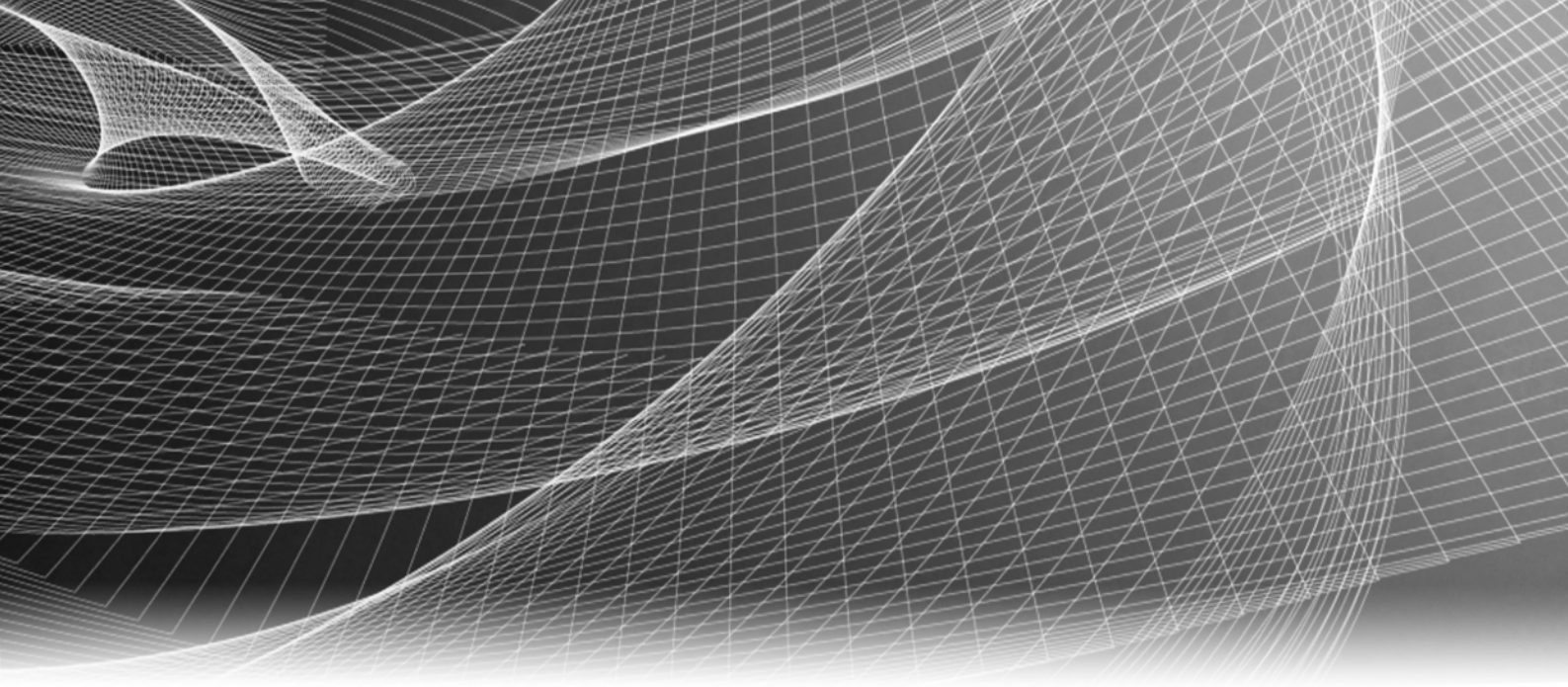

EMC® Smarts®

Foundation 9.3

MODEL Reference Guide

P/N 302-000-678

REV 01

EMC²

Published June, 2014

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

The information in this publication is provided as is. EMC Corporation makes no representations or warranties of any kind with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a particular purpose. Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

EMC^2, EMC, and the EMC logo are registered trademarks or trademarks of EMC Corporation in the United States and other countries. All other trademarks used herein are the property of their respective owners.

For the most up-to-date regulatory document for your product line, go to the technical documentation and advisories section on EMC Online Support.

CONTENTS

Index

FIGURES

TABLES

PREFACE

As part of an effort to improve its product lines, EMC periodically releases revisions of its software and hardware. Therefore, some functions described in this document might not be supported by all versions of the software or hardware currently in use. The product release notes provide the most up-to-date information on product features.

Contact your EMC technical support professional if a product does not function properly or does not function as described in this document.

Note: This document was accurate at publication time. Go to EMC Online Support (https://support.emc.com) to ensure that you are using the latest version of this document.

Purpose

This document is a reference guide for the EMC® Smarts® Managed Object Definition Language (MODEL). It serves as a comprehensive reference for the syntax and use of MODEL declarations.

Audience

This document is intended for programmers who are developing correlation models for EMC Smarts Foundation applications. A correlation model describes the domain or system to be managed, and this model is written in MODEL.

This document is also useful for those users who are extending an existing EMC Smarts application by using Dynamic Model. This document can be used in conjunction with the *EMC Smarts Dynamic Modeling Tutorial*.

Those interested in learning more about MODEL and how it works might also find the Chapter 2, "Introduction by Example," especially useful.

EMC Smarts installation directory

In this document, the term BASEDIR represents the location where EMC Smarts software is installed.

- For Windows, this location is C:\InCharge\‹*productsuite*›.
- For UNIX, this location is /opt/InCharge/‹*productsuite*›.

InCharge names the directory where the software is installed. The ‹*productsuite*› represents the EMC Smarts product suite to which the product belongs. This location is referred to as BASEDIR/smarts.

Optionally, you can specify the root of BASEDIR to be something other than:

- Windows: C:\InCharge
- UNIX: /opt/InCharge

However, you cannot change the ‹*productsuite*› location under the root directory.

The *EMC Smarts System Administration Guide* provides more information about the directory structure.

Related documentation

In addition to this document, EMC Corporation provides a Help system for command line programs as well as product documentation.

Help for command line programs

Descriptions of command line programs 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.

EMC Smarts documentation

Readers of this document might find the following related documentation helpful. These documents are updated periodically. Electronic versions of the updated manuals are available on EMC Online Support:

- *EMC Smarts System Administration Guide*
- *EMC Smarts Common Information Model (ICIM) Reference for Service Assurance Manager*
- *EMC Smarts Common Information Model (ICIM) 1.11 Reference for Non-Service Assurance Manager Products*
- *EMC Smarts ASL Reference Guide*
- *EMC Smarts Perl Reference Guide*
- *EMC Smarts Dynamic Modeling Tutorial*
- *EMC Smarts Foundation EMC Data Access API (EDAA) Programmer Guide*

Conventions used in this document

EMC uses the following conventions for special notices:

⚠DANGER

DANGER indicates a hazardous situation which, if not avoided, will result in death or serious injury.

