InCharge™

MODEL Reference Guide

Version 6.2

Cisco Part Number: OL-6125-01

the technology of Crystal Decisions. The use of all Crystal Decisions software products is subject to a separate license agreement included with the Software electronically, in written materials, or both. YOU MAY NOT USE THE CRYSTAL DECISIONS SOFTWARE UNLESS AND UNTIL YOU READ, ACKNOWLEDGE AND ACCEPT THE TERMS AND CONDITIONS OF THE CRYSTAL DECISIONS' SOFTWARE LICENSE AGREEMENT. IF YOU DO NOT ACCEPT THE TERMS AND CONDITIONS OF THE CRYSTAL DECISIONS' SOFTWARE LICENSE, YOU MAY RETURN, WITHIN THIRTY (30) DAYS OF PURCHASE, THE MEDIA PACKAGE AND ALL ACCOMPANYING ITEMS (INCLUDING WRITTEN MATERIALS AND BINDERS OR OTHER CONTAINERS) RELATED TO THE CRYSTAL DECISIONS' TECHNOLOGY, TO SMARTS FOR A FULL REFUND; OR YOU MAY WRITE, CRYSTAL WARRANTIES, P.O. BOX 67427, SCOTTS VALLEY, CA 95067, U.S.A.

GNU eTeks PJA Toolkit
Copyright © 2000-2001 Emmanuel PUYBARET/eTeks info@eteks.com. All Rights Reserved.
The eTeks PJA Toolkit is resident on the CD on which the Software was delivered to you. Additional information is available at eTEks' web site: http://www.eteks.com. The eTeks PJA Toolkit program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation; version 2 of the License. The full text of the applicable GNU GPL is available for viewing at http://www.gnu.org/copyleft/gpl.txt. You may also request a copy of the GPL from the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. The eTeks PJA Toolkit program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

For a period of three years from the date of your license for the Software, you are entitled to receive under the terms of Sections 1 and 2 of the GPL, for a charge no more than SMARTS' cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code for the GNU eTeks PJA Toolkit provided to you hereunder by requesting such code from SMARTS in writing: Attn: Customer Support, SMARTS, 44 South Broadway, White Plains, New York 10601.

IBM Runtime for AIX
The Software contains the IBM Runtime Environment for AIX(R), Java™ 2 Technology Edition Runtime Modules © Copyright IBM Corporation 1999, 2000 All Rights Reserved.

HP-UX Runtime Environment for the Java™ 2 Platform
The Software contains the HP-UX Runtime for the Java™ 2 Platform, distributed pursuant to and governed by Hewlett-Packard Co. ("HP") software license terms set forth in detail at: http://www.hp.com. Please check the Software to determine the version of Java runtime distributed to you.

DataDirect Technologies
Portions of this software are copyrighted by DataDirect Technologies, 1991-2002.

NetBSD
Copyright © 2001 Christopher G. Demetriou. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
   This product includes software developed for the NetBSD Project. See http://www.netbsd.org/ for information about NetBSD.
4. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. <<Id: LICENSE, v 1.2 2000/06/14 15:57:33 cgd Exp>>

RSA Data Security, Inc.
Copyright © 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved. License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function. License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work. RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind. These notices must be retained in any copies of any part of this documentation and/or software.

AES
Copyright © 2003, Dr Brian Gladman <brg@gladman.me.uk>, Worcester, UK. All rights reserved.

License Terms:

The free distribution and use of this software in both source and binary form is allowed (with or without changes) provided that:

1. distributions of this source code include the above copyright notice, this list of conditions and the following disclaimer;
2. distributions in binary form include the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other associated materials;
3. the copyright holder's name is not used to endorse products built using this software without specific written permission.

ALTERNATIVELY, provided that this notice is retained in full, this product may be distributed under the terms of the GNU General Public License (GPL), in which case the provisions of the GPL apply INSTEAD OF those given above.

Disclaimer: This software is provided 'as is' with no explicit or implied warranties in respect of its properties, including, but not limited to, correctness and/or fitness for purpose. Issue Date: 26/08/2003

Contents

Preface **ix**

Intended Audience ix
Prerequisites x
Document Organization x
Documentation Conventions xi
InCharge Installation Directory xi
Additional Resources xiii
  InCharge Commands xiii
  Documentation xiii
Common Abbreviations and Acronyms xiv
Technical Support xv

1 About MODEL **1**

Modeling Event Information 2
  How InCharge Detects Events 2

2 Introduction by Example **5**

Class Properties 5
  Declaring a Class in MODEL 6
  Interface Declarations 6
Modeling an Object's Properties 7
  Attribute Declarations 8
  Relationship Declarations 9
  Propagate Attribute Declarations 11
Modeling Events 12
  Event Declarations 12
  Problem Declarations 13
  Propagate Symptom Declarations 14

Aggregate Declarations 15
Propagate Aggregate Declarations 16
Export Declaration 17
Refining an Object's Properties 18
The Complete Example 18

3 Compiling MODEL Files 23

Overview of the Compilation Process 23
Specifying Additional Files with #include 24
Using the MODEL Compiler 26
Using the Proper Makefile 26
Compiling MODEL Files 26
Invoking the MODEL Compiler 27

4 Working with MODEL Libraries 29

Tools for Working with MODEL Libraries 29
Using the InCharge Generic Console 30
Using the dmctl Command-Line Interface 30
Loading a Model Library 31
Location of MODEL Libraries 32
Starting a Domain Manager 32
Methods for Loading MODEL Libraries 33
Working with a MODEL Library and a Domain Manager 34
Methods for Listing Models Loaded into a Domain Manager 34
Listing Classes in the MODEL Library 35
Creating Instances of a Class 36
Modifying the Properties of an Instance 37
Notifying Events 38

5 Basic Lexical Elements of MODEL 41

Keywords 41
Identifiers 43
Data Types 44

6 Declaring an Interface 45
Interface Declaration 45
Interface Header Declaration 46

7 Attribute Declarations 49
Access Types for Attributes 49
When the Value of an Attribute is Unavailable 50
Attributes Propagated Over a Relationshipset 51
Minimizing the Effects of Unavailable Attributes 52
Stored Attributes 53
Computed Attributes 54
Instrumented Attributes 55
Propagated Attributes 58
Table Attributes 60

8 Relationship Declarations 63

9 Declaring Events in MODEL 67
MODEL Declarations for Defining Events 67
Event Declaration 68
Problem Declaration 70
Symptom Declaration 72
Propagate Symptom Declaration 73
Aggregate Declaration 74
Propagate Aggregate Declaration 75
Export Declaration 76

10 Operation Declarations 77

11 Writing Expressions in MODEL 79
Lexical Elements for Expressions 79
Literals 80
Operators 81

Evaluation Expression Operators 82
Operators for Set Expressions 84
Precedence of Operators 84
Built-in Functions 85
Syntax for Expressions 90
When the Value of an Expression is Unavailable 91
Boolean Attributes 91

12 Instrument Declarations 93

Example of an Instrumentation Declaration 94
Summary of Runtime Requirements for SNMP Instrumentation 94

13 MODEL Pragmas 97

Required Pragmas 97
#pragma include_c *file-name* 97
#pragma include_h *file-name* 97
Additional Pragmas 98
#pragma ident "string" 98
#pragma Leaf File 98
#pragma Uses Propagation 98
Pragmas Used with SNMP Instrumentation 99
#pragma WrapCounter 99
#pragma ObjectID 99
#pragma DotNotation 99
#pragma HexNotation 100
Pragma Warnings in the MODEL Compiler 100
"Unrecognized Pragma" Warnings 100
"Ignored Pragma" Warnings 100

Preface

The basic operation of InCharge applications is described in *An Introduction to InCharge Technologies*. This description holds true whether you use an off-the-shelf InCharge application or you develop your own.

To develop an InCharge application, you may need to provide one or more of the following items:

- A correlation model that describes the domain or system to be managed. Correlation models are written in the Managed Object Definition Language (MODEL). This document is a reference guide to MODEL. As such, it serves as a comprehensive reference for the syntax and use of MODEL declarations.
- A source of topology that provides up-to-date information about the managed domain to the domain manager. Adapters are often used to retrieve topological data from an authoritative source and relay the data to the domain manager. Refer to *InCharge Software Development Kit ASL Reference Guide* for information about developing adapters. Accurate topological information is essential to ensure accurate correlation and root-cause analysis.

A source of events that enables the domain manager to monitor the managed domain. The domain manager can use different methods to obtain this information. Examples include retrieving data through polling, processing traps, or receiving data from an adapter.

Intended Audience

This guide is intended for programmers who are developing correlation models for InCharge applications. Those interested in learning more about MODEL and how it works may also find the "Introduction by Example" on page 5 especially useful.

Prerequisites

You must install the InCharge Software Development Kit to compile MODEL files. In addition, the examples used in this book are included on the InCharge CD-ROM in the */examples* directory. You may find it useful to copy them to the location where your SMARTS software is installed.

Document Organization

Table provides a brief summary for each section of this document.

1. About MODEL	Describes MODEL, its purpose, and how it models event information.
2. Introduction by Example	Introduces basic MODEL declarations through examples.
3. Compiling MODEL Files	Describes the process for compiling MODEL files.
4. Working with MODEL Libraries	Describes how to load a MODEL library into a domain manager.
5. Basic Lexical Elements of MODEL	Describes the data types, keywords, and literals supported by MODEL.
6. Declaring an Interface	Describes the syntax of an interface declaration.
7. Attribute Declarations	Describes the syntax of attribute declarations.
8. Relationship Declarations	Describes the syntax of relationship declarations.
9. Declaring Events in MODEL	Describes the syntax of the different declarations used to define events.
10. Operation Declarations	Describes the syntax of operations.
11. Writing Expressions in MODEL	Describes the lexical elements for expressions and their syntax.
12. Instrument Declarations	Describes the syntax of instrument declarations.
13. MODEL Pragmas	Describes pragmas and their use.

Document Organization

Documentation Conventions

Several conventions may be used in this document as shown in Table 1.

CONVENTION	EXPLANATION
`sample code`	Indicates code fragments and examples in Courier font
keyword	Indicates commands, keywords, literals, and operators in bold
%	Indicates C shell prompt
#	Indicates C shell superuser prompt
<parameter>	Indicates a user-supplied value or a list of non-terminal items in angle brackets
[option]	Indicates optional terms in brackets
/InCharge	Indicates directory path names in italics
yourDomain	Indicates a user-specific or user-supplied value in bold, italics
File > Open	Indicates a menu path in italics
▼▲	Indicates a command that is formatted so that it wraps over one or more lines. The command must be typed as one line.

Table 1: Documentation Conventions

Directory path names are shown with forward slashes (/). Users of the Windows operating systems should substitute back slashes (\) for forward slashes.

Also, if there are figures illustrating consoles in this document, they represent the consoles as they appear in Windows. Under UNIX, the consoles appear with slight differences. For example, in views that display items in a tree hierarchy such as the Topology Browser, a plus sign displays for Windows and an open circle displays for UNIX.

Finally, unless otherwise specified, the term InCharge Manager is used to refer to InCharge programs such as Domain Managers, Global Managers, and adapters.

InCharge Installation Directory

In this document, the term ***BASEDIR*** represents the location where InCharge software is installed.

- For UNIX, this location is: */opt/InCharge<n>/<productsuite>*.
- For Windows, this location is: *C:\InCharge<n>\<productsuite>*.

The *<n>* represents the InCharge software platform version number. The *<productsuite>* represents the InCharge product suite that the product is part of.

Table 2 defines the *<productsuite>* directory for each InCharge product.

PRODUCT SUITE	INCLUDES THESE PRODUCTS	DIRECTORY
IP Management Suite	• InCharge IP Availability Manager • InCharge IP Performance Manager • InCharge Discovery Manager • InCharge Adapter for HP OpenView NNM • InCharge Adapter for IBM/Tivoli NetView	/IP
Service Assurance Management Suite	• InCharge Service Assurance Manager • Global Console • InCharge Business Dashboard • InCharge Service Assurance Manager Business Impact Manager • InCharge Service Assurance Manager Report Manager • InCharge Service Assurance Manager Failover System • InCharge Service Assurance Manager Notification Adapters • InCharge Service Assurance Manager Adapter Platform • InCharge SQL Data Interface Adapter • InCharge SNMP Trap Adapter • InCharge Syslog Adapter • InCharge XML Adapter • InCharge Adapter for Remedy • InCharge Adapter for TIBCO Rendezvous • InCharge Adapter for Concord eHealth • InCharge Adapter for InfoVista • InCharge Adapter for NetIQ AppManager	/SAM
Application Management Suite	• InCharge Application Services Manager • InCharge Beacon for WebSphere • InCharge Application Connectivity Monitor	/APP
Security Infrastructure Management Suite	• InCharge Security Infrastructure Manager • InCharge Firewall Performance Manager • InCharge Adapter for Check Point/Nokia • InCharge Adapter for Cisco Security	/SIM

PRODUCT SUITE	INCLUDES THESE PRODUCTS	DIRECTORY
SMARTS Software Development Kit	• Software Development Kit	/SDK

Table 2: Product Suite Directory for InCharge Products

For example, on UNIX operating systems, InCharge IP Availability Manager is, by default, installed to */opt/InCharge6/IP/smarts*. This location is referred to as ***BASEDIR****/smarts*.

Optionally, you can specify the root of ***BASEDIR*** to be something other than */opt/InCharge6* (on UNIX) or *C:\InCharge6* (on Windows), but you cannot change the *<productsuite>* location under the root directory.

For more information about the directory structure of InCharge software, refer to the *InCharge System Administration Guide*.

Additional Resources

In addition to this manual, SMARTS provides the following resources.

InCharge Commands

Descriptions of InCharge commands are available as HTML pages. The *index.html* file, which provides an index to the various commands, is located in the ***BASEDIR****/smarts/doc/html/usage* directory.

Documentation

Readers of this manual may find other SMARTS documentation (also available in the ***BASEDIR****/smarts/doc/pdf* directory) helpful.

InCharge Documentation

The following SMARTS documents are product independent and thus relevant to users of all InCharge products:

- *InCharge Release Notes*
- *InCharge Documentation Roadmap*
- *InCharge System Administration Guide*
- *InCharge ICIM Reference*

- *InCharge Dynamic Modeling Tutorial*
- *InCharge ASL Reference Guide*
- *InCharge Perl Reference Guide*

Software Development Kit Documentation

The following SMARTS documents are relevant to users of the Software Development Kit.

- *InCharge Software Development Kit Remote API for Java* (in HTML format)
- *InCharge Software Development Kit Remote API Programmer's Guide*
- *InCharge ICIM Reference* (in HTML format)
- *InCharge Software Development Kit MODEL Reference Guide*

Common Abbreviations and Acronyms

The following lists common abbreviations and acronyms that are used in the InCharge guides.

ASL	Adapter Scripting Language
CDP	Cisco Discovery Protocol
ICIM	InCharge Common Information Model
ICMP	Internet Control Message Protocol
IP	Internet Protocol
MSFC	Multilayer Switch Feature Card
MIB	Management Information Base
MODEL	Managed Object Definition Language
RSFC	Router Switch Feature Card
RSM	Router Switch Module
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
VLAN	Virtual Local Area Network

Technical Support

SMARTS provides technical support by e-mail or phone during normal business hours (8:00 A.M.—6:00 P.M. U.S. Eastern and Greenwich Mean Time). In addition, SMARTS offers the InCharge Express self-service web tool. The web tool allows customers to access a personalized web page and view, modify, or create help/trouble/support tickets. To access the self-service web tool, point your browser to:

https://websupport.smarts.com/SelfService/smarts/en-us

U.S.A Technical Support
E-Mail: *support@smarts.com*
Phone: +1.914.798.8600

EMEA Technical Support
E-Mail: *support-emea@smarts.com*
Phone: +44 (0) 1753.878140

Asia-Pac Technical Support
E-Mail: *support-asiapac@smarts.com*

You may also contact SMARTS at:

	U.S.A World Headquarters	United Kingdom
Address	SMARTS 44 South Broadway White Plains, New York 10601 U.S.A	SMARTS Gainsborough House 17-23 High Street Slough Berkshire SL1 1DY United Kingdom
Phone	+1.914.948.6200	+44 (0)1753.878110
Fax	+1.914.948.6270	+44 (0)1753.878111

For sales inquiries, contact SMARTS Sales at:
sales@smarts.com

SMARTS is on the World Wide Web at:
http://www.smarts.com

1

About MODEL

The Managed Object Definition Language (MODEL) is a declarative language used to develop correlation models for InCharge applications. MODEL provides constructs for defining types of managed objects and describing their properties. Properties include attributes, operations, relationships, and events.

The main purpose of MODEL is to provide a simple means of describing the potential components of a managed domain in a reusable object-oriented framework. Developing your management application with InCharge provides two advantages.

- MODEL is an object-oriented language in which you declare a static class from which many objects (or instances) can be instantiated at runtime.
- InCharge applications separate the generic correlation model (rules) from topology. Because the model you develop is not tied to a particular topology, it can be used to model different domains that consist of similar types of managed objects.

MODEL is based on CORBA IDL (*The Common Object Request Broker: Architecture and Specification,* Object Management Group and Xopen, 1992). MODEL uses CORBA IDL notation where possible, introducing new syntax to address semantic concepts not provided by CORBA IDL. The new syntax supports event information including problems, symptoms, and their propagation.

Modeling Event Information

Events are observable conditions that occur in objects of the managed domain. Event information is essential to the root-cause analysis that is performed by the InCharge Domain Manager.

