements for user interfaces).
Products	• Input and output variable definitions • IN and OUT relations
Decisions	Decisions about required behavior: • What inputs can the software use to determine the value of each monitored variable; how can the software use the inputs to determine the value (the IN relation) • What outputs can the software use to set the value of each controlled variable; how can the software use the outputs to set the value (the OUT relation) Decisions about packaging the specification: • In which class should each input and output variable, IN, or OUT relation be defined

6.2 CoRE IN PRACTICE

In practice, you are concerned with both developing the requirements specification and managing the development. The ideal CoRE process proceeds systematically from system requirements to the software requirements specification, adding detail in each activity. Every decision must be made individually, and the specification is built up as a sequence of detailed decisions and iterations on those decisions. To provide useful guidance in the day-to-day practice of developing a specification, a method should help answer two basic questions at any given level of detail: "What should I do next?" and "How do I know when I am done (or done enough)?" CoRE helps answer these questions through its behavioral model because, for a given set of controlled variables, the model determines exactly what kinds of information are needed to write a complete CoRE specification.

Specification development also tends to be highly concurrent, asynchronous, and subject to change. Several people typically will be developing the requirements at the same time. Some parts of the requirements will be well understood and can be captured in detail, while other parts are fuzzy or highly likely to change. To help manage development, CoRE provides guidance and facilities for breaking a specification into parts that can be developed or changed independently.

6.2.1 SPECIFYING REQUIRED BEHAVIOR

The behavioral model drives the specification of required behavior. Because the model is standardized, the general set of questions that must be answered to provide a complete specification is known in advance. Further, because the model is known in advance, CoRE provides templates to help capture the details of the specification.

The detailed guidance provided by CoRE is keyed to the individual controlled variables. When you decide what the controlled variables will be, the same basic set of questions must be answered for each one. When those questions have been answered for one variable, its specification is complete. When they have been answered for all of the variables, the entire specification is complete. If any of the answers are unknown, this will be clear from the format and content of the specification. The CoRE templates show explicitly which parts of the specification are incomplete. The following summarizes the sequence you follow in specifying the required behavior of a controlled variable. More detailed guidance is provided in the process section.

1. *Define the controlled variables.* Identify a distinct environmental quantity (e.g., the state of a valve) controlled by the software system. Create a controlled variable denoting the quantity, and define the name of the variable, physical interpretation, type and range of values, and any other relevant NAT constraints, such as maximum rate of change.

Additional Guidance: Controlled variable template (Table 4-2)

2. *Define the monitored variables.* Identify the quantities in the environment that the software system will need to track. For each relevant quantity, create a monitored variable modeling the quantity and define the name of the variable, physical interpretation, type and range of values, and any other relevant NAT constraints, such as maximum rate of change. Figure 6-2 illustrates the results of steps 1 and 2.

Additional Guidance: Monitored variable template (Table 4-2)

Boundary Class Definition—Monitored Side	Mode Class	Boundary Class Definition—Controlled Side
Monitored Variable Definition NameType Interpretation FuelLevel Length Level of Fuel... NAT Constraints: 0.0 < FuelLevel < 30 cm		Controlled Variable Definition NameType Interpretation LowAlarm Alarm Blinking Indicator ... NAT Constraints: None

Figure 6-2. Results of Steps 1 and 2

3. *Define the controlled variable function domain.* Define the required controlled variable behavior as a function giving ideal behavior. First, identify in which states of the system the function must be defined (i.e., the domain of the function). If the behavior changes as a function of the monitored variable's history, identify or define the relevant mode machines.

Additional Guidance: Controlled variable function template (Section 4.3.1)

4. *Define mode machines.* Where the value of the controlled variable is a function of the monitored variable's history, define a mode machine to model the state and state changes. Identify the relevant states (i.e., those that partition the domain of the controlled variable function) and model these with the modes of a mode class.

Define the transitions between classes in terms of changes in monitored variables (events). Figure 6-3 illustrates the results of steps 3 and 4.

Additional Guidance: Finite state machine model for mode machines (Section 4.1.5)

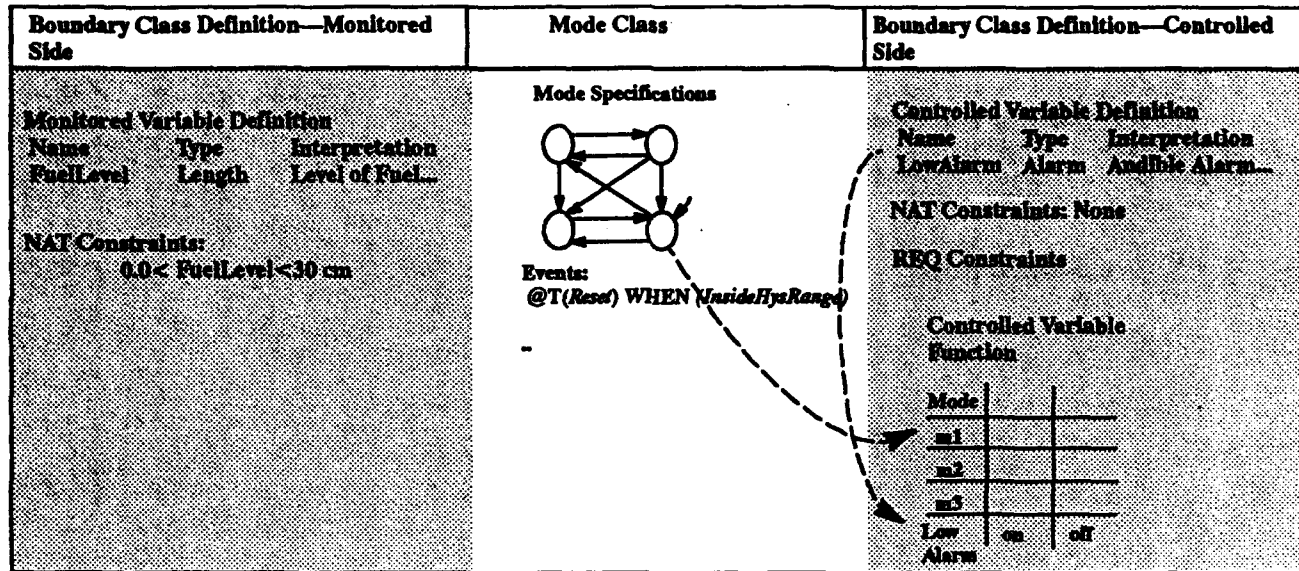


Figure 6-3. Results of Steps 3 and 4

5. Identify the conditions.

For each mode, identify under which circumstances the controlled variable takes on which possible values. Equivalently, you must fill in each cell of the controlled variable function table; this defines the behavior in every possible state of the monitored variables.

Additional Guidance: Controlled variable function template (Section 4.1.5)

6. Define the controlled variable tolerance.

Establish the range of variation allowed in the controlled variable value in each possible state, and represent this as a tolerance in accuracy. Establish the range of variation allowed in the timing, and define the timing and accuracy constraints. Figure 6-4 illustrates the results of steps 5 and 6.

Additional Guidance: Controlled variable function template (Section 4.1.5)

7. Define input and output variables.

Identify the inputs used to determine the values of the monitored variables used, and define an input variable for each. Identify the outputs used to set the value of the controlled variable, and define an output variable for each.

Additional Guidance: Input and output variable templates (Table 11-1)

8. Define the IN and OUT relations.

For each monitored variable, specify how the value of that variable can be determined from the input variables. Define a relation showing how the required value of the controlled variable can be set by assigning values to the output. Figure 6-5 illustrates the results of steps 7 and 8.

Additional Guidance: IN and OUT relation tables (Section 11.3.3)

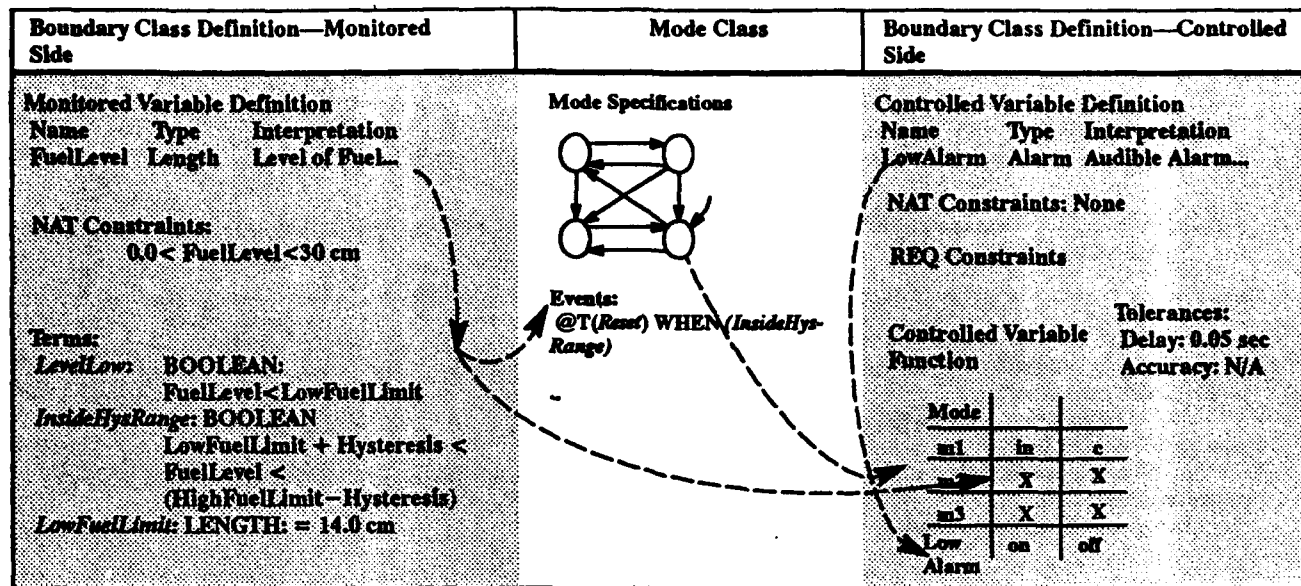
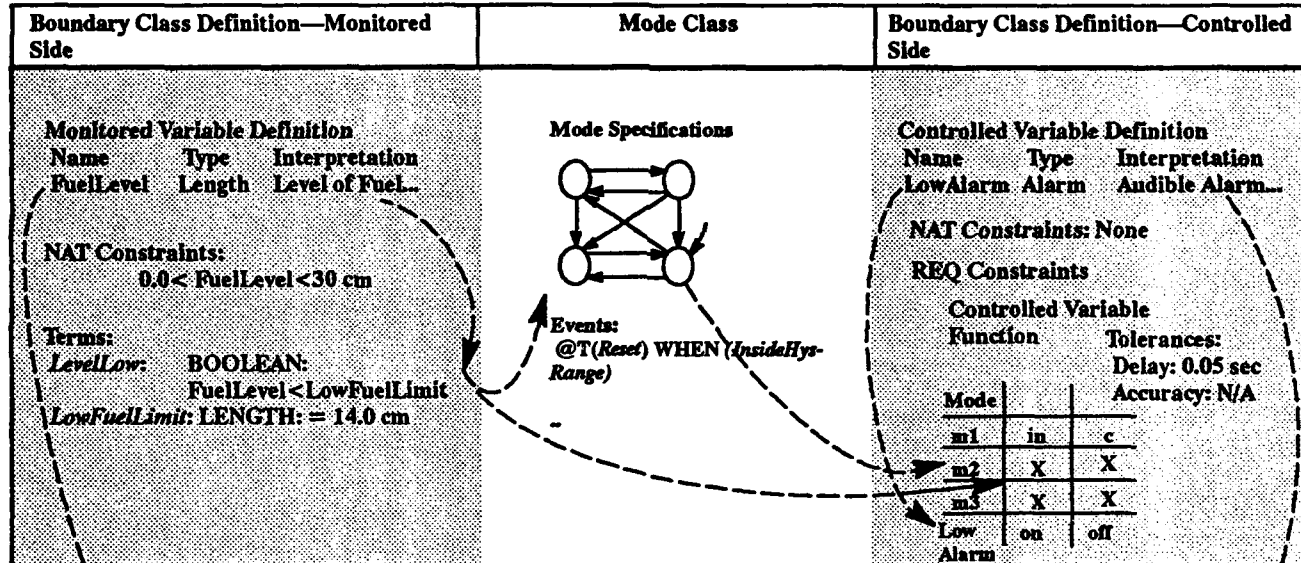


Figure 6-4. Result of Steps 5 and 6



IN Relation Specification	
IN Relation	DiffPress =
Condition	
$\text{LCB} < \text{FuelLevel} < \text{UCB}$	$\text{FuelLevel} - \text{offset}/\text{scale} * 255$
$\text{FuelLevel} < \text{LCB}$	0
$\text{FuelLevel} > \text{UCB}$	255
Input Variable	
Acronym: Differential Pressure	
Hardware: Differential Pressure Unit	
Characteristics of Value: [0..255]	
Data Transfer: ADC(0)	
Data Representation: 8-bit unsigned	

OUT Relation Specification	
Out Relation	
Event	Action
$@T(\text{char} = \text{bel})$	LowAlarm = on
$@T(\text{duration} > 0.5 \text{ s})$	LowAlarm = off
Output Variable	
Acronym: Audible Alarm	
Hardware:	
Characteristics of Value: on 1b, off 0b	
Data Transfer: Port C	
Data Representation: Bit 5 of Byte	

Figure 6-5. Results of Steps 7 and 8

6.2.2 ITERATION AMONG CoRE ACTIVITIES

In practice, you must iterate the CoRE activities as you refine your understanding of the behavioral requirements, the implications of your packaging decisions, or as requirements change. This section discusses how you will iterate between the major CoRE activities. Where additional guidance is needed on iteration within an activity, this is given in the relevant section of the guidebook. In general, it is assumed that you will iterate as necessary within an activity.

The primary reason for iteration among CoRE activities is to resolve discrepancies between packaging decisions (class structuring) and behavioral modeling. The driving problem is that you cannot finalize decisions about the class structure until you understand what information is needed by which classes to complete the detailed behavior specification. Conversely, you will not be able to complete the specification of the behavioral model without making decisions about what information the class structure will encapsulate and what information will be public. Thus, you will typically specify part of the behavioral model, do some class structuring, evaluate the class structuring, then return to specifying the behavioral model until the open issues become sufficiently resolved to stabilize the specification.

In part, this iteration is captured by the major CoRE activities. The goal of Preliminary Behavior Specification is to identify enough of the components of the behavioral model to make sensible packaging decisions. Detailed Behavior Specification revisits the class structuring decisions. In addition, you will iterate as follows:

- You iterate between Preliminary Behavior Specification and Class Structuring. As you begin to understand what information you wish to encapsulate in the classes and what the form of the class interfaces will be, you may revisit and refine the information about the controlled variables functions you gathered in Preliminary Behavior Specification.
- You iterate between Class Structuring and Detailed Behavior Specification. As you understand the detailed requirements, you may determine that you have encapsulated information needed by other classes. This will force you to revisit Class Structuring.

Overall, much of the iteration will involve what information must appear on the class interfaces. This is because there is an inherent conflict between the goals of class structuring and the goals of the behavioral model. Class structuring seeks to reduce complexity and dependency by encapsulating as much information as possible; however, the behavioral model cannot be completed without sufficient information. To balance these concerns requires iteration between these activities.

Particularly affected by the iteration is the choice of terms (expression of monitored variables) defined in the class interfaces. Terms are one of the primary vehicles for abstracting from details while providing exactly the information needed to write the behavioral model. Because of this, the ideal process description allocates the activity of creating terms primarily to Class Structuring. However, because you will write the behavioral specification using those terms, you will typically revisit the choice of terms and their definitions in Preliminary Behavior Specification, Class Structuring, and Detailed Behavior Specification.

In spite of the necessary iteration, your goal is to stabilize as much of the class structure as possible during the Class Structuring activity. It is the fact that classes have stable and well-defined interfaces that will allow independent development of the individual classes.

6.2.3 MANAGING REQUIREMENTS DEVELOPMENT

The CoRE method allows you to divide a requirements specification into a set of distinct and relatively independent parts. At the systems level, you can divide the problem into distinct subsystems. You divide the software specification into a set of classes. Classes may encapsulate additional classes and class dependencies. These provide the packaging facilities needed to manage development issues. While the specific goals and necessary techniques depend on what must be managed, the general approach is to use the class structure to encapsulate parts of the specification that must be treated as a unit, for example:

- **Concurrent Development.** You will often want to break the specification task into parts, each being a distinct work assignment. You can use the class structure to accomplish this. When you define the interface of a class, you can only use information defined in the interfaces of other classes to develop the encapsulated part. One or more classes can be allocated as a work assignment and completed independently. Where there must be changes to a class interface, the CoRE dependency graph allows you to determine which parts of the specification and, hence, which work assignments are affected.
- **Risk Mitigation.** You will want to develop different parts of your specification to different levels of detail at different times. Where parts of the requirements are considered higher risk, you may proceed to detailed specification, design, or implementation to mitigate that risk. For example, you might develop part of the software to better understand the problem or to ensure that your solution is satisfactory. In some cases, one part of the requirements may be better known or better understood than another. In such cases, the class structure can be defined so the parts that will be developed to detail are encapsulated in one or more classes. Development of these classes can then proceed independently of the rest of the software.
- **Fuzzy or Changeable Requirements.** Particularly in the initial stages of development, some requirements will be fuzzy and others will be highly volatile as the software develops. You will want to reduce the likelihood that the refinement of fuzzy requirements or changes in volatile requirements affects other parts of the specification. In CoRE, you can encapsulate fuzzy or volatile requirements in a class, putting only the most stable requirements on the class interface. Then, changes to those requirements will be confined to the defining class. Conversely, you may choose to isolate and develop a part of the software that is considered stable. Either approach can be used in shaping an evolutionary development process (e.g., specify best-defined first or identify and specify high-risk areas).

7. IDENTIFY ENVIRONMENTAL VARIABLES

This guidebook assumes that you start the software requirements activity with a set of system requirements that, from the standpoint of creating a CoRE specification, are relatively unstructured, inconsistent, and incomplete. In addition, the guidebook assumes that relationships among the candidate environmental variables are not simple and well understood; e.g., you may know that related sets of candidate environmental variables characterize distinct things but are not sure that the things should be thought of as instances of the same entity.

The Identify Environmental Variables activity focuses on informally identifying, collecting, and organizing the information you will need to create a CoRE specification. In particular, it focuses on identifying candidate environmental variables and the relationships among them. The creation of an information model drives this activity, where an information model nominally consists of a set of entities, attributes of the entities, and the relationships among instances of the entities. However, the information model is not a part of a CoRE specification; it is solely used to help you gather and organize the information you need to create the formal CoRE specification (described in Sections 8 through 11).

You use the information model to populate both the CoRE behavioral model and class model. In the behavioral model, you consider the entity attributes to determine which environmental quantities will be captured as monitored variables and controlled variables. You also gain a preliminary understanding of the information you need to define the REQ (which monitored variables determine the required value of each controlled variable) and NAT relations (how the environment of the software system constrains the values that the environmental variables can assume).

In the class model, you consider which variables are inherently related and should be defined in the same CoRE class. Also, you identify which variables should be thought of as specializations of a more general variable (described by the generalization/specialization relation).

7.1 GOALS

The goal of this activity is to identify candidate environmental variables (attributes) and to understand and record how they and the things in the environment that they characterize (entities) are related. In this activity, you want to gather and organize information so that you have a complete list of:

- Physical quantities that you use to specify the software requirements and the definitions of the attributes that denote them
- Entities in the environment that the attributes characterize and the subclass and aggregation relations among the entities to organize the information

- Relations among the entities that constrain the values that their attributes can assume
- Likely changes and their impact on the environmental variables

7.2 ENTRANCE CRITERIA

One or more sources of information about the system, its requirements, and its design are required to perform this activity. The following are typical sources of this information:

- System requirements specifications
- System interface specifications
- Technical descriptions of devices that are part of the software system
- Access to domain experts

Specification documents vary considerably in format and degree of formality depending on the nature of the project. Whatever the organization and level of detail of the systems-level specification documents, it is important to identify supplementary sources of information. A technical manual for any device to which the software must interface may be a critical source of information. Furthermore, domain experts, who are familiar with the engineering, physical science, and human factors of the problem domain, are often needed to provide supplementary information. In some projects, they may be the primary source of information.

7.3 ACTIVITIES

The Identify Environmental Variables activity is composed of the following subactivities:

- The Identify and Define Attributes subactivity guides you in identifying the physical quantities (environmental variables) that are relevant to describing the behavior of the software.
- The Identify Entities subactivity guides you to organize the attributes by identifying the entities in the environment that the attributes characterize.
- The Identify Generalization/Specialization Relation subactivity guides you in commonality on the values of the attributes.
- The Identify Aggregation Relation subactivity guides you on the values of the attributes.
- The Identify Application-Specific Relation subactivity guides you in constraints on the values of the attributes.
- The Identify Likely Requirements Changes and Associated Variables subactivity guides you to identify likely changes in the system-level requirements and to relate each change to the environmental variables.

The CoRE information model serves a different purpose than a traditional data model (like the model outlined by Chen [1976]). The emphasis of this activity is to identify the environmental variables and

their constraints, not implications of data integrity, information retrieval, and data manipulation. The CoRE information model also differs in that concepts, such as key attributes, cardinality, and third normal form, are not considered. The common diagramming technique for capturing an information model is an ERD, and building one of these for CoRE is **optional**. You could capture relationships among the attributes in an attribute matrix. The examples given in this section are illustrated using the graphics for an ERD and an attribute matrix (see Section 5.1).

Note that the sequencing of activities does not follow an idealized top-down process. In such an idealized process, for example, entities would be identified first, and then the attributes of each entity would be identified. The activities are shown here in more of a bottom-up fashion, which is appropriate because the sources of information for performing the activities are not likely to be organized to best support a top-down process. This ordering of activities assumes that it will be easier for you first to identify the attributes from the information available to you than to identify the entities first. If you think it will be easier to identify the entities first, you should identify the attributes after you have identified the entities or perform the activities concurrently.

7.3.1 IDENTIFY AND DEFINE ATTRIBUTES

This activity's goal is to identify physical quantities that are relevant to describing the required behavior of the software and to define an attribute to denote each quantity. The software may monitor or control these quantities, or their values may influence quantities that the software does monitor or control. It is not necessary to make such distinctions in this activity; rather you simply identify, define, and collect all of the attributes. It is also not necessary in this activity to ensure that the attributes are independent of one another, i.e., that the value of one cannot be computed from the value of another.

Likely sources of attributes are:

- Variable properties of physical objects in the problem scope, e.g., positions, velocities, and temperatures.
- Physical quantities, such as dimensions of physical objects.
- Information passed across the interface of physical devices, e.g., device status or device commands. Environmental variables typically abstract the interfaces of physical devices. We look at physical devices because they give us insight into which physical quantities the software system can monitor and control.
- Information provided by or supplied to a human user, e.g., user commands or user displays.
- Undesired events, e.g., failures of components of the system or of the software system itself, to which the software system is required to respond.

EXAMPLE: In the FLMS problem, variable properties include the level and flow rate of fuel in the tank. Information passed across device interfaces includes the signal that opens or closes the pump relay. User interface information includes the reset signal provided by the user via a push button. Because the FLMS is a safety system, the failure of it is an important concern and is denoted by the attribute Failure of FLMS. Table 7-1 provides a more complete list of attributes for the FLMS.

Table 7-1. Sample Fuel Monitoring System Attributes

1.	ShutdownRelay
2.	Power
3.	Failure of FLMS
4.	Fuel flow rate
5.	Fuel level
6.	Level too high
7.	Level too low
8.	Reset
9.	Selftest
10.	Fuel level display
11.	Audible alarm
12.	Level too low alarm
13.	Level too high alarm

Define the attributes that you have identified to precisely communicate your decisions to other engineers. Clearly describe the association between the physical quantity and the attribute that denotes it. Describe the association between what can be observed from a view external to the software system and what value the attribute will assume (see Tables 7-2 and 7-3). In Section 8.3.1, you expand upon this definition for attributes that you decide are environmental variables. You may decide that you understand the software system and its environment well enough that you already know which of the attributes are monitored and which are controlled variables. In this case, you may want to provide the more complete and precise definitions of attributes that Section 8.3.1 describes for environmental variables.

Table 7-2. Sample Definition of an Enumerated Attribute

Attribute	Definition
ShutdownRelay	The fuel pump shutdown relay is closed.
	The fuel pump shutdown relay is open.

Table 7-3. Sample Definition of a Numeric Attribute

Attribute	Definitions
Fuel Level	Level of fuel in the tank, in centimeters (cm), along the vertical axis on the left side of the tank, 5 cm from the front edge. The level is measured with respect to the scale.

7.3.2 IDENTIFY ENTITIES

The goal of this activity is to organize the attributes by identifying the entities in the environment that the attributes characterize. The entities are often physical things (e.g., engines, aircraft, radios, and other equipment), but they may include the roles played by persons or organizations (e.g., pilot, operator), incidents, processes, interactions (e.g., the mixing of dyes in a vat, a near-collision of two aircraft), and specifications (e.g., a specification of the attribute values that distinguish pickup trucks from minivans) (Shlaer and Mellor 1988).

In addition to the attributes, consider the following when identifying entities:

- Devices that are monitored or controlled by the software
- Physical entities that are "observed" via sensors by the software
- People or systems that receive information from or provide information to the software
- The software system itself or possibly its subsystems

Associate each attribute with the entity that it characterizes. You may identify entities that no attribute characterizes. When you do so, look for attributes you have overlooked to characterize the entities. To establish a context for the software system, you may want to include in your information model entities characterized by no attributes.

Table 7-4 lists the FLMS entities and the attributes that characterize each of them. Consider whether there may be multiple instances of any of the entities that you have identified (e.g., four-engine entities on an aircraft). Indicate multiple-entity instances by recording the number of instances with the entity.

The entities that you have defined represent a relatively unstructured set of information. Organize the entities by examining them and the specifications that serve as inputs to this activity to determine how the entities are related to one another by generalization/specialization, aggregation, and application-specific relations.

7.3.3 IDENTIFY GENERALIZATION/SPECIALIZATION RELATION

The goal of this activity is to identify similarities among entities and record this information via a generalization/specialization relation. The generalization/specialization relation indicates that one entity is an instance of a more abstract entity; e.g., the reset and selftest switches are instances of a more general two-position switch (see Figure 7-1). This relation allows you to record essential similarity among entities in the system's environment. Recording this similarity will be useful when you try to understand what changes are likely to occur together.