⚠WARNING

WARNING indicates a hazardous situation which, if not avoided, could result in death or serious injury.

⚠CAUTION

CAUTION, used with the safety alert symbol, indicates a hazardous situation which, if not avoided, could result in minor or moderate injury.

NOTICE

NOTICE is used to address practices not related to personal injury.

Note: A note presents information that is important, but not hazard-related.

IMPORTANT

An important notice contains information essential to software or hardware operation.

Typographical conventions

EMC uses the following type style conventions in this document:

Bold	Use for names of interface elements, such as names of windows, dialog boxes, buttons, fields, tab names, key names, and menu paths (what the user specifically selects or clicks)
Italic	Use for full titles of publications referenced in text
`Monospace`	Use for: • System output, such as an error message or script • System code • Pathnames, filenames, prompts, and syntax • Commands and options
`Monospace italic`	Use for variables.
`Monospace bold`	Use for user input.
[ ]	Square brackets enclose optional values
|	Vertical bar indicates alternate selections — the bar means “or”
{ }	Braces enclose content that the user must specify, such as x or y or z
...	Ellipses indicate nonessential information omitted from the example

Where to get help

EMC support, product, and licensing information can be obtained as follows:

Product information — For documentation, release notes, software updates, or information about EMC products, go to EMC Online Support at:

https://support.emc.com

Technical support — Go to EMC Online Support and click Service Center. You will see several options for contacting EMC Technical Support. Note that to open a service request, you must have a valid support agreement. Contact your EMC sales representative for details about obtaining a valid support agreement or with questions about your account.

Your comments

Your suggestions will help us continue to improve the accuracy, organization, and overall quality of the user publications. Send your opinions of this document to:

`techpubcomments@emc.com`

CHAPTER 1
About MODEL

This chapter consists of the following sections:

Overview

The EMC® Smarts® Managed Object Definition Language (MODEL) is a descriptive language used to develop correlation models for EMC Smarts Foundation applications. MODEL provides constructs for defining classes of managed objects and describing their properties. Properties include attributes, operations, relationships, basic events, problems, and symptoms.

The main purpose of MODEL is to provide a means of describing the potential components of a managed domain in a reusable object-oriented framework. Developing your management application with EMC Smarts 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.
- EMC Smarts applications separate the generic correlation model 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-driven information including problems, symptoms, and their propagation.

Modeling event-driven 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 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 a basic event that is directly observable on a class instance. A problem, defined on a class, causes symptoms that may be directly observable on the same class or on related classes. 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. Propagation of events 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.

Detecting events

An EMC Smarts 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 EMC Smarts clients, which include adapters.

When an EMC Smarts 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 accessible to EMC Smarts clients.

CHAPTER 2
Introduction by Example

This chapter consists of the following sections:

Overview

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.

Class properties

A 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 as follows.

- 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. For example, in Figure 1 on page 20 the UnitaryComputerSystem class has an attribute XYZ and an event Down. Each of the subclasses (Router and Host) inherits the attribute XYZ and the Down event.

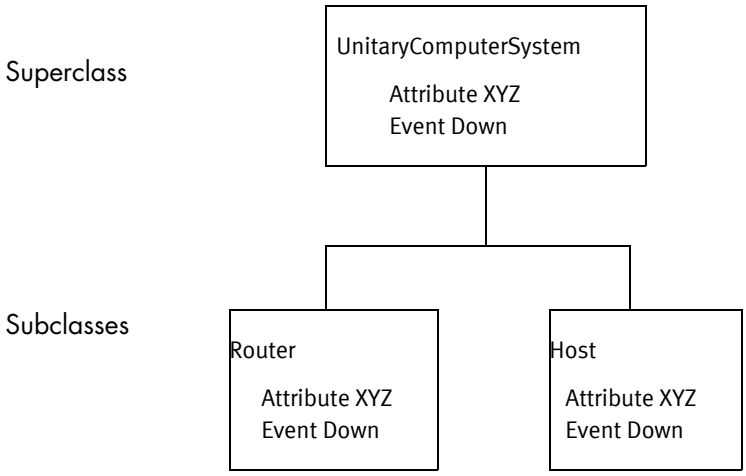

Figure 1 Inheritance of properties by subclasses

  A subclass may refine existing properties or add new properties and operations to introduce unique behavior or states. "Inheritance" on page 61 provides further information.

- A class can be declared as abstract or concrete. Concrete classes can be directly instantiated and must define implementations for all their properties. Abstract classes cannot be instantiated and may have properties for which they do not define implementations. "Interface header declaration" on page 61 provides further information.

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. 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 used less often.

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, as shown in Figure 2 on page 21.

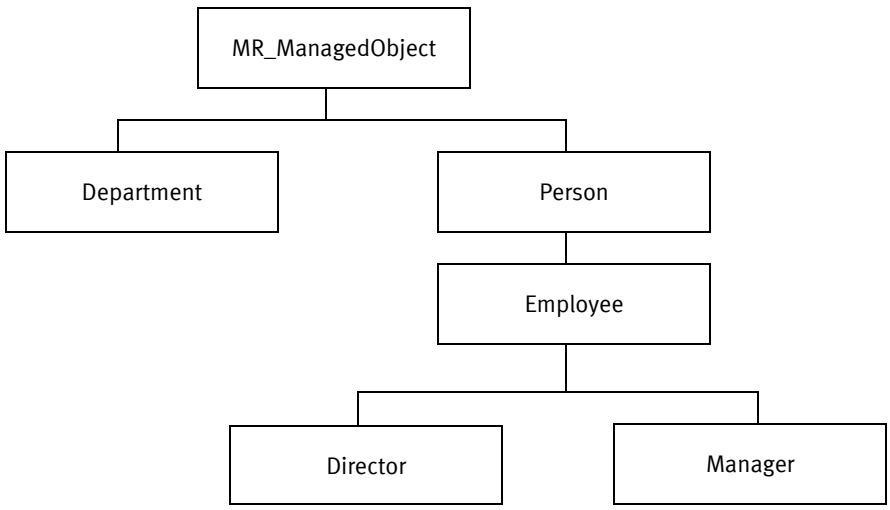

Figure 2 Sample hierarchy

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 EMC Smarts clients. 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 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 obtained, by using an accessor.

A stored 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. "Supported types" on page 55 provides a list of the supported attribute types. 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 data 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. Intrinsically, computed attributes are read only, and therefore, the **readonly** keyword is optional. Their value cannot be set through a EMC Smarts client.