MODEL recognizes the importance of event modeling and supports the following concepts as they relate to the occurrence and effect of events in a managed system.

1. Events occur in objects and are modeled as object properties. Events can also be imported from an external source.
2. Events may be symptoms or problems and sometimes both. A symptom is an event whose presence may indicate a problem. Conversely, a problem is an event that causes symptoms. When the problem is fixed, the symptoms that it causes no longer occur.
3. Events can propagate from one object to another through a relationship. This is important to understand because the symptoms of a problem cannot always be observed in the object where the problem occurs. Instead, the symptoms of such a problem must be detected in related objects to properly diagnose the problem.

How InCharge Detects Events

An InCharge application is event driven. When an event is detected, the domain manager sends a notification to its client(s) indicating that the event has occurred. When the domain manager determines that the notified event is no longer occurring, it clears the notification to indicate that the conditions that caused the event are no longer present.

In MODEL, events are declared as Boolean expressions constructed with the values of attributes or other events. When the domain manager monitors an event, it determines which attributes are required to evaluate the event expression. The domain manager builds a data structure that links these attributes to the event expression and monitors those attributes. When the value of a monitored attribute changes, the domain manager re-evaluates the event expression to determine if its truth value has changed.

Determining Which Events are Monitored

A domain manager does not automatically monitor for every event declared in the correlation model. Instead, the domain manager only monitors for events that have been subscribed to by InCharge clients, which include InCharge consoles and adapters.

When an InCharge client requests to be notified of an event (by subscribing to the event), the domain manager monitors only those attributes and events necessary to determine when the event occurs. Attributes that do not affect the evaluation of a subscribed-to event are not monitored. When the value of an attribute that is not monitored changes, the domain manager does not re-evaluate any event expressions. The values of all attributes, monitored or not, are visible through an InCharge console and accessible to InCharge clients.

2

Introduction by Example

This chapter uses an example model to introduce the basic MODEL declarations that you will use to develop correlation models. The detailed syntax of MODEL declarations is described in later chapters.

The examples of MODEL code used in this chapter are included on the InCharge CD-ROM. The example code is furnished as a complete model, including the necessary makefile for each supported platform. You can compile this model and load the resulting MODEL library into a domain manager.

Class Properties

An InCharge Domain Manager's Repository maintains a runtime database of managed objects instantiated from MODEL classes. The Repository also maintains an index of all instances of a class.

Each class has properties that define it within the Repository. These properties are listed below.

- Inheritance defines the hierarchy of classes. Object classes are related in a subclass/superclass hierarchy. All of the properties defined for a superclass are inherited by its subclasses. A subclass may refine existing properties or add new properties and operations to introduce unique behavior or states.

- A class can be declared as abstract or concrete. Concrete classes are directly instantiable and must define implementations for all their properties. Abstract classes cannot be instantiated and may have properties for which they do not define implementations.

Declaring a Class in MODEL

A class defines the state and behavior for all its instances. In MODEL, a class is defined by an interface declaration. As a rule, MODEL constructs are specified within an interface declaration.

An interface declaration defines a managed object class in the Repository. Each interface must inherit from another interface. There are two base classes from which you can inherit: MR_ManagedObject and MR_MetaObject. MR_ManagedObject serves as the base class for all managed objects. When you develop a correlation model, the classes that represent types of managed objects should inherit from MR_ManagedObject. MR_MetaObject serves as the base class for objects that do not correspond to managed elements. Objects derived from MR_MetaObject may be used for configuration and control purposes. As such, MR_MetaObject is rarely used.

Interface Declarations

An interface is composed of an interface header declaration followed by a sequence of declarations enclosed in curly braces. Properties of the interface are specified between the curly braces.

Examples of Interface Declarations

The following code fragment declares five interfaces. The first two interfaces, Department and Person, are subclasses of MR_ManagedObject. Employee is a subclass of Person and has two subclasses, Director and Manager. Each subclass inherits all of the properties of its superclass. Because Person is declared as abstract, instances of this class cannot be instantiated. Employee is a subclass of Person but is not declared as abstract. Instances of Employee can be created to populate the topology.

The quoted strings that appear in each declaration are descriptions. Most MODEL declarations have an optional description field. The descriptions are stored in the Repository and can be displayed by clients such as the InCharge console. You can use more than one line for a description, but each line must begin and end with quotation marks. When a description spans more than one line, add an extra space before the quotation marks at the end of each line. This ensures that the description is readable when the InCharge console, or other client, concatenates the lines that compose the description.

```
interface Department : MR_ManagedObject
"The departments forming the company"
{
}

abstract interface Person : MR_ManagedObject
"A generic description of a person"
{
}

interface Employee : Person
"A person who works for the company"
{
}

interface Manager : Employee
"An employee who has managerial responsibilities"
{
}

interface Director : Employee
"An employee who has directorial responsibilities"
{
}
```

Modeling an Object's Properties

The state of an object is defined by the values of its state properties. Attributes of an object and relationships between an object and one or more other objects are examples of an object's state properties.

Attribute Declarations

The attribute declaration is one of the basic declarations that comprise an interface declaration. An attribute declaration specifies a property that is present in all instances of the interface.

The value of an attribute can be stored in the Repository, computed on demand, propagated from other objects through a relationship, or instrumented. An instrumented attribute is an attribute whose value is accessed through a standard protocol defined by an instrumentation declaration.

An attribute declaration includes the attribute's type, and optionally, a description, access type, access flag, and initial value. MODEL supports most built-in C++ data types. For a list of supported types, see "Supported Data Types" on page 44. At runtime the attributes obtain values. These values describe the state of the instance.

Examples of Attribute Declarations

The following code fragment declares attributes for the interfaces from the previous example. Each declaration specifies the attribute's type and access type if it is not stored. Note that an expression in a computed attribute must be defined in-line, as shown by the attribute Wages in the class Employee. Computed attributes are read only. Their value cannot be set through an InCharge client.

```
interface Department : MR_ManagedObject
"The departments forming the company"
{
    // Attributes
    attribute short DeptNumber
    "Department Number"
    = 0;

    attribute float DeptBudget
    "Budget for wages"
    = 0;
}
abstract interface Person : MR_ManagedObject
"A generic description of a person"
{
    // Attributes
    attribute string LastName
    "Person's last name";

    attribute string FirstName
    "Person's first name";
}
```

```
interface Employee : Person
"The people that work for the company"
{
    // Attributes
    attribute float LowWage_Thr
    "The minimum salary expected for an employee"
    = 5.15;
    attribute float HoursWorked_Thr
    "The maximum number of hours an employee should be "
    "allowed to work per week"
    = 60;
    attribute string EmployeeID
    "Employee identification number"
    = "0000";
    attribute float HoursWorked
    "Number of hours an employee works per week"
    = 40;
    attribute float HourlyRate
    "The employee's hourly wage"
    = 13;
    readonly computed attribute float Wages
    "The employee's weekly salary"
    = HoursWorked * HourlyRate;
}
```

Relationship Declarations

A relationship maps instances of one class to instances of the same or another class. For each relationship, an inverse relationship may also exist that maps from the related class back to the originating class. We use the term *relation* to refer to the pair of relationships between two objects.

The cardinality of a relationship can be to-one or to-many. A relationship from class C1 to class C2 is to-one when an object of class C1 can be mapped by the relationship to, at most, one object of class C2. A relationship from class C1 to class C2 is to-many when an object of class C1 can be mapped by the relationship to zero or more objects of class C2.

The reciprocal relationship between two objects may have a different cardinality than the original relationship. Because of this, the relation between classes C1 and C2 is one-to-one when the pair of relationships that define the relation are to-one relationships. The relation between two classes is one-to-many when one of the relationships is to-one and the other relationship is to-many. The relation is many-to-many when both relationships are to-many.

A relationship declaration includes the cardinality of the relationship, the name of the relationship, the name of the related class, the name of the inverse relationship, and an optional description. Cardinality is determined by the keyword **relationship**, which declares a to-one relationship, or **relationshipset**, which declares a to-many relationship.

Examples of Relationship Declarations

The Department class declares two relationships: ManagedBy and Contains. ManagedBy is a relationship between a Department and a Manager. Contains is a relationship between a Department and the employees who work in the department. Because a department potentially has many employees, Contains is declared as a relationshipset.

In the Employee class, the relationship WorksIn declares the inverse of the Contains relationship in the Department class. It declares that an employee can work in one department.

The Manager class contains the Manages relationshipset, which is the inverse of the ManagedBy relationship.

```
interface Department : MR_ManagedObject
"The departments forming the company"
{
    // Relationships
    relationship ManagedBy, Manager, Manages
    "The manager who manages this department";

    relationshipset Contains, Employee, WorksIn
    "The employees this department contains";
}
interface Employee : Person
"The people that work for the company"
{
    // Relationships
    relationship WorksIn, Department, Contains
    "The department the employee works in";
}

interface Manager : Employee
"An employee who has managerial responsibilities"
{
    // Relationships
    relationshipset Manages, Department, ManagedBy
    "The department this manager manages. "
    "A manager manages one department";
}
```

Propagate Attribute Declarations

Relationships define connections between classes of managed objects. The value of an attribute can be retrieved from an instance of a related class.

The value of a propagated attribute can be obtained from a single object through a relationship or it can be retrieved from multiple objects through a relationshipset. When the value of the propagated attribute is obtained through a relationshipset, you must specify an aggregate operator. The aggregate operator tells the domain manager how to combine the set of values formed by the value it receives from each instance participating in the relationshipset.

A propagate attribute declaration includes the relationship name the attribute is retrieved over, the related class, and the name of the attribute that propagates.

Examples of Propagated Attributes

The Department interface declares two propagated attributes. The first, AverageDeptWage, computes the average salary of the employees who work in the department. This requires retrieving the value of the Wages attribute from each instance of the Employee class through the Contains relationshipset. Because Contains is a relationshipset, an aggregate operator, avg (for average), must be specified.

The second propagated attribute, ManagerSalary, retrieves the salary of the manager who manages this department. This is accomplished through the ManagedBy relationship. Manager is a subclass of Employee and inherits the Salary attribute.

```
interface Department : MR_ManagedObject
"The departments forming the company"
{
    // Propagated attributes
    propagate attribute float avg AverageDeptWage
    "The average salary of employees in this "
    "department "
    = Employee, Contains, Wages;

    propagate attribute float ManagerSalary
    "The salary of the manager of this department"
    = Manager, ManagedBy, Wages;
}
```

Modeling Events

Events define the behavior for instances of a class. An event is an observable condition that occurs within an object. MODEL provides the following declarations for specifying events.

- Event declaration
- Problem declaration
- Propagate symptom declaration
- Aggregate declaration
- Propagate aggregate declaration
- Export declaration

Note: One of the MODEL declarations is named event declaration. This can lead to confusion because we also use the word event in a generic sense to mean an identifiable condition. We refer to the MODEL declaration as the event declaration and event, by itself, to mean an identifiable condition.

Event Declarations

An event declaration declares an event that is observable in instances of the class for which it is defined. It is important to note that an event declaration specifies an *observable* event. As such, the domain manager does not perform correlation to determine when such an event occurs. Instead, events specified by an event declaration are used by the domain manager to diagnose problems. By monitoring observable events, the domain manager is able to pinpoint problems (specified by problem declarations), which cause the observable events.

An event declaration specifies the name of the event, an event expression, and an optional description. Event expressions are written as Boolean expressions defined over attributes or other events. When an event expression evaluates to true, it means that event is occurring. When the event expression evaluates to false it either means that the event has cleared or that the event is not active.

Examples of an Event Declaration

The Employee class declares three events: UnderPaidEmployee, OverWorked, and UnderAchieved. UnderPaidEmployee occurs when the value of the attribute HourlyRate is less than the attribute LowWage_Thr. Similarly, OverWorked occurs when the attribute HoursWorked is greater than the attribute HoursWorked_Thr. UnderAchieved is defined using two previously declared events. When either UnderPaidEmployee or OverWorked occurs, the event UnderAchieved occurs.

```
interface Employee : Person
"The people that work for the company"
{
    // Events
    event UnderPaidEmployee
    "This employee is making less than the minimum wage"
    = HourlyRate < LowWage_Thr;

    event OverWorked
    "This employee is working too many hours during the "
    "week"
    = HoursWorked > HoursWorked_Thr;

    event UnderAchieved
    "The employee is not performing as expected"
    = UnderPaidEmployee || OverWorked;
}
```

Problem Declarations

A problem declaration declares an event that is detected by observing its symptoms. Symptoms are observable events that uniquely identify the problem. Note that the symptoms described here are not necessarily declared by a symptom declaration. The symptoms of a problem can be declared by event, symptom, and other problem declarations.

Like an event declaration, a problem declared by a problem declaration must be subscribed to by an InCharge client to make the domain manager monitor for it.

A problem declaration specifies the name of the problem, the list of symptoms that the problem causes, and an optional description.

As noted earlier, the domain manager only monitors for problems that an InCharge client has subscribed to. If a problem is not subscribed to, the domain manager does not monitor the attributes and events that the problem depends on.

Example of a Problem Declaration

The PoorPerformance problem causes the UnderAchieved event, making UnderAchieved a symptom of PoorPerformance. When the domain manager detects the UnderAchieved event, it concludes that PoorPerformance is the problem and sends a notification for the PoorPerformance problem. The example for event declarations, "Examples of an Event Declaration" on page 13, shows the attributes and events the domain manager would have to monitor to determine when UnderAchieved occurs.

```
interface Employee : Person
"A person who work for the company"
{
    // Problems
    problem PoorPerformance
    "The employee is not able to complete his or "
    "her activities. Therefore, the employee is "
    "forced to work extra hours"
    => UnderAchieved;
}
```

Propagate Symptom Declarations

It is not unusual for a problem declared in one class to cause events in a related class. MODEL supports this notion through the propagate symptom declaration.

The propagate symptom declaration specifies a set of events that are not observable in the object where they occur but whose effects propagate to related objects. It is important to note that although the name of the declaration is propagate symptom, it is more accurate to think of it as a propagated event. This is because a propagate symptom declaration can be used to propagate events from event, problem, and other propagate symptom declarations.

A propagate symptom declaration specifies the name of the propagated symptom, the name of the related class, the name of the relationship along which the event propagates, and the name of the symptom. By symptom, we mean the event, problem, or propagate symptom declaration that specifies the event in the related class.

Example of a Propagate Symptom Declaration

The Department class declares two propagated symptoms: UnderPaidEmployees and UnderAchieved. Both these events occur in instances of the Employee class. However, neither of these events are observable in instances of the Department class. Instead, they are detected through their symptoms that propagate from instances of the Employee class through the Contains relationship.

Unlike a propagate attribute declaration, a propagate symptom declaration does not use an aggregate operator. If any one of the symptoms of a propagate aggregate occurs, the propagate aggregate occurs.

```
interface Department : MR_ManagedObject
"The departments forming the company"
{
    propagate symptom UnderPaidEmployee
    "The employees in this department that are underpaid"
    = Employee, Contains;

    propagate symptom UnderAchieved
    "The employees in this department that are not "
    "peforming as expected"
    = Employee, Contains;
}
```

Note: If the name of the event (symptom) that propagates from the related class is not specified, it means that the event has the same name as the propagated symptom.

Aggregate Declarations

An aggregate declaration specifies a set of events that are grouped into a single event by disjunction. When any one event in the set occurs, the aggregate event occurs.

Like event and problem declarations, an event specified by an aggregate declaration must be subscribed to for the domain manager to monitor it. When an InCharge client subscribes to an aggregate event, the domain manager only monitors those attributes and events necessary to determine when the aggregate occurs.

An aggregate declaration specifies the name of the aggregated event, an optional description, and a list of one or more events that comprise the aggregate. Events specified by event, problem, and aggregate declarations can be used in an aggregate declaration.

Note: When a notification for an aggregate declaration is displayed in the InCharge console, it is referred to as a compound notification.

Example of an Aggregate Declaration

The LegalException aggregate is a single event declared with the UnderpaidEmployee and Overworked events. When either the UnderpaidEmployee or the Overworked events occur, the LegalException aggregate also occurs. Both events that compose the aggregate are also declared in the Employee class.

```
interface Employee : Person
"The people that work for the company"
{
    // Aggregate
    aggregate LegalException
    "The employee is experiencing an illegal condition in "
    "his/her working environment"
    = UnderPaidEmployee,
      OverWorked;
}
```

Propagate Aggregate Declarations

A propagate aggregate declaration specifies an event whose set of symptoms are events declared in related classes. When one of the symptoms in the set occurs, it propagates to the instance where the propagate aggregate is declared.

The syntax of the propagate aggregate declaration is similar to that of the propagate symptom declaration. A propagate aggregate declaration specifies the name of the propagated aggregate, an optional description, and the names of the related class, the relationship along which the symptoms propagate, and the symptom in the related class.

Example of a Propagate Aggregate Declaration

The LegalException propagate aggregate declared in the Department class propagates from instances of the Employee class through the Contains relationship. LegalException is also the name of the event in the event declaration in the Employee class, so it is not necessary to provide its name. When LegalException occurs in any instance of Employee, the LegalException becomes active in the Employee's Department.

```
interface Department : MR_ManagedObject
"The departments forming the company"
```

```
{
    propagate aggregate LegalException
    "This department contains employees that work "
    "under illegal conditions. The employment contract "
    "for these employees must be reviewed"
    = Employee, Contains;
}
```

Export Declaration

Exported events are those events that are made visible to InCharge clients. If an event is not exported, it remains private and is not accessible outside of the domain manager's Repository. When you declare an event as exported, it is accessible to InCharge clients, so they can subscribe to the event and receive notifications when the event occurs.