Table 7-4. Sample Fuel Level Monitoring System Entities and Attributes

Entity	Attribute
Pump	• ShutdownRelay • Power
FLMS	• Failure
Fuel Tank	• Fuel level • Level too high • Level too low
Fuel	• Fuel flow rate
Operator Interface	• Instances • Reset • Selftest • Fuel level display • Audible alarm • Level too low alarm • Level too high alarm

7.3.4 IDENTIFY AGGREGATION RELATION

The goal of this activity is to provide structure and information that the software must maintain that belongs to an aggregate as opposed to an individual part. The aggregation relation indicates that one or more entities are part of another entity (e.g., engines, wings, and fuselage are part of an aircraft). This relation allows you to record information that may be helpful in understanding the system's environment and structuring your description of it. This relation may lead you to identify additional entities that aggregate entities you have already identified. Figure 7-1 includes the entities Shipboard Fuel System (an aggregate of the entities already identified) and Operator Console (an aggregate of Alarm and Display).

7.3.5 IDENTIFY APPLICATION-SPECIFIC RELATION

The goal of this activity is to develop an understanding of constraints on the values of attributes that the system's environment imposes or that the system is required to impose. In subsequent activities, you will expand on and refine this information to create the NAT and REQ relations, respectively. In this activity, you will not attempt to distinguish between the two or attempt to specify precisely what those constraints are. Rather, you will decide which attribute values may be affected by the values of which other attributes.

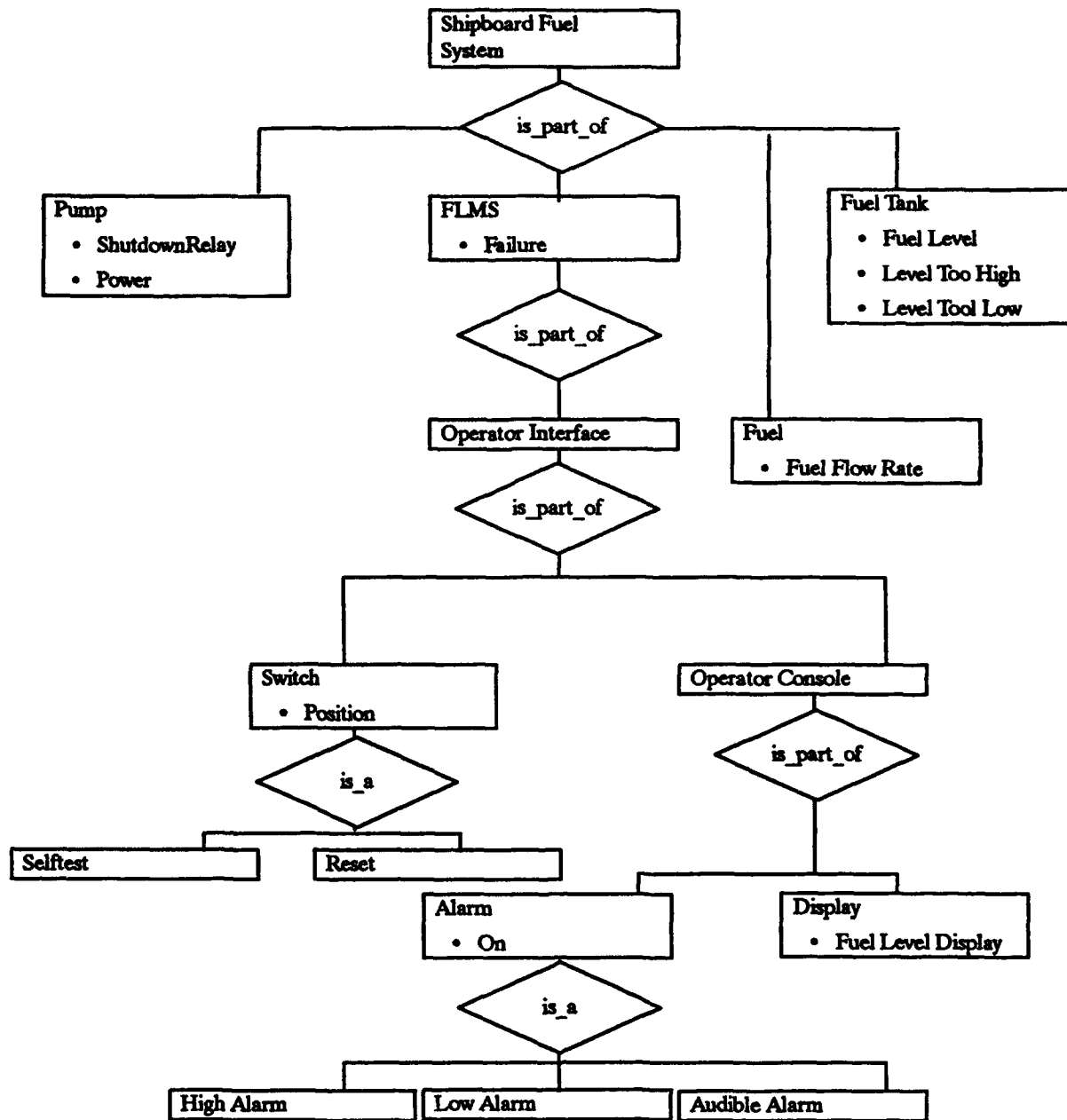


Figure 7-1. Information Model for the Fuel Level Monitoring System

Examine the entities and attributes that you have created, looking for pairs of attributes in which the value of one determines, prescribes, or constrains the value of the other. For each pair, create a relation between the entities characterized by the attributes. If the attributes characterize the same entity, the entity will be related to itself. Annotate the graphical information model with the relation, or use an attribute matrix to record the attributes constrained by the relation (see Table 7-5).

EXAMPLE: Figure 7-2 shows an example of a constraint between two entities, Weapon and Target. This figure shows the relationship between Weapon and Target, where a weapon is designated to a target.



Figure 7-2. Weapon and Target Constraint

7.3.6 IDENTIFY LIKELY REQUIREMENTS CHANGES AND ASSOCIATED VARIABLES

The goal of this activity is to identify likely changes in system-level requirements and to relate each change to its impact on the potential environmental variables (i.e., the attributes that you have identified). Create a list of the likely changes that you identify. You will use this list of likely changes with other system-level information available to you to allocate the environmental variables into classes.

Consider how the potential environmental variables that you developed might change. For example, in the FLMS problem, human factor issues can cause changes to the variables that describe the information that the system provides to the user: Audible Alarm, High Alarm, Low Alarm, and Fuel Level Display. Such a likely change can be described by the following description:

The information that the system provides to the user and how that information is presented is likely to change.

Also, consider information provided by the following in developing the list of likely changes:

- System requirements specifications
- System interface specifications
- Technical descriptions of the devices with which the software interacts
- Discussions with domain experts

In particular, look for areas in which knowledge of the system and its environment appears vague or contradictory. For example, in the case of FLMS, if some of the experts discuss unsafe conditions in terms of the fuel level in the tank and others discuss unsafe conditions in terms of the volume of fuel in the tank, you could consider this description a likely change:

How the FLMS will determine unsafe conditions is likely to change.

An example of the likely change list from the FLMS follows:

1. The means for informing the operator of the FLMS state (i.e., the alarms and displays) is expected to change independently of how the system recognizes unsafe conditions.
2. Level display is likely to change independently of the rest of the system.
3. High and low alarms are unlikely to change independently of one another. They are likely to change independently of the rest of the system.

4. The physical characteristics of the push buttons are unlikely to change independently of one another. They are likely to change independently of the rest of the system.
5. The exact levels that are considered out of range may change from one installation of the FLMS to another or over time. They are likely to change independently of the rest of the system.

7.4 EVALUATION CRITERIA

Evaluate the products of this activity by answering the following questions:

- Have you completed the definitions of each entity, attribute, and relationship?
- Have all relevant relationships among entities been identified?
- Have likely requirements changes and associated entities or attributes been identified?

7.5 EXIT CRITERIA

You have completed this activity when you can answer "yes" to each of the questions in Section 7.4 and you have developed the following products:

- Candidate list of environmental variables
- Candidate list of the information needed to determine the required value of each controlled variable
- Candidate list of how the environment of the system constrains the values that the environmental variables can assume
- Likely change list

8. PRELIMINARY BEHAVIOR SPECIFICATION

In the Preliminary Behavior Specification activity, you make a first cut at identifying and defining the elements of the behavioral model for the software you are developing. You will use the environmental quantities and relations you identified in the previous activity (Identify Environmental Variables) as well as information about sequencing behavior (e.g., system modes or states) from the systems specification to perform the following activities:

- Identify and define monitored and controlled variables
- Identify domain and scheduling type for each controlled variable
- Identify terms
- Identify the modes

The products of this activity include definitions of the monitored and controlled variables, an overview of each controlled variable domain and scheduling type, a description of each term, and a description of each of the system mode machines.

This activity lays the foundation for subsequent packaging and detailed specification activities. It is intended to develop sufficient information about the behavioral requirements so that you can proceed to make sensible packaging decisions. It also lays the groundwork for the detailed specification of the REQ relation (the goal of Detailed Behavior Specification).

8.1 GOALS

The goals of this activity are to:

- Identify which environmental quantities you treat as monitored and controlled and denote these as monitored and controlled variables.
- Determine whether the scheduling type for each controlled variable is periodic or demand. Identify which modes and monitored variables determine the controlled variable's value.
- Identify which terms are needed to define the controlled variable functions and the modes.
- Determine how many mode machines are needed and, for each, the initial mode and the allowed mode transitions.

8.2 ENTRANCE CRITERIA

To perform the Preliminary Behavior Specification activity, you need the following products:

- The information model developed in the Identify Environmental Variables activity
- List of likely changes
- System requirements or other specifications defining system modes and the sequencing of activities controlled by the software

8.3 ACTIVITIES

The Preliminary Behavior Specification activity is composed of the following subactivities:

- **Identify and Define Monitored and Controlled Variables.** Identify which quantities the software will monitor and control and denote them as monitored and controlled variables, respectively. Develop the definition of each variable, and create the system context diagram.
- **Establish Controlled Variable Function Domains.** The REQ relation for each controlled variable must be written in terms of the values or state history of the monitored variables. In this activity, you decide which monitored variables and which modes determine the value of each controlled variable and record the information. You also determine whether the controlled variable must be set periodically or on demand.
- **Define Mode Machines.** Determine the number and characteristics of the mode machines. Decide how many mode machines are needed. For each mode machine, specify the modes, initial mode, and allowed mode transitions. Identify and record where one mode machine depends on another. Identify and record which controlled variables depend on which mode machines.

8.3.1 IDENTIFY AND DEFINE MONITORED AND CONTROLLED VARIABLES

The goal of this activity is to define the monitored and controlled variables you use to write the REQ and NAT relations. To start this activity, you need the set of attributes, attribute definitions, and application-specific relations you identified in the Identify Environmental Variables activity. The products of this activity are the system context diagram and the definition of each monitored and controlled variable in your specification.

Start by identifying the controlled variables. Because the ultimate purpose of the monitored variables is to allow you to write the controlled variable functions, exactly which monitored variables are most appropriate depends on the choice of controlled variables.

8.3.1.1 Identify Controlled Variables

In this activity, you identify which environmental quantities you denote as controlled variables and create the corresponding controlled variable definitions. To identify the controlled quantities, first examine the attributes and application-specific relations created in the Identify Environmental Variables activity. Examine the attribute definitions and the relations to determine which quantities are

partially or completely under control of the software. Look for devices whose states are set by software, situations where information is supplied to the operator or to other systems, and situations where other quantities are produced or affected by the software. If you have previously determined which quantities are controlled and have produced an attribute matrix relating attributes, use that information to identify controlled quantities. You may also use information from systems engineering specifications about the visible behavior of the system to determine which quantities should be treated as controlled.

EXAMPLE: You determine from the information model for the FLMS (Figure 7-1) and the attribute matrix (Table 7-5) that the following attributes can be denoted as controlled variables:

- Audible Alarm
- Low Alarm
- High Alarm
- Fuel Level Display
- ShutdownRelay

The relations in the model indicate that the system must provide the operator with information represented by the first four attributes. The software controls the state of the ShutdownRelay to carry out its primary duty as a safety-shutdown system. The software also has partial control over the fuel flow because the state of the relay affects the pumps.

In most cases, you will be able to develop the clearest and simplest specification if you treat the device state as controlled instead of the environmental quantities affected by the devices. For example, you would treat the FLMS ShutdownRelay as controlled rather than the fuel flow. This is because the states of the environmental quantities are often affected by factors outside the control of the software, making the relation between monitored and controlled complicated and indirect. In contrast, the system's devices are typically directly under software control.

8.3.1.2 Identify Monitored Variables

In this activity, you identify which environmental quantities you denote as monitored variables and create the corresponding monitored variable definitions.

Identifying the monitored variables is inherently an activity that must be revisited during several of the CoRE specification activities. The choice of which quantities to treat as monitored is ultimately driven by what information you will need to write the controlled variable functions and to track mode changes. While you should make a preliminary identification of the monitored quantities as you identify the controlled variables, you will revisit the activity as follows:

- As you determine which monitored quantities each controlled variable value depends on during the Identify Environmental Variables activity
- As you determine which monitored quantities are needed to define the mode transitions in the Detailed Behavior Specification activity

- As you complete the specifications of the controlled variable functions and terms in the Detailed Behavior Specification activity

Rather than repeat the guidance for identifying and specifying the monitored variables in each of these activities, this section gives an overview of the steps and products.

The goal during Preliminary Behavior Specification is to identify all the monitored variables needed to complete the Establish Controlled Variable Function Domains activity. When you complete the Preliminary Behavior Specification activity, you should have a definition for each monitored variable you identified as needed for the controlled variable function domains.

EXAMPLE: In Figure 7-1, the following potential environmental variables can be monitored variables:

- Selftest Switch
- Reset Switch
- Power
- Fuel Level
- Fuel Flow Rate

The operator can set the first two switches, Power indicates whether the system has electrical power, Fuel Level represents the level of fuel in the tank, and Fuel Flow Rate is a measure of the rate of flow of fuel into or out of the tank.

You will choose the monitored quantities based on the information you need to write the controlled variable functions, identify undesired events, and track state changes as follows:

- **Controlled Variable Function Domains.** After you have identified the set of controlled variables, you must decide which quantities you will use to determine the required values of the controlled variables. As a first step, look at the attributes you identified in the Identify Environmental Variables activity. An attribute that can be used to determine the required value of a controlled variable may be modeled as a monitored variable. Use the application-specific relations (e.g., from the attribute matrix, ERD, or other material defining your information model) to identify attributes related to each controlled variable. Remember that a variable can be both monitored and controlled.
- **Undesired Events.** The inability of the software to determine the value of a monitored quantity or set the value of a controlled one is modeled as an undesired event. This includes the inability to monitor the value to the required precision or set a controlled value within the required tolerance. You create a monitored variable that models the undesired state and abstracts from the set of possible failures (e.g., the particulars of device failures, etc.).

EXAMPLES: The FLMS system requirements tell us that, if the FLMS system cannot determine the current fuel level (i.e., the value of `mon_Fuel_Level`), the software must detect the failure and take the fail-safe approach of shutting down the pumps. You can denote this undesired event by modeling the inability to determine with a monitored variable, `mon_Fuel_Level_Unknown`, denoting whether the system is able to determine the value of `mon_Fuel_Level` to the desired accuracy. Thus, you define two monitored quantities relating to the fuel level: `mon_Fuel_Level`,

denoting the level of fuel in the tank, and `mon_Fuel_Level_Unknown`, denoting the system's inability to determine that value.

Consider an operational flight program for an aircraft that must provide the altitude of the aircraft in a pilot display. The required accuracy depends on factors like the current altitude and which devices are functional. Create a monitored variable, `mon_Altitude_Accuracy`, to denote the accuracy with which the software can determine the value of the monitored variable `mon_Altitude`, and use it to specify the required accuracy of the altitude display.

- **Mode Determination.** As you identify and define the modes, you will define the events, causing mode transitions in terms of changes in the monitored variables. You may add monitored variables based on the need to track state changes.

You will often have a choice among quantities you can treat as monitored. For example, in the FLMS, you might choose to denote as a monitored variable the fuel level in the tank, the fuel pressure, the fuel volume, or even the fuel flow into and out of the tank to determine whether there is too much or too little fuel in the tank. You must choose a quantity that is feasible to measure given the overall system specification, your understanding of the hardware, and physical constraints. You then choose the monitored quantity based on the audience for the specification and how easily you can express the required behavior in terms of that quantity. The specification communicates best to its audience if you choose quantities that mirror your audience's understanding of the problem (e.g., choose a quantity that the customer understands). For ease of use, choose the quantities that most directly model the information you need to express the controlled variable behavior (i.e., those that result in the simplest functions).

As you develop the CoRE specification, you should remove redundant and unnecessary monitored variables:

- If two variables denote the same quantity or if you can derive the value of one monitored variable from another, the variables are redundant. Where one quantity can be determined from another, you should consider creating a term to denote the derived quantity. In general, it will be easier to manage change (e.g., if the purpose for the monitored variable changes or goes away) if you remove such redundancies. For example, the quantities Fuel Flow Rate and Fuel Level are redundant for the purposes of the FLMS because one can be calculated from the other.
- A monitored quantity is unnecessary if it is never used to determine the value of a controlled variable or to determine a mode or, equivalently, used in a term that serves one of these purposes. Before you complete the specification, you should remove any unused variables.

8.3.1.3 Define Monitored and Controlled Variables

In this activity, you record definitions of variables to communicate your decisions precisely to other engineers. If you have already created parts of these definitions in the Identify Environmental Variables activity, you review and refine those decisions in this activity.

In creating a monitored or controlled variable, you are abstracting from the (usually physical) quantity in the environment you must monitor or control. The variable represents the essential information about the quantity for which you need to write the requirements while abstracting from incidental

details. For example, you give a monitored variable only the precision required to set the controlled variables to sufficient accuracy although the actual quantity has no limit on its precision.

Name and Type. Choose names that are commonly used to denote the environmental quantity. Choose the type based on how the variable will be used; i.e., choose units that are convenient to represent, easy to use to express the required values, and understandable to the document's users.

Values. Use the NAT relation to determine the range of values a given quantity can assume (e.g., maximum and minimum fuel levels are determined by the configuration of the tank). Define the possible values of the variable within this range. For enumerated variables, list the values (see Table 8-1). For numeric variables, record the lowest and highest values the variable can assume (see Table 8-2). The range of values defines the range over which the software must be able to represent the monitored or controlled variable; thus, you should define only the range that is actually needed. For example, if very large values of altitude for a particular aircraft are never needed, do not include these values in the range.

Table 8-1. Definition of Enumerated Environmental Variable

Name	Type	Values	Physical Interpretation
con_Shutdown_Relay	ENUMERATED	closed	The fuel pump shutdown relay is closed, and the fuel pump is enabled.
		open	The fuel pump shutdown relay is open, and the fuel pump is disabled.

Table 8-2. Definition of Numeric Environmental Variable

Name	Type	Values	Precision	Physical Interpretation
mon_Fuel_Level	LENGTH	0.0..30.0	0.5	Level of fuel in the tank, in centimeters (cm), along the vertical axis on the left side of the tank, 5 cm from the front edge. The level is measured with respect to the scale.

Precision. For a monitored variable, use the precision to express how accurately the software is required to measure the actual quantity that the monitored variable represents. Precision may be implicit in the range of values or it may be explicitly indicated (as in Table 8-2). If precision were not explicit in Table 8-2, the zeroes following the decimal points in the values specification of mon_Fuel_Level would indicate that it has a precision of 0.1 cm (rather than 0.5 cm, which is indicated by Precision in the table). The 0.5 cm precision expresses the requirement that, using the available input resources over time, the software must be able to determine the actual fuel level to plus or minus 0.5 cm. You use the precision to help determine the adequacy of the inputs.

For a controlled variable, use the precision to express how accurately the software must be able to set the actual quantity that the controlled variable represents. For example, if the controlled variable represents the angle on a flap and the precision is 0.5 degree, this expresses the requirement that the software be able to set the angle to within 0.5 degree. Use the precision of a controlled variable to help

determine if you can satisfy the required tolerances on the controlled variable functions based on the available output devices.

Physical Interpretation. Use the physical interpretation to describe the relationship between the monitored or controlled variable and the quantity that the variable denotes. The physical interpretation allows you to relate the quantities used to write the specification to externally visible phenomena. This allows the specification's readers (e.g., customers, system engineers, testers) to correlate the requirements described in the specification to the observable behavior of the software. Thus, the Physical Interpretation should be defined so it is clear to the specification's readers exactly what quantity the variable denotes.

For enumerated variables, describe the physical interpretation of each possible value of the variable (see Table 8-1). Where the variable models relative quantities (e.g., coordinate systems, angle of attack, separation), use figures to give the precise meaning of the quantity being modeled.

8.3.1.4 Create the System Context Diagram

After you have identified the monitored and controlled variables, you can create the system context diagram. An initial version of the diagram can be used as a visual guide to your specification's monitored and controlled quantities and is used in constructing and assessing the level-0 dependency graph. The final context diagram that reflects your ultimate choices of monitored and controlled variables is included in the final CoRE specification.

You create the context diagram for your software following instructions in Section 5.3.1. A context diagram for the FLMS is shown in Appendix B, Figure B.1.

8.3.2 ESTABLISH CONTROLLED VARIABLE FUNCTION DOMAINS

The goal of this activity is to identify information you need to define the requirements for each controlled variable. In particular, you identify the domain of the controlled variable function by identifying the set of modes and the information about the monitored variables that determine the controlled variable's value. You also identify whether the scheduling requirement for setting the controlled variable is periodic or demand.

To perform this activity, you need the definitions you created in the Identify and Define Monitored and Controlled Variables activity. You need the application-specific relations from the Identify Environmental Variables activity. You also need the systems specification or other documentation defining the originating requirements. The product of this activity is a controlled variable overview (e.g., like in Figure 8-1). You use the products of this activity to refine your decisions about which monitored variables and terms are needed and to make initial decisions about the number of mode machines needed and the modes of each (in the Define Mode Machines activity). You also use this information in Class Structuring to make packaging decisions. You refine the products of this activity to develop a complete specification of the controlled variables functions and timing constraints in Detailed Behavior Specification.

For each controlled variable, begin by looking at its definition. For this activity, the goal is to identify the monitored variables, information about monitored variables, or modes that affect the required value of the controlled variable. The product of this activity is a summary of the information needed to specify each controlled variable captured in a form like the Controlled Variable Overview form in Figure 8-1.

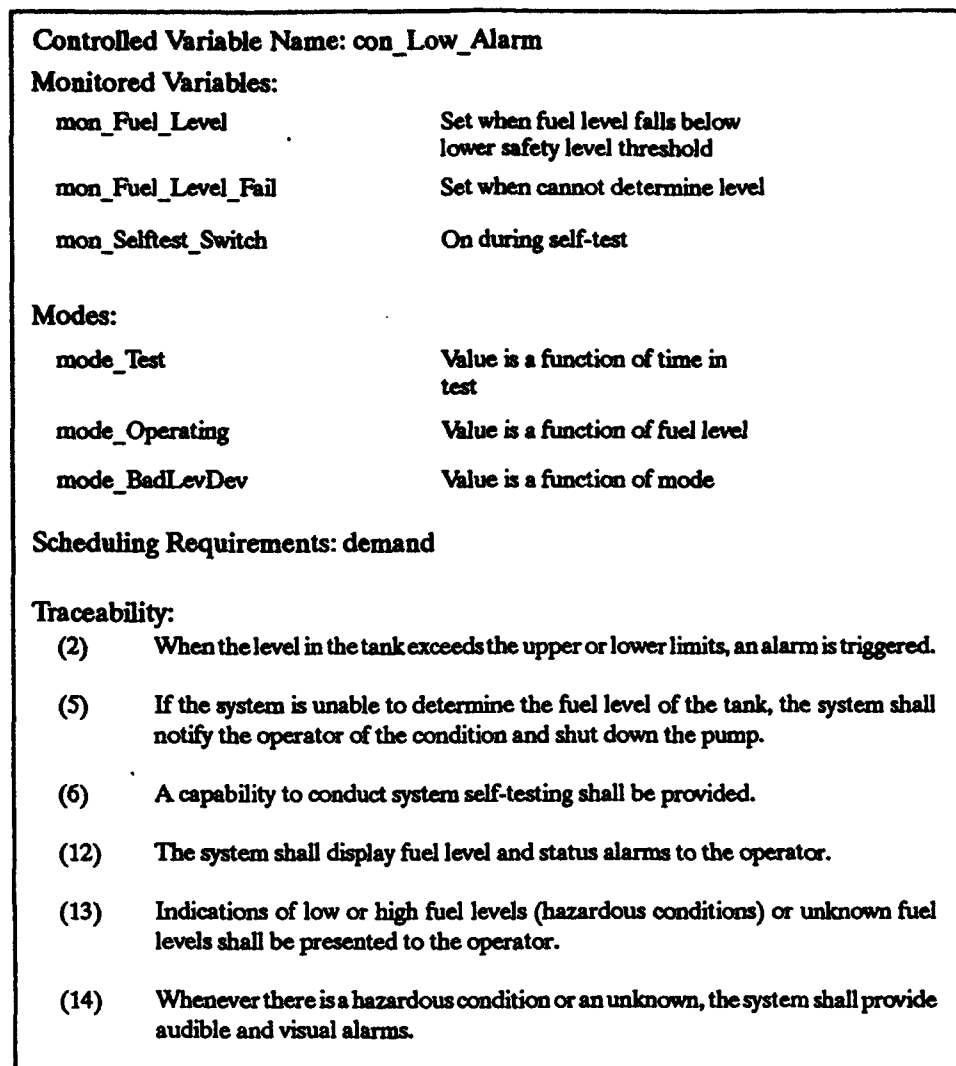


Figure 8-1. Controlled Variable Overview for con_Low_Alarm

8.3.2.1 Identify Monitored Variables

In this activity, you identify the set of monitored variables and the information about those variables that determine the value of the controlled variable. You use this information in Class Structuring to identify the monitored variables and terms that must be defined in the class interfaces.

Look at the set of application-specific relations from the Identify Environmental Variables activity that connect the controlled variable to other attributes. Those attributes that are modeled as monitored variables are prime candidates. Where you identify a monitored variable as affecting the value of a controlled variable, record what information about the variable is needed. You should find this in the originating requirements for the affected controlled or monitored quantities. If the controlled variable is a function of several monitored quantities or otherwise requires a more complex expression to capture, you may choose to create some initial terms to capture the information needed. Add the originating requirements or traceability to the controlled variable overview.

You must also consider how the required behavior changes as the system state changes. To do this, you need to iterate between this activity and identifying the system modes. Look for states where the set of relevant monitored variables changes. Also look for failure states where your ability to set the controlled variable is affected by undesired events. Where these undesired events are modeled as monitored variables, they must be included in the domain.

You iterate between this activity and identifying the monitored variables as you identify places where additional or different information is needed to develop the controlled variable functions.