Note: In a MODEL file, "//" is used to comment a line. "/* and "*/" are used comment a block of lines.

```
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. The term *relation* refers to the pair of relationships between two objects.

The cardinality of a relationship can be to-one or to-many. The cardinality of 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. The cardinality of 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 cardinality of 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 cardinality of 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"
{
   relationship WorksIn, Department, Contains
   "The department the employee works in";
}

interface Manager : Employee
"An employee who has managerial responsibilities"
{
   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 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 event-driven behavior

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 event-driven behavior.

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

Note: An object created with the event declaration is referred to as a *basic event*. Repository objects created with event, problem, aggregate, symptom, propagate symptom, and propagate aggregate declarations are all referred to generically as *events*.

Event declarations

An event declaration declares a basic 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 basic 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 the basic event is occurring. When the event expression evaluates to false it either means that the basic event has cleared or that the event is not active.

Examples of an event declaration

The Employee class declares three basic 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 basic events. When either UnderPaidEmployee or OverWorked occurs, the basic 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 EMC Smarts 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 problems that an EMC Smarts 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.

"Examples of an event declaration" on page 27 describes 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. The symptom refers to 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: UnderPaidEmployee and UnderAchieved. Both these events occur in instances of the Employee class. However, neither of these events is 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. A propagate symptom declaration causes all instances of the target problem/symptom/event to be in the problem signature for the problem that causes the propagated symptom.

```
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 "
   "performing as expected"
   = Employee, Contains;
}
```

Note: If the name of the basic event that propagates from the related class is not specified, it means that the basic 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. Therefore, 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 EMC Smarts 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.

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 EMC Smarts clients. When you declare an event as exported, it is accessible to EMC Smarts clients, so they can subscribe to the event and receive notifications when the event occurs.

If an event is not exported, it remains private and is not accessible outside of the Domain Manager's Repository.

A single export declaration can declare all of the public events for a class. Table 7 on page 100 specifies the types of events that 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 your MODEL code easier to read, 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 a stored 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

The following is the complete model from which the examples in this chapter were taken.

```
#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";

}
```

CHAPTER 3
Working with MODEL Libraries

This chapter consists of the following sections:

Overview

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 dmctl utility, and adapters.

- 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. The *EMC Smarts ASL Reference Guide* provides further information.

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
```

The dmctl utility can be used 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 33.

More information regarding dmctl is available in two places:

- The HTML documentation for EMC Smarts 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, the Domain Manager (and other EMC Smarts commands) is limited to loading libraries from specific directories. When invoked, EMC Smarts programs append the SM_LIBPATH environment variable and the BASEDIR/smarts/lib path to the system library variable described in Table 1 on page 39.

Table 1 Library variables for supported operating systems

Operating system	Library variable
Solaris	LD_LIBRARY_PATH
Red Hat Linux	LD_LIBRARY_PATH
Windows 2003/2008 (also used to locate main programs)	PATH

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 and Red Hat Linux

On UNIX and Red Hat Linux 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 Domain Manager Toolkit 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 *EMC Smarts System Administration Guide* to obtain a permanent license.

There are two commands available to start a Manager: **dmstart** and **sm_server**. The **sm_server** command has more options, including the specification of a configuration directory, and therefore is the command normally used in production. The **dmstart** command is generally used early in the development cycle before configuration files have been created.

Note: The **dmstart** and **sm_server** commands assume that the Broker is already running.

You can use the -help option for a complete list of command line options. Change to the BASEDIR/smarts/bin directory and enter:

```
dmstart --help
```

or

```
sm_server --help
```

One of the required command line parameters of **sm_server** is the domain manager configuration directory ("--config=‹directory›"). The EMC Smarts software includes a directory called BASEDIR/smarts/examples, which contains several different subdirectories. Each subdirectory is for a separate example. Some of these examples include a conf subdirectory, such as BASEDIR/smarts/examples/apollo13/conf. The BASEDIR/smarts/examples/apollo13/conf directory contains a sample bootstrap.conf and sample .import file. These files show how to instantiate key objects and set parameters on them that may be useful for your application.

The installed versions of these files have their contents commented out. You may activate these by running the sm_edit utility to bring up these files in an editor so that you can uncomment selected lines. Alternatively, you can create a new configuration directory named after your application by copying BASEDIR/smarts/conf/SDK to BASEDIR/smarts/conf/‹new-app-name›, and then running sm_edit to edit the files in this new directory.

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.

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.

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 dmctl utility to list the models loaded into a Domain Manager.

Use the getModels option to retrieve the list.

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

Listing classes in the MODEL library

You can use the dmctl utility to list the classes declared in the model. If more than one model is loaded into the Domain Manager, only the concrete classes from the loaded models are retrieved.