A single export declaration can declare all of the public events for a class. Events declared by event, problem, aggregate, and propagate aggregate declarations can be exported.

The export declaration specifies, by name, the events that are to be exported.

Example of an Export Declaration

The Department class exports a single event, the aggregate Legal Exception, while the Employee class exports one problem and three events. To make MODEL code easier to read, we use separate declarations to export events and problems.

```
interface Department : MR_ManagedObject
"The departments forming the company"
{
    // Exported Aggregates
    export
    LegalException;
{

interface Employee : Person
The people that work for the company"
{
    // Exported problems
    export
    PoorPerformance;

    // Exported events
    export
    UnderPaidEmployee,
    OverWorked,
```

```
    UnderAchieved;
}
```

Refining an Object's Properties

Refinement provides a method for modifying a class property in a subclass. Attribute, relationship, event, problem, propagate symptom, aggregate, and propagate aggregate declarations may be refined. The refinement must be declared in a subclass.

A refined property has the same name as the property it modifies and is indicated by the keyword **refine**. A property's description can always be changed through a refinement.

- For an attribute declaration, a refinement can change the attribute's initial value and its access type.
- For a relationship or relationshipset declaration, a refinement can change the class that participates in the relationship. The interface where the relationship is originally declared must be a superclass of the interface where the relationship is refined.
  In addition, the cardinality of the refined relationship must remain the same as that of the original relationship.
- For an event declaration, a refinement can change all the properties of the event.
- For a problem declaration, a refinement can change all the properties of the problem.
- For an aggregate declaration, a refinement can change all the properties of the aggregate.
- For propagate aggregate and propagate symptom declarations, a refinement can change all the properties of the declarations.

The Complete Example

Below is the complete model from which the examples in this chapter were taken. This example is also included on the InCharge CD-ROM with the makefiles necessary to compile it into a MODEL library. See the *InCharge Software Development Kit Installation Guide* appropriate for your operating system for instructions on how to install these files onto your system.

```
#include <repos/managed_object.mdl>

#pragma include_h <repos/managed_object.h>
#pragma include_c "department.h"

interface Department : MR_ManagedObject
"The departments forming the company"
{
    // Export Aggregates
    export
       LegalException;

    // Aggregates
    propagate aggregate LegalException
    "This department contains employees that work under "
    "illegal conditions. The employment contract for "
    "these employees must be reviewed"
    = Employee, Contains;

    // Relationships
    relationship ManagedBy, Manager, Manages
    "The manager who manages this department";

    relationshipset Contains, Employee, WorksIn
    "The employees this department contains";

    // Attributes
    attribute short DeptNumber
    "Department Number"
    = 0;

    attribute float DeptBudget
    "Budget for wages"
    = 0;

    // Propagated attributes
    propagate attribute float avg AverageDeptWage
    "The average salary of employees in this department"
    = Employee, Contains, Wages;

    propagate attribute float ManagerSalary
    "The salary of the manager of this department"
    = Manager, ManagedBy, Wages;

    propagate symptom UnderPaidEmployee
    "The employees in this department that are underpaid"
    = Employee, Contains;

    propagate symptom UnderAchieved
    "The employees in this department that are not "
```

```
    " performing as expected"
    = Employee, Contains;
}
abstract interface Person : MR_ManagedObject
"A generic description of a person"
{
    // Attributes
    attribute string LastName
    "Person's last name";

    attribute string FirstName
    "Person's first name";
}

interface Employee : Person
"A person who works for the company"
{

    // Exported problems
    export
       PoorPerformance;

    // Exported events
    export
       UnderPaidEmployee,
       OverWorked,
       UnderAchieved;

    // Problems
    problem PoorPerformance
    "The employee is not able to complete his or her "
    "activities. Therefore, the employee is forced "
    "to work extra hours "
    => UnderAchieved;

    // Events
    event UnderPaidEmployee
    "This employee is making less than the minimum wage"
    = HourlyRate < LowWage_Thr;
    event OverWorked
    "This employee is working too many hours during the "
    "week"
    = HoursWorked > HoursWorked_Thr;

    event UnderAchieved
    "The employee is not performing as expected"
    = UnderPaidEmployee ||
      OverWorked;
    // Aggregates
    aggregate LegalException
```

```
        "The employee is experiencing an illegal condition "
        "in his/her working environment"
        = UnderPaidEmployee,
          OverWorked;

        // Relationships
        relationship WorksIn, Department, Contains
        "The department the employee works in";

        // Attributes
        attribute float LowWage_Thr
        "The minimum wage for an employee"
        = 5.15;

        attribute float HoursWorked_Thr
        "The maximum number of hours an employee should be "
        "allowed to work per week"
        = 60;

        attribute string EmployeeID
        "Employee identification number"
        = "0000";

        attribute float HoursWorked
        "Number of hours an employee works per week"
        = 40;

        attribute float HourlyRate
        "An employee's hourly wage"
        = 13;

        readonly computed attribute float Wages
        "The employee's weekly salary"
        = HoursWorked * HourlyRate;

        propagate attribute float ManagerSalary
        "The weekly salary of the employee's manager"
            = Department, WorksIn;

    }
    interface Manager : Employee
    "An employee who has managerial responsibilities"
    {
        // Exported problems
        export
           StingyManager;

        // Problems
        problem StingyManager
        "This manager does not compensate employees properly, "
```

```
        "or his/her managerial skills are not provding the "
        "direction his employees require to perform better"
        => MyEmployeesUnderAchieved;

        // Events
        refine event OverWorked
        "This employee is working too many hours during a "
        "week period"
        = HoursWorked > 1.5 * HoursWorked_Thr;

        // Symptoms
        propagate symptom MyEmployeesUnderAchieved
        "The employees that are paid less than the minimun "
        "wage"
        = Department, Manages, UnderAchieved;

        // Relationships
        relationshipset Manages, Department, ManagedBy
        "The department this manager manages. "
        "A manager manages one department";

        relationship ReportsTo, Director, Supervises
        "The director this manager reports to";

    }
    interface Director : Employee
    "An employee who has directorial responsibilities"
    {

        // Relationships
        relationshipset Supervises, Manager, ReportsTo
        "The manager this director supervises";

    }
```

3

Compiling MODEL Files

The process of compiling MODEL code requires both the MODEL compiler and one of the supported C++ compilers. This chapter describes how to compile your MODEL code to produce a MODEL library that can be loaded into a domain manager.

Overview of the Compilation Process

The MODEL compiler translates each MODEL interface into a C++ class, with methods for each of its properties. Invoking the MODEL compiler on a MODEL file produces two output files, which are used as input by the C++ compiler.

- *file.h* contains declarations of the generated classes, with the data members and access type for each property.
- *file.c* contains definitions of the automatically generated methods.

The resulting C++ code is compiled and linked into an operating system-specific shared library. This shared library contains your model, which can be loaded into a domain manager. Figure 1 illustrates this process.

The name of the shared library will vary depending upon the system on which it was compiled. For example, if *example.mdl* is the source file, the name of the resulting shared library would be *libexample.so* on a Solaris system, *libexample.sl* on HP-UX systems, and *example.dll* on Windows systems.

When you load the library into a domain manager, you only specify the base name of the MODEL library. In the previous example, the base name of the MODEL library is *example*.

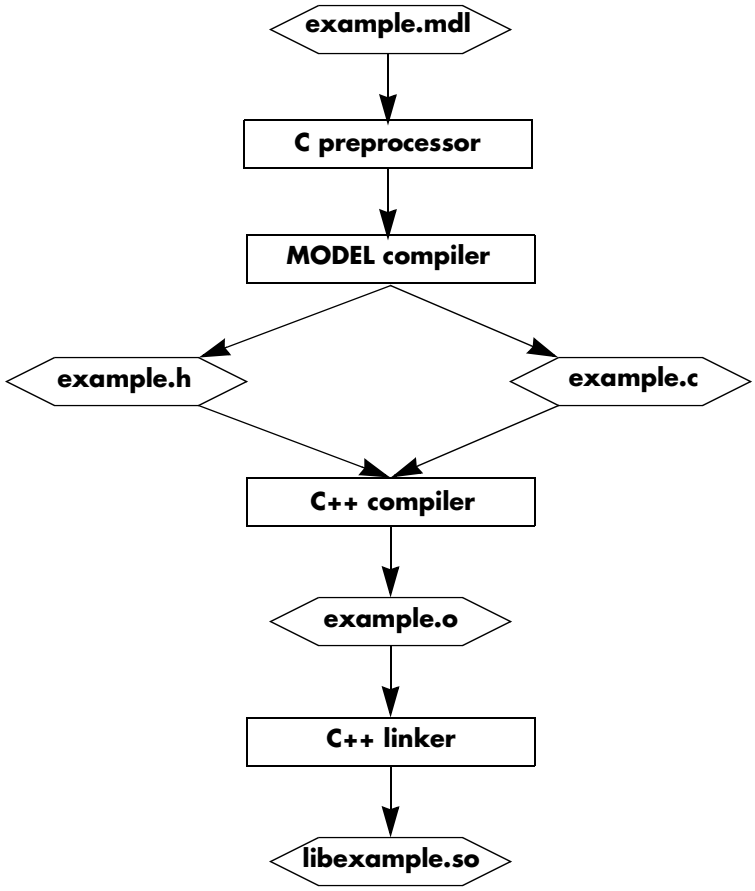

Figure 1: Compiling a MODEL File into a Shared Library

Specifying Additional Files with #include

A MODEL file may include other MODEL files with class definitions. The compiler uses these files for resolving references; it does not generate code for the included files.

Because all MODEL classes inherit from either MR_ManagedObject or MR_MetaObject, you need to include specifications for these files in your MODEL file. Typically, you will only need to include a reference to *managed_object.mdl*.

The MODEL compiler invokes a C preprocessor on MODEL files before it processes them. Because the included files are read by the C preprocessor, follow the conventions for C or C++ on your system. For example:

```
#include <repos/managed_object.mdl>
```

The MODEL compiler defines a preprocessor symbol for use in conditional compilation. It defines SM_COMPILER_MODEL so that you can specify conditional preprocessor directives that only are invoked when the code is processed by MODEL.

You may also break your model into multiple source files. In this case, you need to include any MODEL files that define parent classes of classes declared in the other MODEL files.

For example, if you have two MODEL files, *example.mdl* and *scenario.mdl*, and classes in *scenario.mdl* inherit from classes declared in *example.mdl*, you need to include *example.mdl* in *scenario.mdl*.

You do not need to include *managed_object.mdl* in *scenario.mdl* because it is included in *example.mdl*, the parent of *scenario.mdl*. However, you must include *example.mdl* in *scenario.mdl*.

Similarly, you must include the *example.h* file, but not *managed_object.h,* in the generated *scenario.h* file. The *#pragma include_h* and *#pragma include_c* directives cause the MODEL compiler to add include statements for the given file name to the generated .h and .c files, respectively. For information regarding MODEL pragmas, see "MODEL Pragmas" on page 97.

The include definitions for these files would look like this:

```
# example.mdl

#pragma include_h <repos/managed_object.h>
#pragma include_c <example.h>
#include <repos/managed_object.mdl>

#scenario.mdl

#pragma include_h <example.h>
#pragma include_c <scenario.h>
#include <example.mdl>
```

Using the MODEL Compiler

The recommended method for invoking the MODEL compiler is to use the makefiles supplied with the developer's kit. This ensures that the proper arguments and flags are set for the MODEL and C++ compilers.

Refer to the *InCharge Software Development Kit Installation Guide* appropriate for your operating system for information regarding supported compilers, patch information, and other operating-system specific information.

Using the Proper Makefile

Platform-specific makefiles are included on the InCharge Software Development Kit CD-ROM. To access the makefiles, copy the */examples* directory, and its subdirectories, from the root level of the CD-ROM to the ***BASEDIR****/smarts/*directory. The makefiles will be located in the ***BASEDIR****/smarts/examples/model* directory. The name of the operating system for which the makefile is intended is appended to the name of the makefile; for example, *Makefile.solaris*.

Several sample models are also included so that you may test the MODEL compiler in your environment before compiling models you have developed.

The makefiles included with the InCharge Software Development Kit have been tested with the make program provided by the operating system vendor. GNU, or other make programs, can be used with little or no modification.

WARNING: It is very important that you do not change the flags or options specified in the makefiles for the MODEL or the C++ compiler. These are configured to compile position-independent code that supports multi-threaded execution.

Compiling MODEL Files

To compile your model, edit the makefile to specify the name of your model and the names of any MODEL include files. If the InCharge software or C++ compiler is not installed in its default location, you also need to edit the specified path to this software.

Compiling on Solaris

To compile a model on a Solaris system, issue the following command:

```
% make -f Makefile.solaris
```

The default setting of the makefile assumes that the supported version of the SunPro C++ compiler, CC, is in your PATH environment variable.

Compiling on HP-UX

To compile a model on an HP-UX system, issue the following command:

```
% make -f Makefile.hpux
```

The default setting of the makefile assumes that the supported version of the HPUX ANSI C++ compiler, aCC, is in your PATH environment variable.

Compiling on Windows 2000 and Windows 2003

To compile a model on a Windows 2000/2003 system, issue the following command:

```
c:\ nmake -f Nmakefile.nt
```

It may be necessary to run the VCVARS32.BAT program to set the Visual C++ environment variables.

Invoking the MODEL Compiler

The MODEL compiler is installed into the ***BASEDIR***/*smarts/bin* directory. You can invoke it with the following command:

```
% BASEDIR/smarts/bin/model
```

The MODEL compiler supports the options listed in Table 3.

OPTION	DEFINITION
`-D<name>[=<value>]`	Define *<name>* for the preprocessor.
`-U<name>`	Undefine *<name>* for the preprocessor.
`-E`	Run the preprocessor on the MODEL files and write the results to standard output.
`-w`	Suppress warning messages generated by the MODEL compiler.
`-WlibName=`*<name>*	Specify the name of the shared library that the generated C++ code will be linked into. Specifying this option is not necessary on UNIX systems. Neglecting to specify this option on Windows 2000/2003 will cause the linker to generate unnecessary warning messages when building the library.
`--help`	Display usage information.
`--version`	Display version of the MODEL compiler.

Table 3: Options to the MODEL Compiler

As noted above, SMARTS recommends that you use the makefiles supplied with the InCharge Software Development Kit to invoke the MODEL compiler.

4

Working with MODEL Libraries

After you successfully compile a correlation model, the next step is to load the MODEL library into a domain manager and test it. The purpose of testing is to ensure that:

- The proper events are triggered when the values of attributes change
- The problems and aggregations occur when expected
- The events are propagating along relationships as expected

If your model includes complex expressions in event or attribute declarations, you should test these expressions to ensure that they return the expected results.

You can use several different tools to load and test a MODEL library. This chapter briefly describes these tools and explains various procedures for testing your correlation model.

Tools for Working with MODEL Libraries

You can perform the procedures for loading and testing a correlation model with any of the tools described below. You can choose the tool you prefer to use and follow the procedures for that tool. These tools include the Generic Console, the dmctl utility, and adapters.

- The Generic Console is a graphical interface you can use to connect to a domain manager. It is installed with the Global Console from the InCharge Service Assurance Management Suite CD-ROM.
- The dmctl utility is a command line tool for connecting to a domain manager.
- Adapters can also be used to load and test MODEL libraries. The methods for doing this are not discussed in this book. Please refer to the *InCharge Software Development Kit ASL Reference Guide* book for more information.

Using the InCharge Generic Console

The Generic Console provides a graphical interface to a running domain manager. You can use the Generic Console to:

- View the class hierarchy of a correlation model
- Add or delete instances
- Set the value of attributes
- Insert instances into or remove instances from relationships
- View notifications

The default layout of the Generic Console includes two views: a Topology Browser view and an Alarm Log view. You can add or delete views to configure the console. You may find the Domain Browser view useful because it can display the class hierarchy of a model loaded into the domain manager.

Starting the Generic Console

To start the Generic Console, supply the `--generic` option to the **sm_gui** command.

```
% sm_gui -c generic
```

Using the dmctl Command-Line Interface

The dmctl utility provides a command line interface to a running domain manager. You can use it to perform tasks such as adding or deleting objects, setting the value of attributes, and inserting objects in or removing objects from relationships.

The dmctl utility can operate in two modes: you can establish a continuous connection with a domain manager, or you can open and close a connection for each command. When you perform multiple operations, a continuous connection is easier because you do not have to specify the name of the domain manager for each operation.

To establish a continuous connection with a running domain manager, specify the name of the domain manager with the `--server` option, where <domain> is the name of the domain manager:

```
% dmctl --server=<domain>
```

To use dmctl in such a way that it opens and closes the command for the operation, specify the name of the domain manager and the command. The following command shuts down the specified domain manager.

```
% dmctl --server=<domain> shutdown
```

We use dmctl in the following sections to create instances, insert instances into relationships, supply values to attributes, and generate events for the example model described in "The Complete Example" on page 18.

More information regarding dmctl is available in two places:

- The HTML documentation for InCharge commands, which is installed into the ***BASEDIR****/smarts/doc/html/usage* directory. The help file for dmctl is *dmctl.html*.
- Information is also available by specifying the `--help` option:

  ```
  % dmctl --help
  ```

  You can retrieve the complete list of dmctl commands by specifying the `--commands` option:

  ```
  % dmctl --commands
  ```

Loading a Model Library

You can load a MODEL library into a domain manager when you start it or after it is already running. For security purposes, the domain manager looks in a specific directory for MODEL libraries. You can specify additional locations with an environment variable.