EXAMPLE: If you look at the stated requirements for the FLMS controlled variable `con_Low_Alarm`, you see that the alarm is signaled when the fuel level is too low; therefore, it depends on the value of the monitored variable `mon_Fuel_Level`. The specific information you need about the fuel level is whether the current level is below the minimum safe level. In examining the attribute matrix, you also determine that the controlled variable value is a function of the FLMS attribute `Failure`, which is modeled with the monitored variable `mon_Fuel_Level_Fail`.

8.3.2.2 Identify Modes

In this activity, you identify a set of modes that determine the value of the controlled variable. To identify modes, you must look at the controlled variable's definition and any information about sequencing or state behavior that affects the variable. For example, you must examine the system specification to determine if the behavior depends on sequences of actions by the user; whether the system is in an initiation, operation, or maintenance mode; and so on. In general, you are looking for points at which changes in values of the monitored variables result in changes in the controlled variable function (i.e., points of discontinuity). You represent each such distinct state with a name and brief description of the behavior that distinguishes the state.

You iterate between this activity and the *Define Mode Machines* activity (Section 8.3.3). Where system mode machines have already been defined, you can express this partitioning of system state in terms of the mode names. Otherwise, you use this information to derive the necessary modes and mode machines.

EXAMPLE: The controlled variable function for `con_Low_Alarm` exhibits different behavior in different states of the system. During normal operations, the alarm value is a function of the fuel level. During a system self-test, however, the value is a function of the time since the self-test was initiated. The alarm is also used to indicate failure of the level-measuring device; here, it is a function of the state of the device.

In this example, the function is divided into three parts, depending on whether the system is in a normal operating state or undergoing self-test or whether a device has failed. These states or operating modes partition the domain of the controlled variable function because the system can be in only one of these states at a time.

8.3.2.3 Identify Scheduling Requirements

You use the scheduling requirements to specify when the controlled variable value must be set. Scheduling requirements are classified as periodic or demand: A scheduling requirement is periodic only if the controlled variable is required to be set at certain intervals. For example, a value must be supplied as part of a feedback control loop every 200 milliseconds. Classify the controlled variable as

demand if the setting of the controlled variable is triggered by the change in value of some monitored quantity or mode.

Neither input nor output hardware characteristics determine whether a **controlled variable** is periodic. For example, the fact that the monitor used to display the controlled variable `con_Fuel_Level` is updated at a certain frequency does not make its REQ relation periodic. The requirement is that the value be provided to a certain accuracy and within a certain minimum delay. The designer must work within the constraints of the refresh rate to ensure that these requirements are satisfied. The periodic constraint in this case belongs to the OUT relation, not REQ.

8.3.3 DEFINE MODE MACHINES

The goal of this activity is to identify the mode machines needed to write the behavior specification. To perform this activity, you need the controlled variable overviews, particularly the modes you identified in the Establish Controlled Variable Function Domains activity. You also need any available information from systems specification and related documents on the system modes and other requirements based on system state or sequencing between activities.

The product of this activity is an initial mode machine specification specifying each distinct mode machine needed, the set of modes in each machine, the set of possible mode transitions, and the initial mode for each machine. You use this information in the Class Structuring activity to identify classes to encapsulate the mode machines. You also use the information in the Detailed Behavior Specification activity to complete the definitions of the controlled variable functions and to complete the detailed specifications of the mode machines.

In a CoRE specification, mode machines are typically used to capture state information that is used to specify a number of controlled variable functions. This means that, in developing the modes, you need to consider the software states in the large (i.e., behavior during system initialization, test, maintenance, etc.) and that you will need to consider the effects of these state and state transitions across a number of controlled variables. In particular, you use a mode machine where:

- The externally visible behavior (i.e., the values of one or more controlled variables) differs from one mode to the next.
- The system changes behavior when changes in the values of monitored quantities (events) occur.

You typically iterate between defining the mode machines and identifying the controlled variable function domains, beginning with a sketch of possible mode machines and refining that sketch as the requirements are better understood. You may not completely identify the modes until you have completed much of the Detailed Behavior Specification activity.

8.3.3.1 Identify Mode Machines

Each machine determines the state for a distinct set of the controlled variable functions. Use the mode information from the controlled variable functions and your understanding of the distinct types of activities in the system to identify the different mode machines. The modes of a mode machine are related by a common set of concerns. The following discussion provides some heuristics for finding appropriate mode machines.

EXAMPLE: An aircraft's navigation system must periodically update the aircraft's current position, i.e., provide a new (presumably more accurate) latitude and longitude. There are several ways of updating a position; the system chooses the appropriate method depending on the pilot's actions and the state of the aircraft. Each of these possibilities corresponds to a navigation update mode; collectively, they form the Navigation_Update mode machine. Modes of the machine are related in that the system performs the same basic service (updating position) in each. They differ in the method of acquiring the new position. Similarly, a Weapons_Delivery mode machine would be concerned with delivering the different weapons that the aircraft can carry. Each mode of the machine corresponds to the delivery of a weapon or set of weapons with somewhat different characteristics.

While there is no hard and fast procedure for deciding which mode machines are needed, the following are common in embedded systems:

- **Represent Distinct System Functions.** Most systems have a variety of states in addition to their normal operation. These include states in which the system performs self-test, undergoes maintenance, or operates under a variety of failure or degraded conditions. Modeling such mutually exclusive system functions as modes allows you to simplify the REQ relations.

EXAMPLE: The self-test and failure states of the FLMS are examples.

- **Embody a Sequence of Activities.** The system is the primary agent or one partner (e.g., in cooperation with a person) in carrying out a sequence of activities directed toward some goal. The value of the controlled variable depends on where you are in the sequence. Model the steps of the sequence as the modes of a mode machine.

EXAMPLE: The FLMS is the primary agent in carrying out the safety shutdown sequence for the shipboard fuel system. Roughly, the sequence is to monitor the fuel level for safety. When an unsafe level is detected, sound the alarm and briefly wait to see if the fuel returns to a safe level. If the fuel returns to a safe level, return to monitoring; otherwise, shut down the system. The distinct steps of this activity are modeled with mode_Operating, mode_Hazard, and mode_Shutdown. The behaviors of most of the controlled variables, including con_Audible_Alarm, are defined as the modes of the shutdown procedure.

- **Model System Modes.** The system specification may allocate certain modes to the software or give the software responsibility for maintaining modes visible to the system's user. For example, it is common to organize the pilot's view of an avionics system into different modes of operation, such as the navigation modes or weapon modes. There will be a navigation mode machine and a weapon mode machine. In such cases, it often makes sense to organize the requirements according to these modes. Such an organization is consistent with the customer's view and simplifies the presentation.

8.3.3.2 Identify Modes and Transitions

For each mode machine you have identified, you must determine the machine's modes, the set of possible transitions, and the initial mode.

You choose the set of modes based on your understanding about the states of the system of interest and the information you derived from defining the REQ relation domains in the previous activity. You

need to create a mode for each distinct set of required behaviors (i.e., as distinguished by points of discontinuity in state-related behavior).

Create a name for the mode that corresponds to an external view of the state and concatenate with `mode_class_`. For example, for an avionics system, you would choose mode names that correspond to the pilot's view of the operating modes of the aircraft. Thus, the `mode_class_Attack` might include `mode_Air_to_Air` for air-to-air combat and `mode_Air_to_Ground` for an air-to-ground mission.

Select the initial mode by determining which operations must be performed on system initialization. You may create an initialization mode to capture the unique required behavior during system initialization.

Use the information you derived from defining the REQ relation domains and any sequencing information from the system specification to identify the transitions.

The set of modes in a mode machine must be defined so that the system is always in exactly one mode of the class. In this way, you ensure that the mode machine covers the domain of the REQ functions (i.e., every partition corresponds to at least one mode of the machine). You must also make sure that there is at least one path to every mode and there is at least one path out of every mode unless the mode is a terminal state of the system.

EXAMPLE: In the example for `con_Low_Alarm` in Section 8.3.1, you identified three states affecting the REQ relation. Further analysis shows that the other alarms in the FLMS (`con_High_Alarm` and `con_Audible_Alarm`) also depend on the same states, i.e., whether the system is operating normally, the level detection device has failed, or the system is in self-test. Because the system can only be in one of the operating modes, the test mode, or the failure mode at a given time, you can model all of these states as the modes `mode_Operating`, `mode_BadLevDev`, and `mode_Test` of a single mode machine.

In addition, there are distinct states that the system must track during the safety shutdown procedures. These are states you would identify in writing the REQ relation for the controlled variable `con_Shutdown_Relay` that is used to turn the pumps off during a system shutdown. In addition to the operating state (`mode_Operating`) when the fuel level is within limits, there is a hazard notification state when the fuel level is out of the safe range. When the fuel level stays out of the safe range for too long, the system shuts down. These states can be modeled as two additional modes, `mode_Hazard` and `mode_Shutdown`. This makes five modes needed to write the REQ functions for the FLMS controlled variables see (Figure 8-2).

8.4 EVALUATION CRITERIA

Evaluate the products of this activity by answering the following questions:

- Is the definition of each of the monitored and controlled variables complete?
- Have you identified the monitored variables and modes that each controlled variable value depends on?
- Is the set of modes defined for each controlled variable domain consistent with the contents of the mode machines?

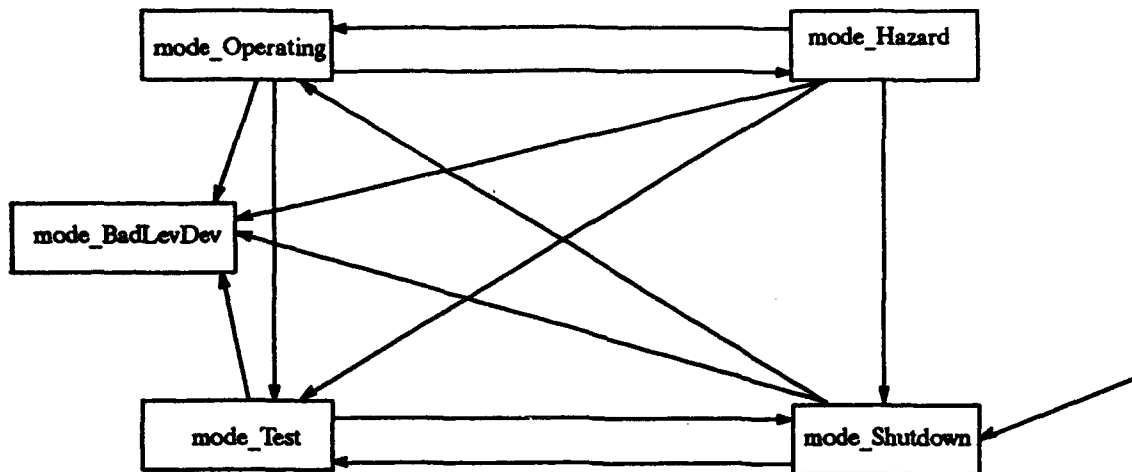


Figure 8-2. Using a Mode Transition Diagram to Represent mode_class_In_Operation

- Is each mode machine well formed according to the criteria in Section 4.2.5?
- Do each of the mode machines satisfy the mode machine criteria that every state is reachable and that every state has an exit unless it is a terminal state?

8.5 EXIT CRITERIA

You have completed this activity when you have produced the following work products and you can answer “yes” to each of the questions in Section 8.4.

- Monitored and controlled variable definitions
- The system context diagram
- An overview for each controlled variable
- Initial mode machine definitions

This page intentionally left blank.

9. CLASS STRUCTURING

In the Class Structuring activity, you make and refine the packaging decisions for the specification. In Preliminary Behavior Specification, you determined the basic structure of the behavioral model. You defined the environmental variables and made preliminary decisions about the behavioral model, including the controlled variable functions, scheduling requirements, modes, and terms. In Class Structuring, you decide which parts of the behavior specification will be allocated to which classes. You decide which parts of the behavior specification will be encapsulated by each class and which will appear on the class interface based on your packaging goals.

As part of creating the class structure, you also make decisions about the encapsulation structure and the generalization/specialization structure. You determine which classes will encapsulate the definitions of other classes and define those internal class structures. You use information about commonality in the requirements to create superclasses and define their subclasses.

Finally, you determine how the definition of each class depends on information defined in other classes. Properties like ease of change, reusability, readability, and the impact of fuzzy requirements are determined by the packaging decisions made in this activity.

9.1 GOALS

The detailed packaging goals of Class Structuring depend on the specific properties you want the specification to have. Typical goals are to create a specification that is easy to read and understand, supports reuse, supports independent development (e.g., for risk mitigation), minimizes the effect of fuzzy requirements, and is stable with respect to likely changes. Of these, you will always be concerned with controlling the effects of requirements changes. Your goal is to encapsulate parts of the specification likely to change and define class interfaces that are unlikely to change. You must have a reasonably stable set of classes and class interfaces to be able to address other packaging concerns, such as reuse or independent development.

In this activity, you create the class structure based on the packaging goals. You allocate the behavioral model to classes and define the class interfaces to satisfy both the behavioral modeling goal of defining the controlled variable functions and the packaging goals, such as ease of change.

If your overall packaging goals have not been decided as part of the system design, you should decide now how important each of these goals is relative to specific parts of the requirements. For example, the specification cannot be equally easy to change for all possible changes. You must decide which changes are most important to accommodate and how likely each change is; this lets you set the more practical goal of making a certain set of requirements easy to change. Similarly, you must decide which parts of the requirements you are likely to reuse, where fuzzy requirements represent a problem that must be controlled, and so on. Identifying these goals gives you a basis for making rational decisions about how the requirements should be packaged.

Within the context of your specific packaging goals, you have the following overall aims:

- Create boundary classes that abstract from details about the software's interface with the environment and encapsulate variables and the REQ, IN, and OUT relations (boundary classes).
- Create classes that provide mode information and encapsulate the detailed specifications of mode machines.
- Create classes that encapsulate fuzzy or changeable requirements.
- Minimize the dependencies among classes.

You create the following work products in the Class Structuring activity:

- A definition of each class interface and its encapsulation structure
- One or more dependency graphs showing the dependency relation among classes

9.2 ENTRANCE CRITERIA

Before beginning Class Structuring, you should have completed the Preliminary Behavior Specification activity and have the following work products:

- A definition for each monitored and controlled variable
- Overviews of the controlled variable function domains, including the monitored variables, the modes needed, and the scheduling requirements
- Initial definitions of the system modes
- Any specification of packaging goals from systems engineering or prior CoRE activities, including:
 - A list of anticipated requirements changes
 - A list of fundamental stabilities and aspects of the requirements, problem, and system not expected to change

9.3 CLASS STRUCTURING ACTIVITIES

Class Structuring comprises the following activities:

- **Create Boundary Classes.** Identify and create those classes that contain the definitions of the monitored and controlled variables.
- **Create Mode Classes.** Create the classes that encapsulate mode machine definitions and provide mode information.
- **Create Term Classes.** Identify and create classes that provide any terms not provided by the boundary and mode classes.

- **Define the Encapsulation Structure.** Identify those classes that will encapsulate the definitions of other classes and develop the encapsulated class structure.
- **Define the Generalization/Specialization Structure.** Identify common elements of the behavioral model and package them as a superclasses. Create the superclass and subclass definitions.
- **Establish Dependencies.** Establish and document which classes use what information provided by other classes and evaluate the class structure based on your packaging goals.

In practice, you typically begin developing the class structure by creating a dependency graph to illustrate the effects of your packaging decisions. Begin by creating the boundary classes. First allocate the monitored and controlled variables to classes, creating a partial dependency graph showing boundary classes and the monitored and controlled variables. Work to connect the classes defining controlled variables and their functions to the classes providing monitored variables, terms, and modes. Add mode classes and show which classes depend on which modes. Add classes to encapsulate groups of terms that you wish to package together, and show how other classes depend on those terms. You have completed the Class Structuring activity when all of the elements of the behavioral model have been allocated to some class and the dependency graph for each encapsulation structure shows all of the dependencies between those classes.

Repeat the process of partitioning the behavioral model into classes and showing the dependencies in increasing levels of detail as you decompose class definitions into further classes. You have completed the entire Class Structuring activity when the entire behavioral model is allocated to some part of the class structure, when no class will be further decomposed, and when all of the dependencies have been identified.

9.3.1 CREATE BOUNDARY CLASSES

In this activity, you create the boundary classes. A boundary class defines monitored and controlled variables and potentially encapsulates the corresponding variables and the REQ, IN, and OUT relations. Define the boundary class interface to provide the definitions of monitored variables that are used by other classes. Specify terms on the interface that provide information about the monitored variables defined by the class.

This section provides heuristics that help guide your decisions about which parts of a boundary class specification should be encapsulated and which parts should be defined on the interface.

9.3.1.1 Allocate Monitored and Controlled Variables

Begin creating the boundary classes by allocating the set of monitored and controlled variables among a set of boundary classes. Allocate the variables to boundary classes by applying the following heuristics.

- Assign to the same class those variables whose definitions are likely to change together. Use information about stabilities in the requirements to help identify variables likely to change together.
- Assign to different classes those variables whose definitions are likely to change independently. Use the list of likely changes to help you decide which variable definitions are likely to change independently.

- Where there are relationships between environmental variables you must preserve, allocate these to the same class. For example, allocate variables to the same class where the variables are part of a coherent abstraction.

If you created an information model in the Identify Environmental Variables activity, use the entities and relationships in the model to help make decisions about whether particular variables should be allocated to the same class. Examine each of the entities in the model to determine whether the environmental variables represented by the model's attributes are related and should be part of the same class. For example, consider whether an entity and its attributes represent a single coherent abstraction. Consider whether a set of monitored variables associated with an entity must be evaluated together. Consider whether a set of controlled variables associated with an entity must be set together.

EXAMPLE: Consider the information model and list of likely changes developed in Section 7. Table 7-4 lists the FLMS entities and attributes. For this example, the elements of the operator interface are likely to change together because they are all handled by the same device. Create class `Operator` and allocate variables `mon_Reset_Switch`, `mon_Selftest_Switch`, `con_Low_Alarm`, `con_High_Alarm`, and `con_Audible_Alarm` to it.

9.3.1.2 Define the Boundary Class Interface

Use the decisions you have made about allocating monitored variables to classes as well as information about the behavioral model (e.g., from the Preliminary Behavior Specification activity) to decide which monitored and controlled variables (if they are also treated as monitored) and terms will be provided on the class interface. Summarize your decisions about what information the class will provide in the Class Description (see Table 5-3).

Monitored and Controlled Variables. You may choose to encapsulate an environmental variable or allow it to be used elsewhere in the specification. Use the preliminary behavior specification to determine whether you need the value of the variable (e.g., to define some controlled variable function) or if you need information about the variable (e.g., a predicate on its value).

Provide the definition of a monitored variable on the interface only if that variable is needed in the definitions of other classes. Provide the definition of a controlled variable on the interface only if the variable is both monitored and controlled by the software. If, instead, some property of the variable or a set of variables is needed, define a term providing the needed information.

If the variable is encapsulated, its definition must appear only in the Encapsulated Information section of the class specification. Otherwise, the definition appears in the Class Interface section.

EXAMPLE: Allocate the monitored variable `mon_Fuel_Level` to class `Fuel_Tank`. Because the monitored variable is needed by other classes (e.g., to describe the value function for `con_Level_Display`), put the definition on the class interface (see Figure 9-1).

Terms. In this activity, you specify any terms provided by the boundary class. Provide terms on the interface to abstract from details of the environmental variables. Reduce complexity by defining terms that provide only information about the variables that are actually needed to define other classes. Manage change by defining terms that provide only the information that is not likely to change while abstracting from details likely to change.

CLASS_FUEL_TANK

class_Fuel_Tank provides the information needed to determine the current fuel level; whether the fuel level is above, below, or within safe limits; and whether the fuel level is within the hysteresis bounds. It encapsulates the constants and rules for determining whether the fuel level is within safe or hysteresis limits. It also encapsulates how the software can determine the values of **mon_Fuel_Level** and **mon_Fuel_Level_Unknown**.

CLASS INTERFACE**Environmental Variable Glossary**

Name	Type	Values	Precision	Physical Interpretation
mon_Fuel_Level	LENGTH	0.0..30.0	0.5	Level of fuel in the tank, in centimeters (cm), along the vertical axis on the left side of the tank, 5 cm from the front edge. The level is measured with respect to the scale.
mon_Fuel_Level_Unknown	BOOLEAN	false		The system is able to obtain the information required to determine the value of fuel level to the required accuracy.
		true		The system is unable to obtain the information required to determine the value of fuel level to the required accuracy.

Constants, Events, and Terms Other Classes Are Allowed to Use**Name****term_Fuel_Level_Range****term_Inside_Hys_Range****Environment-Imposed Behavior (NAT)**

$$\left| \frac{d\text{mon_Fuel_Level}}{d\text{mon_Time}} \right| \leq \text{const_Max_Level_Rate}$$

$$-0.4 \text{ cm} \leq \text{mon_Fuel_Level} \leq 30.0 \text{ cm}$$

Figure 9-1. Partial Definitions of class_Fuel_Tank

Use the Preliminary Behavior Specification, the set of environmental variables defined by the class, and your packaging constraints (e.g., list of likely changes) to decide which terms the class should provide on its interface. Use the Preliminary Behavior Specification to identify which terms are needed that are a function of the variables defined by the class. Use the packaging constraints to decide what information about those terms must be provided on the interface. Apply the following heuristics to decide which terms you should create:

- Where the meaning of a monitored variable's value rather than the value of the variable itself is needed by other classes, create a term that captures the properties other classes need.

EXAMPLE: An avionics system monitors the weight-on-gear switch to determine whether the aircraft is airborne. It is the information about the state of the aircraft that is needed by other classes, so you should create a condition modeling whether the aircraft is airborne.

`class_Operator` defines a monitored variable that denotes the state of the reset switch (`mon_Reset_Switch`). The software determines whether a reset has occurred based on how long the switch is held down (at least 3 seconds). However, the essential information needed to write the other classes is that a reset has occurred, not the state of the switch. Abstract from the arbitrary rules for determining that a reset has occurred, and provide the essential information by creating the `event_Reset` event that occurs when all of the conditions satisfying a reset become true.

- Where the actual quantities monitored are only a means for getting another value, encapsulate the monitored quantities and define a term providing the needed value. For example, this can occur when the required quantities are relative measures among variables. In such cases, provide the needed value on the interface and encapsulate the monitored variables and other information needed to derive the value.

EXAMPLE: For an aircraft collision avoidance system, the system typically monitors the current positions of the host aircraft and any threat aircraft. Which aircraft represents a threat depends on its relative position, heading, and velocity. The interface for a class that encapsulates the aircraft position might provide information for such relative measures.

- Where there are arbitrary constraints or constants that add complexity or are likely to change, define a term that captures the essential information about the environmental quantities and encapsulate the details.

EXAMPLE: `class_Fuel_Tank` defines the monitored variables `mon_Fuel_Level` and `mon_Fuel_Level_Unknown`. You have determined that the current operating mode of the system depends on the state of the fuel level. In particular, it depends on whether the fuel level is too low, too high, or within safe limits. The operating mode also depends on whether the fuel level is within certain hysteresis limits (to ensure stable transitions).

Other classes must use information about whether the fuel level is within safe limits or within the hysteresis limits to determine the current mode or sound an alarm. However, other classes need not use or depend on the specific constants that determine what the limits are.

The definition of `class_Fuel_Tank` addresses these packaging concerns by providing two terms on the interface: `term_Fuel_Level_Range`, an expression denoting whether the current fuel level is too low, too high, or within limits, and `term_Inside_Hys_Range`, a predicate indicating whether the current fuel level is within the hysteresis bounds (see Figure 9-1). The definitions of the terms and the related constants are given in the encapsulated information for the class (See Appendix B, Section B.4).

9.3.2 CREATE MODE CLASSES

In this activity, you create the classes that encapsulate mode machines and create the interface for each mode machine. The class interface provides information about the current mode and mode transitions that can be used to define the behavioral model. It encapsulates the detailed definition of the mode machine, such as which events result in which specific transitions.

Each mode machine must be allocated to one mode class. Create the mode classes based on the mode machines you identified in the Preliminary Behavior Specification activity.

Define a mode class interface by providing the information that other classes are allowed to use about the mode machine. You have few decisions to make about the class interface because the class definition is constrained to provide only certain mode information as follows:

- All of the modes of the class must be defined on the interface (to support completeness checking)
- The events ENTERED(*m*) and EXITED(*m*) are defined for each mode *m* on the interface
- The condition INMODE(*m*) is defined for each mode *m* on the interface

To define the interface, use the form provided in Section 5-3. For example, you would create a class to encapsulate the definition of the mode machine for the FLMS as shown in Figure 9-2. The interface definition provides the names of the modes of the mode machine for the FLMS (see Section 8.3.3.1).

CLASS_IN_OPERATION

class_In_Operation defines the FLMS modes. It encapsulates the rules for determining the current mode and the events that trigger transitions among the modes.

CLASS INTERFACE

Modes Other Classes Are Allowed to Use

Mode

mode_Operating
mode_Hazard
mode_Shutdown
mode_Test
mode_BadLevDev

Constants, Events, and Terms Other Classes Are Allowed to Use

Name

term_Test_Time

Figure 9-2. Mode Class Interface Example

9.3.3 CREATE TERM CLASSES

In this activity, you create any additional classes needed to provide terms that are not provided by the boundary classes. Create a class that defines additional terms from the monitored variables or other terms under the following circumstances:

- The information needed is an expression of two or more monitored variables defined in different boundary classes or used to define the controlled variable functions or modes in two or more classes.
- Some subset of the requirements represents a distinct concern relative to the packaging goals.
- The specification is simpler if you define a class that abstracts from the details of some part of the requirements.

Use the controlled variable overviews, boundary classes, change list, and any additional information from the system specification to identify shared information that should be provided by a term class. In general, the same criteria that govern the creation of terms on boundary class interfaces also govern the creation of term class interfaces.

EXAMPLE: An aircraft collision avoidance system monitors the position of other aircraft relative to the host, determines the threat of collision for each, and provides a visual display of the current threat status to the pilot. The aircraft that represent a potential threat are classified according to a set of rules determined by the Federal Aviation Administration. Because these rules represent data that independently changes from the rest of the specification and the required behavior of the display depends only on the threat class (not on what determines the rules that determine the threat class), the specification is clearer and easier to change if the Federal Aviation Administration rules are encapsulated in a term class.

9.3.4 DEFINE THE ENCAPSULATION STRUCTURE

In this activity, you identify and define encapsulated classes. Make a first pass at defining the encapsulation structure following Preliminary Behavior Specification based on the information allocated to each class. Revisit this activity during the subsequent Detailed Behavior Specification activity as the detailed requirements are better understood.

Define encapsulated classes when doing so addresses your packaging goals. You create encapsulated classes following the same guidelines and heuristics by which you created the higher level (in the encapsulation hierarchy) class structure, e.g., if additional organization into classes helps reduce complexity, manage change, and so on.