The getClasses command can be used to list the classes.

```
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 dmctl utility to create instances of classes declared in your model. By creating instances, you build a topology of managed elements.

The **create** command is used to create an instance. You must specify the class name and an instance name.

In this example, 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 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 dmctl utility

You can change the value of an attribute by using the **put** command:

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

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

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.

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

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

CHAPTER 4
Basic Lexical Elements of MODEL

This chapter consists of the following sections:

Overview

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 2 on page 46 lists the MODEL keywords, and Table 3 on page 54 lists the C++ keywords and other words 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 lists are currently used by MODEL. Unused keywords are reserved for future use.

Table 2 MODEL keywords (page 1 of 8)

Keyword	Description	Refer to
_time	Represents the current time as an int in UNIX epoch format. Used in expressions.	
abstract	Defines an interface for which instances cannot be instantiated.	"Interface header declaration" on page 61.
aggregate	A type of declaration that groups one or more basic events, problems, aggregates or propagated aggregates declared for the same class into a single abstract event by disjunction.	"Aggregate declaration" on page 96.
and	An aggregate operator that is used to determine the results when propagating attributes over a relationshipset.	"Attributes propagated over a relationshipset" on page 65. "Propagated attributes" on page 75.
apriori	Indicates the probability of a problem occurring before looking at any of the symptoms as evidence.	"apriori keyword" on page 89.
ascending	Indicates the direction when an access key has been defined for a table.	The description of ascending on page 78.
attribute	Attributes describe the state of a managed object. Their values can be queried and manipulated by EMC Smarts clients and displayed in a console.	Chapter 6, "Attribute Declarations"on page 63.

Table 2 MODEL keywords (page 2 of 8)

Keyword	Description	Refer to
avg	An aggregate operator that is used to determine the results when propagating attributes over a relationshipset.	"Attributes propagated over a relationshipset" on page 65. "Propagated attributes" on page 75.
binary	A data type supported by MODEL.	"Data types" on page 55.
boolean	A data type supported by MODEL.	"Data types" on page 55.
byte	A data type supported by MODEL.	"Data types" on page 55.
case	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.	"Case operator" on page 115.
check	A separator used to identify a guarded expression.	"Guarded events" on page 86.
computed	Specifies an access type for an attribute. The value of a computed attribute is calculated on demand by evaluating an expression, when the value of the attribute is required.	"Computed attributes" on page 70.
const	Indicates that the operation does not change the instance in the repository, it cannot assign to attributes of the class. However, it may change other parts of the repository.	"Operation Declarations" on page 103. "Instance-only read lock" on page 110.
const value	Declares a constant value.	
constraint	Constraint expressions are used to enforce limits on the values of attributes or combinations of attributes. Constraints are enforced at runtime, unlike ranges. Constraints can also be used for relationshipsets.	"Constraints" on page 127.
default	A key in a case evaluation expression operator.	"Case operator" on page 115.
definition	A keyword used when defining an operation.	"Operation Declarations" on page 103.
delta	A built-in function that is supplied by MODEL. The term `delta(A)`, where A is an attribute, returns the difference between the current value of A and its previous value.	"Built-in functions" on page 117.
descending	Indicates the direction when an access key has been defined for a table.	The description of descending on page 78.
double	A data type supported by MODEL.	"Data types" on page 55.

Table 2 MODEL keywords (page 3 of 8)

Keyword	Description	Refer to
else	An operator that is used when evaluating expressions.	"Else operator" on page 114.
enum	A data type supported by MODEL. Enumerated type assigns integer values to each of its tags.	"Enumerations" on page 56.
event	A type of declaration that defines a basic event, meaning it is observable in the object where it occurs.	"Event declaration" on page 85.
events_only	Indicates that a proxy declaration is restricted to proxy only events, as opposed to events and attributes.	
explains	Indicates that this symptom of a problem is used for display purposes and does not affect the analysis. Indicates that this member of the ‹symptom_list› of a symptom is used for display purposes and does not affect the analysis.	"explains keyword" on page 90. "explains keyword" on page 93.
export	Indicates that the associated event-driven behavior is visible outside of the Repository. EMC Smarts clients, such as the Global Console and adapters, can subscribe to exported event-driven behavior and receive notifications when they occur.	"Export declaration" on page 99.
external	Indicates whether the literal text specified as the initial value of a string attribute will be externalized. The external keyword implies both static and readonly. When including the external keyword in a declaration, do not include the static and readonly keywords.	"Stored attributes" on page 67.
FALSE	A valid value for a boolean data type.	"Data types" on page 55.
float	A data type supported by MODEL.	"Data types" on page 55.
foreach	This keyword is used in constraint expressions to iterate over the contents of a relationshipset, applying a test to each element.	"Constraints" on page 127.
fragment	A computation expression that my be embedded in other expressions.	
get_row	A keyword used in expressions to get a row from a table.	
hard	Modifies a constraint to indicate that it is always enforced and fatal to violate.	"Constraints" on page 127.

Table 2 MODEL keywords (page 4 of 8)

Keyword	Description	Refer to
idempotent	Indicates that the operation, if called multiple times without an intervening put, will return the same value each call.	The description of idempotent on page 105.
if	Indicates a guard clause in an event expression.	"Guarded events" on page 86.
implementation	Designates text that is copied verbatim to the generated C++ code. This keyword is used at the end of an interface declaration.	"Interface header declaration" on page 61.
imported	Reserved for future use.	
in	When the **in** keyword is included with an argument to an operation, it indicates that information passes from the caller to the operation.	"in keyword" on page 107.
inout	An operation argument direction modifier.	"inout keyword" on page 108.
instrument	A type of declaration that specifies the access method for all of the instrumented attributes of the class for which it is declared.	"Instrument Declarations" on page 131.
instrumented	Indicates an access type for an attribute. An instrumented attribute obtains its value through a standard protocol, such as SNMP, and requires a separate instrumentation declaration that ties the attribute to the protocol.	"Instrumented attributes" on page 72.
instrumented_op	Reserved for future use.	
int	A data type supported by MODEL.	"Data types" on page 55.
interface	Defines a managed object class or a type of managed element in the Repository.	"Declaring an Interface" on page 59.
internal	Indicates that the attribute or operation is not visible outside of the repository.	"Operation Declarations" on page 103.
key	A means to access a table. If a key is declared, then the table row must be a struct and the key must be a field of the struct.	"Table attributes" on page 77.
long	A data type supported by MODEL.	"Data types" on page 55.
loss	Specifies the probability that the observable basic event is lost or not observed (its expression does not evaluate to true) even though it is occurring.	"Event declaration" on page 85.

Table 2 MODEL keywords (page 5 of 8)

Keyword	Description	Refer to
max	An aggregate operator that is used to determine the results when propagating attributes over a relationshipset.	"Attributes propagated over a relationshipset" on page 65. "Propagated attributes" on page 75.
min	An aggregate operator that is used to determine the results when propagating attributes over a relationshipset.	"Attributes propagated over a relationshipset" on page 65. "Propagated attributes" on page 75.