Location of MODEL Libraries

Because the domain manager may run with root or administrator privileges, we have limited the domain manager, and other SMARTS commands, to loading libraries from specific directories. When invoked, SMARTS programs append the SM_LIBPATH environment variable and the ***BASEDIR****/smarts/lib* path to the system library variable described in Table 4.

OPERATING SYSTEM	LIBRARY VARIABLE
HP-UX	SHLIB_PATH
Solaris	LD_LIBRARY_PATH
Windows 2000/2003 (also used to locate main programs)	PATH

Table 4: Library Variables for Supported Operating Systems

You can specify additional directories by setting the SM_LIBPATH environment variable. By default, the SM_LIBPATH environment variable is not set.

Setting SM_LIBPATH on Windows

On Windows systems, you can specify additional directories for the current user session from the command prompt.

```
C:> SET SM_LIBPATH=<path>;<path>;<path>
```

Setting SM_LIBPATH on UNIX

On UNIX systems, the method for setting an environment variable differs according to the shell. The instructions below assume sh or ksh.

```
% SM_LIBPATH=<path>:<path>:<path>
% export SM_LIBPATH
```

Where <*path*> is a directory where MODEL libraries are located. More than one path may be specified by separating the paths with a colon (:).

Starting a Domain Manager

The InCharge Software Development Kit includes a runtime license for a domain manager. This enables you to test your model and any adapters you have developed. Follow the procedures in the *InCharge System Administration Guide* to obtain a permanent license.

The steps below assume that the InCharge Broker is already running, as described in the InCharge Software Developer's Kit Installation Guide.

To start a domain manager, use the dmstart command. The syntax of the command is same for UNIX and Windows 2000/2003 systems,

```
$ BASEDIR/smarts/bin/dmstart --name=<domain>
```

where <domain> is the name of the domain manager. You can view the complete list of options to the **dmstart** command by specifying the `--help` option.

```
C:\BASEDIR\smarts\bin\dmstart --help
```

Methods for Loading MODEL Libraries

There are several methods for loading a MODEL library into a domain manager.

- You can load a MODEL library when you start the domain manager.
- You can use the dmctl utility to load a MODEL library into a running domain manager.
- You can load a MODEL library into a running domain manager through the Generic Console.

Regardless of what method you use, you only specify the base name of the library when you load it into a domain manager. For example, on a Solaris system, the name of the MODEL library might be *libexample.so*. When you load the library, simply use *example* to specify the library.

Loading a MODEL Library at Domain Manager Startup

To load a MODEL library when you start the domain manager, specify the MODEL library with the `--model` option to the **dmstart** command.

```
% BASEDIR/smarts/bin/dmstart --name=<domain> --model=<model>
```

Where <*domain*> is the name of the domain manager and <*model*> is the name of the MODEL library.

Loading a MODEL Library with dmctl

To load a MODEL library to a running domain manager, use the dmctl command-line interface and the loadModel command.

```
% BASEDIR/smarts/bin/dmctl --server=<domain> \
loadModel <model>
```

This command must be run on a single line. The <*domain*> parameter is the name of the domain manager and <*model*> is the name of the MODEL library.

Loading a MODEL Library with the Generic Console

To load a MODEL library through the Generic Console, you must first start a domain manager and the Generic Console.

1. Add the Domain Browser view to the Generic Console.
2. Right-click on the icon for the domain manager and select **Properties** from the pop-up menu. This displays the property sheet for the domain manager in a new window.
3. Select the Modules tab.
4. Click on the **Load** button. This displays the Load Module dialog.
5. Select **Model** from the Type of Module drop-down menu.
6. Type the name of the model in the Module Location field. To be accessible by the domain manager, the MODEL library must be located in the proper directory, as specified by "Location of MODEL Libraries" on page 32.
7. Click **OK**, or **Apply** if you wish to load additional models, and then click **Close** to close the domain manager's property sheet.

Working with a MODEL Library and a Domain Manager

Once your model is compiled and loaded into a domain manager, you are ready to start testing. This section describes different methods for viewing your model, listing its contents, creating instances, and notifying events.

Methods for Listing Models Loaded into a Domain Manager

You can use the Generic Console and the dmctl utility to list the models loaded into a domain manager.

Listing Models with the Generic Console

To list the models loaded into a domain manager using the console, complete the following steps.

1. Open the property sheet for the domain manager.

   In the Domain Browser view, this is accomplished by right-clicking on the icon for the domain manager and selecting **Properties** from the pop-up menu. In the Topology Browser view, this is accomplished by selecting the domain manager.

2. Select the Modules tab. The models loaded into the domain manager are listed in the Models list box.

Listing Models with the dmctl Utility

To list the models loaded into a domain manager using dmctl, complete the following steps.

Use the getModels option to retrieve the list of models loaded into the domain manager.

```
% BASEDIR/smarts/bin/dmctl --server=<domain> getModels
```

Listing Classes in the MODEL Library

You can use the Generic Console and the dmctl utility to list the classes declared in the model. Note that classes declared as **abstract** are not visible through the console.

If more than one model is loaded into the domain manager, the methods described below list all the concrete classes from the loaded models.

Listing Classes with the Generic Console

To list the classes using the console, expand the plus sign next to the icon for the domain manager in either the Topology Browser view or the Domain Browser view.

You can also view the class hierarchy of your model through the Domain Browser view. Select the Class Browser tab at the bottom left of the view and then expand the plus sign next to the icon for the domain manager.

By default, the Topology Browser view does not list classes for which there are no instances. To remedy this, perform the following steps:

1. Right-click in a blank area of the Topology Browser view and select **Select** from the pop-up menu. This displays the Select tree content window.
2. Check the *Show classes with no instances* checkbox.
3. Click **OK**.

Listing Classes with the dmctl Utility

To list the classes using dmctl, use the getClasses command.

```
% BASEDIR/smarts/bin/dmctl --server=<domain> getClasses
```

Note: The list of classes retrieved by dmctl will also include classes used to configure and control the operation of the domain manager. Removing these classes or modifying the properties of their instances is not recommended.

Creating Instances of a Class

You can use the Generic Console and the dmctl utility to create instances of classes declared in your model. By creating instances, you build a topology of managed elements.

Creating Instances with the Generic Console

To create an instance of a class using the console, complete the following steps.

1. Select *Edit > Create Instance*. This displays the Create Instance window.
2. Select the proper domain (only necessary if the console is attached to more than one domain manager).
3. Select the class that the new instance should be instantiated from.
4. Type the name of the instance in the Instance field.
5. Click **Apply** to create additional instances or click **OK** to close the Create Instance window.

Creating Instances with the dmctl Utility

To create an instance using dmctl, use the **create** command. You must specify the class name and an instance name.

In the example below, dmctl is already connected to a domain manager. This enables you to type commands without having to first attach to the domain manager.

```
% dmctl> create <class_name>::<instance_name>
```

Modifying the Properties of an Instance

You can use the Generic Console and the dmctl utility to modify the properties of instances. This includes changing the values of attributes and inserting instances into relations.

Note: When you insert an instance into a relation, the instance that is on the other end of the relation must already exist.

Changing an Attribute Value with the Generic Console

To change the value of an attribute using the console, complete the following steps.

1. Select the instance with the attribute you want to change and then choose *Edit > Edit Instance*. This displays the property sheet for the selected instance.
2. Double-click in the Value field of the attribute you want to change so that the current value is selected or the cursor is blinking in the Value field.
3. Type a new value, followed by the **Enter** key.
4. Click **Apply** to save your changes.

Changing an Attribute Value with the dmctl Utility

To change the value of an attribute with dmctl, use the **put** command as shown below.

```
dmctl> put <class>::<instance>::<attribute> <value>
```

Note: The value of attributes declared as read-only, which includes computed attributes, cannot be changed by an InCharge client.

Inserting an Instance into a Relation with the Generic Console

To insert an instance into a relation using the console, complete the following steps.

1. Select the instance that you want to insert into a relation and then choose *Edit > Edit Instance*. This displays the property sheet for the selected instance.
2. Select the Relations tab to list the relations that the selected instance can participate in.

3 Select the relation you want to insert the instance into and then click **Insert**. This displays the Insert Relation window.

Note: The name of the inverse relation is enclosed in brackets after the relation name.

4 Select the class of the other instance participating in the relation.

5 Type the name of the other instance in the Instance field.

6 Select **Apply** to insert the selected instance into additional relations or click **OK** to close the Insert Relation dialog.

An error is returned and the operation fails if an instance is inserted into an invalid relation or if the other instance does not exist.

Insert an Instance into a Relation with the dmctl Utility

To insert an instance into a relation with dmctl, use the **insert** command. You may find it helpful to retrieve the class properties to verify the relations in which the instance can participate.

To retrieve the properties of a class:

```
% dmctl> getProperties <class_name>
```

To insert an instance into a relation, you must specify an instance for each end of the relation. For the instance to be inserted, specify the class name, instance, and relation. For the other end of the relation, specify the class name and instance.

```
% dmctl> insert <class>::<instance>::<relation> \
  <class>::<instance>
```

Notifying Events

Notifying events provides a method for simulating the occurrence of events in a managed system. The domain manager treats a notified event as though it were a real event.

You simulate events with the notify command. Events specified by problem or aggregate declarations cannot be notified. Instead, you notify the symptoms, specified by event declarations, that the problem would cause if it were to occur. If the problem affects other objects, as specified by an aggregate declaration, then the domain manager notifies the appropriate aggregate events as well.

Notifying Events with the Generic Console

To notify an event using the console, complete the following steps.

1. Select the instance you wish to notify the event for and open its property sheet, as follows:

   In the Topology Browser view, the property sheet of a selected instance is displayed in the left panel of the view.

   In the Domain Browser view, double-click on an instance to open its property sheet.

2. Select the Events tab in the property sheet.
3. Right-click on the event you wish to notify and select **Notify** from the pop-up menu.

Notifying Events with the dmctl Utility

To notify an event with dmctl, specify the class name, instance, and event.

```
% dcmtl> notify <class>::<instance>::<event>
```

5

Basic Lexical Elements of MODEL

The remaining chapters of this document provide a reference for the syntax and usage of MODEL declarations. This chapter describes the basic lexical elements of MODEL including keywords, identifiers, and data types. Because MODEL files are compiled to C++, the lexical elements follow many of the conventions for keywords, identifiers, and data types established by C++.

Keywords

Table 5 lists the MODEL and C++ keywords that are reserved and should not be used as identifiers. Identifiers with a double underscore (__) or those that begin with an underscore and an uppercase letter are also reserved for use by C++ implementations and standard libraries. You should avoid C++ keywords because the code generated by the MODEL compiler is compiled by a C++ compiler.

Note: Not all of the keywords in the following list are currently used by MODEL. Unused keywords are reserved for future use.

_time	abstract	aggregate
and	and_eq	apriori
ascending	asm	attribute
auto	avg	bitand
bitor	bool	boolean
break	case	catch
causes	char	check
class	compl	computed
const	const_cast	constraint
continue	control	CORBA
default	definition	delete
delta	descending	do
double	dynamic_cast	else
enum	error_code	event
events_only	explains	explicit
export	extern	external
FALSE	float	for
foreach	fragment	friend
function	get_row	goto
handler	hard	idempotent
if	implementation	imported
in	inline	inout
instrument	instrumented	instrumented_op
int	interface	internal
key	long	loss
max	min	mutable
namespace	new	not
not_eq	obj	old
operator	or	or_eq
out	polling_frequency	private

problem	prod	propagate
protected	proxy	public
rate	readonly	refine
register	reinterpret_cast	relationship
relationshipset	REMOTE_REPOSITORY	required
return	row	self
set	short	signed
sizeof	SNMP	soft
spurious	static	static_cast
stored	string	struct
sum	switch	symptom
table	TCL	template
this	throw	timestamp
timestamped	TRUE	try
type	typedef	typeid
typename	union	unique
unsigned	unstringable	using
value	virtual	void
volatile	wchar_t	while
with	xor	xor_eq

Table 5: MODEL and C++ Keywords

Identifiers

A MODEL identifier must start with an upper or lowercase letter but may contain any number of letters, numbers, and underscores after the first letter. You cannot use any MODEL or C++ keyword as an identifier.

The scope of an identifier declared in an interface is limited to the scope of that interface. Subclasses, however, extend the scope of an interface. All identifiers of a class are visible to its subclasses.

Data Types

Table 6 lists the data types supported by MODEL.

DATA TYPE	DEFINITION
boolean	Two possible values: TRUE or FALSE.
char	Any single ASCII character.
double	Any double-precision floating point number (similar to C++).
enum	Enumerated type which assigns integer values to each of its tags. Integer values may be positive, negative or zero.
float	Any single-precision floating point number (similar to C++).
int	A 32-bit integer ranging from -2^31 to +(2^31 -1).
long	A 64-bit integer ranging from -2^63 to +(2^63 -1).
short	A 16-bit integer ranging from -32,768 to 32,767.
string	An ordered list of characters.
struct	A collection of fields, each field specifies its own type.
unsigned int	Unsigned 32-bit integer ranging from 0 to (2^32 - 1).
unsigned long	Unsigned 64-bit integer ranging from 0 to (2^64 - 1).
unsigned short	Unsigned 16-bit integer ranging from 0 to 65,535.

Table 6: Supported Data Types

6

Declaring an Interface

An interface defines a managed object class or a type of managed element in the Repository. An interface can be defined by an interface declaration or a forward declaration.

```
<interface>          ::=    <interface_dcl>
                     |      <forward_dcl>

<forward_dcl>        ::=    interface <identifier>
```

A forward declaration declares the name of an interface and signals the MODEL compiler that its definition appears later in the MODEL file. Unlike C++, a MODEL class cannot be referenced before it is declared. You can use the forward declaration to declare a class whose full declaration appears later. A forward declaration includes the keyword **interface** and the name of the interface.

Note: The relationship declaration is an exception to the rule that dictates that a MODEL class cannot be referenced before it is declared. A relationship declaration may declare a reciprocal class for the relationship, and the reciprocal class may not yet be declared.

Interface Declaration

An interface declaration is composed of an interface header statement followed by a series of interface definition statements enclosed by curly braces. Interface definition statements describe the properties of the class.

- Attributes, relationships, and operations on instances of the class.

- Problems and events associated with instances of the class and the causal relations among the instances.
- Access types for the attributes of the interface.

The syntax of an interface declaration:

```
<interface_dcl>::=<interface_header>
                  <interface_description>
                  "{"
                  <declarations>
                  [ <opt_implementation> ]
                  "}"
                  [ ; ]
```

The declarations that define the properties of an interface are described in the following chapters.

Interface Header Declaration

An interface header declaration specifies the name of an interface and its inheritance specification.

```
<interface_header>::=   [ abstract ][ unique ]
                        interface
                        <interface_name>
                        <inheritance_spec>
<inheritance_spec>::=   ":" <parent_class>
```

The optional keyword **abstract** defines an interface for which instances cannot be instantiated. Classes that are directly instantiable must define implementations for all of their attributes. Classes that are declared as abstract and are not directly instantiable may specify attributes for which implementations are not defined. To instantiate instances of a subclass of an abstract class, the subclass must define implementations for all of its attributes. By default, an interface is not abstract.

The optional keyword **unique** defines an interface where only one instance of the interface, or any derived interface, may be instantiated at any given time. If you create an instance, and destroy it, you can then create another instance.

Inheritance

MODEL is a single-inheritance language: an interface can only inherit from one other interface. This makes the class hierarchy of your model tree-shaped, with MR_ManagedObject as the root.

Interfaces are related in a subclass/superclass tree. The inheritance specification, *<inheritance_spec>*, declares the class that the interface inherits from. The inheritance specification parameter is specified with a colon (**:**) followed by the name of the superclass.

All properties defined for a class are inherited by its subclasses. You can refine properties specified by attribute, relationship, and event declarations in the subclass. You can also add properties to a subclass to introduce a unique behavior or state.

7

Attribute Declarations

Attributes describe the state of a managed object. Their values can be queried and manipulated by InCharge clients and displayed in an InCharge console.

A data attribute can have a value of scalar, structure, or table type. A scalar value is one of the pre-defined set of types. A table is a collection of rows, each of which can be a scalar or a structure.

Access Types for Attributes

The value of an attribute can be accessed using one of four methods: *stored* in the Repository, *computed* on demand, *instrumented* (i.e., accessed through a standard protocol that is defined by a separate instrumentation declaration), or *propagated* from other objects through a relationship.

- Stored attributes can have an initial value that is a constant. If the stored attribute is not declared as readonly, an InCharge client such as an adapter can update or change the attribute's value.
- The value of a computed attribute is calculated on demand, when the value of the attribute is required.
- Instrumented attributes obtain their value through a standard protocol, such as SNMP. Instrumented attributes require a separate instrumentation declaration that ties the attribute to the protocol.

- The value of a propagated attribute is obtained from one or more instances of a related class through a relationship or relationshipset. A propagated attribute whose value is retrieved through a relationship returns a single value from the related object. A propagated attribute whose value is retrieved through a relationshipset is computed by applying an aggregate operator to a set of values, one value from each related object.