If you developed an information model, examine the model for any is-part-of relations. If there are requirements that apply to the whole structure rather than individually to the parts, consider defining the structure as a class and the components as encapsulated classes.

You should stop decomposing into encapsulated classes when each class satisfies your packaging goals, each class is understandable, the information in a class is related, the information in a class is likely to change together, and so on.

EXAMPLE: class_Operator encapsulates how the FLMS is required to interact with the operator and the mechanisms available to the software to support the required interactions. Within class_Operator, the outputs to the operator and the inputs from the operator represent parts of the specification that can be usefully represented as two encapsulated classes.

The controlled variables (con_Audible_Alarm, con_High_Alarm, etc.) sent to the operator satisfy CoRE's heuristics for class formation in several ways. First, all of the outputs use the same hardware

device so they share parts of the OUT relation. Second, because they share the same hardware, they are also likely to change together if the display hardware changes. Finally, they are temporally related because the alarms and displays must be updated to reflect the conditions the alarms are signaling.

The monitored variables from the operator (`mon_Selftest_Switch` and `mon_Reset_Switch`) do not share hardware with the controlled variables, are likely to change independently of those variables, and have some reasonable expectation of changing together. Thus, the switches should be defined in a distinct class.

Create the interface for `class_Operator` by showing which variables and terms are defined by the encapsulated classes and exported by `class_Operator`. These are the events `event_Selftest` and `event_Reset`. Create the encapsulated information by providing the definitions of the two encapsulated classes and the dependency graph of the encapsulated classes. The interface and dependency graph are shown in Figure 9-3. The definitions of `class_Operator_Communication` and `class_Switch` classes are given in Appendix A.

CLASS_OPERATOR

`class_Operator` encapsulates how the FLMS is required to interact with the operator and the mechanisms available to the software to support the required interactions.

CLASS INTERFACE

Constants, Events, and Terms Other Classes Are Allowed to Use

`event_Selftest`

`event_Reset`

ENCAPSULATED INFORMATION

Classes Defined

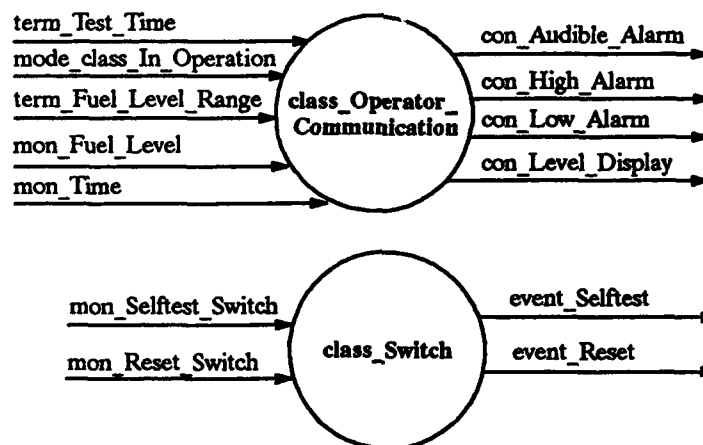


Figure 9-3. Dependency Graph for Operator Interface

9.3.5 DEFINE THE GENERALIZATION/SPECIALIZATION STRUCTURE

If you have identified cases where two or more classes have common requirements, you may choose to represent the commonality by creating a superclass. In CoRE, you use the superclass mechanism

for both representing requirements more compactly and for making the commonality in requirements explicit to others using the requirements (e.g., the designers). This means that you should use the superclass mechanism only if the commonality embodied in the superclass represents an essential part of the problem you do not expect to change. Conversely, you should not use it to represent incidental or accidental commonality.

If you developed an information model, you may have identified potential superclass or subclass relationships in the form of is-a relations. Otherwise, use information on common parts of the specification developed in this and previous activities to identify potential superclass or subclass relationships.

EXAMPLE: The RTCPs and CDU share all the display and much of their data entry characteristics. Further, these characteristics are essential to the problem and potentially useful to future designers as discussed in Section 5. You choose to create a superclass `_Radio_Display` to represent the parts of the behavior specification common to the RTCPs and CDU. You identify the following as common elements:

- `mon_Radio_Selection` is a monitored variable assuming one of six values (UHF1, UHF2, VHF1, VHF2, HF1, HF2) indicating which radio the pilot has selected.
- `mon_Scratchpad_String` is an alphanumeric string entered into the scratchpad.
- `con_Scratchpad` can clear or blink the scratchpad contents.
- `con_Display_Line` displays the current frequency of each radio.

Define each of these variables on the interface of superclass `_Radio_Display`. Then define two subclasses as follows:

- `class_RTCP` adds no additional information, but its definition constrains the type of `mon_Scratchpad_String` to be numeric because there are no alphabetic keys on the RTCP.
- `class_CDU` adds two monitored variables to represent a preset frequency and a mnemonic. It also adds two terms representing a valid preset and a valid mnemonic being received.

You then define class `_CDU` as shown in Figure 9-4.

9.3.6 ESTABLISH DEPENDENCIES

In this activity, you establish how classes depend on one another. In CoRE, a class X depends on class Y if X uses a term provided on the interface of Y in its definition. The number and kind (i.e., which modes and terms a class depends on) of dependencies help you determine how well the class structure meets packaging goals like ease of change.

CLASS_CDU

CLASS DESCRIPTION

`class_CDU` is a subclass of `superclass_Radio_Display`. It encapsulates the rules for entering radio frequencies as presets and mnemonics and for determining if a preset or mnemonic is valid. It provides conditions denoting a preset or mnemonic selected, the radio selected, and the string entered.

CLASS INTERFACE

Defines: `term_Valid_Preset`
 `term_Valid_Mnemonic`
 `mon_Scratchpad_String.CDU`
 `mon_Radio_Selection.CDU`

ENCAPSULATED INFORMATION

Dependency Graph

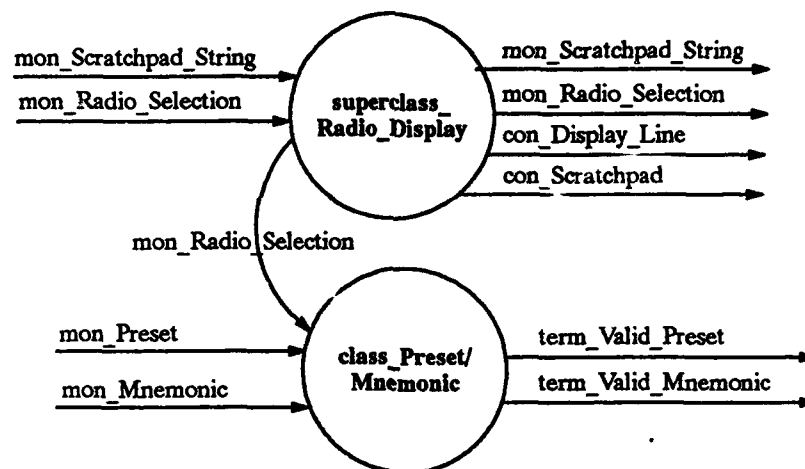


Figure 9-4. Generalization/Specialization Example

You are likely to find additional dependencies or change existing ones during Detailed Behavior Specification. For example, you may find that you cannot write the functions for a particular controlled variable without using an additional monitored variable. However, to understand the implications of your class structuring decisions and evaluate those decisions against the packaging goals, you need an initial set of dependencies.

The goal of this activity is to identify and record the dependencies between the classes. To perform this activity, you need the class definitions. The product of this activity is the dependency graphs that show exactly which class uses which environmental variable, term, event, or mode machine. Create a dependency graph for each distinct part of the Encapsulation Structure; i.e., each set of classes that are encapsulated by the same parent class will have a distinct dependency graph.

In an idealized view of the process, you create the dependency graphs after defining the class interfaces because you cannot complete the graphs until the class specifications are complete. In

practice, you begin constructing the dependency graph in parallel with choosing the classes and defining their interfaces because the graph illustrates the consequences of your choices and helps guide subsequent decisions.

You create a dependency graph for the classes you have defined. To create a dependency graph, examine each class in turn. Determine which of the terms defined in other classes are used to define the current class's interface or is needed to define its encapsulated information. Use the dependency graph notation defined in Section 5.3.2.

EXAMPLE: Figure 9-5 shows a partial dependency graph for the FLMS. Because the mode machine in class `In_Operation` changes modes if a self-test occurs, it uses `event_Selftest` in the definition of the mode transition table. Thus, class `In_Operation` depends on class `Operator`. Draw an arrow from the class `Operator` bubble to the class `In_Operation` bubble and label it `event_Selftest`.

Similarly, the modes defined by class `In_Operation` are used to define the controlled variable functions in both class `Operator` and class `Pump`. Create a dependency from the class `In_Operation` to each class using the mode machine.

You have finished establishing dependencies when there is a dependency for every term that is defined by one class and used by another. In practice, the exact dependencies between classes become better understood and change during subsequent development, especially in the Detailed Behavior Specification activity. You need to iterate the entire development process to identify the dependencies fully.

9.4 EVALUATION CRITERIA

On each iteration and upon finishing the specification, evaluate the packaging and interface creation decisions against your objectives.

9.4.1 EVALUATING CLASSES

Evaluate the classes chosen and the class interfaces against both general criteria and the specific packaging goals established for your specification. General criteria to consider are:

- **Appropriate Abstraction.** Each of the class interfaces you create provides information used to define other classes (e.g., to write the controlled variable functions). The interface is considered a good one if it provides an abstract representation of the information defined in the class that is easy to understand and use. Verify that the names used clearly indicate the content of each term, that each term is well defined, and that the interface is as simple as possible while providing the information needed by other classes. Check the number of dependencies originating in the class; if the number is large, look for a simpler interface or consider creating an additional class.
- **Appropriate Encapsulation.** Evaluate each class to ensure that the division of the requirements into classes provides appropriate separation of concerns. Verify that the class description makes clear what information the class encapsulates. Verify that the class interface does not provide information that should be encapsulated; i.e., check that each piece of information provided by the interface is not part of the information the class should encapsulate.

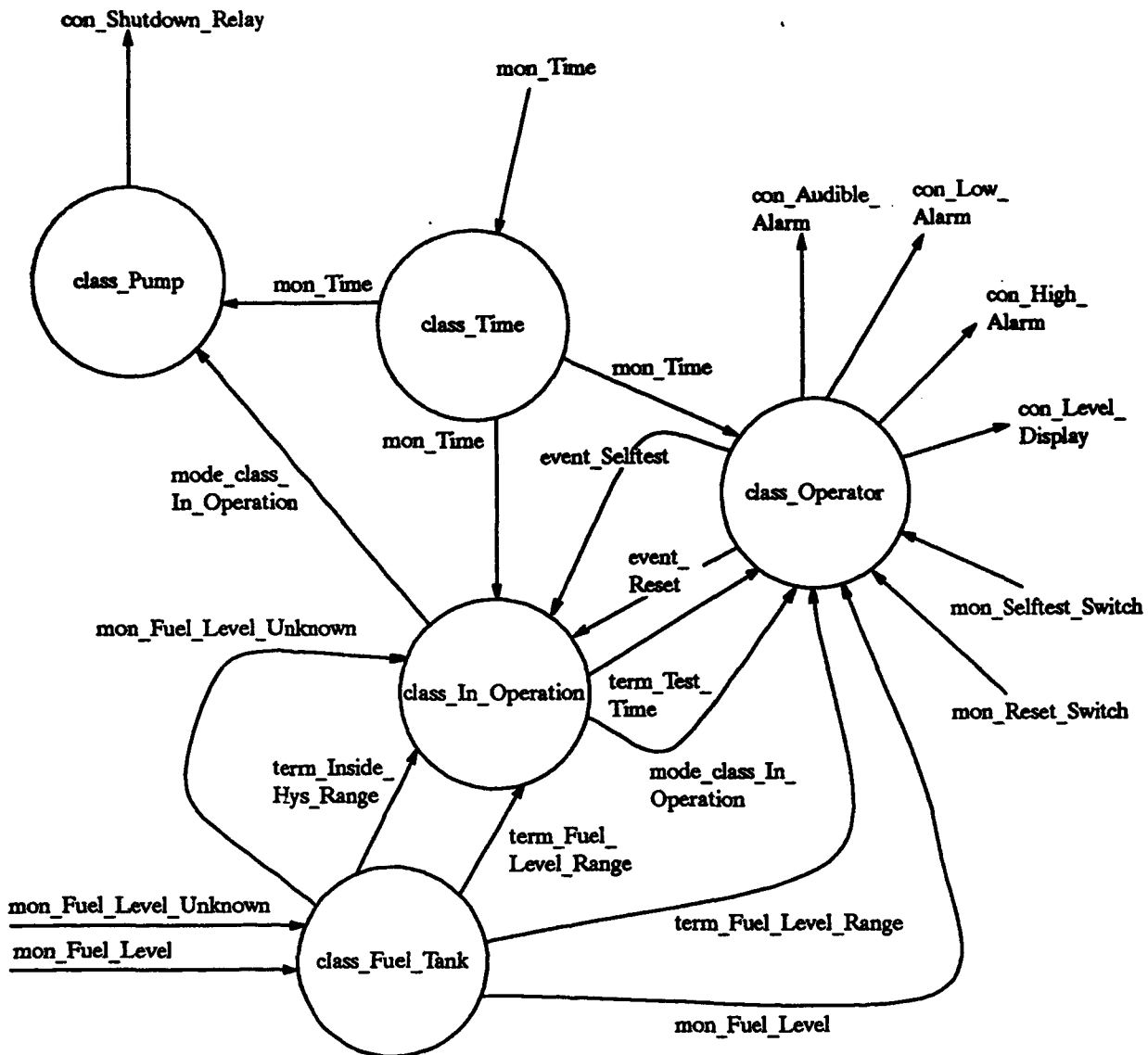


Figure 9-5. Fuel Level Monitoring System Dependency Graph

Each class should encapsulate related information. Verify that the requirements in a class can change together. If they change independently, consider creating an additional class.

You must also evaluate the class structure against any specific packaging goals you established. For example:

- **Change Management.** For each requirement considered likely to change, trace the requirement to the defining class. Verify that the requirement is encapsulated by the class. If a requirement that is likely to change is on a class interface, reexamine the class interface to see if it can be hidden.
- **Fuzzy Requirements.** Limit the impact of fuzzy requirements by encapsulating the fuzzy requirement in a class. For such cases, verify that any fuzzy requirements are encapsulated by a class.

- **Reuse.** The class is typically the atomic unit of reusability in a CoRE specification. Verify that each reusable component is encompassed by a class definition and that the class provides an appropriate abstraction.
- **Risk Mitigation.** Verify that each set of related requirements considered risky is allocated to a distinct class and that the class interface is well defined. You can then proceed with the design and implementation of that class, independent of the other classes, until the risk issues are sufficiently resolved.

9.4.2 EVALUATING CLASS DEPENDENCIES

The dependency graph allows you to evaluate the requirements architecture against the packaging goals. For example, you must determine whether the structure easily accommodates the changes you identified as likely.

- **Closure.** Use the preliminary behavioral model and the dependency graph to determine whether you have identified all the information needed to write the controlled variable functions.
 - Evaluate each boundary class that defines a controlled variable to determine if the monitored variables, terms, and modes used (as shown by the dependency graph) are adequate to define the controlled variable functions (as identified in Preliminary Behavior Specification).
 - Examine the classes that define modes or terms used by the boundary classes. For each class, ensure that the terms or modes provided by the object are defined using the information used by the object (as shown by the dependency graph).
- **Change Management.** Tracing dependencies allows you to analyze the effect of changes to a class interface on other parts of the requirements. For each change that you identify that appears on a class interface, trace the dependencies and determine which other classes will be affected by the change. If there are a relatively large number of dependencies on these or any other interface requirements, consider altering the interface to reduce the dependencies.

9.5 EXIT CRITERIA

Class Structuring is complete when:

- All the elements of the behavioral model (i.e., all the variables and relations) have been allocated to some class.
- Classes and class interfaces have been defined.
- The Encapsulation Structure is defined.
- The Generalization/Specialization Structure is defined.
- All the dependencies are shown on the dependency graphs.

10. DETAILED BEHAVIOR SPECIFICATION

In the Detailed Behavior Specification activity, you complete the specification of the behavioral model relations except for IN and OUT. You use the controlled variable overviews you developed in Preliminary Behavior Specification and the class definitions you developed in Class Structuring to perform the following activities:

- Develop a detailed specification of the value function for each controlled variable.
- Develop a detailed specification of the scheduling, timing, and tolerance constraints for each controlled variable.
- Complete the specification of each mode machine.
- Revisit the class structuring decisions.

When you have finished this activity, you will have completed the definition of the encapsulated information for each class except for the definitions of the input and output variables and the IN and OUT relations (the subject of Section 11).

10.1 GOALS

The goal of this activity is to complete the behavioral model specification. This includes:

- A complete specification of required behavior for each controlled variable as follows:
 - A specification of the controlled variable value function that is complete and consistent
 - A specification of the controlled variable's initial value
 - A specification of the controlled variable function's timing and value tolerances
 - A specification of the periodic or demand scheduling parameters
- A complete specification of each mode class
- A set of class definitions, including interface and encapsulated information, that is consistent with the information needed to complete the behavioral model for each class

When you finish the Detailed Behavior Specification activity, you have a complete description of the externally visible behavior of the software.

10.2 ENTRANCE CRITERIA

To perform Detailed Behavior Specification, you need the following products:

- Monitored variable definitions from the class interfaces
- Controlled variable definitions and overviews
- Term definitions from the class interfaces
- Modes from mode class interfaces
- NAT relation information from the information model, system requirements, and domain knowledge
- Dependency graphs
- Timing and scheduling requirements from the system requirements

10.3 ACTIVITIES

In Detailed Behavior Specification, you revisit and refine the class definitions. The major focus of this activity is on completing the detailed specification of the controlled variable functions and timing constraints and the detailed specification of the mode machines. The subactivities are as follows:

- **Define Controlled Variable Behavior.** Refine the definition of each boundary class that defines the behavior of a controlled variable. For each controlled variable, complete the behavioral specification. Define the value function, the timing and tolerance constraints, the scheduling parameters, and initial value. Define any relevant NAT constraints.
- **Refine Mode Classes.** Complete the mode machine definition by defining the encapsulated information for each mode class. Identify the events that result in each mode transition.
- **Refine Remaining Classes.** Complete the definitions of the encapsulated parts of remaining classes.
- **Revisit Class Structuring.** Where completing the detailed definitions of the classes requires changing Class Structuring decisions, revisit the appropriate Class Structuring activity.

10.4 DEFINE CONTROLLED VARIABLE BEHAVIOR

The goal of this activity is to complete the detailed specification of the functions and timing constraints for each controlled variable. You do this by refining the definition of each boundary class that defines one or more controlled variables. In Preliminary Behavior Specification, you identified the information needed to define the controlled variable functions. In Class Structuring, you developed the class interfaces to provide the monitored variables, modes, and terms needed. At this point, you should have sufficient information to develop the detailed specification for each leaf class defining a controlled variable independently; i.e., you can divide this activity into separate work assignments for each class. Thus, the following discussion is written in terms of completing the detailed controlled variable specification for a single class, one variable at a time.

To perform this activity, you need the definitions of all monitored variables, modes, or terms on which the controlled variable depends. Use the Controlled Variable Overview from the Preliminary Behavior Specification to identify these quantities. Use the dependency graphs to identify which classes

provide needed information on their interfaces. The class definitions then provide the detailed information you can use about each monitored variable, mode, or term. You will also need any timing constraints applying to the controlled variable from the system requirements and any NAT constraints from system requirements or other domain information.

The process of completing the controlled variable behavior specification is guided by the detailed behavior specification template (see Section 4.4). You complete the specification by systematically filling out the template where it applies to the controlled variable being defined. This activity is complete when all the relevant parts of the template have been filled out. The following sections provide guidance on filling out the sections of the controlled variable form for both periodic and demand controlled variables.

10.4.1 SPECIFY INITIAL VALUE

In this activity, you identify and define the initial value that the software must provide for the controlled variable. You specify the value by filling out the Initial Value section as follows:

- *None.* No particular initial value is required.
- *Value function.* Initial value is defined by the controlled variable function.
- *Initial value expression and initiating event.* Use these forms if the initial value must be set based on a certain event (e.g., at system generation, system initialization, or at run-time) and the assignment is not covered by the controlled variable function.
 - *Initial value expression* is an expression giving the initial value to be assigned to the controlled variable.
 - The *initiating event* is the event at which the value must be assigned (e.g., system initialization).

10.4.2 DEFINE SUSTAINING CONDITIONS

The sustaining conditions give the conditions under which the controlled variable function remains defined after initialization (i.e., the conditions under which it makes sense to evaluate the function and set the controlled variable). For certain kinds of controlled variables, the entity being controlled exists only if other systems are operating correctly. For instance, displayed values can be set only as long as the display subsystem is operating properly. Under the sustaining condition, you list any conditions that must be true for the controlled variable to be able to assume its required values.

To identify the sustaining conditions, examine the quantities needed to define the controlled variable functions as well as any conditions relevant to setting the controlled variable itself and create the sustaining conditions as follows:

- Any undesired events that prohibit getting the values of the monitored quantities or related terms
- Any modes (e.g., failure modes) in which the function cannot be evaluated or the controlled variable cannot be set

- Any undesired events indicating that the controlled variable cannot be set

Where the variable always has a value, the sustaining condition is given the value true.

EXAMPLE: The system monitors the state of a display and defines a condition that is true if the display has failed, called `mon_Display_Failure`. The controlled variable `con_Altitude_Display` cannot be set if the display has failed; thus, `NOT mon_Display_Failure` is a sustaining condition.

10.4.3 SPECIFY DEMAND BEHAVIOR

In this activity, you create the detailed specification for a controlled variable with a demand scheduling constraint. Use the controlled variable overview to determine the variable's scheduling constraint.

10.4.3.1 Specify Demand Functions

In this activity, you create the function specifying the ideal value for the controlled variable. Demand functions are characterized by the need to respond to changes in the environmental quantities or modes. Use an event table to represent the value function.

Use the table format (see Section 4.2.6.2) to guide creation of the function and capture its specification. You analyze the function to determine how the behavior depends on the modes. You create one row of the event table for each set of modes for which the behavior of the function is the same.

Unless you have already identified all of the distinct behaviors, it is easiest to begin by creating one row for each mode of the relevant mode machine. As you begin to fill in the rows, you can combine any rows that contain identical conditions.

You create one column in the table for each distinct expression needed to define the value of the controlled variable. If the controlled variable is an enumerated type, create a column for each distinct value. If the controlled variable value is determined by a set of expressions, create a column for each such expression. Put the expression at the bottom of the column.

EXAMPLE: In the FLMS, the controlled variable `con_Audible_Alarm` must be set; i.e., the alarm must be sounded when certain events occur, such as the fuel level going beyond safe limits. You represent the function using an event table as shown in Table 10-1. You create a row for each of the modes of the FLMS and, because `con_Audible_Alarm` is an enumerated type, one column for each possible value of the variable.

Visit each cell in the event table row by row. For a given cell, you provide the event that should cause the controlled variable to take on the corresponding value in the column for the modes in that row. Use information about the mode transitions and the assumptions on the mode class interface to help determine which events are needed. If there is more than one such event, list each event connected by an OR operator. As you complete each row, verify that only one of the events in that row can occur at any time whenever the system is in one of the modes in that row.

Table 10-1. Event Table Example (Incomplete)

Mode	Events	
mode_Operating		
mode_Hazard		
mode_Shutdown		
mode_Test		
mode_BadLevDev		
con_Audible_Alarm	sound	silent

EXAMPLE: Begin filling in the table, starting with mode_Operating (see Table 10-2). In mode_Operating, the alarm must be sounded if the fuel level goes out of safe limits so the corresponding event is added to the column with the value sound. The alarm is turned off (if it is on) upon entry to the mode so the event ENTERED is put in the silent column. In mode_Shutdown, the system never changes the state of the alarm, so an X is entered in each column. In mode test, the stimulating events are based on the time in test. The alarm is turned on upon entry to test mode, then turned off after 4 seconds. These events are entered in the table.

Table 10-2. Event Table Example (Completed)

Mode	Events	
mode_Operating	X	ENTERED
mode_Hazard	ENTERED OR @F(term_Fuel_Level_Range = withinlimits)	@T(term_Fuel_Level_Range = withinlimits)
mode_Shutdown	X	X
mode_Test	@T(term_Test_Time $\geq$ 0 ms)	@T(term_Test_Time $\geq$ 4000 ms)
mode_BadLevDev	ENTERED	X
con_Audible_Alarm	sound	silent

To support readability of the functions, you provide additional textual overview of the specified behavior as necessary. However, it is the function specification that represents the binding requirement in case of conflict.

10.4.3.2 Demand Scheduling and Timing Constraints

In this activity, you specify the timing constraints for the controlled variable function. The timing constraints define the range of acceptable behavior in time. For demand variables, this is expressed

by defining the interval over which the software is allowed to set the controlled variable following an initiating event. You also specify any NAT constraints, such as the minimum interval between events of interest.

To perform this activity, you use scheduling and timing information from the system requirements as well as information about the environmental quantities from systems requirements, domain experts, hardware constraints, and any other sources describing the timing characteristics of the environmental events that affect the controlled variable. You have completed this activity when you have filled out all the relevant parts of the timing and scheduling portion of the behavior specification template (Section 4.4).

The ability of the software to respond to every event (i.e., satisfy the ideal behavior defined by the value function) may depend on the timing characteristics of the initiating events. As part of the NAT relation, you must record any assumptions about how frequently initiating events can occur and what the minimum interval between occurrences is. If events can occur closely enough together (e.g. another event can occur before the completion deadline corresponding to the previous event), you must specify the tolerance in behavior. For example, specify whether the software must respond to every event or is allowed to ignore events under certain circumstances.

10.4.4 SPECIFYING PERIODIC BEHAVIOR

In this activity, you create the detailed specification for a controlled variable with a periodic scheduling constraint. Use the controlled variable overview to determine the variable's scheduling constraint.

10.4.4.1 Specify Periodic Functions

The value function for a controlled variable with a periodic timing constraint defines the values of the controlled variable in terms of the prevailing conditions when the periodic timing event occurs. Use a condition table or selector table (see Section 4.2.6) to guide creation of the function and capture the results.

Analyze the function to determine how the behavior depends on the modes. In particular, you must determine each distinct set of modes where the value of the controlled variable depends on different conditions. You create one row of the condition table for each set of modes for which the behavior of the function is the same, and enter the names of the modes in that row.

EXAMPLE: The FLMS must provide status information to a secondary status and trend recording system. If the fuel level is within safe limits, the current fuel level is reported. If the fuel level is out of safe limits, a special value indicating whether the level is low or high is provided. The status information must be provided every 0.5 second in mode_Operating, mode_Hazard, and mode_Shutdown. It is not provided in mode_Test or mode_BadLevDev.