obj	A built-in function in MODEL. The term `obj( <string_expression>` returns the object whose name matches the result of the string expression.	"Object" on page 118.
old	The **old** keyword can be used before an attribute name in a constraint expression. When used, it indicates "check the attribute's old value".	"Constraints" on page 127.
or	An aggregate operator that is used to determine the results when propagating attributes over a relationshipset.	"Attributes propagated over a relationshipset" on page 65. "Propagated attributes" on page 75.
out	When the **out** keyword is included with an argument to an operation, it indicates that value of the argument is set by the operation and returned to the caller.	"out keyword" on page 108.
polling_frequency	A built-in function in MODEL. The term `polling_frequency(A)`, where A is an attribute, returns the interval, in seconds, between successive polls of an attribute's value.	"Polling frequency" on page 118.
problem	A type of declaration that 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.	"Problem declaration" on page 87.
prod	An aggregate operator that is used to determine the results when propagating attributes over a relationshipset.	"Attributes propagated over a relationshipset" on page 65. "Propagated attributes" on page 75.
propagate	Declares that the value of the attribute is computed or copied from other objects through a relation.	"Propagated attributes" on page 75.
proxy	Declares an interface (class) whose instances represent the information of other interfaces.	

Table 2 MODEL keywords (page 6 of 8)

Keyword	Description	Refer to
rate	A built-in function in MODEL. 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.	"Rate" on page 120.
rate_last	An alternative to rate(A,T) that may be used for polled attributes in cases when rate interval T equals the polling period of A.	"Rate_last" on page 120
readonly	Prevents write access to an attribute's value. Provided for computed relationships. Performs a repository-wide read lock when used with an operation. The **readonly** keyword, when used in conjunction with #pragma Local Operation, locks (for reading) only the object on which the operation is invoked.	"Propagated attributes" on page 75. "Computed attributes" on page 70. "Instrumented attributes" on page 72. "Relationship Declarations" on page 79. "Repository-wide read lock" on page 109. "Instance-only read lock" on page 110.
refine	All properties defined for a class are inherited by its subclasses. The **refine** keyword is used to modify the properties specified by attribute, relationship, and event declarations in the subclass.	"Attribute Declarations" on page 63. "Relationship Declarations" on page 79. "Declaring Event-Driven Behavior in MODEL" on page 83.
relationship	Defines a to-one connection between classes	"Relationship Declarations" on page 79.
relationshipset	Defines a to-many connection between classes.	"Relationship Declarations" on page 79.
REMOTE_REPOSITORY	The **REMOTE_REPOSITORY** keyword is used in conjunction with the **instrument** keyword.	"Instrument Declarations" on page 131.
required	Indicates that an attribute's value must be initialized when an instance of the class is created. A transaction must be used to create and initialize the attribute.	The description of the **required** keyword on page 68.
return	Specifies a return value from a function written in MODEL.	"Operation Declarations" on page 103.
row	Declares an identifier in an SNMP row.	
self	A reference to the current instance in an expression.	"Syntax for expressions" on page 122.
set	Indicates that the operation returns a set of values.	"Return_type parameter" on page 106.
short	A data type supported by MODEL.	"Data types" on page 55

Table 2 MODEL keywords (page 7 of 8)

Keyword	Description	Refer to
SNMP	The **SNMP** keyword is used in conjunction with the **instrument** keyword. The **instrument SNMP** declaration connects the instrumented attributes to an SNMP object identifier (OID).	"Instrument Declarations" on page 131. "Instrumented attributes" on page 72.
soft	Constraints may be either hard or soft. Soft constraints are currently treated similar to hard constraints.	"Constraints" on page 127.
spurious	Specifies the probability that this observable basic event is not occurring, even though the event expression evaluates to true.	"Event declaration" on page 85.
static	Declares the scope of a stored attribute. If the stored attribute is declared static, a single copy of the attribute value is shared by all instances of the interface.	"Stored attributes" on page 67
stored	Specifies an access type for an attribute. The value of a stored attribute is specified in the model or set at runtime when the instance is created. It is also a keyword used to specify the access type of a relationship. It specifies where and how information about the relationship is obtained.	"Stored attributes" on page 67. "Relationship Declarations" on page 79.
string	A data type supported by MODEL.	"Data types" on page 55.
struct	A data type supported by MODEL.	"Structures" on page 57.
sum	An aggregate operator that is used to determine the results when propagating attributes over a relationshipset.	"Attributes propagated over a relationshipset" on page 65. "Propagated attributes" on page 75.
symptom	A type of declaration that groups a set of basic events, symptoms, propagated symptoms and other problems that it can cause into a single event by conjunction.	"Symptom declaration" on page 92.
table	A type of data attribute. It is a collection of rows, each of which can be a scalar or a structure.	"Table attributes" on page 77.
timestamp	The term `timestamp`(A), where A is an attribute, returns the time when the value of A was last changed. Used in conjunction with the **timestamped** keyword.	"Timestamp" on page 121.

Table 2 MODEL keywords (page 8 of 8)

Keyword	Description	Refer to
timestamped	When an attribute is declared with the optional keyword **timestamped**, the Domain Manager creates a read-only unsigned integer whose value is the time that the attribute was last updated.	"Stored attributes" on page 67.
TRUE	A valid value for a boolean data type.	"Data types" on page 55.
unique	Defines an interface where only one instance of the interface, or any derived interface, may be instantiated at any given time. When used with a table, it specifies that each row must have a unique key. There is also a unique operator for set expressions.	"Interface header declaration" on page 61. The description of the **unique** keyword on page 78. "Operators for set expressions" on page 116.
unsigned	Indicates an unsigned int data type in a class declaration.	
unstringable	Indicates an external type has no means to convert to/from a string representation.	
void	The value type returned when an operation has no return value.	"Return_type parameter" on page 106.
with	An alternative to events_only. It is used in a proxy declaration to proxy all of the events and attributes of an interface (class).	

Table 3 C++ and other reserved words

and_eq	extern	sizeof
asm	external	static_cast
auto	for	switch
bitand	friend	template
bitor	function	this
bool	goto	throw
break	handler	try
catch	inline	type
causes	mutable	typedef
char	namespace	typeid
class	new	typename
compl	not	union
const_cast	not_eq	using
continue	operator	value
CORBA	or_eq	virtual
delete	private	volatile
do	protected	wchar_t
dynamic_cast	public	while
error_code	register	xor
explicit	reinterpret_cast	xor_eq
extern	signed	

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 4 on page 55 lists the data types supported by MODEL.

Table 4 Supported types

	Data Type	Definition
Fundamental Types	binary	Binary data.
	boolean	Two possible values: TRUE or FALSE.
	byte	An unsigned 8-bit integer ranging from 0 to 255.
	double	Any double-precision floating point number (similar to C++).
	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	A NUL-terminated array of UTF-8 data.
	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.
User-defined Types	enum	Enumerated type that assigns integer values to each of its tags. Integer values may be positive, negative or zero.
	struct	A collection of fields, each field specifies its own type.

Enumerations

An enumerated type contains a list of tags and assigns integer values to each of the tags. The integer values may be positive, negative or zero. Enumerations that are declared without numerical assignments start with 0 and increase by 1.

Each tag must be unique within a namespace (global or within an interface); you cannot have two enumerations in the same namespace that contain the same tag.

The format for declaring an enumeration is as follows:

```
<enum>              ::=    enum
                           <enum_name>
                           "{" <tags_list> "}"
                           ";"
<tags_list>         ::=    <tag>
                           [","  <tags_list>]
<tag>               ::=    <tag_name>
                           [ <tag_value> ]
<tag_value>         ::=    "="
                           int
```

The following example defines the enumeration OperationalState.

```
enum OperationalState {OFF, ON, UNKNOWN, TESTING};
```

Since the declaration does not specify values for any of the tags, the first tag is assigned the value 0. The value of each subsequent tag increments by one. Therefore, OFF is equal to 0, ON is equal to 1, UNKNOWN is equal to 2, and TESTING is equal to 3.

In this example, ON is assigned a specific value in the declaration.

```
enum OperationalState {OFF, ON = 4, UNKNOWN, TESTING};
```

OFF is equal to 0, and ON is equal to 4. Since UNKNOWN and TESTING have not been declared with a specific value, they increment by 1 from the value of the preceding tag. Therefore, UNKNOWN is equal to 5, and TESTING is equal to 6.

Enumerations can also be defined outside of an interface, in the global name space. However, as a best practice you should avoid global enumerations. You can declare an abstract interface and then define all of your enumerations in that one interface.

Once you have an enumeration defined, you can use the enumeration as a data type in other declarations, such as attributes and events. For example, the enumeration state_e is declared in the MyEquip class. The attribute OperationalState is then declared with the state_e data type, and initialized to one of the tags in the state_e enumeration.

```
interface MyEquip : ICIM_ManagedElement
{
   enum state_e { OK, MALFUNCTION, UNKNOWN };
   attribute state_e OperationalState = UNKNOWN;
   event malfunctioning
       = OperationalState == MALFUNCTION;
}
```

You can also reference enumerations that are defined in other classes. If the class containing the enumeration is in another model file, then the model file with the enumeration must be referenced in your model file with the *#include* and *#pragma include_h* statements.

An enumeration can be used as a data type in other interface declarations, using the syntax *‹interface_name›*::*‹enumeration_name›.*

An enumeration tag can also be declared in one interface and referenced in another interface declaration, using the syntax *‹interface_name›*::*‹tag_name›*.

In this example, the class OtherEquipment is defining an attribute that will use the enumeration defined in the MyEquip class. In addition, the UNKNOWN and MALFUNCTION tags are also referenced.

```
interface MyEquip : ICIM_ManagedElement
{
   enum state_e { OK, MALFUNCTION, UNKNOWN };
   attribute state_e OperationalState = UNKNOWN;
   event malfunctioning
       = OperationalState == MALFUNCTION;
}

interface OtherEquipment : ICIM_ManagedElement
{
   attribute MyEquip::state_e OperState = MyEquip::UNKNOWN;
   event Down
       = OperState == MyEquip::MALFUNCTION;
}
```

Structures

Structures can be defined within an interface or outside of an interface, in the global name space. Within a namespace (interface or global), the structure name must be unique. However, as a best practice you should avoid global structures.

Each structure defines a namespace for their fields. Therefore, two or more structures can have the same field names.

The format for declaring a structure is as follows:

```
<struct>            ::=    struct
                           <struct_name>
                           "{"
                           <fields_list>
                           "}"
                           ";"
<fields_list>       ::=    <field>
                           [ <fields_list> ]
<field>             ::=    <field_type>
                           <field_name>
                           ";"
<field_type>        ::=    double
                    |      float
                    |      [unsigned] int
                    |      [unsigned] long
                    |      [unsigned] short
                    |      boolean
                    |      binary
                    |      byte
                    |      string
                    |      <enum_type_name>
```

In this example, the structure AuditTrailEntry is declared. This structure contains five fields.

```
struct AuditTrailEntry
{
        unsigned SerialNumber;
        unsigned Timestamp;
        string   User;
        string   ActionType;
        string   Text;
};
```

You can declare an attribute with a struct type:

```
attribute demoStruct AuditTrailEntry;
```

When you get the values, you use a dotted notation on the attribute name and the field name.

You can also use a structure when declaring a table attribute. In this example, each row in the AuditTrail table consists of the five fields of the AuditTrailEntry structure. In addition, the SerialNumber field is used as the unique key in the table.

```
table AuditTrail AuditTrailEntry
   descending unique key SerialNumber
   "The audit trail for the event.";
```

CHAPTER 5
Declaring an Interface

This chapter consists of the following sections:

Overview

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.

Forward declaration

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

A forward declaration declares the name of an interface and indicates that its definition appears later in the MODEL file. 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> ]
                                "}"
                                [ ; ]
<opt_implementation>    ::=     implementation
                                ":"
                                <user-entered text>
```

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

The names of each of these properties, such as an attribute, problem, relationship and so on, must be unique within an interface. For example, if you have an attribute named "Target1", then you cannot name any other property of this class "Target1".

The optional ‹*opt_implementation*› parameter defines text that is copied verbatim to the generated C++ code when the MODEL is compiled. In this example, the text “void init ()” would be copied into the C++ file that is generated from the .mdl file.

```
interface MyClass1: TopClass
{
    refine FileName
    = "mymod/layout.xml";

    implementation:
            void init();
};
```

IMPORTANT

Do not call a Java implemented construct from within the C++ init() class.

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 can be directly instantiated must define implementations for all of their attributes. Classes that are declared as abstract and cannot be directly instantiated 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. 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.

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.

```
interface NetworkAdapter;     /* Forward Declaration */

abstract interface ComputerSystem : MR_ManagedObject
{
}

interface Router : ComputerSystem
{
}
```

CHAPTER 6
Attribute Declarations

This chapter consists of the following sections:

Overview

Attributes describe the state of a managed object. Their values can be queried and manipulated by EMC Smarts clients and displayed in a 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* (that is, accessed using an accessor), 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 EMC Smarts 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 an accessor. 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. “When the value of an expression is unavailable” on page 124 provides additional 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 5 on page 65 lists the identity value of an empty relationshipset and the corresponding aggregate operator.

Table 5 Identity result for aggregate operators

Aggregate Operator	Identity Value
and	True
or	False
max	Lowest possible value
min	Highest possible value
sum	0
prod	1
avg	Error

The following example interface declares one propagated attribute for each aggregate operator. The value of each attribute propagates from a related class through a relationshipset. This example explains the identity values listed in Table 5 on page 65.

```
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 EMC Smarts client. A typical use of a stored attribute is to provide a name for an object as a string.

```
<attribute_dcl>      ::=     [ internal ]
                             [ static ][ readonly ]
                             [ stored ]
                             attribute
                             [ external ]
                             <attribute_type>
                             [ <value_range> ]
                             <attribute_name>
                             [ required ]
                             [ timestamped ]
                             [ <attribute_description> ]
                             [ "=" <initial_value> ]
                             ";"
                        |
                             refine
                             stored
                             <attribute_name>
                             [ required ]
                             [ timestamped ]
                             [ <attribute_description> ]
                             [ "=" <initial_value> ]
                             ";"
<value_range>        ::=     "["
                             <integer_literal>
                             " "
                             ".."
                             " "
                             <integer_literal>
                             "]"
```

The **internal** keyword indicates that this attribute is only visible within the generated C++ code. The attribute is not visible through an external interface, and therefore it cannot be viewed in a console.

Note: A computed attribute that is not marked as internal can use the internal attribute in an expression, thus making the internal attribute's value visible to an external interface.

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. In this example, the value of the attribute is set to 865.0 and cannot be changed.