When the Value of an Attribute is Unavailable

A domain manager monitors those attributes necessary to determine when events that have been subscribed to occur. When the status of a monitored attribute changes, the domain manager evaluates the event expressions that depend on this attribute. If the domain manager is unable to retrieve the value of a monitored attribute, for whatever reason, the domain manager marks the attribute as unavailable.

When an attribute is marked as unavailable, the domain manager no longer attempts to retrieve its value. An event expression that uses the value of this attribute may also become unavailable, depending on how the attribute value is used in the event expression. See "When the Value of an Expression is Unavailable" on page 91 for more information regarding unavailable events.

There are several reasons why a domain manager might not be able to retrieve the value of an attribute.

- An instance that contains the attribute does not exist (was not created or was deleted). Typically, this affects propagated attributes whose value is retrieved through a relationship or a relationshipset.
- If the attribute is instrumented, the domain manager may not be able to retrieve its value because of external problems. For example, network congestion may impede the domain manager's attempts to retrieve the value of an attribute that is instrumented through a protocol.

When the status of an unavailable attribute changes, the domain manager automatically retrieves its value and evaluates any dependent event expressions.

MODEL also provides a default value for an attribute whose value is obtained by propagation from an empty relationshipset. In this case, the default value of the attribute is the identity value for the aggregate operator.

Attributes Propagated Over a Relationshipset

Attributes that propagate over a relationshipset must be of a numeric or Boolean type. The results returned by propagation over an empty relationshipset depend on the aggregate operator. Table 7 lists the identity value of an empty relationshipset and the corresponding aggregate operator.

AGGREGATE OPERATOR	IDENTITY VALUE
and	True
or	False
max	Lowest possible value
min	Highest possible value
sum	0
prod	1
avg	Error

Table 7: Identity Result for Aggregate Operators

The example interface below declares one propagated attribute for each aggregate operator. The value of each attribute propagates from a related class through a relationshipset. We use this example to explain the identity values listed in Table 7.

```
interface Ex2 : MR_ManagedObject
{
    propagate attribute boolean and A3 <= C3, R3;
    propagate attribute boolean or A4 <= C4, R4;
    propagate attribute int max A5 <= C5, R5;
    propagate attribute int min A5 <= C6, R6;
    propagate attribute int sum A7 <= C7, R7;
    propagate attribute int prod A8 <= C8, R8;
    propagate attribute int avg A9 <= C9, R9;
}
```

The *and* aggregate operator returns true for an empty set. You expect that inserting a true value to any set leaves the *and* aggregate unchanged. This can only occur if the identity value of the *and* aggregate operator is true. Similarly, the *or* aggregate operator returns a value of false when R4 is empty. You expect that inserting a value of false to any set leaves the *or* aggregate unchanged. This can only occur if the identity value of *or* is false.

The *max* aggregate operator returns the lowest possible value for an empty set. When you insert a value that is higher than the current maximum to R5, the value of A5 increases to reflect the new maximum value. This can only occur if the identity value of the *max* operator is the lowest possible value. Similarly, when you insert a value that is lower than the current minimum to R6, you expect the value of A5 to change accordingly. This can only occur if the identity value of the *min* operator is the highest possible value.

The *sum* aggregate operator returns a zero for an empty set. If you insert a value of one to R7, you expect the value of A7 to increase by one. This can only occur if the identity value of the *sum* operator is zero.

The *prod* aggregate operator returns a value of one for an empty set. When you insert a value of two to R8, you expect its value to double. This can only occur if the identity value of the *prod* operator is one.

The *avg* aggregate operator is different from the other aggregate operators in that it has no identity value. If A9 is initially empty and you insert a value of one, the average should be one, the only value in the set. However, the value of A9 in this case is A9 + 1/2. For A9 to equal one, its initial value would have to be one. If you substitute two for the one, the same result occurs. Because of this, there is no identity value that works with every possible value inserted into A9.

Minimizing the Effects of Unavailable Attributes

You can, for certain situations, write MODEL code that prevents an attribute from becoming unavailable. For example, when the value for an attribute propagates from a related instance, you can write an event expression that determines whether that related instance exists before the domain manager retrieves the value of the attribute.

The following example illustrates this technique. The value of attribute A1 propagates from a related instance through relationship R1. Event E1 is only evaluated when the number of instances participating in R1 is greater than zero. The vertical bars (| |) surrounding R1 produce a count of the related instances. If the count of R1 is not greater than zero, the Boolean expression evaluates to False. If the count of R1 is greater than zero, the first term of the Boolean expression evaluates to True and the domain manager can safely retrieve the value of A1 to determine whether the second term, and the entire event expression, evaluate to true.

```
interface Example : MR_ManagedObject
{
    relationship R1, Example2;
    propagate attribute boolean A1 = Example2, R1;
    event E1 = |R1| > 0 && A1;
}
```

Stored Attributes

The value of a stored attribute is specified in the model or set at runtime when the instance is created. If the attribute is declared as readonly, its value cannot be updated or changed by an InCharge client. A typical use of a stored attribute is to provide a name for an object as a string.

```
<attribute_dcl>  ::=    [ static ][ readonly ]
                        stored
                        attribute
                        <attribute_type>
                        <attribute_name>
                        [ timestamped ]
                        [ <attribute_description> ]
                        [ "=" <initial_value> ]
                        ";"
                 |
                        refine
                        [ access_type ]
                        <attribute_name>
                        [ attribute_description> ]
                        [ <initial_value> ]
                        ";"
<access_type>    ::=    stored
                 |      computed
                 |      instrumented
                 |      propagated
```

The optional keyword **static** declares the scope of the attribute. If the stored attribute is declared static, a single copy of the attribute value is shared by all instances of the interface.

The optional keyword **readonly** prevents write access to an attribute's value. A readonly attribute must be given an initial value.

The keyword **attribute** identifies this as an attribute declaration. The keyword **stored** declares the attributes access type.

The *<attribute_type>* parameter specifies an attribute's data type. A stored attribute can be any one of the supported types listed in "Data Types" on page 44.

When an attribute is declared with the optional keyword **timestamped**, the domain manager creates a readonly unsigned integer whose value is the time that the attribute was last updated. To retrieve this value, use the `timestamp()` function described in "Timestamp" on page 89. The time is calculated by counting the number of seconds that have elapsed since Midnight of 1 January 1970 (GMT). The last update is the most recent call to the put method for a stored attribute and the most recent poll for an instrumented attribute. The keyword **timestamped** cannot be used with the keyword **readonly** except for attributes declared as instrumented.

The optional *<initial_value>* parameter sets an initial value for the attribute when the object is instantiated.

The keyword **refine** declares this attribute as a refinement of an attribute declared in a superclass of this class. A refined attribute can change the original attribute's access type, description, and initial value, but not its type. When you refine the access type of a stored attribute to propagate, you need to provide the class name and relation name. In addition, you must place a #pragma Uses Propagation before the attribute declaration in the parent class. If the pragma is not provided, the MODEL compiler will issue a warning message. For more information regarding this pragma, see "#pragma Uses Propagation" on page 98.

Computed Attributes

A domain manager determines the value of a computed attribute by evaluating an expression. You define an expression using the operators, functions, and types supported by MODEL. For information regarding operators, see "Lexical Elements for Expressions" on page 79 and for information regarding types see "Data Types" on page 44.

Computed attributes are used for calculating rates and displaying the information to an InCharge client or for presenting information in readable format.

Computed attributes are treated as readonly, even if the keyword **readonly** is omitted.

```
<attribute_dcl>  ::=    [ readonly ]
                        computed
```

```
                        attribute
                        <attribute_type>
                        <attribute_name>
                        [ timestamped ]
                        [ <attribute_description> ]
                        "=" <expression>
                        ";"
                    |
                        refine
                        [ <access_type> ]
                        <attribute_name>
                        [ timestamped ]
                        [ <attribute_description> ]
                        "=" <expression>
                        ";"
<access_type>   ::=     stored
                |       computed
                |       instrumented
                |       propagated
```

The keyword **attribute** indicates that this is an attribute declaration. The keyword **computed** declares the access type of the attribute.

The MODEL syntax for a computed attribute and a stored attribute are similar. One important difference is the *<expression>* parameter. This parameter provides a way for you to write an expression that the domain manager uses to compute the attribute's value. For information regarding the syntax of expressions, see "Syntax for Expressions" on page 90.

When you refine the access type of a computed attribute to propagate, you need to provide the class name and relation name that the value propagates from. In addition, you must place a #pragma Uses Propagation before the attribute declaration in the parent class. If the pragma is not provided, the MODEL compiler will issue a warning message. For more information regarding this pragma, see "#pragma Uses Propagation" on page 98.

Instrumented Attributes

The value of an instrumented attribute is retrieved by a domain manager through a standard protocol. The runtime support for instrumentation is provided by a component called an accessor. A domain manager includes an SNMP accessor for retrieving the values of SNMP MIB variables.

When an attribute is instrumented through the SNMP accessor, the accessor retrieves the value of the attribute from a remote SNMP agent. The SNMP accessor does this by polling the values of the SNMP MIB variables that correspond to the instrumented attribute. Only attributes necessary to evaluate events that have been subscribed to are actually polled.

The type of an instrumented attribute must match the type of the MIB variable retrieved by the SNMP accessor. When the SNMP accessor retrieves a MIB variable, it checks to see that its type matches the type declared for the corresponding MODEL attribute. The SNMP accessor rejects a MODEL attribute of type enum because the type it retrieves from the SNMP agent is an integer. When the SNMP accessor detects a type mismatch, the domain manager logs an error and the user receives the error SNMP_EAGENTBUG.

The variable types that appear in MIBs are defined by Abstract Syntax Notation 1 (ASN.1). SNMP defines a base set of types on top of the ASN.1 types. Table 8 lists the common MIB types, the corresponding SNMP type, and the corresponding MODEL attribute type.

MIB Type	SNMP Type	MODEL Attribute Type
DisplayString	OCTET STRING	string
IpAddress	OCTET STRING	string - Requires the #pragma DotNotation, see "#pragma DotNotation" on page 99.
Counter	INTEGER	unsigned int - Requires the #pragma WrapCounter, see "Additional Pragmas" on page 98.
Gauge	INTEGER	unsigned int
TimeTicks	INTEGER	unsigned int
Integer32	INTEGER	unsigned int
Counter32	INTEGER	unsigned int - Requires the #pragma WrapCounter, see "Additional Pragmas" on page 98.
Gauge32	INTEGER	unsigned int
Unsigned32	INTEGER	unsigned int
OID	OBJECT IDENTIFIER	string - Requires the #pragma Object ID, see "#pragma ObjectID" on page 99.
Counter64	COUNTER64	unsigned long

Table 8: MIB, SNMP, and MODEL Types

Developing a MODEL with instrumented attributes is a two-step process that requires you to complete steps during both MODEL development and at runtime. This section describes the MODEL requirements for attribute declarations and the syntax of instrumented attributes. A summary of the runtime requirements appears in "Summary of Runtime Requirements for SNMP Instrumentation" on page 94.

To properly instrument an attribute, MODEL code must meet two conditions:

- It must contain attributes declared as instrumented. Furthermore, all of the instrumented attributes declared for a class must get their value from the same SNMP agent. When you retrieve values from an SNMP table, such as ifTable, the rule of thumb is to map a MODEL class to a row of an SNMP table. In the class, declare a separate instrumented attribute for each column of the table and create an instance to represent each row.
- It must contain an instrumentation declaration for each class with instrumented attributes. The instrumentation declaration maps the instrumented attributes of a class to OIDs. For the syntax of an instrumentation declaration, see "Instrument Declarations" on page 93.

The syntax of an instrumented attribute declaration is shown below.

```
<attribute_dcl>  ::=    [ readonly ]
                        instrumented
                        attribute
                        <attribute_type>
                        <attribute_name>
                        [ <attribute_description> ]
                        ";"
                 |
                        refine
                        [ <access_type> ]
                        [ <attribute_name> ]
                        [ <optional_description> ]
                        ";"
```

The keyword **attribute** identifies this as an attribute declaration. The keyword **instrumented** declares the access type of this attribute.

An instrumented attribute does not require the use of the keyword **timestamped** because the SNMP accessor automatically maintains this information. You can retrieve the most recent time at which the value of an instrumented attribute changed as you would for a stored or computed attribute declared as timestamped. See "Timestamp" on page 89 for more information.

The keyword **refine** indicates that this declaration is a refinement of an attribute in a related class. You can refine an instrumented attribute's access type, name, and description.

Example of Instrumented Attribute

The following example shows an instrumented attribute, sysUpTime, whose value is retrieved from an SNMP-enabled device that supports MIB-II. The connection between this attribute and the sysUpTime MIB variable is demonstrated in "Instrument Declarations" on page 93.

```
interface AgentStatus
{
    instrument SNMP{
       sysUpTime = "1.3.6.1.2.1.1.3"
    };
    #pragma WrapCounter
    readonly instrumented attribute unsigned sysUpTime
       "The time (in hundredths of a second) since "
       "the network management portion of the system "
       "was last re-initialized.";
}
```

The WrapCounter pragma is used with instrumented attributes of an unsigned numeric type that get their value from a MIB counter variable. This pragma prevents rates and deltas that are computed over the attribute from returning a negative value. If the current value of the attribute is smaller than the previously retrieved value, the domain manager assumes that the counter has wrapped.

For information regarding MODEL pragmas, see "MODEL Pragmas" on page 97. For information about the rate and delta operators, see "Built-in Functions" on page 85.

Propagated Attributes

An attribute declared as propagated obtains its value from an attribute in one or more instances of a related class. The value is retrieved from one instance for a relationship and one or more instances for a relationshipset. The type of a propagated attribute must be compatible with type of the attribute in the related class. For example, you could declare a propagated attribute with the sum, prod, or avg aggregate operator as a float or a double, even if the values propagate from an attribute declared as an int.

MODEL supports two types of relationships: to-one, which are declared with the **relationship** keyword; and to-many, which are declared with the **relationshipset** keyword. The cardinality of a relationship determines the attribute type that can propagate.

- Any MODEL-supported type can propagate through a to-one relationship.
- Only Boolean or numeric types can propagate through a to-many relationshipset. You can, however, use a computed attribute or a set expression to avoid this limitation.

```
<attribute_dcl>  ::=    propagate
                        attribute
                        <attribute_type>
                        [ <aggregate_operator> ]
                        <attribute_name>
                        [ <attribute_description> ]
                        "<="
                        <class_name>
                        ","
                        <relationship_name>
                        [ "," <attribute_name> ]
                        ";"
                  |     refine
                        propagate
                        attribute
                        [ <aggregate_operator> ]
                        <attribute_name>
                        [ <attribute_description> ]
                        "<="
                        <class_name>
                        ","
                        <relationship_name>
                        [ "," <attribute_name> ]
                        ";"
<aggregate_operator>    ::=    max
                        |      min
                        |      sum
                        |      prod
                        |      avg
                        |      and
                        |      or
```

The keyword **attribute** identifies this an attribute declaration.

The keyword **propagate** declares that the value of this attribute is computed or copied from other objects through a relation. If the relation that the attribute propagates through is a relationshipset, then the *<aggregate_operator>* parameter is required to specify how the attribute values from related objects are to be handled. An attribute that propagates through a relationshipset must be a numeric or Boolean type.

The *<class_name>* and *<relationship_name>* parameters specify the class and the relationship that the value of the attribute propagates from. The optional *<attribute_name>* parameter identifies the original attribute if it has a different name. If you do not provide a name, the MODEL compiler assumes that the attribute has the same name in the related class.

The keyword **refine** identifies this declaration as a refinement of a propagate attribute declared in a superclass. The keyword **attribute** is required for a refinement of a propagate attribute declaration. A refinement can modify the class from which values propagate, the relationship through which the values propagate, the attribute that the value propagates from, and the aggregate operator.

Table Attributes

Tables provide a useful means for storing configuration and other data in a persistent and accessible manner. Because the domain manager's Repository is persistent across restarts, the information is always available to an InCharge client.

Tables should not be used to access variables from SNMP tables if you want to use this data in an event expression. Data stored in a MODEL table cannot be accessed through an event expression. For information regarding the technique for accessing and storing data from an SNMP table, see "Instrumented Attributes" on page 55.

A table is a collection of rows, each of which can be an integer, string or struct. Table parameters, and their definitions, are similar to those of the attribute declaration.

```
<table_dcl>       ::=    [ readonly ]
                         table
                         <table_name>
                         <row_type>
                         [ <table_key> ]
                         [ <table_description> ]
                         ";"
```

```
                        |       refine
                                <table_name>
                                [ <table_description> ]
                                ";"

                        |       propagate
                                table
                                <table_name>
                                <row_type>
                                [ <table_key> ]
                                [ <table_description> ]
                                "<="
                                <class_name>
                                ","
                                <relationship_name>
                                [ "," <table_name> ]
                                ";"
                        |       refine
                                propagate
                                table
                                <table_name>
                                [ <table_description> ]
                                "<="
                                <class_name>
                                ","
                                <relationship_name>
                                [ "," <table_name> ]
                                ";"
<table_key>         ::=         <key_direction>
                                [ unique ]
                                key
                                <key_name>
<key_direction>     ::=         ascending
                        |       descending
```

The keyword **table** declares this attribute as a table.

The *<row_type>* parameter specifies the type of table row. Valid types include integer, string, and struct.

The optional *<table_key>* parameter, which includes the keyword **key** followed by the key's name, specifies that the table can be accessed by a key. If a key is declared, then the table row must be a struct and the key must be a field of the struct. You can also declare a direction for the key, ascending or descending, which affects how the table is read. When the optional keyword **unique** is specified, each row must have a unique key.