Unless you have already identified all the distinct behaviors, it is easiest to begin by creating one row for each mode of the relevant mode machine. As you begin to fill in the rows, you can combine any rows that contain identical conditions.

You create one column in the table for each distinct expression needed to define the value of the controlled variable. If the controlled variable is an enumerated type, create a column for each distinct

value. If the controlled variable value is determined by a set of expressions, create a column for each such expression. Put the expression at the bottom of the column.

EXAMPLE: Create one row for mode_Operating, mode_Hazard, and mode_Shutdown because the behavior is the same in each. Create another row for mode_Test or mode_BadLevDev. Create a column in the table for each of the possible values reported, the fuel level, low, and high (see Table 10-3).

Table 10-3. Initial Condition Table Example (Incomplete)

Mode	Condition		
mode_Operating mode_Hazard mode_Shutdown			
mode_Test mode_BadLevDev			
con_Status	mon_Fuel_Level	low	high

Finally, visit each cell in the condition table row by row. Fill each cell with a condition that must hold in the mode defined by the row for the variable to be assigned the value at the bottom of the column. As you complete each row, verify that exactly one of the conditions in a row must be true whenever the system is in one of the modes in that row.

EXAMPLE: For mode_Operating, mode_Hazard, and mode_Shutdown, the controlled variable is determined by the value of mon_Fuel_Level if the fuel level is within range. Thus, you fill the corresponding cell with the condition term_Fuel_Level_Range = withinlimits. The remaining cells in the row correspond to the level being low or high. The function is not defined in mode_Test or mode_BadLevDev, so place an X in these cells (see Table 10-4).

Table 10-4. Condition Table Example (Completed)

Mode	Condition		
mode_Operating mode_Hazard mode_Shutdown	term_Fuel_Level_Range = withinlimits	term_Fuel_Level_Range = levellow	term_Fuel_Level_Range = levelhigh
mode_Test mode_BadLevDev	X	X	X
con_Status	mon_Fuel_Level	low	high

The condition table is easier to check if you use conditions whose consistency is obvious from their very form. For example, conditions of the form A and NOT A clearly exclude one another and add up to true.

10.4.4.2 Specify Periodic Scheduling and Timing

In this activity, you specify the scheduling and timing constraints for a periodic controlled variable function. To perform this activity, you use scheduling and timing information from the system specification as well as information about the environmental quantities from systems specifications, domain experts, hardware constraints, and any other sources describing the controlled variable's periodic timing constraints. You have completed this activity when you have filled out all the relevant parts of the timing and scheduling portion of the behavior specification template (Section 4.4). Use the timing and scheduling portions of the template to guide identification of the timing and scheduling quantities.

For a periodically produced value, use the initiation delay and completion deadline to express the allowed variation between the exact interval given by the period and when the controlled variable's value may be changed. In general, there will be a range of times in a period that is acceptable; this range is expressed by giving the initiation delay and completion deadline.

EXAMPLE: If you give a period of 500 milliseconds, an initiation delay of 50 milliseconds, and a completion deadline of 400 milliseconds, then the software is required to update the controlled variable in the interval between 50 and 400 milliseconds after the start of the 500-millisecond period.

Some variables with periodic constraints are set only under certain conditions. For example, where the user chooses what is currently displayed on a screen, the system only needs to update the displayed values. If this information is not captured in the modes, use the initiation and termination section to define when the values of the controlled variable need to be updated and when they do not. Do this by giving the event that signals when the value must be provided (initiating event) followed by the event that signals when the event no longer needs to be provided (terminating event). If the initiation and termination events depend on the mode, then use an event table to specify the initiation and termination events for each mode.

EXAMPLE: `con_Status` is only required to be updated in `mode_Operating`, `mode_Hazard`, and `mode_Shutdown` but is not updated in `mode_Test` or `mode_BadLevDev`. Provide an event table as shown in Table 10-5.

Table 10-5. Initiation and Termination Events for `con_Status`

Mode	Event	
<code>mode_Operating</code> <code>mode_Hazard</code> <code>mode_Shutdown</code>	ENTERED	X
<code>mode_Test</code> <code>mode_BadLevDev</code>	X	ENTERED
Initiation and Termination	Initiation	Termination

Where the initiation and termination events are provided, the requirement is that the value be periodically updated between the initiating event and the terminating event at the specified interval. If the process continuously runs, you only provide the initiating event (e.g., system initialization).

10.4.5 SPECIFY TOLERANCE CONSTRAINTS

In this activity, you determine and specify the allowed deviation from the controlled variable's ideal value (as expressed by the value function). You perform this activity only for variables that model continuous quantities (e.g., degrees of angle) or otherwise have a notion of more and less accurate values. You do not express tolerance for variables that have only one ideal value (e.g., Boolean or enumerated types). Because the acceptable tolerance is expressed in terms of the controlled variable's value (i.e., the externally visible quantity affected by the system including the output devices), you must determine the allowed tolerance based on the originating system requirements.

You specify the tolerance by defining a function that maps the system state to the corresponding allowed deviation from the ideal. In most cases, this function is a constant; i.e., the controlled variable must be set with the same tolerance at all times. Where the required tolerance is a constant, it is sufficient to specify the error bound on the controlled variable value.

EXAMPLE: FLMS system requirements dictate that the fuel level must be displayed within 0.5 cm of the actual value. Thus, the controlled variable `con_Level_Display` has a constant tolerance function with the value ± 0.5 cm of the ideal value.

Where the required tolerance is not a constant, express the tolerance as a function of the relevant monitored variables, terms, or modes, as necessary. If the tolerance is a direct function of the monitored variables, write the expression. If the tolerance is a function only of the modes, use a selector table. If the tolerance is a function of modes and conditions, use a condition table. Look for tolerance to vary with the mode under the following circumstances:

- **Value-Dependent Variation.** The required tolerance may vary depending on changes in value of the quantities being monitored or controlled. In particular, less tolerance may be required as a measured quantity increases in value.

EXAMPLE: The displayed altitude of an aircraft is a controlled variable. The required tolerance depends on height: the greater the altitude, the less tolerance is needed. You are required to provide the altitude within 2 feet below 500 feet of altitude, within 10 feet from 500 to 5,000 feet of altitude, and within 50 feet above 5,000 feet of altitude.

- **Degraded Modes of Operation.** Systems frequently have degraded modes of operation. Less tolerance may be required in a degraded mode than a normal operating mode. In such a case, the required tolerance should be expressed as a function of the mode.
- **Varying Load.** The required tolerance may vary in proportion to the system load. This can occur when there are fixed computing resources but a varying load at run time (e.g., where a system must track a set of targets). In some cases, you must trade tolerance for the ability to handle an increased load; e.g., the system tracks more targets with less tolerance. In such cases, the tolerance should be expressed as a function of the system loading (e.g., number of targets tracked).

10.5 REFINE MODE CLASSES

In this activity, you complete the definition of the mode classes by completing the specification of the mode machines. To perform this activity, you need the initial mode class definitions you created in

Class Structuring, including the modes and transitions. You also need the definitions of any monitored variables or events that the mode class depends on.

Complete the definition of a mode class by defining all of the transition events that cause mode changes and any auxiliary terms needed in the class's encapsulated information. Use the dependency graph for the mode class to locate the definitions of the terms and events on which the mode machine depends. Use the mode transition graph or mode transition table to help ensure that you have identified all of the necessary events.

10.6 REFINE REMAINING CLASSES

In this activity, you complete the definitions of the encapsulated parts of the boundary and term classes except the specification of the the input and output variables and the IN and OUT relations. To perform this activity, you need the class specifications developed in the Class Structuring activity. In this activity, you do the following:

- Provide or complete the definitions of any encapsulated terms required to support the class interface definitions.
- Provide or complete the definitions of any encapsulated constants needed to support the class interface definitions.
- Provide overview and explanatory material as needed.

You have completed this activity when you have defined all of the encapsulated parts of each class specification except the input and output variables and the IN and OUT relations.

10.7 REVISIT CLASS STRUCTURING

Throughout the Detailed Behavior Specification, you revisit decisions you made in Class Structuring. You develop the Detailed Behavior Specification working from the boundary classes that define the controlled variables back to the classes that define the monitored variables in the controlled variable function domain. You begin by developing the encapsulated information of a boundary class that defines a controlled variable. The detailed specifications for different classes may be developed independently.

As you develop the Detailed Behavior Specification, you may determine that you need terms, monitored variables, or possibly modes that are not currently provided by other classes. In such cases, you either define the needed information in the class you are specifying or you need to revisit the Class Structuring activity as follows:

- Where a new monitored variable or term must be provided, determine whether the definition should be created locally or provided by another class. The variable or term should be provided by another class if its definition requires the information encapsulated by that class or is part of the abstraction provided by that class. Create a new term class or boundary class if necessary to address the Class Structuring criteria.
- Where a monitored variable or term must be changed, you must coordinate the change with any other classes that use the variable or term. Use the dependency graph to determine which class definitions may be affected.

- Where a mode definition must be added or changed, you must revisit all of the classes using the mode class. Use the dependency graph to determine which class definitions may be affected.

As you revisit the Class Structuring activity, record any changes in classes and class dependencies in the affected dependency graph. When you have completed this activity, you should have a set of dependency graphs that is consistent with the information provided and used by each class.

10.8 EVALUATION CRITERIA

When the Detailed Behavior Specification is complete, the required behavior specification for every controlled variable should be complete and consistent. The following sections discuss CoRE's evaluation criteria for a single controlled variable.

10.8.1 COMPLETENESS

For controlled variable functions, there is a well-defined notion of completeness. A function is complete when every value in the domain is mapped to some value in the range. For CoRE functions, this means that every possible value of the relevant monitored variables must be mapped to some value of the controlled variables. Thus, you assess completeness by ensuring that this property holds for each of the functions specifying the controlled variable behavior (value function, tolerance, and timing):

- **Initial Value.** If you supplied a value function, the value function assigns a value on entering the initial system mode. Otherwise, an initiating event and initial value must be given.
- **Mode Class.** If the initial value is none, the controlled variable functions should be the same for all system modes. If one or more modes are specified, the controlled variable functions should be defined for all of the listed mode machines.
- **Sustaining Conditions.** The sustaining conditions must give a set of conditions or true.
- **Value Function.** The function is complete if it defines the behavior over all the possible values of the function's domain. If the controlled variable is a function only of the monitored variables, ensure that the function is specified for all the possible values of those variables. If it is a function of a particular mode class, ensure that the function covers every mode of the relevant mode class. Section 4.2.6 describes the inspection procedure you should follow for each type of table (i.e., condition, event, or selector).

For an enumerated type of controlled variable, the columns should cover every possible enumerated value or there should be a statement that the values are never used. For continuous valued controlled variables, the columns should cover the range of the variable (e.g., $x < 0$, $x = 0$, $x > 0$) or there should be a statement specifying which ranges are never used.

EXAMPLE: The function for the controlled variable `con_Low_Alarm` is defined in terms of the mode class `class_In_Operation`. You verify that every mode of the mode machine appears in some row of the event table defining the value function. You verify that every possible value of `LowAlarm` appears in a column of the event table.

- **Timing and Scheduling Requirements.** Verify that each relevant part of the timing and scheduling template is filled out.

10.8.2 CONSISTENCY

Apply the following consistency checks to the specification of REQ:

- **Value Function.** Check that the controlled variable's value function specifies exactly one required behavior for each state of the system.
 - **Event Table.** An event table describes consistent behavior if the same event does not assign two different values to the controlled variable. You must check that the events in a given row of the table are not the same and do not imply one another (i.e., the events are defined so that the occurrence of one means that the other also occurs). You must also check that events in different columns cannot occur simultaneously or, if they can, that the specification states which column value applies.
 - **Condition Table.** A condition table is consistent if the conditions in a given row of the table are mutually exclusive and the disjunction (i.e., all conditions joined by OR operators) is true. The conditions are mutually exclusive if only one of the conditions in a row can be true at a given time; e.g., the conditions `LevelLow` and `NOT LevelLow` necessarily exclude one another. The conditions add up to true if one of the conditions in the row must be true at a given time; e.g., one of the conditions `LevelLow` or `NOT LevelLow` must be true.
- **Tolerance Function.** Check that the function assigns a tolerance for every state for which the value function assigns a value. You must also check that the expressions do not assign two possible accuracies to the same state (i.e., the expressions are indeed a mathematical function). If event or condition tables are used, apply the same checks that are described for the value function.
- **Timing Constraints.** Check that the timing and scheduling constraints satisfy the properties defined in Section 4.3.

10.9 EXIT CRITERIA

Detailed Behavior Specification is complete when the required behavior for every controlled variable is completely defined.

11. DEFINE HARDWARE INTERFACE

The previous activities described the required behavior of the software in terms of monitored and controlled variables. In the Define Hardware Interface activity, you describe the resources and the input and output variables available to the software to get the values of monitored variables and to set the value of controlled variables, respectively.

11.1 GOALS

The goal of the Define Hardware Interface activity is to complete detailed specification of the boundary classes by defining the input and output variables and the IN and OUT relations. The hardware interface specification defines the resources the software may use and serves as a demonstration that it is feasible to get the monitored values and set the controlled ones. Where the use of particular inputs or outputs represents requirements, this should be stated explicitly and the appropriate traceability specified. The goals of identifying the variables and relations are as follows:

- In defining the input variables, you explicitly identify the input resources available to the software to determine the values of the monitored variables.
- In defining the output variables, you explicitly identify the output resources available to the software to affect the values of the controlled variables.

The IN relation defines the relationship between the monitored variables and the input variables. Similarly, the OUT relation defines the relationship between the output variables and the controlled variables. In defining these relationships, you have the following goals:

- The specification of each monitored variable defines a range of values over which the software must be able to measure the monitored quantity and a precision that expresses how accurately the quantity must be measured. The goal of the IN relation specification is to show that the inputs are sufficient to measure the monitored variables across the required range and to the required precision.
- The specification of each controlled variable defines a range of values over which the software must be able to set the controlled quantity and a precision that expresses how accurately the quantity must be set. The goal of the OUT relation specification is to show that the outputs are sufficient to set the controlled variables across the required range and to the required precision.

A final goal is to specify for the subsequent designers and implementers how the hardware resources are used to get the monitored variables or set the controlled variables. This includes the device protocols, timing characteristics, and data conversion if necessary.

11.2 ENTRANCE CRITERIA

For the Define Hardware Interface activity, you need the following products from previous activities:

- Boundary class definitions
- Input and output device interface specifications from system requirements

11.3 ACTIVITIES

The Define Hardware Interface activity is composed of the following subactivities:

- Assign Input and Output Variables to Boundary Classes
- Define Input and Output Variables
- Define IN and OUT Relations

11.3.1 ASSIGN INPUT AND OUTPUT VARIABLES TO BOUNDARY CLASSES

Allocate the definitions of the input and output variables to the same class that encapsulates the corresponding environmental variable. Examine the environmental variables that you have defined. For each monitored variable, decide which input variables the software can use to determine the variable's value. For each controlled variable, decide which output variables the software can use to set its value. Encapsulate the variable in the class that contains the definition of the environmental variable.

Because the boundary class that contains the monitored variables also encapsulates all the input variables that can determine the values of the monitored variables, the class encapsulates the IN relation for the input variables. Define the IN relation as part of the encapsulated information of the class. Similarly, define the OUT relation as part of the encapsulated information for the class defining the corresponding output.

If the software uses the variable to access environmental variables defined in different classes, reassess the decision to assign the environmental variables to different boundary classes. You may decide that the environmental variables should be assigned to the same class.

11.3.2 DEFINE INPUT AND OUTPUT VARIABLES

In the Define Input and Output Variables activity, you specify the detailed characteristics of the input and output variables and describe how they are accessed by the software. Identify the input and output variables by denoting each resource that independently changes value (Heninger 1980) by a variable.

Create the variable specification by filling out the applicable parts of the input and output variable template as shown in Table 11-1. First, provide a unique name for each variable. Users of the requirements need to know with what hardware the variable is associated, how to read or write the variable, the values that they can read or write, and the representation of the values that the variable assumes (typically a bit pattern). Table 11-2 is a sample definition of a numeric input variable from the FLMS. The abstract values of the input variable `ln_Diff_Press` are integers in the range 0 to 255. An 8-bit unsigned integer represents the values. Table 11-3 is a sample definition of a nonnumeric output variable.

The abstract values of out_Shutdown are true and false. Zero in bit 1 of the byte represents true. One in bit 1 represents false.

Table 11-1. Input and Output Variable Template

Input and Output Variable	Definition
Acronym	Unique identifier for the variable
Hardware	The hardware unit with which the variable is associated
Values	The abstract values you can read or write: • Numeric. Specify range and resolution of numeric variables. • Nonnumeric. List the abstract values of nonnumeric variables.
Data Transfer	How to read or write the variable
Data Representation	How the variable assumes the abstract values are represented

Table 11-2. Sample Definition of Input Variable Diff_Press

Acronym	in_Diff_Press
Hardware	Differential Pressure Unit
Values	[0..255]
Data Transfer	ADC(0)
Data Representation	8-bit unsigned integer

Table 11-3. Sample Definition of Output Variable Shutdown Signal

Acronym	out_Shutdown
Hardware	Pump Shutdown Relay
Values	False (1b) True (0b)
Data Transfer	PortC
Data Representation	Bit 1 of byte

The information you need to complete the variable's definitions should come from the input and output interface specifications. You often will have to manipulate the information to obtain a form suitable for completing the variable definition. If the information that you need to complete the definition is not included in the input and output interface specifications, you may have to interview the engineers who designed the input and output interface or experiment with the hardware.

11.3.3 DEFINE IN AND OUT RELATIONS

The goal of this activity is to define the relationship between the monitored and controlled variables and the input and output variables, respectively. As part of specifying this relationship, you establish

that the input variables are sufficient to measure the monitored quantities and that the outputs are sufficient to set the controlled quantities. For numeric quantities, you also define the conversion.

11.3.3.1 Define IN for a Monitored Variable

Define an IN relation for each monitored variable. You define the relation to show that the software can determine the value of the monitored variable, using the available inputs, to that variable's range and precision requirements.

Use the monitored variable's definition to determine the range of values and precision. Demonstrate that the available inputs meet the monitored variable's range requirements by giving a mapping from the range of the monitored variable to the corresponding inputs. For discrete monitored variables (e.g., Boolean or enumerated types), this requires showing the exact mapping between values. For continuous quantities (e.g., degrees, feet), show the mapping over the range of values.

For continuous quantities, it is usually easier to describe the expected value, error, and delay separately. In this case, you define its expected value as a function of the values of monitored variables. You then give the minimum accuracy and maximum delay associated with the input device.

Figure 11-1 illustrates an example of an IN relation for the input `in_Diff_Press`. `in_Diff_Press` is used to get the value of the monitored variable `mon_Fuel_Level`.

The following table describes how `in_Diff_Press` reflects the value of `mon_Fuel_Level`. The device is calibrated so that it has a value in the range [1..254] when `mon_Fuel_Level` is between 13.0 cm (`const_LCB`) and 27.0 cm (`const_UCB`). A value of 0 for `in_Diff_Press` indicates that `mon_Fuel_Level` has fallen below the lower calibration bound (`const_LCB`) or that the device has failed. A value of 255 indicates that `mon_Fuel_Level` has risen above the upper calibration bound (`const_UCB`) or that the device has failed.

Determining the Value of `in_Diff_Press`

Variable	in_Diff_Press	
Tolerance	0.1 cm	
Delay	0.2 s	
Condition		
mon_Fuel_Level < const_LCB	const_LCB ≤ mon_Fuel_Level ≤ const_UCB	mon_Fuel_Level > const_UCB
in_Diff_Press = 0	in_Diff_Press in [1 .. 254]	in_Diff_Press = 255

The table describes how `in_Diff_Press` reflects the value of `mon_Fuel_Level_Unknown`. When `in_Diff_Press` has a value of either 0 or 255, we assume that `in_Diff_Press` has failed and that the FLMS is unable to determine the value of `mon_Fuel_Level`.

The following expression describes how the value of `mon_Fuel_Level` can be calculated from the value of `in_Diff_Press`:

$$\text{mon_Fuel_Level} = \frac{\text{in_Diff_Press} - 1}{253} \times (\text{const_UCB} - \text{const_LCB}) + \text{const_LCB} \pm 0.28 \text{ cm}$$

Figure 11-1. IN Relation for `in_Diff_Press`

After you have defined the mapping from the range of the monitored variable to the inputs, you can define the conversion. The expression following the table defines the conversion from the values of `in_Diff_Press` to the values of `mon_Fuel_Level`. The specification of `in_Diff_Press` is completed by specifying the delay associated with getting the inputs and the accuracy relative to the monitored quantity. The specification for `in_Diff_Press` is given in Figure 11-2.

The input variable `in_Diff_Press` has a precision of

$$\frac{\text{const_UCB} - \text{const_LCB}}{254} = \frac{27.0 - 13.0}{254} = 0.0551\text{cm}$$

which is sufficient to represent `mon_Fuel_Level` to the required 0.5 cm. The maximum error introduced by the delay of 0.2 s is

$$0.2 \text{ s} \times \text{const_Max_Fuel_Rate} = 0.2\text{s} \times 0.375\text{cm/s} = 0.075\text{cm}$$

The maximum error introduced by the device error and delay of `in_Diff_Press` is the sum of the two, 0.175 cm, which satisfies the needs of the FLMS to determine the value of `mon_Fuel_Level` to 0.5 cm.

Figure 11-2. Accuracy Specification for `in_Diff_Press`

11.3.3.2 Define OUT for a Controlled Variable

Define an OUT relation for each controlled variable. You define OUT to show that the software can set the values of the controlled variable, using the available outputs, to that variable's range and precision requirements.

Use the controlled variable's definition to determine the range of values and precision. Demonstrate that the available outputs meet the controlled variable's value requirements by giving a mapping from the set of possible outputs to the set of possible controlled variable values. For discrete controlled variables (e.g., Boolean or enumerated types), this requires showing the exact mapping between values. For continuous quantities (e.g., angle of a flap, degrees of rotation of a dial), show the mapping over the range of values.

For continuous quantities, it is usually easier to describe the expected value, error, and delay separately. In this case, you define its expected value of the controlled variable as a function of the output values. You then give the minimum accuracy and maximum delay associated with the output devices.

Any of the CoRE notations can be used to define the OUT relation. If the controlled variable is a continuous function of the output variables, use standard mathematical notation. If the controlled variable depends on conditions or modes, use the condition or event tables described in Section 4.1.6.

Table 11-4 illustrates the use of a condition table to specify the mapping from the output variable `out_Shutdown` to the values of the Boolean controlled variable `Shutdown_Relay`. Because the values are discrete, the table gives the mapping for each value; this shows that the available output values cover the range of the controlled variable. The table gives the expected value of the controlled variable `con_Shutdown_Relay` as a function of the output variable `out_Shutdown`. The first column of the table indicates that if the output variable `out_Shutdown` = true, then `con_Shutdown_Relay` = open.

The second column of the table specifies that if `out_Shutdown = false`, then `con_Shutdown_Relay` is set to closed.

Table 11-4. Sample OUT Relation for Controlled Variable `con_Shutdown_Relay`

Condition	
<code>out_Shutdown = true</code>	<code>out_Shutdown = false</code>
<code>con_Shutdown_Relay = open</code>	<code>con_Shutdown_Relay = closed</code>

After you have specified the expected value of the controlled variable as a function of the output variables, define the error and delay introduced by the output devices. Discrete values have no associated error. For continuous values, these are typically constants. Table 11-5 indicates that if the software sets the value of the output variable `out_Shutdown` to `true`, for example, then the controlled variable `con_Shutdown_Relay` must assume the value `open` within 10 milliseconds. Because `Shutdown_Relay` is an enumerated variable, there is no tolerance associated with it (indicated by NA in the table).

Table 11-5. Sample Tolerance and Delay for Controlled Variable `con_Shutdown_Relay`

Error	NA
Delay	10 ms

For continuous values, you should also specify the conversion from the output to the values of the controlled variable.

Each controlled variable's REQ relation defines an associated tolerance in value and time. You must show that the outputs can be set with sufficient accuracy to satisfy the tolerance in value. You must show that the maximum delay in setting the values is less than the delay allowed by the REQ relation.

11.4 EVALUATION CRITERIA

This section describes how to evaluate the consistency and completeness of the hardware interface that you have defined. You should be able to answer "yes" to each of the questions listed below:

- For each of the input and output variables that you have identified:
 - Is the variable template filled in?
- For each of the input variables that you have defined:
 - Is there at least one IN relation?
 - Are all of the monitored variables in the domain of IN defined in the class?
- For each of the controlled variables that you have defined:
 - Is there at least one OUT relation?
 - Are all of the output variables in the domain of OUT encapsulated by the boundary class?

- Can each output variable be used to set the value of at least one controlled variable?
- Can the value of each monitored variable be determined from the IN relation?

11.5 EXIT CRITERIA

Define Hardware Interface is complete when you can answer "yes" to all of the questions listed in Section 11.4.

This page intentionally left blank.

12. ANALYZING A CoRE SPECIFICATION

CoRE's models, the use of mathematics, and the use of formats for capturing requirements all support analysis of a CoRE specification for completeness and consistency. Detailed procedures for analyzing the products of individual activities have been provided in the detailed process sections (Sections 8 through 11). This section summarizes the analysis process and its goals.

This section describes a series of steps for analyzing completeness and consistency of a CoRE specification. Wherever there are guidelines not supported by strict rules, you are given heuristics.

12.1 MONITORED AND CONTROLLED VARIABLES

All monitored and controlled variables that are measured or affected by the system have been defined in a boundary class.

Completeness

- There is a definition for each monitored or controlled variable in the specification. Each variable is defined in a boundary class.
- All the relevant attributes for all monitored and controlled variables have been specified.

Consistency

- There is only one definition for each monitored or controlled variable in the specification.
- The attributes of each variable (i.e., type, range, and precision) are consistent with any NAT constraints for the variable.
- The context diagram shows one incoming arrow from an entity to the software for each monitored variable and one outgoing arrow from the software to an entity for each controlled variable.
- The dependency graph shows one incoming arrow for each monitored variable terminating at the boundary class that defines the variable and one outgoing arrow for each monitored variable originating at the boundary class that defines the variable.

12.2 CONTROLLED VARIABLE FUNCTIONS

The REQ relation has been fully and consistently defined for every controlled variable.

Completeness

- There is a REQ relation defined for each controlled variable in the boundary class that defines the controlled variable. All parts of the relation are defined, in particular:

- There is a value function defined.
- Where applicable, there is an accuracy tolerance function defined.
- There is a timing tolerance function.

- The definition of each function is mathematically complete according to the detailed process.
- The function is defined for every mode of the applicable mode machine. It depends on every state permitted by the NAT relation.