```
readonly attribute float LowOxygenPressureThreshold
        "Minimum operational oxygen pressure (psi)."
        = 865.0;
```

Note: A readonly attribute must be given an initial value.

The keyword **attribute** identifies this as an attribute declaration. The keyword **stored** declares the attribute's access type. The **stored** keyword is optional because if you declare an attribute without a keyword specifying the access type, a stored attribute is created, by default.

The keyword **external** indicates whether the literal text specified as the initial value of a **string** attribute will be externalized. The external keyword implies both static and readonly. When including the **external** keyword in a declaration, do not include the **static** and **readonly** keywords. In this example, the ‹*initial value*›, Jim Smith, will be externalized.

```
interface Pitcher : MR_ManagedObject
   "The man who throws the baseball"
   {
      attribute external string Name
      "The name of the pitcher" = "Jim SMith";
   }
```

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 55.

The **required** keyword indicates that the value of the attribute must be initialized when an instance of the class is created. Therefore, the creation and initialization must be done using a transaction. Transactions can be written in ASL or using one of the remote APIs (Java, Perl or C). To set values it must be a write-transaction (read only transactions cannot create instances or assign values). This example shows the basic format of a write-transaction in ASL:

```
tx = transaction(WRITE_LOCK);
create instance
set instance's required attribute(s)' s values.
tx->commit();
```

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 121. 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.

Note: Attributes that are declared as stored cannot use the **timestamped** keyword with the keyword **readonly**. Only attributes declared as instrumented can use the **timestamped** keyword with the **readonly** keyword, as described in "Instrumented attributes" on page 72.

The optional *‹initial_value›* parameter sets an initial value for the attribute when the object is instantiated. In this example, whenever an instance of the Engine class is created, the Temperature attribute for that instance will have an initial value of 0.

```
interface Engine : ICIM_Managed_Element
{
   stored attribute float Temperature
         "Internal gas temperature (fahrenheit)."
         = 0;
}
```

The optional *‹value_range›* parameter indicates the expected range of the value. Currently, it does not impose any constraints on the variable's value. However, you can use the range to set things, such as the end-points of slider bar. In this example, the MaxResponseTime attribute has a range of 0 to 60 with an initial value of 2.

```
attribute unsigned long [0 .. 60] MaxResponseTime
   "The maximum response time (in seconds). The actual
   response time is compared against this threshold to
   determine if the process is running slow."
   = 2;
```

In this example, the enumeration *state_e* is defined. state_e is then used as the *‹attribute_type›* in the attribute declaration.

```
enum state_e { OK, MALFUNCTION, UNKNOWN };
readonly attribute state_e OperationalState
   "Operational State of this device. Possible states are: "
   " OK, MALFUNCTION, or UNKNOWN "
   = UNKNOWN;
```

You can also define a structure and then use the name of the struct as the *‹attribute_type›* in your attribute declaration.

Refine keyword

The keyword **refine** declares this attribute as a refinement of an attribute declared in a superclass of this class.

The **stored** keyword is required if you are refining the access type of an attribute to the stored access type.

The **stored** keyword is optional if you declared or refined the attribute to be the stored access type in the parent class. If you do not specify the access type in the refine statement, the access type defaults to the access type defined in the previous specification going up the class hierarchy. The previous specification is either a refinement of the attribute that includes the access type keyword or the original attribute declaration.

You can add/modify the description and the initial value of the attribute.

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. "Lexical elements for expressions" on page 112 provides additional information regarding operators, and "Data types" on page 55 describes the supported data types.

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

```
<attribute_dcl>      ::=    [ internal ]
                            [ readonly ]
                            computed
                            attribute
                            <attribute_type>
                            [ <value_range> ]
                            <attribute_name>
                            [ required ]
                            [ timestamped ]
                            [ <attribute_description> ]
                            ["=" <expression>]
                            ";"
                       |
                            refine
                            computed
                            <attribute_name>
                            [ required ]
                            [ timestamped ]
                            [ <attribute_description> ]
                            "=" <expression>
                            ";"
<value_range>        ::=    "["
                            <integer_literal>
                            " "
                            ".."
                            " "
                            <integer_literal>
                            "]"
```

The **internal** keyword indicates that this attribute is only visible within the generated C++ code. The attribute is not visible through an external interface, and therefore it cannot be viewed in a console.

The optional keyword **readonly** prevents write access to an attribute's value. The default behavior of a computed attribute is readonly, even if the **readonly** keyword is omitted. However, when you refine a computed attribute the **readonly** flag will only be applied if you specifically set the **readonly** keyword in the original computed attribute. The refined computed attribute inherits the **readonly** flag from the computed attribute in the parent class.

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

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 55.

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. This expression must be a MODEL expression; it cannot be procedural code. "Syntax for expressions" on page 122 describes the syntax of expressions.

This example computes a value for the attribute CurrentUtilization.

```
computed attribute float CurrentUtilization
   = ((TotalRate * 8) / (MaxSpeed + 0.001)) * 100;
```

Refine keyword

The keyword **refine** declares this attribute as a refinement of an attribute declared in a superclass of this class.

The **computed** keyword is required if you are refining the access type of an attribute to the computed access type.

The **computed** keyword is optional if you declared or refined the attribute to be the computed access type in the parent class. If you do not specify the access type in the refine statement, the access type defaults to the access type defined in the previous specification going up the class hierarchy. The previous specification is either a refinement of the attribute that includes the access type keyword or the original attribute declaration.

When you refine a computed attribute the **readonly** flag will only be applied if you specifically set the **readonly** keyword in the original computed attribute. The refined computed attribute inherits the **readonly** flag from the computed attribute in the parent class.

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 verifies 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 6 on page 72 lists the common MIB types, the corresponding SNMP type, and the corresponding MODEL attribute type.

Table 6 MIB, SNMP, and MODEL Types

MIB Type	SNMP Type	MODEL Attribute Type
DisplayString	OCTET STRING	string
IpAddress	OCTET STRING	string - Requires the #pragma DotNotation. “#pragma DotNotation” on page 139 provides additional information.
Counter	INTEGER	unsigned int - Requires the #pragma WrapCounter. “Pragmas used with SNMP instrumentation” on page 139 provides additional information.
Gauge	INTEGER	unsigned int
TimeTicks	INTEGER	unsigned int
Integer32	INTEGER	int
Counter32	INTEGER	unsigned int - Requires the #pragma WrapCounter. “Pragmas used with SNMP instrumentation” on page 139 provides additional information.
Gauge32	INTEGER	unsigned int
Unsigned32	INTEGER	unsigned int
OID	OBJECT IDENTIFIER	string - Requires the #pragma Object ID. “#pragma ObjectID” on page 139 provides additional information.
Counter64	COUNTER64	unsigned long

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 133.

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

- It must contain attributes declared as instrumented. Furthermore, all of the instrumented attributes of a given instance 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. For SNMP instrumentation, the instrumentation declaration maps the instrumented attributes of a class to OIDs. "Instrument Declarations" on page 131 describes the syntax of an instrumentation declaration.

The syntax of an instrumented attribute declaration is shown below.