The keyword **refine** declares this table as a refinement of a table declared in a superclass. A refinement can change the table's access type, name, and description. For a propagated table, a refinement can change the class from which the table propagates, the relationship through which the values propagate, and the table from which the values propagate.

8

Relationship Declarations

A relationship declaration defines a connection between classes. A relation can connect instances of different classes or instances of the same class. Relationships are important because attributes and events propagate along the traversal path specified by the relationship.

When you declare a relationship from classA to classB, you are implicitly declaring a reciprocal relationship from classB to classA. We refer to this relationship pair as a *relation*.

Both relationships and relations have a cardinality. The cardinality of a relation is determined by the cardinality of its constituent relationships. A relationship from classA to classB may have cardinality of to-one or to-many. A cardinality of to-one means there is a single instance of classB related to a given instance of class A. A cardinality of to-many means that one or more instances of classB are related to a given instance of classA. The cardinality of the relation is one-to-one if both relationships are to-one, one-to-many or many-to-one if one relationship is to-one and the other is to-many, or many-to-many if both relationships are to-many.

MODEL provides two keywords to indicate the cardinality of a relationship: **relationship** and **relationshipset**. As you might expect, relationshipset declares a to-many relationship.

If you insert an instance into one end of a two-way relationship, the Repository guarantees that the inverse traversal path is correctly set. When an object is added or removed from the Repository, the Repository maintains the integrity of the relationship. If an object that participates in a relationship is deleted, the domain manager automatically removes it from the inverse traversal path.

Relationship declarations, like attribute declarations, are a basic building block for describing real-world objects. Attribute and relationship declarations share a similar syntax.

```
<relation_dcl>    ::=    [ readonly ]
                         [ <access_type> ]
                         <cardinality>
                         <relationship_name>
                         ","
                         <class_name>
                         [ "," <inverse_name> ]
                         [ <optional_key_spec> ]
                         [ <relationship_description> ]
                         [ "=" <expression> ]
                         ";"
                   |     refine
                         <relationship_name>
                         [ "," <class_name> ]
                         <relationship_description>
                         ";"
<access_type>     ::=    stored
                         computed
<cardinality>             ::=    relationship
                         |       relationshipset
```

The optional keyword **readonly** is provided for computed relationships. Computed relationships are treated as readonly, even if the keyword **readonly** is omitted.

The optional *<access_type>* parameter specifies where and how information about this relationship is obtained. By default, relationships are **stored**. Relationships declared as **computed** are limited to operators that result in a single object reference. The expression for a computed relationshipset can return a set of object references. The result of the expression is assigned to the other end of the relationship or ends of the relationshipset.

The *<cardinality>* parameter specifies whether the traversal path of this relationship declaration refers to one class or many classes. The keyword relationship specifies a connection to, at most, one class; the keyword relationshipset specifies a connection to any number of classes.

The *<class_name>* parameter specifies the class with which the declaration establishes a relationship.

The optional <*inverse_name*> parameter specifies the name of the inverse traversal path in the other class. If specified, you must also declare the corresponding relationship or relationshipset in the other class. If the <*inverse_name*> parameter is omitted, a one-way relationship or relationshipset is created.

The keyword **refine** declares this relationship as a refinement of an inherited relationship. To refine a relationship, both ends of the relationship must belong to the same inheritance hierarchy. A refined relationship uses the name of the inverse traversal path that appears in the original relationship declaration and its cardinality must be the same as that of the original relationship.

9

Declaring Events in MODEL

This chapter describes the MODEL declarations that are used to model the behavior of objects in a managed system. These constructs enable you to specify the static knowledge required for event management.

When you begin adding events to your model, remember three important points:

1. Problems propagate across related objects.
2. A single problem can cause numerous observable events (symptoms).
3. Problems may not be observable in the object where they occur.

MODEL Declarations for Defining Events

MODEL provides a number of declarations for defining events. Each of these has a particular purpose, as described below.

- An event declaration defines a local event, meaning it is observable in the object where it occurs. An event declaration is defined by an expression over attributes or other events of the class. An event can also be declared as imported, meaning the domain manager subscribes to it from an external source. See "Event Declaration" on page 68.
- A problem declaration is defined by the symptoms that it can cause. You can specify a probability for each symptom because the causal relationship between the problem and its symptoms may be probabilistic. See "Problem Declaration" on page 70.

- A symptom declaration groups a set of events into a single event by conjunction. These are not related to the symptom parameters used to define a problem declaration. See "Symptom Declaration" on page 72.
- A propagate symptom declaration defines an event that propagates from a related object through a relationship. An event declared as a propagated symptom is intended to be used as a symptom in a problem declaration. See "Propagate Symptom Declaration" on page 73.
- An aggregate declaration defines a disjunction of one or more events declared for the same class. An aggregate groups a collection of events into a single abstract event. See "Aggregate Declaration" on page 74.
- A propagate aggregate declaration defines a set of events that are not observable in the object where they occur but whose effects propagate to related objects through a given relationship. See "Propagate Aggregate Declaration" on page 75.

Any of these events, except propagated symptoms and aggregations, may be exported to InCharge clients that request to be notified when the event occurs.

Event Declaration

An event declaration declares an event that is defined using an expression. An event declared using an expression is defined over attributes and other events of its class.

Events declared by an event declaration are intended to be used as symptoms in problem declarations. The domain manager creates a data structure called a codebook that lists the problems to which InCharge clients have subscribed. The codebook also includes the events, specified by event declarations, that each problem can cause. These events are used by the domain manager to determine when a problem occurs. The domain manager reduces the codebook so that each problem is identifiable by a unique set of symptoms.

```
<event_dcl>       ::=    event
                         <event_name>
                         [ <event_description> ]
                         [ <opt_loss> ]
                         [ <opt_spurious> ]
                         [ <event_implementation> ]
                         ";"

    |                    refine
                         event
```

```
                          [ <event_name> ]
                          [ <event_description> ]
                          [ <opt_loss> ]
                          [ <opt_spurious> ]
                          [ <event_implementation> ]
                          ";"
<opt_loss>          ::=   loss
                          "(" <expression> ")"
<opt_spurious>      ::=   spurious
                          "(" <expression> ")"
<event_implementation> ::=    "="
                              [ <event_guard> ]
                              <expression>
<event_guard>       ::=   if
                          <expression>
                          check
```

The *<opt_loss>* parameter, indicated by the keyword **loss**, specifies the probability that this observable event is lost or not observed (its expression does not evaluate to true) even though it is occurring. Setting a value closer to one increases the likelihood that the symptom will be lost. The loss expression must evaluate to a floating point number greater than or equal to zero but less than or equal to one.

The *<opt_spurious>* parameter, indicated by the keyword **spurious**, is the converse of *<opt_loss>*. It specifies the probability that this observable event is not occurring, even though the event expression evaluates to true. The expression must evaluate to a floating point number greater than or equal to zero but less than or equal to one.

The *<event_implementation>* parameter specifies the expression that defines the event and, optionally, whether the event is guarded.

The optional keyword **refine** declares this event as a refinement of an event declared in a superclass. You can change all the parameters of an event through a refinement.

Guarded Events

The *<event_guard>* parameter provides a method for preventing an event expression from being evaluated unless a specified condition is met. If the guard expression specified by the *<event_guard>* parameter, which must return a Boolean value, is true, then the event expression is evaluated. Guarded events are useful when there is a high cost associated with evaluating an event expression. Factors that can contribute to a high cost include expensive computations or operations such as SNMP polling.

Event Expressions

An event expression can be declared using the names of other events as well as the operators and syntax described in "Syntax for Expressions" on page 90. There is one notable constraint on event expressions: they must return a Boolean value.

When the event is evaluated, the expression is computed. If the return value changes the status of the event (from true to false or from false to true), the domain manager sends the appropriate notify or clear notification to each client. The domain manager only sends a notification when the status of an event changes.

When the attributes used to create an event expression depend on values within the Repository, the domain manager can compute those expressions as necessary and send a notification precisely when the status of the event changes. However, if the attributes are instrumented and obtain their values from an external source, the process for obtaining those attribute values can affect when the domain manager determines that an event's status has changed. For example, if the value of an attribute is retrieved by polling an SNMP agent, the responsiveness of the SNMP agent and the duration of the polling interval can affect when the event is recognized.

Problem Declaration

A problem declaration declares an event that may occur in instances of a class. A problem causes a set of symptoms and the problem declaration specifies these symptoms and a probability that each symptom will be observed when the problem occurs. There are three types of symptoms.

- Locally observable symptoms can be observed in the same object where the problem occurs. Local symptoms must be declared by separate event declarations in the same class as the problem declaration.
- A problem can cause a second problem within the same object. The second problem must be declared by a separate problem declaration in the same class as the original problem.
- A propagated event or problem can only be observed in related objects. A propagated event must be declared by a propagate symptom declaration in the same class as the problem. In addition, the events or problems must be declared in the class to which they propagate.

Symptoms are events specified by event, symptom, propagate symptom, and problem declarations. Note that a symptom can be a problem that causes other symptoms.

When an InCharge client subscribes to a problem, the domain manager adds the problem to its codebook. The domain manager also adds the set of symptoms necessary to uniquely identify this problem from other problems listed in the codebook.

```
<problem_dcl>     ::=    problem
                         <problem_name>
                         [ <problem_description> ]
                         [ <symptoms_list> ]
                         ";"
                  |      refine
                         problem
                         [ <problem_name> ]
                         [ <problem_description> ]
                         <symptoms_list>
                         ";"
<symptoms_list>   ::=    "=>"
                         symptom
                         <symptoms>
<symptoms>        ::=    <symptom>
                         ["," <symptoms> ]
<symptom>         ::=    <symptom_name>
                         [ <probability> ]
                         [ <condition> ]
<condition>       ::=    if
                         <expression>
```

The keyword **problem** indicates that is a problem declaration.

The optional *<symptoms_list>* parameter includes the keyword **symptom** and lists the symptoms that this problem causes. Each symptom can include a *<probability>* parameter that specifies the probability that the problem causes this symptom to occur. The *<probability>* parameter must evaluate to a floating point number greater than or equal to zero but less than or equal to one. The default probability is one, meaning that the problem always causes this symptom.

The optional *<condition>* parameter provides a method for conditionally removing a symptom from a problem. The *<expression>* parameter must return a Boolean value. When the expression evaluates to false, the event is removed from the list of symptoms that this problem causes. When the expression evaluates to true, the event remains in the list of symptoms caused by the problem. Note that the value of the Boolean expression can change during runtime. However, the codebook must be recomputed before the correlator recognizes the change.

The optional keyword **refine** declares this problem as a refinement of a problem declared in a superclass of this class. You can change all the parameters of a problem declaration through a refinement. The refine declaration, however, must include at least one symptom.

Symptom Declaration

A symptom declaration groups a set of events under a single name. A symptom declaration can group events declared by event, problem, symptom, or propagated symptom declarations.

The purpose of a symptom declaration is to combine events for defining a problem declaration. An event defined by a symptom declaration is not added to the codebook and the correlator does not diagnose its occurrence. Furthermore, a symptom declaration cannot be exported and is not available to InCharge clients.

```
<symptom_dcl>     ::=    symptom
                         <symptom_name>
                         [ <symptom_description> ]
                         [ <symptoms_list> ]
                         ";"
                  |      refine
                         symptom
                         [ <symptom_name> ]
                         [ <symptom_description> ]
                         [ <symptoms_list> ]
                         ";"
<symptoms_list>   ::=    "=>"
                         <symptoms>
<symptoms>        ::=    <symptom>
                         ["," <symptoms> ]
<symptom>         ::=    <symptom_name>
                         [ <probablility> ]
                         [ <condition> ]
<condition>       ::=    if
                         <expression>
```

The keyword **symptom** identifies this as a symptom declaration.

The *<symptoms_list>* parameter lists the events that this symptom causes. Each event can be followed by an optional probability that specifies the probability that the symptom causes this event to occur. The *<probability>* parameter must evaluate to a floating point number greater than or equal to zero but less than or equal to one. The default probability is one, meaning that the symptom always causes this event.

The optional *<condition>* parameter provides a method for conditionally removing an event from a symptom declaration. The *<expression>* parameter must return a Boolean value. When the expression evaluates to false, the event is removed from the list of events that are caused by this symptom. When the expression evaluates to true, the event remains in the list of events caused by the symptom. Note that the value of the Boolean expression can change during runtime. However, the codebook must be recomputed before the correlator recognizes the change.

The optional keyword **refine** declares this symptom as a refinement of a symptom declared in a superclass of this class. You can change all of the parameters of a symptom declaration through a refinement.

Propagate Symptom Declaration

A propagate symptom declaration defines an event that is not observable in the object where it occurs. The symptoms of this event propagate to one or more objects of related classes. The events that comprise a propagate symptom must be declared by event, problem, or propagate symptom declarations in the class where they occur.

```
<propagate_symptom_dcl> ::=    propagate
                               symptom
                               <symptom_name>
                               [ <symptom_description> ]
                               "=>"
                               <class_name>
                               ","
                               <relationship_name>
                               [ "," <symptom_name> ]
                               ";"
                           |   refine
                               propagate
                               [ <symptom_name> ]
                               [ <symptom_description> ]
                               "=>"
                               <class_name>
                               ","
```

```
                                    <relationship_name>
                                    [ "," <symptom_name> ]
                                    ";"
```

The keywords **propagate** and **symptom** identify this as a propagate symptom declaration.

The *<class_name>* and *<relationship_name>* parameters identify the class and the relationship that the symptom propagates from. The *<symptom_name>* parameter must be used when the propagated symptom has a different name than the event in the originating class.

The keywords **refine** and **propagate** identify this declaration as a refinement of a propagate symptom declared in a superclass of this class. You can change all the parameters of a propagate symptom through a refinement.

Aggregate Declaration

An aggregate declaration groups one or more events into a single abstract event by disjunction. The aggregate becomes active when any one of the events in the set is active. The events grouped into an aggregate must be declared in event, problem, or propagate aggregate declarations in the same class.

```
<aggregate_dcl>  ::=    aggregate
                        <aggregate_name>
                        [ <aggregate_description> ]
                        "<="
                        <events>
                        ";"
                 |      refine
                        aggregate
                        [ <aggregate_name> ]
                        [ <aggregate_description> ]
                        "<="
                        <events>
                        ";"
<events>         ::=    <event_name>
                        [ "," <events> ]
```

The keyword **aggregate** identifies this as an aggregate declaration.

The *<events>* parameter lists the events that comprise the aggregate.

The optional keyword **refine** identifies this declaration a refinement of an aggregate declared in a superclass of this class. You can change all the parameters of an aggregate through a refinement.

Propagate Aggregate Declaration

A propagate aggregate is an event that is not observable in the object where it occurs. Instead, its symptoms occur in related objects and propagate to the object where the propagate aggregate is declared. When one of its symptoms becomes active, the propagate aggregate becomes active. The events that comprise a propagate aggregate must be defined as event, problem, or propagate symptom declarations in the class where they originate.

```
<propagate_aggregate_dcl>::=   propagate
                               aggregate
                               <prop_aggregate_name>
                               [ <description> ]
                               "<="
                               <class_name>
                               ","
                               <relationshipset_name>
                               [ "," <event_name> ]
                               ";"

                             |  refine
                               propagate
                               <prop_aggregate_name>
                               [ <description> ]
                               "<="
                               <class_name>
                               ","
                               <relationship_name>
                               [ "," <event_name> ]
                               ";"
```

The keywords **propagate** and **aggregate** identify this as a propagate aggregate declaration.

The *<class_name>* parameter identifies the class the event propagates from and *<relationship_name>* identifies the relationship that the event traverses. The *<event_name>* parameter must be used if the name of the propagate aggregate is different from the name of the event in the originating class.

The keywords **refine** and **propagate** identify this declaration as a refinement of a propagate aggregate declared in a superclass of this class. You can change all the parameters of a propagate aggregate declaration through a refinement.

Export Declaration

An export declaration is required to make the events declared for a class visible outside of the Repository. InCharge clients, such as the InCharge Console and adapters, can subscribe to exported events and receive notifications when they occur. You can export events defined by event, problem, and aggregate declarations.

```
<export>          ::=    export
                         <events>
                         ";"
<events>          ::=    <event_name>
                         [ "," <events> ]
```

The keyword **export** identifies this as an export declaration.

The *<events>* parameter lists the events to be exported. Each event must be declared in the same class as the export declaration. You can list multiple events, separating them with a comma, in a single export declaration. You can also specify multiple export declarations for a class.

10

Operation Declarations

Operations provide a method for manipulating objects and their properties. An operation declaration includes the name of the operation, the type of its arguments, and the type of any returned values. Operation declarations provide one method for defining a set expression. Note that there is no operation keyword.

```
<operation_dcl>          ::=   <return_type>
                               <operation_name>
                               "(" [ <arguments> ] ")"
                               [ <operation_description> ]
                               [ <operation_definition> ]
                               ";"

<return_type>            ::=   void
                         |     <value_type>
                         |     set "(" <value_type> ")"
<arguments>              ::=   in
                               <argument_type>
                               <argument_name>
                               "="
                               <expression>
                               [ "," <arguments> ]
<argument_type>          ::=   <value_type>
                         |     set "(" <value_type> ")"

<operation_definition>   ::=   "definition:" <assignments>

<assignments>            ::=   <assignment>","<assignments>
                         |     <assignment>
                         |     <return_expression>

<assignment>             ::=   <identifier>"="<expression>
```

```
<return_expression>     ::=    "return" <expression>
```

The *<return_type>* parameter declares the value type returned by the operation. The return value must be one of the supported types listed in "Data Types" on page 44. If the operation has no return value, use the keyword **void**. An operation can also return a set of values. If this is the case, the keyword **set** should precede the value type. The syntax for set expressions is the same as that for logical or arithmetic expressions. However, a special group of operators must be used. For information regarding the syntax of expressions, see "Syntax for Expressions" on page 90 and for information regarding set operators, see "Operators for Set Expressions" on page 84.