Consistency

- The parts of the REQ relation definition are mutually consistent, and the relation is consistent with the controlled variable definition (i.e., with the type, range, and precision of the variable).
- The value function maps each value of the monitored variables in its domain to, at most, one value in the range.
- Every mode used is defined by exactly one mode machine.
- Every term used is defined in the encapsulating class or on the interface of some other class.
- There is an arrow in the dependency graph for each term defined by one class and used by another class. The arrow originates from the defining class and terminates at the using class.

12.3 TERMS AND MODES

Every term is fully defined and properly used. Every mode class is completely defined.

Completeness

- Every term has a definition.
- Every mode is defined by exactly one mode machine. The definition of each mode machine is complete as specified in the detailed process for evaluating mode machines.

Consistency

- Any term defined in one class and used in another is provided on the interface section of the defining class.
- Any term defined in the encapsulated part of a class is not used outside the class.
- Every term is defined exclusively as an expression on monitored variables, constants, and other terms.
- The dependency graph shows an arrow for every term or mode class used by the defining class but defined by another class. The arrow originates in the defining class and terminates in the using class.

- The definition of every mode machine satisfies the criteria for consistency in the detailed process for evaluating mode classes.

12.4 IN AND OUT RELATIONS

There is an IN relation defined for every monitored variable and an OUT relation defined for every controlled variable. The IN, OUT, and the input and output variable specifications are complete and internally consistent.

Completeness

- There is at least one IN relation defined for each input variable in the specification. The definition of each IN relation is complete and internally consistent.
- There is at least one OUT relation defined for each output variable in the specification. The definition of each OUT relation is complete and internally consistent.
- There is a definition for each input or output variable in the specification. The variable specification is complete according to the detailed process description (i.e., all applicable parts of the template are filled in).

Consistency

- There is at least one IN relation for each input variable and at least one OUT relation for each output variable.
- Each IN and OUT relation is complete and consistent according to the detailed process description.
- The IN relation and input variable are defined in the class that defines the corresponding monitored variable.

12.5 GLOBAL CHECKS

Make additional checks for consistency or completeness between CoRE relations.

Completeness

- Check that every monitored variable and every term is used in at least one REQ relation.

Consistency

- For a given controlled variable, the delays allowed by the IN and OUT relations must be consistent with the timing tolerance for the REQ relation; i.e., the maximum delay required to read the inputs plus the maximum delay required to set the outputs must be less than the maximum delay allowed by the controlled variable tolerance.

This page intentionally left blank.

APPENDIX A. SOFTWARE REQUIREMENTS FOR THE FUEL LEVEL MONITORING SYSTEM

A.1 INTRODUCTION

The design of the FLMS comprises automatic or manual control mechanisms (engine and fuel-level controls) and safety monitoring devices. The safety monitoring devices include fuel gauges and gauge cocks that convey the fuel level in the tank, fusible plugs or fuse alarms that alert the operator when the fuel level is too low, and fuel flow rate gauges or other gauges showing the engine operating condition. The FLMS is intended to replace or complement the above-mentioned devices. It monitors and displays the fuel level in the tank and provides visible and audible alarms for high and low fuel levels. With the currently selected hardware configuration, fuel level is displayed in a window on a CRT display, two "annunciation" windows on the CRT provide visible indication of exceeded fuel-level limits, and the computer's speaker provides an audible alarm.

A.2 REQUIREMENTS FOR THE FUEL LEVEL MONITORING SYSTEM

- (1) The FLMS shall monitor the fuel level in the tank.
- (2) When the level in the tank exceeds the upper or lower limits, an alarm is triggered. (3) If the fuel level is out of limits for more than the shutdown lock time, the pump shall be shut down. (4) It shall be possible to restart the system when the fuel levels are again within limits.
- (5) If the system is unable to determine the fuel level of the tank, the system shall notify the operator of the condition and shut down the pump.
- (6) A capability to conduct system self-testing shall be provided. (7) System self-test shall be possible at any time. (8) On initiation of a self-test, the system shall shut down the pumps. (9) A self-test shall not keep the system offline for longer than 15 seconds. (10) At the conclusion of a self-test, it shall be necessary to restart the system.
- (11) The operator shall be provided with reset and test switches. (12) The system shall display fuel level and status alarms to the operator. (13) Indications of low or high fuel levels (hazardous conditions) or unknown fuel levels shall be presented to the operator. (14) Whenever there is a hazardous condition or an unknown, the system shall provide audible and visual alarms.

This page intentionally left blank.

**APPENDIX B. CoRE SPECIFICATION OF THE
SOFTWARE REQUIREMENTS FOR THE FUEL
LEVEL MONITORING SYSTEM**

This page intentionally left blank.

APPENDIX B CONTENTS

B.1 System Context	B-5
B.2 Fuel Level Monitoring System Dependency Graph	B-6
B.3 mode_class_In_Operation	B-7
B.3.1 Class Interface	B-7
B.3.2 Encapsulated Information	B-7
B.3.3 Traceability	B-8
B.4 class_Fuel_Tank	B-9
B.4.1 Class Interface	B-9
B.4.1.1 NAT Relation	B-10
B.4.2 Encapsulated Information	B-10
B.4.2.1 Input Variables	B-10
B.4.2.2 IN Relation	B-11
B.4.3 Traceability	B-12
B.5 class_Pump	B-13
B.5.1 Class Interface ..	B-13
B.5.2 Encapsulated Information	B-13
B.5.2.1 REQ Relation	B-13
B.5.2.2 Output Variables	B-14
B.5.2.3 OUT Relation	B-14
B.5.3 Traceability	B-14
B.6 class_Time	B-15
B.6.1 Class Interface	B-15
B.6.2 Encapsulated Information	B-15
B.6.2.1 Input Variables	B-15
B.6.2.2 IN Relation	B-15

B.6.3 Traceability	B-15
B.7 class_Operator	B-16
B.7.1 Class Interface	B-16
B.7.2 Encapsulated Information	B-16
B.7.3 Traceability	B-16
B.8 class_Operator_Communication	B-17
B.8.1 Class Interface	B-17
B.8.2 Encapsulated Information	B-17
B.8.2.1 REQ Relation	B-18
B.8.2.2 Output Variables	B-21
B.8.2.3 OUT Relation	B-21
B.8.3 Traceability	B-22
B.9 class_Switch	B-23
B.9.1 Class Interface	B-23
B.9.2 Encapsulated Information	B-23
B.9.2.1 Input Variables	B-23
B.9.2.2 IN Relation	B-24
B.9.3 Traceability	B-24
B.10 Safety Requirements	B-24
B.11 Security Requirements	B-25
B.12 Other Requirements	B-25
APPENDIX B INDEX	B-26

APPENDIX B FIGURES

Figure B-1. Fuel Level Monitoring System: Context Diagram	B-5
Figure B-2. Fuel Level Monitoring System Dependency Graph	B-6
Figure B-3. Fuel Level Monitoring System Pump and Tank Configuration (Front View) ..	B-9

B.1 SYSTEM CONTEXT

Figure B-1 illustrates the context of the FLMS. The bubble represents the FLMS system. Inputs to the FLMS are the monitored variables. Outputs are the controlled variables.

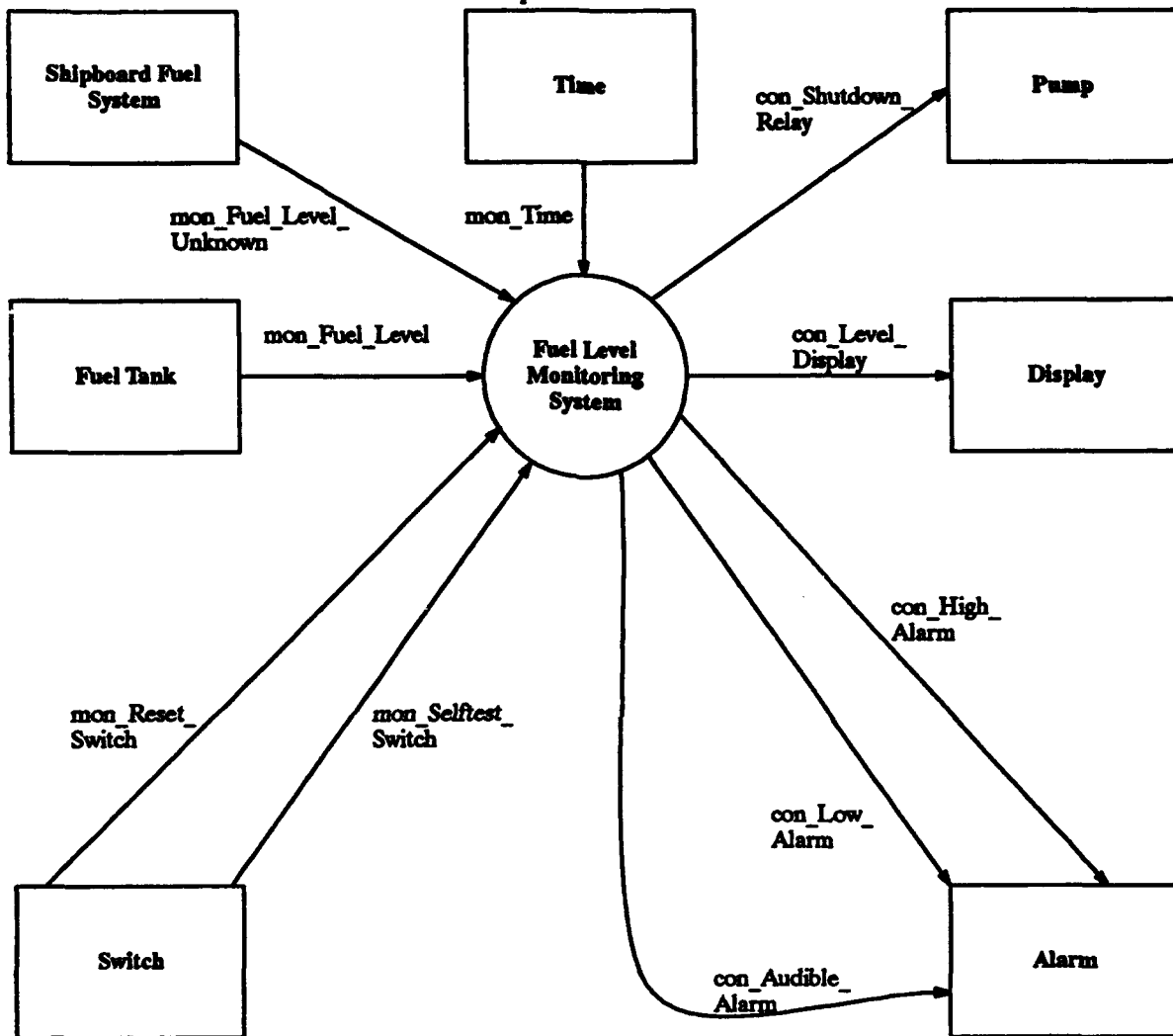


Figure B-1. Fuel Level Monitoring System: Context Diagram

B.2 FUEL LEVEL MONITORING SYSTEM DEPENDENCY GRAPH

Figure B-2 illustrates the dependency graph of this specification. It is the detailed view of the FLMS bubble in Figure B-1. Arcs entering the diagram from outside represent monitored variables, and arcs leaving the diagram represent controlled variables. There is an internal arc where a class uses requirements information defined by the interface of another class.

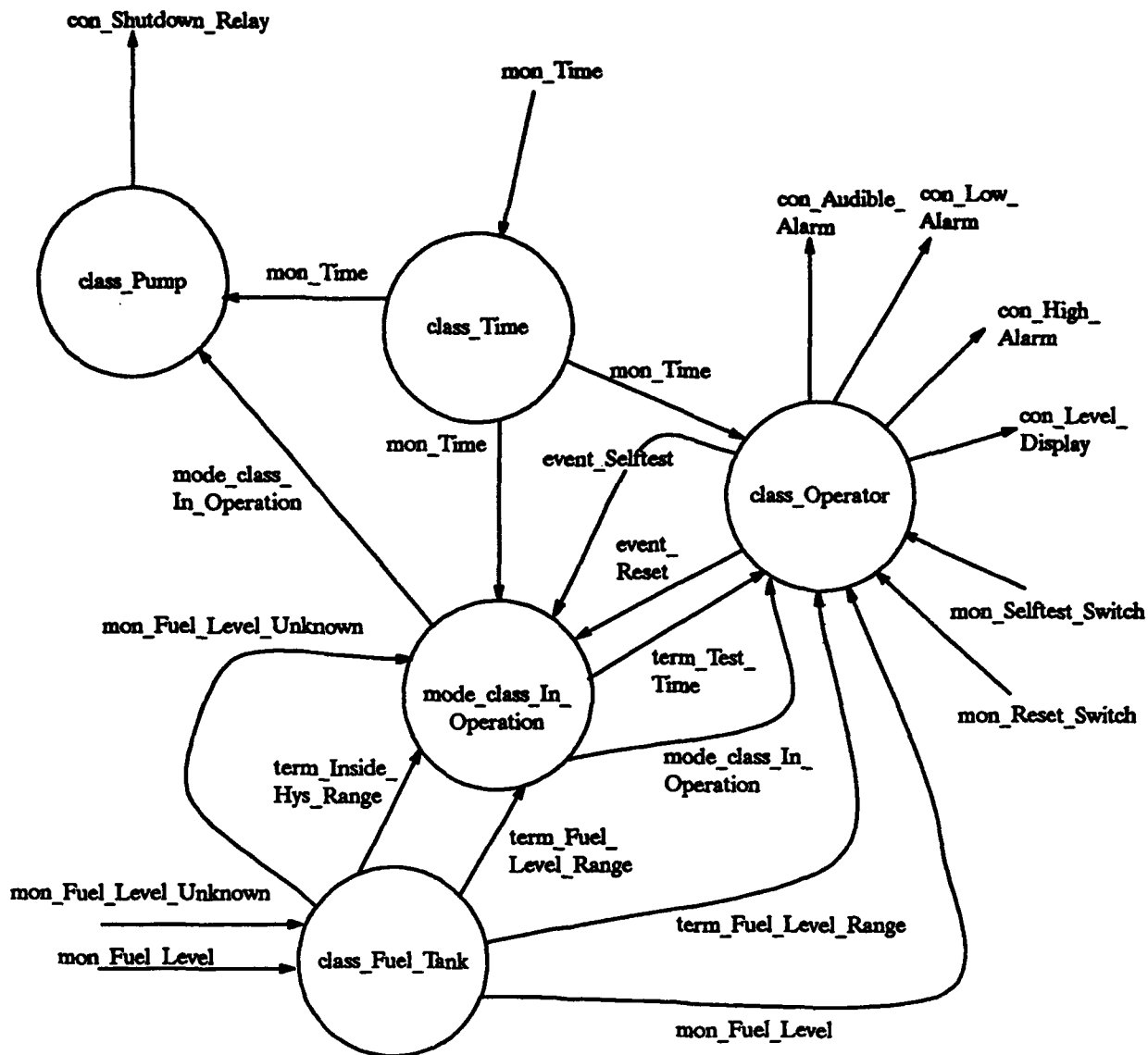


Figure B-2. Fuel Level Monitoring System Dependency Graph

B.3 MODE_CLASS_IN_OPERATION

mode_class_In_Operation defines the FLMS modes. It encapsulates the rules for determining the current mode and the events that trigger transitions among the modes.

B.3.1 CLASS INTERFACE

Modes Other Classes Are Allowed to Use

Mode

mode_Operating

mode_Hazard

mode_Shutdown

mode_Test

mode_BadLevDev

Constants, Events, and Terms Other Classes Are Allowed to Use

Name

term_Test_Time

B.3.2 ENCAPSULATED INFORMATION

Constant, Event, and Term Glossary

Name	Type	Values	Definition
const_Shutdown_Lock_Time	TIME	2.0 s	
const_Max_Test_Time	TIME	14.0 s	
term_Test_Time	TIME		DURATION(INMODE(mode_Test))

Mode Transitions: mode_class_In_Operation

Current Mode	term_Inside_Hys_Range	DURATION(INMODE(mode_Hazard)) > = const_Shutdown_Lock_Time	event_Reset	term_Fuel_Level_Range = withinlimits	DURATION(INMODE(mode_Test)) > = const_Max_Test_Time	event_Selftest	mon_Fuel_Level_Unknown	New Mode
mode_Operating				@F				mode_Hazard
						@T		mode_Test
							@T	mode_BadLevDev
mode_Hazard	@T	f						mode_Operating
		@T						mode_Shutdown
						@T		mode_Test
							@T	mode_BadLevDev
mode_Shutdown	t		@T					mode_Operating
						@T		mode_Test
							@T	mode_BadLevDev
mode_Test					@T			mode_Shutdown
							@T	mode_BadLevDev
mode_BadLevDev								

B.3.3 TRACEABILITY

The following capabilities from Software Requirements for the FLMS (see Appendix A) are fully or partially satisfied by this class:

- (3) If the fuel level is out of limits for more than the shutdown lock time, the pump shall be shut down.
- (9) A self-test shall not keep the system offline for longer than 15 seconds.
- (10) At the conclusion of a self-test, it shall be necessary to restart the system.

B.4 CLASS_FUEL_TANK

class_Fuel_Tank provides the information needed to determine the current fuel level; whether the fuel level is above, below, or within safe limits; and whether the fuel level is within the hysteresis bounds. It encapsulates the constants and rules for determining whether the fuel level is within safe or hysteresis limits. It also encapsulates how the software can determine the values of mon_Fuel_Level and mon_Fuel_Level_Unknown.

B.4.1 CLASS INTERFACE

Environmental Variable Glossary

Name	Type	Values	Physical Interpretation
mon_Fuel_Level	LENGTH	0.0..30.0	Level of fuel in the tank, in centimeters (cm), along the vertical axis on the left side of the tank, 5 cm from the front edge. The level is measured with respect to the scale (see Figure B-3).
mon_Fuel_Level_Unknown	BOOLEAN	false	The system is able to obtain the information required to determine the value of fuel level to the required accuracy.
		true	The system is unable to obtain the information required to determine the value of fuel level to the required accuracy.

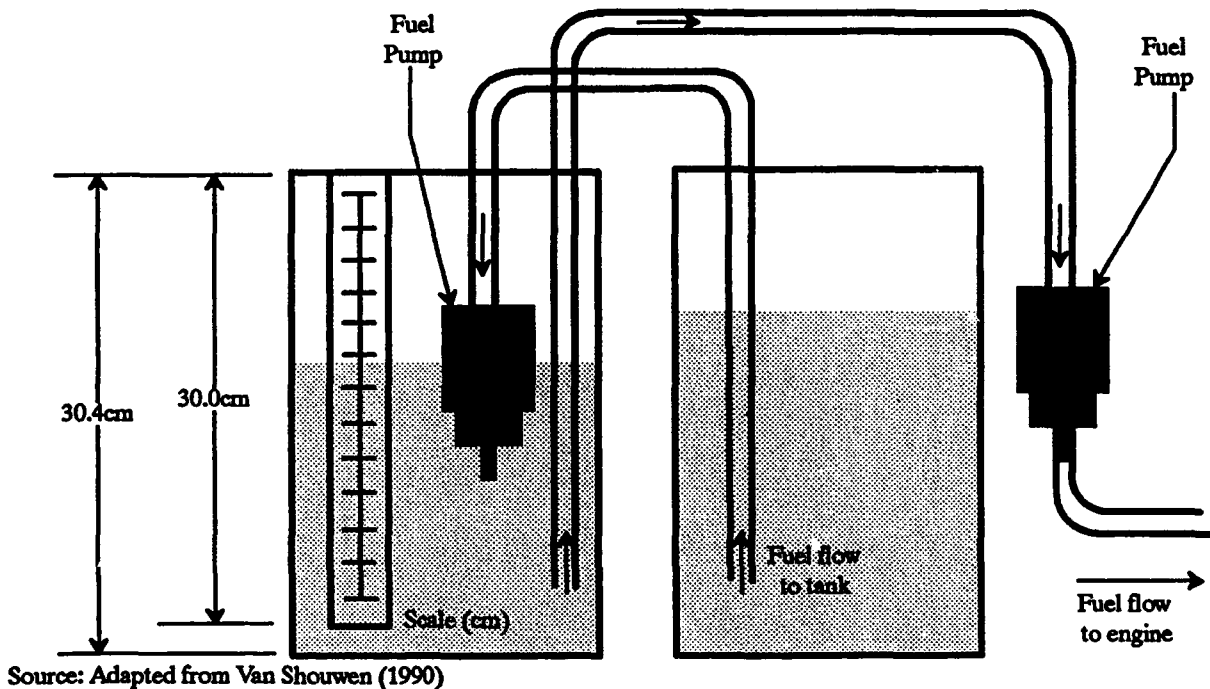


Figure B-3. Fuel Level Monitoring System Pump and Tank Configuration (Front View)

Constants, Events, and Terms Other Classes Are Allowed to Use**Name****term_Fuel_Level_Range****term_Inside_Hys_Range****B.4.1.1 NAT Relation**

$$\left| \frac{d\text{mon_Fuel_Level}}{d\text{mon_Time}} \right| \leq \text{const_Max_Level_Rate}$$

$$-0.4 \text{ cm} \leq \text{mon_Fuel_Level} \leq 30.0 \text{ cm}$$

B.4.2 ENCAPSULATED INFORMATION**Constant, Event, and Term Glossary**

Name	Type	Value	Definition
const_High_Fuel_Limit	LENGTH	26.0 cm	
const_Hysteresis	LENGTH	0.5 cm	
const_LCB	LENGTH	13.0 cm	
const_Low_Fuel_Limit	LENGTH	14.0 cm	
const_Max_Level_Rate	LENGTH/TIME	0.375 cm/s	
const_UCB	LENGTH	27.0 cm	
term_Fuel_Level_Range	ENUMERATED	levellow	$\text{mon_Fuel_Level} \leq \text{const_Low_Fuel_Limit}$
		withinlimits	$\text{const_Low_Fuel_Limit} < \text{mon_Fuel_Level} < \text{const_High_Fuel_Limit}$
		levelhigh	$\text{mon_Fuel_Level} \geq \text{const_High_Fuel_Limit}$
term_Inside_Hys_Range	BOOLEAN	true	$(\text{const_Low_Fuel_Limit} + \text{const_Hysteresis}) < \text{mon_Fuel_Level} < (\text{const_High_Fuel_Limit} - \text{const_Hysteresis})$
		false	$(\text{const_Low_Fuel_Limit} + \text{const_Hysteresis}) \geq \text{mon_Fuel_Level} \text{ OR } \text{mon_Fuel_Level} \geq (\text{const_High_Fuel_Limit} - \text{const_Hysteresis})$

B.4.2.1 Input Variables**Differential Pressure Unit**

Acronym	in_Diff_Press
Hardware	Differential Pressure Unit
Values	[0..255]
Data Transfer	ADC(0)
Data Representation	8-bit unsigned integer

B.4.2.2 IN Relation

in_Diff_Press reflects the value of mon_Fuel_Level. The device is calibrated so that it has a value in the range [1..254] when mon_Fuel_Level is between 13.0 cm (const_LCB) and 27.0 cm (const_UCB). A value of 0 for in_Diff_Press indicates that mon_Fuel_Level has fallen below the lower calibration bound (const_LCB) or that the device has failed. A value of 255 indicates that mon_Fuel_Level has risen above the upper calibration bound (const_UCB) or that the device has failed. When we cannot distinguish failure of in_Diff_Press from mon_Fuel_Level going outside of the calibration bounds, we assume that the device has failed.

Determining the Value of in_Diff_Press

Variable	in_DiffPress
Error	0.1 cm
Delay	0.2 s

Condition		
mon_Fuel_Level < const_LCB	const_LCB ≤ mon_Fuel_Level ≤ const_UCB	mon_Fuel_Level > const_UCB
in_Diff_Press = 0	in_Diff_Press in [1 .. 254]	in_Diff_Press = 255

in_Diff_Press reflects the value of mon_Fuel_Level_Unknown. When in_Diff_Press has a value of either 0 or 255, we assume that in_Diff_Press has failed and that the FLMS is unable to determine the value of mon_Fuel_Level.

Determining the Value of in_Diff_Press

Variable	in_Diff_Press
Error	N/A
Delay	0.2 s

Condition	
mon_Fuel_Level_Unknown	NOT mon_Fuel_Level_Unknown
in_Diff_Press = 0 OR in_Diff_Press = 255	in_Diff_Press in [1 .. 254]

The following expression describes how the value of mon_Fuel_Level can be calculated from the value of in_Diff_Press:

$$\text{mon_Fuel_Level} = \frac{\text{in_Diff_Press}-1}{253} \times (\text{const_UCB} - \text{const_LCB}) \div \text{const_LCB} \pm 0.28 \text{ cm}$$

The input variable in_Diff_Press has a precision of

$$\frac{\text{const_UCB}-\text{const_LCB}}{254} = \frac{27.0-13.0}{254} = 0.0551 \text{ cm}$$

which is sufficient to represent mon_Fuel_Level to the required 0.5 cm. The maximum error introduced by the delay of 0.2 s is

$$0.2 \text{ s} \times \text{const_Max_Fuel_Rate} = 0.2 \text{ s} \times 0.375 \text{ cm/s} = 0.075 \text{ cm}$$

The maximum error introduced by the device error and delay of in_Diff_Press is the sum of the two, 0.175 cm, which satisfies the needs of the FLMS to determine the value of mon_Fuel_Level to 0.5 cm.

B.4.3 TRACEABILITY

The following capability from Software Requirements for the FLMS (see Appendix A) is fully or partially satisfied by this class:

- (1) The FLMS shall monitor the fuel level in the tank.

B.5 CLASS_PUMP

class_Pump encapsulates the rules that determine the required behavior of the pump and the mechanisms available to the software to support the required behavior.

B.5.1 CLASS INTERFACE

None.

B.5.2 ENCAPSULATED INFORMATION

Environmental Variable Glossary

Name	Type	Values	Physical Interpretation
con_Shutdown_Relay	ENUMERATED	closed	The fuel pump shutdown relay is closed, and the fuel pump is enabled.
		open	The fuel pump shutdown relay is open, and the fuel pump is disabled.

B.5.2.1 REQ Relation

The FLMS disables the pump under unsafe unconditions and tests the mechanism for disabling the pump.