An operation declaration takes a list of arguments. Each argument requires the keyword **in**, which indicates that information passes from the caller to the operation. The *<argument_type>* parameter specifies the argument's type, which must be one of the data types supported by MODEL. You can declare a set expression with the *<argument_type>* parameter, in which case the keyword **set** precedes the value type. You can use the *<expression>* parameter to specify a default value, which may be either a literal or a more complicated expression. The syntax of *<expressions>* is described in "Syntax for Expressions" on page 90. If you declare an argument with a default value, any successive argument must also have a default value.

For *<assignment>*, the <identifier> is an attribute name with *<expression>* described in "Syntax for Expressions" on page 90, where the result of *<expression>* is assigned to *<identifier>*.

If there is no *<return_expression>* argument and the operation has a return type other than *void*, the last *<assignment>* argument is used to provide the return value.

11

Writing Expressions in MODEL

Unlike other chapters that discuss specific MODEL declarations, expressions are not a declaration in and of themselves. Expressions are used to define computed attributes, set expressions, and event expressions. The purpose of this chapter is to describe the syntax for these types of expressions.

Expressions allow you to manipulate information by combining it, comparing it, and performing other operations on it. The expressions in MODEL are composed of *terms* and *operators*. Terms are the basic units that you can combine and operators describe what actions are to be performed on the terms.

Set expressions are a special type of expression that enable you to evaluate, compare, and perform operations on sets of values. The operators for set expressions are described in "Operators for Set Expressions" on page 84. The syntax for set expressions is the same as that for expressions.

Lexical Elements for Expressions

The lexical elements for expressions include literals, operators, and the built-in functions provided by MODEL.

Literals

A literal is a value that represents a constant. Literals in MODEL may be integers, floating-point numbers, strings, or character literals.

Integer Literals

Integer literals may be decimal, hexadecimal or octal. Integer literals may start with an optional unary minus sign or unary plus sign.

A decimal integer is a sequence of the digits zero through nine; the first digit cannot be a zero. An octal integer starts with a zero and may contain the numbers zero through seven. A hexadecimal integer starts with a zero followed by the upper or lowercase letter x. It may contain the digits zero through nine and the upper or lowercase letters A through F.

Floating-Point Literals

Floating-point literals have many parts, some of which may be omitted. A floating-point literal may start with an optional minus sign followed by a series of digits, a decimal point, another series of digits, an upper or lowercase E, an optional plus or minus sign, a series of digits representing the exponent, and an optional upper or lowercase 'F'. The decimal point is required, but you can omit the digits that appear before the decimal point or the digits that appear after the decimal point, but not both. The exponent, which starts with the letter E, may also be omitted. By default, a floating-point literal is treated as a sixty-four bit, floating-point number. When the trailing F is included, it is treated as a thirty-two bit, floating-point number.

String Literals

String literals are a sequence of characters enclosed by double quotes. You can create a multi-line string literal by terminating the first line of a large string literal with a double quote and a new line and enclosing the remaining section of the literal in double quotes on the next line, and so on, until the string is complete. We use this technique to provide descriptions for MODEL classes and their properties. You can also use escape characters in a string literal. An escape character consists of a backslash followed by one of the characters: `n`, `t`, `v`, `b`, `r`, `f`, `a`, `\`, `?`, `"`, or up to three octal digits representing the octal value of a character.

Character Literals

Character literals are enclosed in single quotes. They may be either a single character, or an escape character. The escape characters for character literals are the same as for string literals.

Enumeration Literals

Enumeration literals are named integer literals. You can use negative values within the enumeration declaration. While you can assign fixed values to the enumerators, SMARTS does not recommend declaring multiple enumerators with the same value.

Operators

MODEL supports the C and C++ operators listed in Table 9. These operators behave the same in MODEL and C++ except where noted below. The precedence of operators, also similar to C++, is described in "Precedence of Operators" on page 84.

Operator Type	Symbol	Definition
Unary	+ - ! ~	unary plus unary minus logical negation bitwise complement
Arithmetic	* / % + -	multiply divide modulus plus minus
Shift	>> <<	right shift left shift
Relational	< > <= >=	less than greater than less than or equal greater than or equal
Equality	== !=	equal to not equal to
Logic	&& ||	logical and logical or

OPERATOR TYPE	SYMBOL	DEFINITION

Bitwise	`|`
	`^`
	`&`
		bitwise or
		bitwise exclusive or
		bitwise and

Conditional	`?:`	conditional evaluation; if the test proves invalid, the result is the error from the test. *operand1* `?` *operand2* : *operand3*;
On Error	`else`	on error evaluation

Selection	`case`
	`key`
	`default`
		selective evaluation

Table 9: Operators Supported by MODEL

MODEL does not use the following operators:

- Assignment operators (`=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=`)

MODEL defines special behavior for the operators listed below.

- In MODEL, an expression that divides by zero produces an error result. For example, if an expression involving a zero divide defines a computed attribute, a **get** of that attribute's value will return an error.
- The logical negation operator (!) may only be used on Boolean values.

Evaluation Expression Operators

The following evaluation operators differ slightly in MODEL from how they are generally used in C++ compilers.

Else Operator

The else operator evaluates the left hand expression; if that result is not an error, then the value becomes the result of the else expression. The result types of the left and right hand expressions must agree. If the left hand expression does not return an error, the right hand expression is not evaluated. For example:

```
instrumented attribute int i_state "In MIB as an enum, SNMP
int";
computed attribute state_e safe_state
"Give a default value for failed get or conversion"
= state_e(i_state) else UNKNOWN;
```

Case Operator

The case operator uses the value of a selection expression to select one of a number of keys and the value expression associated with that key. If there is no default key then the MODEL compiler checks the key expressions to see if they cover all possible key values. A compile time warning is produced if not all key values are covered. At run-time, an error is produced if the selection expression value does not match any of the key expression values. An example of the case operator:

```
interface MiddleClass : ConvClass
{
    enum state_e {
              UP = 1
              DOWN = 4
              UNKNOWN = 6
              };
attribute state_e tEnum;

computed attribute int tCaseInt
    =
    case ( tEnum ) {
     key UP : 1;
    key DOWN :4;
    default :    //falls through to key UNKNOWN value
    key UNKOWN : 6;
    };
};
```

Combining Expression Operators

You can combine the previous operators with other expression operators. The following example illustrates how to convert an integer, instrumented attribute coming over the network to an an enumeration:

```
readonly instrumented attribute int ifAdminStatus;
refine computed AdminStatus =
    case (ifAdminStatus) {
       key 1:  UP;
       key 2:  DOWN;
       key 3:  TESTING;
       default:  OTHER;
    } else UNKOWN;
```

Operators for Set Expressions

Set expressions use their own group of terms and operators, as shown in Table 10. The precedence of set operators is at the same level as the corresponding symbol for logical and arithmetic operators. For more information, see Table 11 in "Precedence of Operators" on page 84.

SET OPERATOR	DEFINITION
`<term> in <setA>` <term> `in` <vector_expression>	Returns a Boolean value. The `in` operator checks that the given value from the left term is a member of the right term. The left term must be a scalar, the right term must be a set or vector expression.
`| <setA> |`	Returns an integer that is the count of the number of members in setA.
<class_name>`(`<relationshipset>`)`	Returns a set of all the objects in the relationshipset whose class matches the specified class name.
<setA> `&` <setB>	Returns the set of elements that are members of both setA and setB. Both sets must have the same underlying type.
<setA> `|` <setB>	Returns a set that includes all of the elements of both setA and setB. The members of both sets must be of the same type.
<setA> `-` <setB>	Returns the members of setA that are not members of setB. Both sets must have the same underlying type.
<setA> `->` <relationship>	Determines the number of elements each member of setA is related to through the relationship. All these individual sets are then combined into a multiset. setA must be a set of objects.
`unique(`<vector_expression>`)`	Returns a set of elements from the multiset where no element in the set appears more than once.

Table 10: Operators for Set Expressions

Precedence of Operators

Operators have rules of precedence and associativity to determine how expressions are evaluated. You can put expressions in parentheses to change the order in which operations are performed.

Outside of parentheses, unary operators have the highest precedence followed by arithmetic operators. The convention of arithmetic operators is that the multiplicative operators have precedence over the additive operators. Table 11 lists the exact order of precedence and associativity for operators in MODEL.

OPERATOR PRECEDENCE	ASSOCIATIVITY
conversion, case, key, default	Right to left
+ - ! ~ (unary)	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >= in	Left to right
== ! =	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
else	Left to right
?:	Right to left

Table 11: Precedence and Associativity of Operators

Built-in Functions

The terms described below are built-in functions supplied by MODEL. They typically appear in expressions, as described in "Syntax for Expressions" on page 90. Although the syntax of MODEL permits you to use a function as a term in a complex expression, we have found that it is better to use a function in one attribute and store it in a variable.

Delta

The term `delta(A)`, where A is an attribute, returns the difference between the current value of A and its previous value. This is typically used for an instrumented attribute whose value is obtained by polling.

```
interface EthernetInterfaceStats : Instrumentation
{
    instrument SNMP{
        ifInOctets = "1.3.6.1.2.1.2.2.1.10",
    };

#pragma WrapCounter
    readonly instrumented attribute unsigned ifInOctets
    "The total number of octets received on the "
    "interface.";

    /* Compute Delta
     --------------- */
    readonly computed attribute int deltaifInOctets
    = delta(ifInOctets);
};
```

Conversion

The term `type_name ( )` allows you to convert an expression to a predefined numeric type or to an enumeration type. Write the type name followed by the expression in parentheses. For example:

```
enum state_e {UP, DOWN, UNKNOWN};
attribute state_e tEnum;
attribute int tInt;
attribute int tEnum2Int "Convert to int"
    =int(tEnum);
attribute state_e tInt2Enum "Convert to state_e"
    =state_e(tInt);
```

Object

The term `obj( <string_expression> )` returns the object whose name matches the result of the string expression. This is useful for determining whether a particular object exists at runtime.

Polling Frequency

The term `polling_frequency(A)`, where A is an attribute, returns the interval, in seconds, between successive polls of an attribute's value.

```
interface EthernetInterfaceStats : Instrumentation
{
    instrument SNMP{
        ifInOctets = "1.3.6.1.2.1.2.2.1.10",
    };

#pragma WrapCounter
    readonly instrumented attribute unsigned ifInOctets
    "The total number of octets received on the "
```

```
    "interface.";

    /* Compute Polling Frequency
     -------------------------- */
    readonly computed attribute int frequencyifInOctets
    = polling_frequency(ifInOctets);
};
```

Errnum

The term `errnum(attribute_name)` returns the error number for the last **get** of the attribute. For stored attributes the return value will always be zero (0). For all other attributes the return value can be either zero or non-zero.

Errstr

The term `errstr(attribute_name)` returns the string representation of the error for the last **get** of the attribute.

No_response

The `no_response(attribute_name)` returns TRUE if the error number for the last **get** of the attribute was MR_TIMEOUT or if the agent is marked unavailable.

```
abstract interface ICIM_Instrumentation : ICIM_MetaObject
"Instrumentation is the mechanism whereby information"
"regarding a SystemElement is retrieved from the managed"
"domain."
{
    attribute boolean InstrumentationOK
       "TRUE if the instrumentation is functioning
"properly."
       =TRUE

abstract interface NetworkAdapter_Fault_SNMP :
NetworkAdapter_Fault
"Instrumentation super-class for instrumented Network"
"Adapters using SNMP."
{
    instrument SNMP;
    readonly instrumented attributes int ifAdminStatus;
    refine computed InstrumentationOK
       =errnum(ifAdminStatus) ==0 &&
no_response(ifAdminStatus) == FALSE
}
```

For both rate() and average(), below, arguments other than the first argument may be either an integer literal or the name of a numeric attribute whose value can be converted to an integer.That is, the value cannot be a string but may be integer, float, or unsigned. If the arguments are attribute names, then the value of the attribute is obtained when the containing instance is connected to an accessor.

The values of the argument attributes are not tracked over time, making it necessary to explicitly request that the accessor get and use the new values. The accessor interface method synch_rates(instance) is used to get the values for a particular instance. The other way to accomplish this is to call update_rates(), directly, on an instance. Either call results in the accessor obtaining the values of all the attribute arguments at the time of the call and adjusting the rate and average parameter values. These calls do not cause tracking of the argument attribute values. To track them, one of the methods must be called every time the value changes. Normal usage is to have the values be a value propagated from a settings class where the values do not change very frequently.

Rate

The term `rate(A,T)`, where A is an attribute and T is a time interval, returns the rate at which A changed during the last T seconds. The time interval may be an expression that evaluates to a numeric value.

```
interface EthernetInterfaceStats : Instrumentation
{
    instrument SNMP{
        ifInOctets = "1.3.6.1.2.1.2.2.1.10",
    };
#pragma WrapCounter
    readonly instrumented attribute unsigned ifInOctets
        "The total number of octets received on the "
        "interface.";

    attribute unsigned pollingPeriod
        "The interval, in seconds, between "
     "successive polls."
        = 120;

   /* Compute the rate of ifInOctets
      ------------------------------ */
    computed attribute float ifInOctetsRate
        = rate(ifInOctets, pollingPeriod);
};
```

Average

The term `average(A, T, N, P)`, where A is an attribute,T is a time interval, N is a count, and P is a percent, computes the average value of A over the interval T.

A is an attribute name whose values will be sampled and averaged.

T is the window size and is an integer representing the number of seconds over which the samples of A are collected and averaged.

N is the minimum number of samples. It is an integer which represents the minimum number of samples of A to collect during T. N and T are used to provide input to the polling period for A. For example, if T is 600 and N is 10 then you want a sample every 600/10 = 60 seconds. Picking a small T and large N will cause very frequent polling of the attribute. The actual number of samples collected over an interval T is at least N. If you choose a large T and small N, then the actual number of samples may be many more than N because the repository collects one sample each time A is polled.

P is the percentage of samples required without error to compute the average. It is an integer between 0 and 100. The check is the percentage of the actual number of samples that do not have an error. As noted above, the actual number of samples may be many more than N.

Timestamp

The term `timestamp(A)`, where A is an attribute, returns the time when the value of A was last changed. A timestamp is returned in UNIX time format: an integer representing seconds since Midnight, 1 January, 1970 (GMT). You can only use the `timestamp()` operator on an attribute that is instrumented or has been declared with the **timestamped** keyword.

```
interface EthernetInterfaceStats : Instrumentation
{
    instrument SNMP{
        ifInOctets = "1.3.6.1.2.1.2.2.1.10",
    };

    #pragma WrapCounter
    readonly instrumented attribute unsigned ifInOctets
    "The total number of octets received on the "
    "interface";

    readonly computed attribute unsigned ifInOctetsTimestamp
    = timestamp(ifInOctets);
};
```

Syntax for Expressions

The *<expression>* parameter is the top level of the expression hierarchy. The way the grammar is constructed embeds the operator precedence as well as describes the expressions.

The MODEL compiler distinguishes between set expressions and mathematical or logical expressions by the operands on either side of the operator.

```
<expression>      ::=   <term>
                        <expression>
                  |     <term>
                        <binary_operator>
                        <term>
<term>            ::=   <simple_term>
                  |     <unary_operator>
                        <simple_term>
<simple_term>     ::=   <literal>
                  |     <reference>
                  |     "(" <expression> ")"
<reference>       ::=   <name>
                  |     <function_name>
                        "(" <arg_list> ")"
                  |     "|" <expression> "|"
<name>            ::=   <identifier>
                  |     <identifier> "." <identifier>
                  |     self
```

The *<binary_operator>* and *<unary_operator>* parameters represent the operators described in "Operators" on page 81. With minor exceptions, the syntax of MODEL operators is similar to those of C and C++.

Literals, indicated by the *<literal>* parameter, include the familiar arithmetic, string, and character literals of C and C++. These are described in "Literals" on page 80.

The *<function_name>* and *<arg_list>* parameters refer to operations defined in the MODEL code and "Built-in Functions" on page 85.

The vertical bars around an expression, | *<expression>* |, return a count of the number of members in a set. See "Operators for Set Expressions" on page 84 for the list of set operators.

The *<identifier>* . *<identifier>* parameters select a field in a previously declared struct.

When the Value of an Expression is Unavailable

When the domain manager marks an attribute as unavailable, it does so because it cannot determine the attribute's value. For event expressions written with Boolean operators, the result of the expression, in certain situations, can be determined when the value of one of the attributes is known.

Boolean Attributes

There are three possible results for an event expression written with a Boolean operator: True, False, and Unavailable.

Events E1 and E2 are both defined using attributes A1 and A2. The expression for event E1 uses the Boolean AND operator while the expression for event E2 uses the Boolean OR operator.

```
interface Ex1 : MR_ManagedObject
{
    event E1 = A1 && A2;
    event E2 = A1 || A2;

    attribute boolean A1;
    attribute boolean A2;
}
```

When either attribute A1 or A2 has a value of unavailable, the result of expressions E1 and E2 may not be unavailable. The following tables illustrate how the domain manager evaluates each event expression.