Behavior of con_Shutdown_Relay

Variable	con_Shutdown_Relay
Initial Value	See value function
Mode Class	mode_class_In_Operation
Sustaining Condition	true

Mode	Events	
mode_Operating	X	ENTERED
mode_Hazard	X	X
mode_Shutdown	ENTERED	X
mode_BadLevDev		
mode_Test		
con_Shutdown_Relay	open	closed
Tolerance	N/A	
Initiation Delay	0 ms	
Completion Deadline	50 ms	

B.5.2.2 Output Variables**Shutdown Signal**

Acronym	out_Shutdown
Hardware	Pump Shutdown Relay
Values	false (1b) true (0b)
Data Transfer	PortC
Data Representation	Bit 1 of byte

B.5.2.3 OUT Relation**Setting the Value of con_Shutdown_Relay**

Variable	con_Shutdown_Relay
Error	N/A
Delay	10 ms

Condition	
out_Shutdown = true	out_Shutdown = false
con_Shutdown_Relay = open	con_Shutdown_Relay = closed

B.5.3 TRACEABILITY

The following capabilities from Software Requirements for the FLMS (see Appendix A) are fully or partially satisfied by this class:

- (3) If the fuel level is out of limits for more than the shutdown lock time, the pump shall be shut down.
- (4) It shall be possible to restart the system when the fuel levels are again within limits.
- (5) If the system is unable to determine the fuel level of the tank, the system shall notify the operator of the condition and shut down the pump.
- (6) A capability to conduct system self-testing shall be provided.
- (7) System self-test shall be possible at any time.
- (8) On initiation of a self-test, the system shall shut down the pumps.

B.6 CLASS_TIME

class_Time encapsulates the mechanisms available to the software to determine the values of the monitored variables associated with the passage of time.

B.6.1 CLASS INTERFACE

Environmental Variable Glossary

Name	Type	Values	Physical Interpretation
mon_Time	TIME	$0 \dots 31.536 \times 10^9$	Time, in milliseconds (ms), since system initialization

B.6.2 ENCAPSULATED INFORMATION

B.6.2.1 Input Variables

System Timer

Acronym	in_ClkPulse
Hardware	System timer
Values	None
Data Transfer	Interrupt (1Ch)
Data Representation	None

B.6.2.2 IN Relation

Determining the Value of in_Clk_Pulse

Variable	in_Clk_Pulse
Error	N/A
Delay	1 ms

Events

@T(mon_Time mod 55 = 0)	@F(mon_Time mod 55 = 0)
in_Clk_Pulse <i>defined</i>	in_Clk_Pulse <i>undefined</i>

B.6.3 TRACEABILITY

The following capabilities from Software Requirements for the FLMS (see Appendix A) are fully or partially satisfied by this class:

- (3) If the fuel level is out of limits for more than the shutdown lock time, the pump shall be shut down.
- (9) A self-test shall not keep the system off line for longer than 15 seconds.

B.7 CLASS_OPERATOR

class_Operator encapsulates how the FLMS is required to interact with the operator and the mechanisms available to the software to support the required interactions.

B.7.1 CLASS INTERFACE

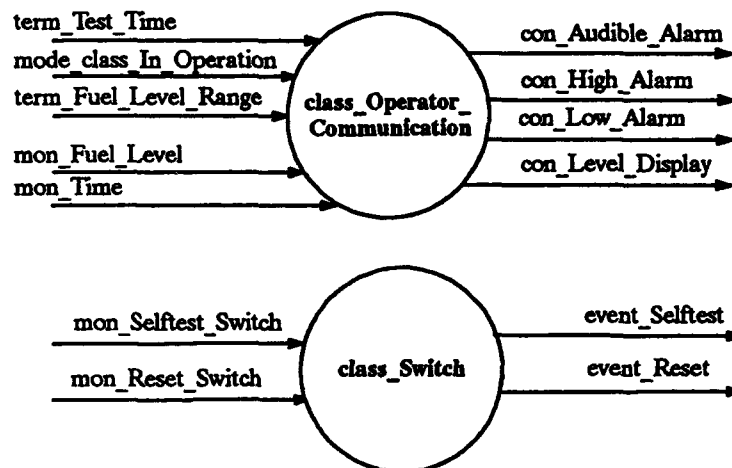
Constants, Events, and Terms Other Classes Are Allowed to Use

Name

event_Selftest

event_Reset

B.7.2 ENCAPSULATED INFORMATION



Dependency Graph for Operator Interface

B.7.3 TRACEABILITY

See encapsulated classes.

B.8 CLASS_OPERATOR_COMMUNICATION

`class_Operator_Communication` encapsulates the rules that determine how the FLMS communicates to the operator and the mechanisms available to the software to support this communication.

B.8.1 CLASS INTERFACE

None.

B.8.2 ENCAPSULATED INFORMATION

Environmental Variable Glossary

Name	Type	Values	Physical Interpretation
con_High_Alarm	ENUMERATED	on	The alarm labeled "FUEL LEVEL HIGH" is visible to the operator.
		off	The alarm labeled "FUEL LEVEL HIGH" is not visible to the operator.
con_Level_Display	REAL	0.0..99.9	The value conveyed by the display labeled "FUEL LEVEL."
con_Low_Alarm	ENUMERATED	on	The alarm labeled "FUEL LEVEL LOW" is visible to the operator.
		off	The alarm labeled "FUEL LEVEL LOW" is not visible to the operator.
con_Audible_Alarm	ENUMERATED	sound	The audible alarm is sounding.
		silent	The audible alarm is silent.

Constant, Event, and Term Glossary

Name	Type	Values	Definition
const_High_Alarm_Col	INTEGER	9	
const_High_Alarm_Row	INTEGER	17	
const_Level_Display_Row	INTEGER	6	
const_Low_Alarm_Col	INTEGER	29	
const_Low_Alarm_Row	INTEGER	17	
const_MaxCol	INTEGER	39	
const_MaxRow	INTEGER	24	
const_MinCol	INTEGER	0	
const_MinRow	INTEGER	0	
term_Digit(x, k)	CHARACTER		$\begin{cases} \frac{x \bmod 10^{k+1}}{10^k} & \text{if } \frac{x}{10^k} \neq 0 \\ \text{space} & \text{if } \frac{x}{10^k} = 0 \end{cases}$
term_Flash_On	BOOLEAN	true	$\text{mon_Time mod } 1000 < 500$
		false	$\text{mon_Time mod } 1000 \geq 500$

Name	Type	Values	Definition
term_Level_Display_Digit(i)	INTEGER		$\begin{cases} 20 & \text{if } i = 0 \\ 18 & \text{if } i = 1 \\ 17 & \text{if } i = 2 \\ 16 & \text{if } i = 3 \end{cases}$
event_SetDigit(x, i)	EVENT		@T(out_Character = term_Digit(x, i)) WHEN out_Cursor_Row = Const_Level_Display_Row AND out_Cursor_Col = term_Level_Display_Digit(i)

B.8.2.1 REQ Relation

The FLMS informs the operator when mon_Fuel_Level is too high, informs the operator when the FLMS is unable to determine whether mon_Fuel_Level is too high, and tests the mechanism for informing the operator.

Behavior of con_High_Alarm

Controlled Variable Name	con_High_Alarm
Initial Value	(con_High_Alarm = on), system init
Mode Class	mode_class_In_Operation
Sustaining Conditions	true

Mode	Events	
mode_Operating	X	ENTERED
mode_Hazard	ENTERED WHEN term_Fuel_Level_Range = levelhigh OR @T(term_Fuel_Level_Range = levelhigh)	@F(term_Fuel_Level_Range = levelhigh)
mode_Shutdown	X	X
mode_Test	ENTERED	@T(term_Test_Time $\geq$ 2)
mode_BadLevDev	@T(term_Flash_On)	@F(term_Flash_On)
con_High_Alarm	on	off

Tolerance	N/A
Initiation Delay	0 ms
Completion Deadline	100 ms

The FLMS informs the operator of the value of mon_Fuel_Level and tests the mechanism for informing the operator.

Behavior of con_Level_Display

Controlled Variable Name con_Level_Display
Initial Value See value function
Mode Class mode_class_In_Operation
Sustaining Conditions true

Mode	Event			
mode_Operating mode_Hazard mode_Shutdown	ENTERED OR @T(mon_Fuel_Level - con_Level_Display ≥ 0.5 cm	X	X	X
mode_Test	X	ENTERED OR @T(term_Test_Time ≥ 14)	@T(term_Test_Time ≥ 4)	X
mode_BadLevDev	X	X	X	ENTERED
con_Level_Display	mon_Fuel_Level	0.0	[(term_Test_Time - 4)] x 11.1	99.9
Tolerance	0.5 cm			
Initiation Delay	0 ms			
Completion Deadline	500 ms			

The FLMS informs the operator when mon_Fuel_Level is too low, informs the operator when the FLMS is unable to determine whether mon_Fuel_Level is too low, and tests the mechanism for informing the operator.

Behavior of con_Low_Alarm

Controlled Variable Name con_Low_Alarm
Initial Value (con_Low_Alarm = on), System init
Mode Class mode_class_In_Operation
Sustaining Condition True

Mode	Events	
mode_Operating	X	ENTERED
mode_Hazard	ENTERED WHEN term_Fuel_Level_Range = levellow OR @T(term_Fuel_Level_Range = levellow)	@F(term_Fuel_Level_Range = levellow)
mode_Shutdown	X	X
mode_Test	@T(term_Test_Time $\geq$ 2)	@T(term_Test_Time $\geq$ 4)
mode_BadLevDev	@T(term_Flash_On)	@F(term_Flash_On)
con_Low_Alarm	on	off

Tolerance N/A
 Initiation Delay 0 ms
 Completion Deadline 100 ms

The FLMS attracts the operators attention when mon_Fuel_Level has an unsafe value and when it is unable to determine the value of mon_Fuel_Level, and it tests the mechanism for attracting the operator's attention.

Behavior of con_Audible_Alarm

Controlled Variable Name con_Audible_Alarm
 Initial Value (con_Audible_Alarm = silent), System initialization
 Mode Class mode_class_In_Operation
 Sustaining Condition True

Mode	Events	
mode_Operating	X	ENTERED
mode_Hazard	ENTERED OR @F(term_Fuel_Level_Range = withinlimits)	@T(term_Fuel_Level_Range = withinlimits)
mode_Shutdown	X	X
mode_Test	@T(term_Test_Time $\geq$ 0)	@T(term_Test_Time $\geq$ 4)
mode_BadLevDev	ENTERED	X
con_Audible_Alarm	sound	silent

Tolerance N/A
 Initiation Delay 0 ms
 Completion Deadline 100 ms

B.8.2.2 Output Variables**Cursor Position**

Acronym	out_Cursor_Row out_Cursor_Col
Hardware	Console
Values	const_Min_Row ≤ out_Cursor_Row ≤ const_Max_Row const_Min_Col ≤ out_Cursor_Col ≤ const_Max_Col
Data Transfer	Softint (10h), function 02h out_Cursor_Row 8088 register DH out_Cursor_Col 8088 register DL
Data Representation	8-bit unsigned integer

Screen

Acronym	out_Character
Hardware	Console
Values	space 32 period 46 block 219
Data Transfer	Softint (10h), function 0Eh, 8088 register AL
Data Representation	8-bit unsigned integer

B.8.2.3 OUT Relation**Setting the Value of con_High_Alarm****Events**

@T(out_Character = space) WHEN out_Cursor_Row = const_High_Alarm_Row AND out_Cursor_Col = const_High_Alarm_Col	@T(out_Character = block) WHEN out_Cursor_Row = const_High_Alarm_Row AND out_Cursor_Col = const_High_Alarm_Col	@T(out_Character ≠ block AND out_Character ≠ space) WHEN out_Cursor_Row = const_High_Alarm_Row AND out_Cursor_Col = const_High_Alarm_Col
con_High_Alarm = off	con_High_Alarm = on	con_High_Alarm <i>undefined</i>

Setting the Value of con_Low_Alarm

Events		
@T(out_Character = space) WHEN out_Cursor_Row = const_Low_Alarm_Row AND out_Cursor_Col = const_Low_Alarm_Col	@T(out_Character = block) WHEN out_Cursor_Row = const_Low_Alarm_Row AND out_Cursor_Col = const_Low_Alarm_Col	@T(out_Character ≠ block AND out_Character ≠ space) WHEN out_Cursor_Row = const_Low_Alarm_Row AND out_Cursor_Col = const_Low_Alarm_Col
LowAlarm = off	LowAlarm = on	LowAlarm <i>undefined</i>

Setting the Value of con_Level_Display

Events
event_Set_Digit(X, 0) AND event_Set_Digit(X, 1) AND event_Set_Digit(X, 2) AND event_Set_Digit(X, 3)
con_Level_Display = X

Setting the Value of con_Audible_Alarm

Events	
@T(out_Character = bel)	@T(DURATION(out_Character = bel) > 0.5 s)
con_Audible_Alarm = sound	con_Audible_Alarm = <i>silent</i>

B.8.3 TRACEABILITY

The following capabilities from Software Requirements for the FLMS (see Appendix A) are fully or partially satisfied by this class:

- (2) When the level in the tank exceeds the the upper or lower limits, an alarm is triggered.
- (5) If the system is unable to determine the fuel level of the tank, the system shall notify the operator of the condition and shut down the pump.
- (6) A capability to conduct system self-testing shall be provided.
- (12) The system shall display fuel level and status alarms to the operator.
- (13) Indications of low or high fuel levels (hazardous conditions) or unknown fuel levels shall be presented to the operator.
- (14) Whenever there is a hazardous condition or an unknown, the system shall provide audible and visual alarms.

B.9 CLASS_SWITCH

class_Switch encapsulates the mechanisms available to the software to determine the values of switches that can be set by the user and the rules that determine when the FLMS considers the user to have requested a selftest or reset of the FLMS.

B.9.1 CLASS INTERFACE

Definitions Other Classes Are Allowed to Use

Name

event_Reset

event_Selftest

B.9.2 ENCAPSULATED INFORMATION

Environmental Variable Glossary

Name	Type	Values	Physical Interpretation
mon_Reset_Switch	ENUMERATED	pressed	The push button labeled RESET is pressed.
		released	The push button labeled RESET is not pressed.
mon_Selftest_Switch	ENUMERATED	pressed	The push button labeled SLFTST is pressed.
		released	The push button labeled SLFTST is not pressed.

Constant, Event, and Term Glossary

Name	Type	Definition
event_Reset	EVENT	@T(DURATION(mon_Reset_Switch = pressed) $\geq$ 3 s)
event_Selftest	EVENT	@T(DURATION(mon_Selftest_Switch = pressed) $\geq$ 0.5 s)

B.9.2.1 Input Variables

Reset Pushbutton

Acronym	in_Reset_Device
Hardware	FLMS Pushbutton Array
Values	on (1b) off (0b)
Data Transfer	PortC
Data Representation	ResetDevice Bit 5 of byte

Selftest Pushbutton

Acronym	in_Selftest_Device
Hardware	FLMS Pushbutton Array
Values	on (1b) off (0b)
Data Transfer	PortC
Data Representation	SelftestDevice Bit 7 of byte

B.9.2.2 IN Relation

Determining the Value of in_Reset_Device

Variable	in_Reset_Device
Error	N/A
Delay	10 ms

Condition

mon_Reset_Switch = pressed	mon_Reset_Switch = released
in_Reset_Device = on	in_Reset_Device = off

Determining the Value of in_Selftest_Device

Variable	in_Selftest_Device
Error	N/A
Delay	10 ms

Condition

mon_Selftest_Switch = pressed	mon_Selftest_Switch = released
in_Selftest_Device = on	in_Selftest_Device = off

B.9.3 TRACEABILITY

The following capability from Software Requirements for the FLMS (see Appendix A) is fully or partially satisfied by this class:

- (11) The operator shall be provided with reset and test switches.

B.10 SAFETY REQUIREMENTS

1. The pump shall not operate when the system is unable to determine the fuel level.
2. The pump shall not operate for longer than the shutdown lock time with the fuel at an unsafe level.

3. The system shall always require operator intervention to enable the pumps when they have been disabled.
4. The pump shall not operate while system self-test is being performed.

B.11 SECURITY REQUIREMENTS

None.

B.12 OTHER REQUIREMENTS

None.

APPENDIX B INDEX

Class

class_Fuel_Tank, definition of, B-9–B-12
 class_In_Operation, definition of, B-7–B-8
 class_Operator, definition of, B-16
 class_Operator_Communication, definition of,
 B-17–B-22
 class_Pump, definition of, B-13–B-14
 class_Switch, definition of, B-23–B-24
 class_Time, definition of, B-15

Constant

const_High_Alarm_Col
 definition of, B-17
 used, B-21
 const_High_Alarm_Row
 definition of, B-17
 used, B-21
 const_High_Fuel_Limit
 definition of, B-10
 used, B-10
 const_Hysteresis
 definition of, B-10
 used, B-10
 const_LCB
 definition of, B-10
 used, B-11
 const_Level_Display_Row
 definition of, B-17
 used, B-18
 const_Low_Alarm_Col
 definition of, B-17
 used, B-22
 const_Low_Alarm_Row
 definition of, B-17
 used, B-22
 const_Low_Fuel_Limit
 definition of, B-10
 used, B-10
 const_Max_Level_Rate
 definition of, B-10
 used, B-10
 const_Max_Test_Time
 definition of, B-7
 used, B-8
 const_MaxCol
 definition of, B-17
 used, B-21
 const_MaxRow
 definition of, B-17
 used, B-21
 const_MinCol
 definition of, B-17

used, B-21

const_MinRow, definition of, B-17
 const_Shutdown_Lock_Time
 definition of, B-7
 used, B-8
 const_UCB
 definition of, B-10
 used, B-11

Controlled variable

con_Audible_Alarm
 definition of, B-17
 OUT relation, B-22
 REQ relation, B-20
 con_High_Alarm
 definition of, B-17
 OUT relation, B-21
 REQ relation, B-18
 con_Level_Display
 definition of, B-17
 OUT relation, B-22
 REQ relation, B-19
 con_Low_Alarm
 definition of, B-17
 OUT relation, B-22
 REQ relation, B-19
 con_Shutdown_Relay
 definition of, B-13
 OUT relation, B-14
 REQ relation, B-13

Event

event_Reset
 definition of, B-23
 used, B-8
 event_Selftest
 definition of, B-23
 used, B-8
 event_Set_Digit
 definition of, B-18
 used, B-22

Input variable

in_ClkPulse
 definition of, B-15
 IN relation, B-15
 in_Diff_Press
 definition of, B-10

IN relation, B-11
in_Reset_Device
 definition of, B-23
 IN relation, B-24
in_Selftest_Device
 definition of, B-24
 IN relation, B-24

Mode

mode_BadLevDev
 definition of, B-8
 used, B-13, B-18, B-19, B-20
mode_class_In_Operation
 definition of, B-8
 used, B-13, B-18, B-19, B-20
mode_Hazard
 definition of, B-8
 used, B-13, B-18, B-19, B-20
mode_Operating
 definition of, B-8
 used, B-13, B-18, B-19, B-20
mode_Shutdown
 definition of, B-8
 used, B-13, B-18, B-19, B-20
mode_Test
 definition of, B-8
 used, B-7, B-13, B-18, B-19, B-20

Monitored variable

mon_Fuel_Level
 definition of, B-9
 IN relation, B-11
 NAT relation, B-10
 used, B-10, B-19
mon_Fuel_Level_Unknown
 definition of, B-9
 IN relation, B-11
 used, B-8
mon_Reset_Switch
 definition of, B-23
 IN relation, B-24
 used, B-23
mon_Selftest_Switch
 definition of, B-23
 IN relation, B-24
 used, B-23

mon_Time
 definition of, B-15
 IN relation, B-15
 NAT relation, B-10
 used, B-17

Output variable

out_Character
 definition of, B-21
 OUT relation, B-21, B-22
 used, B-18
out_Cursor_Col
 definition of, B-21
 OUT relation, B-21, B-22
 used, B-18
out_Cursor_Row
 definition of, B-21
 OUT relation, B-21, B-22
 used, B-18
out_Shutdown
 definition of, B-14
 OUT relation, B-14

Term

term_Digit
 definition of, B-17
 used, B-18
term_Flash_On
 definition of, B-17
 used, B-18, B-20
term_Fuel_Level_Range
 definition of, B-10
 used, B-8, B-18, B-20
term_Inside_Hys_Range
 definition of, B-10
 used, B-8
term_Level_Display_Digit
 definition of, B-18
 used, B-18
term_Test_Time
 definition of, B-7
 used, B-18, B-19, B-20

This page intentionally left blank.

APPENDIX C. CoRE MAPPING TO DOD-STD-2167A

C.1 INTRODUCTION

DOD-STD-2167A (DoD 1988) is the Defense System Software Development standard that helps you establish uniform requirements for software development that are applicable throughout the system life cycle. This section provides guidelines so that you can format the information found in a CoRE specification into a Software Requirements Specification (SRS). This section was written assuming that you have access to the standard and the SRS data item descriptions.

This section offers suggestions for using an existing CoRE software specification to produce an SRS. The Consortium designed the CoRE requirements specification to be a “living” document (e.g., undergoing constant change and revision); the documents required by the 2167A standard are viewed as static and produced to fulfill the needs of the contract. Two strategies may be employed to create the SRS:

- Create the CoRE specification as a “living” document, and create the SRS from a snapshot of the CoRE specification.
- Create the SRS and the CoRE specification in parallel.

The mapping from the elements of a CoRE specification to the SRS sections is summarized in Table C-1. The first column provides the table of contents for the SRS. For each SRS section, the CoRE specification element that should be incorporated into a section is given along with a brief comment. The sections that have no direct mapping from the CoRE specification are left blank. The sections that follow the table elaborate on the comments given in Table C-1.

Table C-1. Relationship of CoRE Specification Elements to the Software Requirements Specification

Software Requirements Specification	CoRE Specification	Comment
1. Scope		
1.1 Identification		
1.2 CSCI overview		
1.3 Document overview		
2. Applicable documents		
2.1 Government documents		
2.2 Nongovernment documents		
3. Engineering requirements		

Table C-1, continued

Software Requirements Specification	CoRE Specification	Comment
3.1 CSCI external interface requirements	Relations tables between input and monitored variables (IN) and between controlled and output variables (OUT)	Provide a brief description for each IN and OUT relation.
3.2 CSCI capability requirements	Mode classes and their dependency diagrams	Examine each mode class to determine the various system modes. Each dependency from a mode class will correlate each capability to each mode.
3.2.X (Capability name and project-unique identifier)	Class structure	Each CoRE class is considered a capability. The encapsulation structure for each class dictates the constituent capabilities (Sections 3.2.x.y).
3.3 CSCI internal interfaces	Monitored and controlled variables	The system context diagram may be presented in this section.
3.4 CSCI data element requirements	Monitored variables, controlled variables, input variables, output variables	List the definitions of the monitored and controlled variables for the internal data items and the input and output variables for the external data items.
3.5 Adaptation requirements	Obtained from NAT relation	NAT relation is intended to capture constraints on the environmental variables. Could also be listed as nonfunctional requirements.
3.5.1 Installation-dependent data	Obtained from NAT relation	NAT relation is intended to capture constraints on the environmental variables.
3.5.2 Operational parameters	Obtained from NAT relation	NAT relation is intended to capture constraints on the environmental variables.
3.6 Sizing and timing requirements	Obtained from REQ relation	REQ relation template provides sizing and timing information (tolerance and scheduling).
3.7 Safety requirements		Listed as nonfunctional requirements.
3.8 Security requirements		Listed as nonfunctional requirements.
3.9 Design constraints		
3.10 Software quality factors		

Table C-1, continued

Software Requirements Specification	CoRE Specification	Comment
3.11 Human performance/human engineering requirements		
3.12 Requirements traceability		
4. Quality requirements		
4.1 Qualification methods		
4.2 Special qualification requirements		
5. Preparation for delivery		
6. Notes		

C.2 SOFTWARE REQUIREMENTS SPECIFICATION

The SRS specifies the engineering and qualification requirements for a computer software configuration item (CSCI). The SRS is used to provide the detailed requirements for a CSCI allocated from the System Segment Specification. It is also used by the contractor as the basis for the design and formal testing of a CSCI. The sections that follow elaborate on the mapping provided by Table C-1.

C.2.1 SRS PARAGRAPH 3.1: CSCI EXTERNAL INTERFACE REQUIREMENTS

The CoRE IN and OUT relations and corresponding variables provide a description of how the software reads or writes from or to required devices and software subsystems. The devices and software subsystems that the CSCIs are required to use are specified in the Interface Requirements Specification or Interface Control Document for the CSCI. The variables and IN and OUT relations provide traceability information needed by this section. You need to provide a brief description of each interface.

C.2.2 SRS PARAGRAPH 3.2: CSCI CAPABILITY REQUIREMENTS

System modes and states provide a history of one or more of the monitored variables. Modes and states are both encapsulated in CoRE's mode classes. To identify the system modes and states, examine each mode class and determine which modes are used by other classes. The dependency diagram for each mode class gives you the capabilities that are affected by the mode. Each capability maps to a CoRE class. This information can be captured in a tabular format like the one shown in Table C-2. An X in a row correlates a capability or constituent capability to a particular mode.

Table C-2. Example of CSCI System States Mapping to Capabilities

mode_class_in_Operation	class_Pump	class_Operator
mode_Operating	X	X
mode_Hazard		X
mode_Shutdown	X	
mode_Test	X	X
mode_BadLevDev	X	X

C.2.3 SRS PARAGRAPH 3.2.X: (CAPABILITY NAME AND PROJECT-UNIQUE IDENTIFIER)

The capabilities of a CSCI correspond to the classes identified on the top-level dependency graph (level 0). Each of these classes, in turn, can be decomposed into subordinate classes. This imposes a hierarchy, called the encapsulation structure, where the subordinate classes become constituent capabilities. Hierarchical requirements structures have normally been used to simplify expressing the capabilities of a CSCI; with CoRE, a hierarchical structure is created to help manage the requirements development. This structure should be reflected in numbering the capabilities.

The names of the classes should be meaningful and unique. When taken with the information provided by the class template, a class name should provide a clear picture of the capability being expressed by the class. The class description provides the purpose of the capability. The dependencies into the class are described via information found in the class template.

C.2.4 SRS PARAGRAPH 3.3.: CSCI INTERNAL INTERFACES

The interfaces between capabilities are defined in terms or expressions of monitored variables. The system context diagram shows the boundaries between the software and the monitored and controlled variables.

C.2.5 SRS PARAGRAPH 3.4.: CSCI DATA ELEMENT REQUIREMENTS

For data elements internal to the CSCI, list the definitions of the monitored and controlled variables and terms. The information that CoRE requires you to capture about each of these provides the information required by this section.

For data elements external to the CSCI, list the definitions of the input and output variables. The information that CoRE requires you to capture about each of these provides the information required by this section.

C.2.6 SRS PARAGRAPH 3.5.: ADAPTATION REQUIREMENTS

This section is intended to specify the requirements for adapting the CSCI to site-unique conditions and to changes in the system environment. These requirements are captured in the NAT relation, which captures constraints placed on the environmental variables in which the system is expected to operate.

C.2.7 SRS PARAGRAPH 3.5.1.: INSTALLATION-DEPENDENT DATA

Site-unique data can be considered a constraint on the environment. If you choose to capture this information as a constraint, the appropriate NAT relations would be listed in this section.

C.2.8 SRS PARAGRAPH 3.5.2.: OPERATIONAL PARAMETERS

Operational needs can be considered constraints on the environment. If you choose to capture this information as constraints, the appropriate NAT relations would be listed in this section.

LIST OF ABBREVIATIONS AND ACRONYMS

CDU	Command and Display Unit
C³I	command, control, communications, and intelligence
cm	centimeter
CoRE	Consortium Requirements Engineering
CRT	cathode-ray tube
CSCI	computer software configuration item
ERD	entity-relationship diagram
FLMS	Fuel Level Monitoring System
IN	input
ms	millisecond
NAT	nature
OUT	output
REQ	required
RTCP	Radio Tune Control Panel
s	second
SRS	Software Requirements Specification

This page intentionally left blank.

GLOSSARY

Abstraction	A view of the problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information (IEEE 1983).
Accuracy	A quantitative measure of the magnitude of error (IEEE 1990). Use it to characterize the discrepancy between the actual values of monitored and input variables and of controlled and output variables.
Attribute	Characterizes some important aspect or fact about an entity.
Behavioral model	The behavioral model defines the required, externally visible behavior in terms of two relations from monitored variables to controlled variables. These relations are NAT and REQ.
Boundary class	Defines monitored and controlled variables and potentially encapsulates the corresponding IN and OUT relations. Boundary classes serve to abstract from details about the software's interface with the environment.
Class	A template for an object or a set of related objects. A class defines a set of requirements or terms common to one or more objects. A CoRE requirements specification is written in terms of CoRE classes.
Class model	The set of mechanisms provided by CoRE for defining classes and their relationships. Provides a set of facilities for packaging the behavioral model as a set of classes and provides the mechanisms to manage requirements changes, create reusable requirements, and develop parts of the software in parallel.
Class structure	The set of classes and their relationships for a particular CoRE specification. The class structure for a CoRE specification is constructed using the elements of the class model.

Complete	A CoRE specification is complete when the behavioral model defines the required values of the controlled variables for all possible values of the monitored variables, the values of the input for all possible values of the monitored variables, and the values of the output for all possible values of the controlled variables.
Compound condition	Formed by connecting two or more conditions using the logical operators AND, OR, or NOT.
Condition	Boolean expression (predicate) of the environmental variables that holds for a continuous, measurable period of time. A condition characterizes some aspect of the environmental state.
Condition table	A tabular representation of a function where the domain of the function comprises mode and a set of mutually exclusive conditions.
Consistent	A CoRE specification is internally consistent when the CoRE controlled variable functions map each value in the domain to exactly one value in the range and when no two parts of the specification are mutually contradictory.
Controlled variable	Denotes a quantity in the environment that the software sets.
Controlled variable functions	Refers to the set of functions used to describe the REQ relation for a controlled variable. This includes the value function, accuracy, and timing.
Demand	Scheduling requirement that is associated with a controlled variable when the controlled variable must be set in response to a periodic event.
Dependency	Exists between classes X and Y when class X uses T, where T can be a monitored variable, term, mode, or event provided by class Y only if X employs T in its definition.
Depends-on relation	Denotes which classes use what information provided by other classes.
Domain	A relation pairs the elements of one set with the elements of a second set. The first set is called the domain of the relation (see Range).

Dynamic view	Captures the required timing behavior and scheduling characteristics, i.e., when the software must initiate or complete the required behavior.
Encapsulation	Process of hiding details and other decisions by providing an abstract interface.
Encapsulates relation	Class X encapsulates class Y if the definition of Y is part of the encapsulated information of X.
Encapsulation structure	The encapsulates relation induces a hierarchy on the set of classes called the encapsulation structure.
Environmental variables	CoRE views a system as existing within and interacting with an environment. The quantities in the environment that are relevant to the software are denoted by mathematical variables called environmental variables.
Entity	A representation of any aspect of the system environment of interest to the system that can describe physical things, roles played by persons or organizations, incidents, and interactions that are significant to the software.
Event	Occurs when a condition changes value.
Event expression	An expression used to represent an event occurrence.
Event occurrence	A moment in time when a condition's value changes. Each event occurrence is instantaneous (takes zero time) and atomic (all or none occurs).
Event table	Tabular representation of a function where the domain of the function comprises modes and events.
Expression	A formula that defines the computation of a value using a logical symbol or a meaningful combination of one or more variables.
Finite state machine	A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions (IEEE 1990).
Four-variable model	The CoRE behavioral model is based on a four-variable model in which the four variables are monitored, controlled, input, and output.

Function	A relation between two sets in which each element of the one set (the domain) maps to no more than one element of the other set (the range).
Functional view	A view of the software behavioral requirements as a set of functions, i.e., what values the system must produce.
Generalization/specialization structure	Hierarchical relationship denoting inheritance among a set of CoRE classes. A superclass defines a set of common properties inherited by its subclasses.
Hidden information	Details or decisions that are hidden in a class.
IN relation	Describes the relationship between the monitored variables and the available inputs.
Inheritance	Denotes the requirements or terms that are defined by a superclass and shared among its subclasses.
Input variable	A variable representing a discrete input to the software. The complete definition provides a precise description of how the software reads from an input device, including the protocol for reading from a device and a mapping between abstract values and the bit patterns read from the device.
Interface	Information defined by a class that can be used by other classes in their definitions.
Mode	A state of a mode machine.
Mode machine	A form of finite state machine. Includes a collection of system modes and the transitions between them. Differs from a finite state machine in that a mode machine does not define actions.
Monitored variable	Denotes an environmental quantity that the software must track.
NAT relation	Describes those constraints placed on the system by the external environment (nature). These constraints are properties of the environment that affect the software but exist whether the software exists or not.

Object	An object is a subset of the definition of REQ, NAT, IN, and OUT (including definitions of the variables) for a given specification written in terms of the four-variable model. Every object must be an instance of a class.
OUT relation	Describes the relationship between the controlled variables and the available software outputs.
Output variable	A variable representing a discrete output of the software. The complete definition provides a precise description of how the software writes to an output device, including the protocol for writing to a device and a mapping between abstract values and the bit patterns sent to the device.
Periodic	A scheduling requirement associated with a controlled variable when the controlled variable must be set or updated at regular, fixed time intervals.
Precision	The degree of exactness or discrimination with which a quantity is stated (IEEE 1990). For a monitored variable, the precision expresses how accurately the software is required to measure the actual quantity that the monitored variable denotes. For a controlled variable, the precision expresses how accurately the software must be able to set the actual quantity that the controlled variable denotes.
Predicate	A function whose range is the elements {true, false}.
Range	A relation pairs the elements of one set with the elements of a second set. The second set is called the range of the relation (see Domain).
Relation	Pairs the elements of one set (the domain) with the elements of another (the range). Each element of the first set can be paired with one or more elements of the second.
REQ relation	Describes properties that the software is required to maintain between the monitored and controlled variables.
Selector table	Tabular representation of mode-dependent information.

State	The values assumed at a given instant by the set of environmental variables (monitored and controlled).
Subclass	A class that is defined as an instance of a superclass. A subclass specializes the definition of its superclass by adding or constraining requirements.
Superclass	A class that defines a set of requirements or terms that are common among two or more CoRE classes.
Term	Named expression of one or more monitored variables (or other terms). A formula that defines the computation of a value using one or more monitored variables to which you have assigned a name.
Testable	A requirement is considered testable if it is possible to determine, for any given test case (i.e., an input and output), whether the output represents an acceptable behavior of the software given the input and the system state. In other words, testable requirements distinguish precisely the set of acceptable software behaviors in terms of the observable behavior of the system.
Tolerance	The amount of variation allowed from an ideal value of a controlled variable.
Transition	Occurs between modes as a result of a change in one or more environmental variables.
Undesired event	Failure of a system component or of the system itself.
Value function	Maps each value of the monitored variables in its domain to the ideal value of the controlled variable. The value function specifies the ideal behavioral requirements of a controlled variable.

REFERENCES

- Alspaugh, Thomas A., Stuart R. Faulk, Kathryn Heninger Britton, R. Alan Parker, David L. Parnas, and John E. Shore
1992
Software Requirements for the A-7E Aircraft, NRL/FR/5530-92-9194. Washington, D.C.: Naval Research Laboratory.
- Chen, Peter
1976
The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems* 1, 1:9-36.
- Department of Defense
1988
Defense System Software Development, DOD-STD-2167A. Washington, D.C.: Department of Defense.
- Hatley, D., and I. Pirbhai
1988
Strategies for Real-Time System Specification. New York, New York: Dorset House.
- Heninger, Kathryn L.
1980
Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *IEEE TSE SE6*:2-13.
- IEEE
1990
IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE STD 610.12-1990. Institute of Electrical and Electronic Engineers.
- 1983
IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE STD 729-1983. Institute of Electrical and Electronic Engineers.
- Parnas, David L., and Jan Madey
1990
Functional Documentation for Computer Systems Engineering, (Version 2), CRL Report No. 237. Hamilton, Ontario: McMaster University.
- Shlaer, S., and S. Mellor
1988
Object-Oriented Systems Analysis: Modeling the World in Data. Englewood Cliffs, New Jersey: Prentice-Hall.
- Van Schouwen, A.J.
1990
The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems, Technical Report 90-276. Hamilton, Ontario: Queen's University.

This page intentionally left blank.

BIBLIOGRAPHY

Embley, David W., Barry D. Kurtz, and Scott N. Woodfield. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Englewood Cliffs, New Jersey: Prentice-Hall, 1992.

Faulk, Stuart, John Brackett, Paul Ward, and James Kirby, Jr. The Core Method for Real-Time Requirements. *IEEE Software* 5, 9:22-33, 1992.

Heitmeyer, C., and J. McLean. Abstract Requirements Specification: A New Approach and Its Application. *IEEE TSE SE* 95:580-589, 1983.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

This page intentionally left blank.

INDEX

Attribute

- definition of, 5-2
- enumerated, 7-4
- example of, 5-2
- identification of, 7-5
- in FLMS, 7-6
- matrix, 5-3
- numeric, 7-5
- on ERD, 5-2

Balancing

- definition of, 5-9
- illustration, 5-9

Behavioral model, 2-5

- definition of, 2-5
- description, 1-2
- reasons for, 2-7, 4-1
- relations

- illustration, 2-7
- IN, 2-6
- NAT, 2-6
- OUT, 2-6
- REQ, 2-6

- state machine, 4-6

Behavioral requirement

- behavioral model, 2-1
- class model, 2-1
- definition of, 2-1
- tolerance, 4-1

Boundary class

- construction of, 9-2
- input and output variables, 11-2
- interface, 9-4
 - example of, 9-4
 - versus encapsulation, 9-4
- purpose of, 8-1

Class

- class_Fuel_Tank, definition of, B-9-B-12
- class_In_Operation, definition of, B-7-B-8
- class_Operator, definition of, B-16
- class_Operator_Communication, definition of, B-17-B-22
- class_Pump, definition of, B-13-B-14
- class_Switch, definition of, B-23-B-24
- class_Time, definition of, B-15
- definition of, 2-12
- template for, 5-14

Class interface

- definition of, 5-9
- example of, 9-7, B-7
- goals of, 5-9
- illustration, 5-10

Class model, 2-11

- See also* Packaging
- concepts, 5-1
- contents of, 5-4
- interface, 5-1
 - dependencies on, 5-1
- semantics, 5-1
- syntax, 5-1

Class structure

- See also* Packaging
- capability name, C-2
- construction of, 9-2
- definition of, 2-11
- evaluation of, 9-12
- inheritance, 2-14
- messages, 2-16
- notation, 5-6
- representation, 5-5
- revisiting, 10-11
- specification, 5-9
- subclass, 2-14
- superclass, 2-14
- use of dependency graph, 9-3

Condition

- compound, 4-4

- definition of, 4-1
- example of, 4-4
- predefined
 - ENTERED, 4-9
 - EXITED, 4-9
 - INMODE, 4-9
 - semantics of, 4-9
- sustaining, 10-11
- Condition table
 - See also* Mode machine
 - completed sample, 10-7
 - create value function, 10-6
 - defining IN relation, 11-4
 - defining OUT relation, 11-5
 - modes, 10-7
 - template for, 4-10
- Constant
 - const_High_Alarm_Col
 - definition of, B-17
 - used, B-21
 - const_High_Alarm_Row
 - definition of, B-17
 - used, B-21
 - const_High_Fuel_Limit
 - definition of, B-10
 - used, B-10
 - const_Hysteresis
 - definition of, B-10
 - used, B-10
 - const_LCB
 - definition of, B-10
 - used, B-11
 - const_Level_Display_Row
 - definition of, B-17
 - used, B-18
 - const_Low_Alarm_Col
 - definition of, B-17
 - used, B-22
 - const_Low_Alarm_Row
 - definition of, B-17
 - used, B-22
 - const_Low_Fuel_Limit
 - definition of, B-10
 - used, B-10
 - const_Max_Level_Rate
 - definition of, B-10
 - used, B-10
 - const_Max_Test_Time
 - definition of, B-7
 - used, B-8
 - const_MaxCol
 - definition of, B-17
 - used, B-21
 - const_MaxRow
 - definition of, B-17
 - used, B-21
 - const_MinCol
 - definition of, B-17
 - used, B-21
 - const_MinRow
 - definition of, B-17
 - used, B-21
 - const_Shutdown_Lock_Time
 - definition of, B-7
 - used, B-8
 - const_UCB
 - definition of, B-10
 - used, B-11
- Constraint
 - demand, 10-5
 - example of, 10-8
 - periodic, 10-8
 - example of, 10-8
 - scheduling, 10-8
 - timing, 10-8
 - consistency check, 10-12
 - tolerance, 10-9
- Context diagram. *See* System context diagram
- Controlled state function
 - definition of, 2-8
 - range, 2-9
- Controlled variable
 - See also* Environmental variable
 - con_Audible_Alarm
 - definition of, B-17
 - OUT relation, B-22
 - REQ relation, B-20
 - con_High_Alarm
 - definition of, B-17
 - OUT relation, B-21
 - REQ relation, B-18
 - con_Level_Display
 - definition of, B-17
 - OUT relation, B-22
 - REQ relation, B-19
 - con_Low_Alarm
 - definition of, B-17
 - OUT relation, B-22
 - REQ relation, B-19

- con_Shutdown_Relay
 - definition of, B-13
 - OUT relation, B-14
 - REQ relation, B-13
 - data element requirement, C-2
 - defined with OUT relation, 4-2
 - defining function, 4-1
 - definition of, 2-5
 - delay requirement, 4-3
 - determining range of values, 11-5
 - example of, B-13
 - identification of, 8-2
 - internal interface, C-2
 - output resources, 11-1
 - periodic or demand, 8-10
 - precision specification, 8-6
 - scheduling requirement, 4-3
 - value of, 4-2
- Controlled variable function
- analysis
 - completeness, 12-1
 - consistency, 12-2
 - definition of, 8-7
 - domain, 8-4
- CoRE
- definition of, 1-1
 - DoD-STD-2167A specification (SRS), C-1
 - dynamic view, 4-3
 - representation, 4-12
 - functional view, 4-1
 - illustration, 4-2
 - representation, 4-3
 - process, 6-1
 - purpose of, 2-2
 - representation of, 4-1
 - tables, 4-9
 - condition, 4-10
 - event, 4-10
 - selector, 4-12
- Demand behavior, specification of, 10-5
- Dependency graph
- See also* Class structure
 - example of, B-6
 - notation, 5-7
 - reasons for, 9-3
 - relationships, 5-7
- Depends-on relation
- definition of, 2-13
 - illustration of, 2-13
- Encapsulated information
- decomposition, 5-10
 - example of, B-7
 - identification of, 9-3
 - illustration, 5-10
 - IN and OUT relations, 11-2
 - input and output variables, 11-2
 - notation, 5-10
- Entity
- definition of, 5-2
 - example of, 5-2
 - on ERD, 5-2
 - template of, 5-2
- Environmental variable, 2-5
- analysis
 - completeness, 12-1
 - consistency, 12-1
 - controlled variable, 2-5
 - definition of, 8-5
 - evaluation criteria
 - completeness, 10-11
 - consistency, 10-12
 - identifying, 8-2
 - initial value, 10-3
 - sustaining condition, 10-3
 - from information model, 5-2
 - monitored variable, 2-5
 - definition of, 8-5
 - state characterization, 4-1
 - template
 - physical interpretation, 4-3
 - type, 4-3
 - values, 4-3
- Event
- based on, 4-2
 - definition of, 4-1
 - event_Reset
 - definition of, B-23
 - used, B-8
 - event_Selftest
 - definition of, B-23
 - used, B-8
 - event_Set_Digit
 - definition of, B-18
 - used, B-22
 - example of, 9-9, B-16
 - expression, 4-4
 - @F expression, 4-4
 - @T expression, 4-4
 - example of, 4-4

- order of evaluation, 4-5
- occurrence, 4-4
- periodic timing, 10-6
- Event table, template for, 4-11
- Existing skills
 - requirement for, 1-1
 - where met, 2-15
- FLMS, B-5
 - illustration, 3-1
 - narrative, 3-1
- Formal model
 - requirement for, 1-1
 - where met, 2-7
- Four-variable model
 - definition of, 2-5
 - relations of, 4-2
 - use in CoRE, 4-1
- Global checking
 - completeness, 12-3
 - consistency, 12-3
- Graphical specifications
 - requirement for, 1-1
 - where met, 2-15
- Guidance, requirement for, 1-2
- Guidebook
 - description, 1-2
 - organization, 1-3
 - scope, 1-3
 - use of, 1-4
- Hardware resources
 - definition of, 11-1
 - example of, 11-1
- Hidden information. *See* Encapsulated information
- Idealized CoRE process
 - definition of, 6-1
 - stages of, 6-3
 - class structuring, 9-1
 - define hardware interface, 11-1
 - detailed behavior specification, 10-1
 - identify environmental variables, 7-1
 - preliminary behavior specification, 8-1
- IN relation
 - analysis
 - completeness, 12-3
 - consistency, 12-3
 - definition of, 2-6, 2-9, 11-3
 - domain of, 2-10
 - example of, B-15
 - external interface requirements, C-2
 - hardware interface, 6-6
 - range of, 2-10
- Information model
 - See also* Class model
 - attributes, 7-3
 - example of, 7-4
 - sources of, 7-3
 - composition of, 7-3
 - construction of
 - behavioral model, 5-2
 - class model, 5-2
 - definition of, 7-2
 - entities, 7-5
 - example of, 7-6
 - sources of, 7-5
 - example of, 7-6
 - identifying
 - classes, 5-2
 - environmental variables, 5-2
 - relations, 7-5
 - example of, 7-9
 - sources of, 7-6
 - representation of
 - attribute matrix, 7-9
 - ERD, 7-6
- Inheritance
 - description of, 5-12
 - generalization/specialization, 5-12
 - notation, 5-13
- Initial mode
 - identification of, 4-7
 - transition from, 4-7
- Input variable
 - allocation of, 11-2
 - completing definition, 11-3
 - data element requirement, C-2
 - definition of, 2-6
 - evaluation criteria, 11-6
 - example of, B-10
 - IN relation, 11-1
 - in_ClkPulse
 - definition of, B-15
 - IN relation, B-15

- in_Diff_Press
 - definition of, B-10
 - IN relation, B-11
- in_Reset_Device
 - definition of, B-23
 - IN relation, B-24
- in_Selftest_Device
 - definition of, B-24
 - IN relation, B-24
- mapping to monitored variable, 11-5
- specification of, 11-2
- template for, 11-3
- Interface
 - definition of, 2-12
 - specification of, 11-2
- Internal class, purpose of, 6-4
- Mode
 - analysis
 - completeness, 12-2
 - consistency, 12-2
 - based on, 4-2
 - example of, 9-7, B-7
 - identification of, 8-9
 - mode_BadLevDev
 - definition of, B-8
 - used, B-13, B-18, B-19, B-20
 - mode_class_In_Operation
 - definition of, B-8
 - used, B-13, B-18, B-19, B-20
 - mode_Hazard
 - definition of, B-8
 - used, B-13, B-18, B-19, B-20
 - mode_Operating
 - definition of, B-8
 - used, B-13, B-18, B-19, B-20
 - mode_Shutdown
 - definition of, B-8
 - used, B-13, B-18, B-19, B-20
 - mode_Test
 - definition of, B-8
 - used, B-7, B-13, B-18, B-19, B-20
 - system, 8-11
- Mode class. *See* Internal class; Mode machine
- Mode machine
 - See also* Internal class
 - capability requirements, C-2
 - construction of, 9-2, 10-6
 - sample, 10-7
 - contents of, 4-6
 - defining set of modes, 8-12
 - definition of, 8-10
 - example of, 8-12
 - identification of, 8-10
 - illustration, 4-2
 - initial value, 10-11
 - properties, 4-9
 - purpose of, 4-1
 - refinement of, 10-9
 - representation of
 - See also* Mode transition diagram
 - example, 4-7
 - mode transition table, 4-7
 - state transition diagram, 8-12
 - specification of, 6-4
 - state machine, 4-6
- Mode transition diagram
 - definition of, 4-7
 - transition events, 4-7
- Mode transition table
 - definition of, 4-7
 - example of, 4-8
- Monitored state function
 - definition of, 2-8
 - domain of REQ, 2-9
- Monitored variable
 - See also* Environmental variable
 - data element requirement, C-2
 - defined with IN relation, 4-2
 - definition of, 2-5
 - determining range of values, 11-4
 - example of, 9-5, B-9
 - for undesired event, 4-17
 - ideal value function, 4-1
 - identification of, 8-3, 8-8
 - input resources, 11-1
 - internal interface, C-2
 - mon_Fuel_Level
 - definition of, B-9
 - IN relation, B-11
 - NAT relation, B-10
 - used, B-10, B-19
 - mon_Fuel_Level_Unknown
 - definition of, B-9
 - IN relation, B-11
 - used, B-8
 - mon_Reset_Switch
 - definition of, B-23
 - IN relation, B-24
 - used, B-23

- mon_Selftest_Switch
 - definition of, B-23
 - IN relation, B-24
 - used, B-23
 - mon_Time
 - definition of, B-15
 - IN relation, B-15
 - NAT relation, B-10
 - used, B-17
 - timing constraint, 4-1
 - tolerance function, 4-1
 - use in terms, 4-1
- NAT relation**
- adaptation requirement, C-2
 - definition of, 2-6, 2-8
 - domain of, 2-8
 - operational parameter, C-4
 - range of, 2-8
 - specification of, 4-16
- Nonalgorithmic specification**
- requirement for, 1-1, 2-3
 - where met, 2-7
- Notation**
- conventions, 1-4
 - requirement for, 1-1
- Object**
- as class instance, 5-10
 - definition of, 2-12
 - specification of, 5-11
- Object-oriented model**
- requirement for, 1-1
 - where met, 2-1, 2-11
- OUT relation**
- analysis
 - completeness, 12-3
 - consistency, 12-3
 - definition of, 2-6, 2-9, 11-3
 - domain of, 2-10
 - example of, B-14
 - external interface requirements, C-2
 - hardware interface, 6-6
 - range of, 2-10
- Output variable
 - allocation of, 11-2
 - completing definition, 11-3
 - data element requirement, C-2
 - definition of, 2-6, 11-2
 - evaluation criteria, 11-6
 - example of, B-14
 - mapping to controlled variable, 11-5
 - OUT relation, 11-1
- out_Character
 - definition of, B-21
 - OUT relation, B-21, B-22
 - used, B-18
- out_Cursor_Col
 - definition of, B-21
 - OUT relation, B-21, B-22
 - used, B-18
- out_Cursor_Row
 - definition of, B-21
 - OUT relation, B-21, B-22
 - used, B-18
- out_Shutdown
 - definition of, B-14
 - OUT relation, B-14
- specification of, 11-2
- template for, 11-3
- Packaging**
- See also* Class structure; Encapsulated information
 - allocation of information, 9-1
 - illustration (canonical), 2-15
 - creating classes for, 9-2
 - definition of, 2-1
 - goals of, 9-1
 - reasons for, 2-4
 - relationships, 2-12
 - dependency, 2-13
 - encapsulation, 2-12
 - generalization/specialization, 2-14
- Periodic behavior, specification of, 10-6**
- Physical interpretation**
- enumerated variables, 8-7
 - use of, 8-7
- Precision**
- specification of, 8-6
 - use of, 8-6
- Procedure**
- example of, FLMS, 8-11
 - specification of, 8-11
- Recursion. *See* Recursion**

- Relationship
 - aggregation, 5-3
 - illustration, 5-4
 - application-specific, 5-4
 - description of, 5-2
 - generalization/specialization, 5-3
 - illustration, 5-3
- REQ relation
 - consistency, 10-12
 - definition of, 2-6, 2-9
 - derived information, 8-12
 - domain of, 2-9
 - example of, B-13
 - range of, 2-9
 - sizing and timing requirement, C-2
 - specification of, 4-15
- Rigorous specifications
 - requirement for, 1-1
 - where met, 2-1, 2-4
- Scheduling
 - demand, 4-14
 - illustration, 4-14
 - identify requirements, 8-9
 - periodic, 4-13
 - constraint, 10-6
 - illustration, 4-14
- Selector table
 - create value function, 10-6
 - definition of, 4-12
 - example of, 4-12
 - template for, 4-12
- Separation of concerns
 - requirement for, 2-3
 - where met, 2-5
- State
 - representation of
 - decision table, 4-6
 - state transition diagram, 4-6
 - value of, 4-6
- System constraint
 - evaluation criteria, 7-10
 - exit criteria, 7-10
 - identification of
 - environmental variable, 7-3
 - likely requirements changes, 7-9
- System context diagram
 - development of, 8-7
 - example of, B-5
- leveling, 5-8
 - illustration, 5-8
 - notation, 5-6
 - representation of overview, 5-7
 - use of, 5-6
- System function
 - example of, 8-11
 - representation of, 8-11
- Term
 - analysis
 - completeness, 12-2
 - consistency, 12-2
 - construction of, 9-4
 - example of, 9-5, B-10
 - reasons for, 4-6
 - term_Digit
 - definition of, B-17
 - used, B-18
 - term_Flash_On
 - definition of, B-17
 - used, B-18, B-20
 - term_Fuel_Level_Range
 - definition of, B-10
 - used, B-8, B-18, B-20
 - term_Inside_Hys_Range, definition of, B-10
 - term_Level_Display_Digit
 - definition of, B-18
 - used, B-18
 - term_Test_Time
 - definition of, B-7
 - used, B-19, B-20
 - type, 4-6
 - use of, 4-1
 - value, 4-6
- Timing and scheduling requirements
 - constraints, 10-12
 - template for, 10-11
- Tolerance, function, 10-12
- Undesired event
 - description of, 4-17
 - inability to determine value, 8-4
 - specification of, 4-17
- Value function
 - See also* Controlled variable
 - analysis of, 10-6
 - completeness, 10-11

- condition table, 10-12
- creation of, 10-4
- event table, 10-12
- initial value, 10-11
- periodic timing constraint, 10-6
- response to event, 10-6
- specification of, 10-1
- Value-dependent variation
 - definition of, 10-9
 - example of, 10-9