Table 12 shows all possible outcomes for event expression E1 when the value for one of its attributes, A1 or A2, is known.

	ATTRIBUTE A1		
ATTRIBUTE A2	TRUE	FALSE	UNAVAILABLE
TRUE	True	False	Unavailable
FALSE	False	False	False
UNAVAILABLE	Unavailable	False	Unavailable

Table 12: Truth Table for Event E1 = A1 && A2

Table 13 shows all of the possible outcomes for event expression E2 when the value for one of its attributes, A1 or A2, is known.

	ATTRIBUTE A1		
ATTRIBUTE A2	TRUE	FALSE	UNAVAILABLE
TRUE	True	True	True
FALSE	True	False	Unavailable
UNAVAILABLE	True	Unavailable	Unavailable

Table 13: Truth Table for Event E2 = A1 || A2

When the value of an unavailable attribute does not affect the result of the event expression, that result is given. For example, when A1 is `False` and A2 is `True` for event E2, the result is `True` because the Boolean OR operator requires that only one of the terms evaluate to `True`.

MODEL also applies a short circuit logic to expressions written with Boolean operators. For an event defined with Boolean AND, A2 is not evaluated when A1 is `False` because the result of the expression is already `False`. If A1 is `True` or `Unavailable`, the domain manager evaluates A2. For an event defined with Boolean OR, the domain manager does not evaluate A2 when A1 is `True` because the result is `True` regardless of the value of A2. If A1 is `False` or `Unavailable`, then the domain manager evaluates A2.

12

Instrument Declarations

An instrumentation declaration specifies the access method for all of the instrumented attributes of the class for which it is declared. All of the attributes for a class must use the same instrumentation access method. MODEL currently supports SNMP instrumentation. For SNMP, an instrumentation declaration connects the instrumented attributes to an SNMP object identifier (OID). The SNMP accessor, the domain manager component that performs SNMP polling, retrieves the value of the corresponding MIB variable and updates the value of the instrumented attribute.

The domain manager automatically manages the interconnection between the SNMP accessor, the monitoring system, and the subscription mechanism. As noted previously, the domain manager only monitors for those attributes and events necessary to diagnose the problems to which an InCharge client has subscribed. The SNMP accessor is aware of the monitoring structures connected to attributes that it instruments. If the value of an instrumented attribute is not required to evaluate an event expression, the SNMP accessor does not retrieve or update the value of that attribute.

```
<instrumentation_dcl>   ::=    instrument
                               SNMP
                               "{" <instrument_mapping> "}"
                               ;
<instrument_mapping>    ::=    <instrument_pair>
                               ["," <instrument_mapping> ]
<instrument_pair>::=           <attribute_name> "="
                               <instrumentation_string>
```

The keywords **instrument** and **SNMP** declare this as an SNMP instrumentation declaration.

Each optional <*instrument_mapping*> parameter joins an instrumented attribute with the OID of the SNMP object. If the <*instrumentation_string*> parameter is omitted, the <*attribute_name*> is mapped to an SNMP object with the same name. The syntax for declaring an instrumented attribute is described in "Instrumented Attributes" on page 55.

When the instrumented attributes of a class get their value from an SNMP table, specify the OID to the table's index, not the OID of a row in the table.

Example of an Instrumentation Declaration

```
interface EthernetInterfaceStats : Instrumentation
{
    instrument SNMP{
    ifDescr = "1.3.6.1.2.1.2.2.1.2",
    ifInOctets = "1.3.6.1.2.1.2.2.1.10",
    ifOutOctets = "1.3.6.1.2.1.2.2.1.16"
    };

    relationship
    Instruments, EthernetInterface, InstrumentedBy;

    readonly instrumented attribute string ifDescr
    "User-settable description of this interface";

#pragma WrapCounter
    readonly instrumented attribute unsigned ifInOctets
    "The total number of octets received on the "
    "interface.";

#pragma WrapCounter
    readonly instrumented attribute unsigned ifOutOctets
    "The total number of octets transmitted on the "
    "interface.";
}
```

Summary of Runtime Requirements for SNMP Instrumentation

The MODEL requirements for SNMP instrumentation include instrumented attributes and SNMP instrumentation. When you are ready to load your model into a domain manager, there are several additional steps you must complete.

The SNMP accessor is the component within the domain manager responsible for polling SNMP agents. You must provide the SNMP accessor with the following information.

- The polling parameters for the SNMP accessor. These include how often to poll, the number of retries to attempt for unsuccessful polls, and how long to wait before a timeout.
- The read community string for the SNMP agent.
- The IP address or host name of the SNMP agent.
- Index number of the SNMP table. This is optional.

There are two methods for configuring the SNMP accessor. The recommended method is to use the InCharge Framework. You may also connect an instance to the SNMP accessor through the InCharge Manager's C API.

13

MODEL Pragmas

Pragmas are added to MODEL files to guide the MODEL compiler. In many cases, pragmas are required; neglecting to use them will result in errors and code that does not properly compile.

Required Pragmas

The pragmas described in this section are almost always required for meaningful MODEL code.

#pragma include_c *file-name*

This pragma, which may appear anywhere in a MODEL file, causes the MODEL compiler to add a C preprocessor include statement for the given *file-name* to the generated *.c* file. The file name must be enclosed by quotes (" ") or angle brackets (<>). The quotes or brackets are preserved in the generated file.

#pragma include_h *file-name*

This pragma, which may appear anywhere in the MODEL file, causes the MODEL compiler to add a C preprocessor include statement for the given *file-name* to the generated *.h* file. The file name must be enclosed by quotes (" ") or angle brackets (<>). The quotes or brackets are preserved in the generated file.

Example of #pragma include_h and #pragma include_c

The example below is from the *department.mdl* file.

```
#include <repos/managed_object.mdl>

#pragma include_h <repos/managed_object.h>
#pragma include_c "department.h"
```

Additional Pragmas

The pragmas described in this section are not always required. However, you may be required to use these pragmas in certain situations to produce correct MODEL code.

#pragma ident "string"

This pragma provides a string for constructing an "RCS ident" comment in the generated code. The MODEL compiler will generate the following code in the generated *.c* file:

```
static const char SM_RCSIDSTRING[] = "string";
```

You can use a string of the form `RCS $Id: $` to be substituted by RCS. Strings of this form can be extracted from the MODEL library and printed by the UNIX `what` program.

#pragma Leaf File

This pragma, which may appear anywhere in a MODEL file, helps the MODEL compiler generate more efficient code. It tells the MODEL compiler that the classes declared in this file do not have subclasses declared for them in any other file. The MODEL compiler generates an error message if another source file includes the file with this pragma and declares subclasses.

#pragma Uses Propagation

This pragma, which must appear before the attribute declaration to which it applies, tells the MODEL compiler that access to this attribute may require access to other Repository instances. This pragma should be used before any attribute that can be refined as propagated or as computed with an expression referring to other propagated attributes.

Pragmas Used with SNMP Instrumentation

The following pragmas typically appear before instrumented attributes whose values are retrieved from an SNMP agent.

#pragma WrapCounter

This pragma appears before attributes of type unsigned int that represent a wrapping counter, such as an octets counter on a router interface. This means that rates and deltas computed over this attribute can never be negative; if the later value is smaller than the earlier value, it is assumed that the counter has wrapped and the value is adjusted accordingly.

```
Example of #pragma WrapCounter
traced interface EthernetInterfaceStats : Instrumentation
{
    instrument SNMP{
    ifInOctets = "1.3.6.1.2.1.2.2.1.10",
    ifOutOctets = "1.3.6.1.2.1.2.2.1.16"
    };

#pragma WrapCounter
    readonly instrumented attribute unsigned ifInOctets
        "The total number of octets received on the "
        "interface.";

#pragma WrapCounter
    readonly instrumented attribute unsigned ifOutOctets
        "The total number of octets transmitted on the "
        "interface.";
}
```

#pragma ObjectID

This pragma appears before an instrumented attribute declaration of type string whose value is the OID of an SNMP device. This pragma tells the Repository to convert the OID type returned by the SNMP agent into a MODEL string.

#pragma DotNotation

This pragma appears before an instrumented attribute declaration of type string whose value is an IP address. This pragma tells the Repository to convert the OCTECT STRING type returned by the SNMP agent into a MODEL string.

#pragma HexNotation

This pragma appears before an instrumented attribute declaration of type string whose value is in hexadecimal form. This pragma tells the Repository to convert the hexadecimal value into a string. The only legal characters in such a string are the digits zero through nine and the letters A through F. Successive pairs represent the value of a single byte.

Pragma Warnings in the MODEL Compiler

Using the proper pragmas is necessary to make MODEL generate the correct code. The MODEL compiler issues warnings about unrecognized pragmas as well as pragmas which are ignored.

"Unrecognized Pragma" Warnings

The MODEL compiler issues "Unrecognized Pragma" warnings for the following pragma syntax errors:

- Incorrect spelling
- Incorrect capitalization
- Missing or extra spaces

"Ignored Pragma" Warnings

Pragmas must be applied to top-level declarations. If you try to apply a pragma to a declaration refinement, it is ignored when you run the compiler. If the MODEL compiler discovers a pragma applied to a refinement, it will ignore it and issue a warning.

Index

A

Abstract 6
abstract keyword 35, 46
Accessor
  SNMP 55, 56, 93
Adapter ix, 30
Aggregate
  Declaration 15, 68, 74
  Operator 50, 60
    and 51
    avg 52
    Identity value 50
    max 52
    min 52
    or 51
    prod 52
    sum 52
  Refine 74
aggregate keyword 74
Alarm Log 30
and aggregate operator 51
Arithmetic operator 81
Attribute 8, 49
  Access type 49
  Computed 49, 54, 79
    Refine 55
  Instrumented 49, 55, 70
    Refine 58
    Type 56
  Propagate 50, 58
    Refine 60
  Stored 49, 53
    Refine 54
  Table 60
    Refine 62
  Unavailable value 50
attribute keyword 53, 55, 57, 59
average() 89
avg aggregate operator 52

B

BASEDIR xi
Bitwise operator 82
Boolean
  Expression 12
  Operator 91
Broker 33

C

C preprocessor 97
C++ compiler 23
Cardinality 9, 63
Character literal 80
  Escape character 80
Class 6
  Abstract 6
  Concrete 6
  Inheritance 5
  Interface declaration 6
  Listing 35
  see also Interface
Codebook 71
  Recompute 72
Compiler
  C++ 23
  make 26
  Makefile 26
  MODEL 23, 26
  Options 27
  Preprocessor 25, 97
  Using 26
Compound notification 16
Computed
  Attribute 49, 54, 79
  Relationship 64
computed keyword 55, 64
Concrete 6
Console
  Alarm Log 30
  Changing attribute values 37
  Creating instances 36
  Domain Browser 30
  Generic 29, 30
  Inserting into a relation 37
  Listing classes 35
  Listing models 34

Loading MODEL library 34
Notifying events 39
Topology Browser 30
CORBA IDL 1
Correlation model ix
Correlator 72

D

Data type 44
Declaration
Aggregate 15, 68, 74
Refine 74
Attribute 8
Computed 54
Instrumented 55
Propagate 11, 58
Stored 53
Table 60
Event 12, 67
Refine 69
Export 17, 76
Forward 45
Instrument 93
Interface 6, 45
Interface header 6, 46
Operation 77
Problem 13, 67, 70
Refine 72
Propagate aggregate 16, 68, 75
Refine 75
Propagate symptom 14, 68, 73
Refine 74
Relationship 9, 45, 63
Refine 65
Symptom 68, 72
Refine 73
delta() 85
dmctl 30, 31
Changing attribute values 37
create command 36
Creating instances 36
getClasses command 36
Help 31
insert command 38
Inserting into a relation 38
List classes 36
Listing models 35
Loading MODEL library 33
notify command 39
Notifying events 39
put command 37
Starting 31
dmstart 33
Loading MODEL library 33
Domain Browser 30
Listing classes 35
Domain manager
Codebook 71
Recompute 72
Correlator 72
dmstart 33
Listing models 34
Loading MODEL library 33
Repository 5, 63
Starting 33

E

Environment variable
SM_LIBPATH 32
Equality operator 81
Escape character 80
Event 2
Declaration 12, 67
Refine 69
Expression 2, 12, 50, 70
Guard 69
Modeling 12
Notification 2, 70, 76
Notifying 38
Source ix
Subscribe 2, 76
Example model 18
Export declaration 17, 76
export keyword 76
Expression 79
Boolean 12
Set 77, 78, 79, 84
Operator 84
Syntax 90
Unavailable value 91
Event
see Event expression

F

file.c 23, 97
file.h 23, 97
Floating point literal 80
Forward declaration 45

Function
delta() 85
obj() 86
polling_frequency() 86
rate() 88
timestamp() 54, 89

G

Generic Console
see Console

H

Hexadecimal integer 80

I

Identifier 43
Identity value 50
List of 51
InCharge Broker 33
#include 24
Inheritance 5, 46
Instance
Creating 36
Modifying properties 37
Instrument
Declaration 93
instrument keyword 93
Instrumented
Attribute 49, 55, 70
Refine 58
Type 56
instrumented keyword 57
Integer
Hexadecimal 80
Literal 80
Octal 80
Interface 45
Abstract 6
Declaration 6, 45
Header declaration 6, 46
interface keyword 45

K

key keyword 61
Keyword
abstract 35, 46
aggregate 74
attribute 53, 55, 57, 59
computed 55, 64
export 76
instrument 93
instrumented 57
interface 45
key 61
List of 41
loss 69
problem 71
propagate 60, 74, 75
propagate aggregate 75
propagate symptom 74
readonly 53, 54, 64
refine 18
relationship 10, 59, 63
relationshipset 10, 59, 63
set 78
SNMP 93
spurious 69
stored 53, 64
symptom 71, 73
table 61
timestamped 54, 57, 89
unique 61
void 78

L

Lexical element 79
Library
see MODEL Library
Literal 80
Character 80
Escape character 80
Floating point 80
Integer 80
String 80
Escaped character 80
Logic operator 81
loss keyword 69

M

make 26
Makefile 26
Managed Object Definition Language
see MODEL
max aggregate operator 52
min aggregate operator 52
MODEL ix, 1
Built-in functions 85

Compiler 23, 26
make 26
Makefile 26
Options 27
Preprocessor 25, 97
Using 26
Data type 44
Identifier 43
Keyword 41
Lexical element 79
Library 23
Loading 31, 33
Location 32
Naming 23
Operator 81
Pragma
see Pragma
Model
Correlation ix
Example 18
MR_ManagedObject 6, 46
MR_MetaObject 6

N

Notification 2, 70, 76
Compound 16
Notifying events 38

O

obj() 86
Object identifier 93
Octal integer 80
OID 93
Operation
Declaration 77
Operator 81
Arithmetic 81
Associativity 84
Bitwise 82
Boolean 91
Equality 81
Logic 81
Precedence 84
Relational 81
Set expression 84
Shift 81
Unary 81
Aggregate
see Aggregate Operator
or aggregate operator 51

P

polling_frequency() 86
Pragma 97
#pragma DotNotation 99
#pragma HexNotation 100
#pragma ident 98
#pragma include_c 25, 97
#pragma include_h 25, 97
#pragma Leaf File 98
#pragma ObjectID 99
#pragma Uses Propagation 54, 55, 98
#pragma WrapCounter 58, 99
Preprocessor 25, 97
Problem
Declaration 13, 67, 70
Refine 72
problem keyword 71
prod aggregate operator 52
Propagate
Aggregate
Declaration 16, 68, 75
Refine 75
Attribute 50
Declaration 11, 58
Refine 60
Symptom
Declaration 14, 68, 73
Refine 74
propagate aggregate keywords 75
propagate keyword 60, 74, 75
propagate symptom keywords 74

R

rate() 88
readonly keyword 53, 54, 64
Refine 18, 47
Aggregate 74
Attribute
Computed 55
Instrumented 58
Propagate 60
Stored 54
Event 69
Problem 72
Propagate aggregate 75
Propagate symptom 74
Relationship 65

Symptom 73
refine keyword 18
Refinement
see Refine
Relation 9, 63
Cardinality 63
Inserting 37
Relational operator 81
Relationship 58
Cardinality 9, 63
Computed 64
Declaration 9, 45, 63
Refine 65
Stored 64
relationship keyword 10, 59, 63
Relationshipset 58
Empty 50
Refine 65
relationshipset keyword 10, 59, 63
Repository 5, 63

S

Set
Expression 77, 78, 79, 84
Syntax 90
Operator 84
set keyword 78
Shared library
see MODEL Library
Shift operator 81
SM_LIBPATH 32
SNMP 49, 93
Accessor 55, 56, 93
Instrumentation 93
OID 93
SNMP keyword 93
spurious keyword 69
Stored
Attribute 49
Relationship 64
stored keyword 53, 64
String literal 80
Escaped character 80
Subscribe 2, 13, 76
sum aggregate operator 52
Symptom
Declaration 68, 72
Refine 73
symptom keyword 71, 73

T

Table 60
refine 62
table keyword 61
Technical Support xv
timestamp() 54, 89
timestamped keyword 54, 57, 89
Topology ix
Topology Browser 30
Listing classes 35

U

Unary operator 81
unique 46
unique keyword 61

V

View
Alarm Log 30
Domain Browser 30
Listing classes 35
Topology Browser 30
Listing classes 35
void keyword 78