```
<attribute_dcl>      ::=    [ internal ]
                            [ readonly ]
                            instrumented
                            attribute
                            <attribute_type>
                            [ <value_range> ]
                            <attribute_name>
                            [ <attribute_description> ]
                            ";"
                     |
                            refine
                            instrumented
                            <attribute_name>
                            [ <optional_description> ]
                            ";"
<value_range>        ::=    "["
                            <integer_literal>
                            " "
                            ".."
                            " "
                            <integer_literal>
                            "]"
```

The **internal** keyword indicates that this attribute is only visible within the generated C++ code. The attribute is not visible through an external interface, and therefore it cannot be viewed in a console.

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

The *‹attribute_type›* parameter specifies an attribute's data type. An instrumented attribute can be any one of the supported types listed in "Data types" on page 55.

Note: SNMP does not support enumeration types. They may appear in a MIB as a list, however, they come across the network as integers.

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. "Timestamp" on page 121 provides additional information.

Refine keyword

The keyword **refine** indicates that this declaration is a refinement of an attribute in a related class.

The **instrumented** keyword is required if you are refining the access type of an attribute to the instrumented access type.

The **instrumented** keyword is optional if you declared or refined the attribute to be the instrumented access type in the parent class. If you do not specify the access type in the refine statement, the access type defaults to the access type defined in the previous specification going up the class hierarchy. The previous specification is either a refinement of the attribute that includes the access type keyword or the original attribute declaration.

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 131.

```
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.

"MODEL Pragmas" on page 135 describes the available MODEL pragmas. "Built-in functions" on page 117 describes the rate and delta operators.

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 the 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>       ::=   [ internal ]
                            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 **internal** keyword indicates that this attribute is only visible within the generated C++ code. The attribute is not visible through an external interface, and therefore it cannot be viewed in a console.

The keyword **attribute** identifies this as 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, then the assumption is that the attribute has the same name in the related class.

Refining an existing propagated attribute

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.

Refining an attribute to be propagated

When you refine the access type of a stored or 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, a warning message is issued when the MODEL is compiled. "#pragma Uses Propagation" on page 138 provides additional information regarding this pragma.

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 EMC Smarts 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. "Instrumented attributes" on page 72 describes the technique for accessing and storing data from an SNMP table.

A table is a collection of rows. 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> ]
                            ";"
<row_type>           ::=    double
                     |      float
                     |      [unsigned] int
                     |      [unsigned] long
```

```
                        |       [unsigned] short
                        |       byte
                        |       binary
                        |       string
                        |       enum
                        |       struct
<table_key>             ::=     [ ascending | descending ]
                                [ unique ]
                                key
                                <key_name>
```

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

The *‹row_type›* parameter specifies the type of table row. For example, this declaration defines the SupportedRates table with a *‹row_type›* of string.

```
table SupportedRates string
```

You can also declare a structure and then use the structure as the ‹row_type› in a table declaration. In this example, the structure circuit_alarms is declared in the Circuit class. The table AlarmTable is then declared in the MyNet class, using the circuit_alarms structure as its row type.

```
Interface Circuit : LogicalConnection
{
struct circuit_alarms
  {
   string tableKey;
   string Name;
   int End;
   unsigned receivedAt;
   int Alarm;
  };
}
Interface MyNet
{
table AlarmTable Circuit::circuit_alarms unique key tableKey;
}
```

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. In the example above, the *‹row_type›* is the structure circuit_alarms, and the key is the field tableKey.

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.

CHAPTER 7
Relationship Declarations

This chapter consists of the following sections:

Overview

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. This relationship pair is referred to as a *relation*.

Cardinality

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.

Declaring a relationship

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>       ::=    [ internal ]
                            [ readonly ]
                            [ <access_type> ]
                            <cardinality>
                            <relationship_name>
                            ","
                            <class_name>
                            [ "," <inverse_name> ]
                            [ <optional_key_spec> ]
                            [required]
                            [ <relationship_description> ]
                            [ "=" <expression> ]
                            ";"
                     |      refine
                            [ <access_type> ]
                            <relationship_name>
                            [ "," <class_name> ]
                            [ required ]
                            <relationship_description>
                            ";"
<access_type>        ::=    stored
                     |      computed
<cardinality>        ::=    relationship
                     |      relationshipset
<optional_key_spec>  ::=    [ ascending | descending ]
                            [ unique ]
                            key
                            <key_name>
```

The **internal** keyword indicates that this relationship is only visible within the interface (class) in which it is declared. The relationship or relationshipset is not visible through an external interface, and therefore it cannot be viewed in a console.

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 instance or many class instances. The keyword relationship specifies a connection to, at most, one class instance; the keyword relationshipset specifies a connection to any number of class instances.

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 *‹optional_key_spec›* parameter is only provided for relationshipsets. The *‹key_name›* parameter specifies the name of an attribute in the class. The attribute name must have a separate declaration in the class.

```
readonly attribute string KeyStr;

stored relationshipset
        MemberOf, FurnitureStore, Members
        descending key KeyStr;
```

The **required** keyword indicates that the relationship must not be empty. If a class contains a required relationship, whenever an instance of the class is created, the relationship must be populated.

The following example declares a relation between engines and a space craft module. Each engine only provides thrust to maneuver one space craft module: a to-one relationship. However, each space craft module can be maneuvered by one or more engines: a to-many relationship.

```
interface Engine : ICIM_ManagedSystemElement
{
   relationship Maneuvers, SpaceCraftModule, ManeuveredBy;
}
interface SpaceCraftModule
{
   relationshipset ManueveredBy, Engine, Maneuvers;
}
```

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.

CHAPTER 8
Declaring Event-Driven Behavior in MODEL

This chapter consists of the following sections:

Overview

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 event-driven behavior to your model, remember three important points:

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

MODEL declarations for defining event-driven behavior

MODEL provides a number of declarations for defining event-driven behavior in a managed system. Each of these has a particular purpose, as described below.

- An object created with the event declaration is referred to as a *basic event*. Repository objects created with event, problem, aggregate, symptom, propagate symptom, and propagate aggregate declarations are all referred to generically as *events*.
- An event declaration defines a basic 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, as described in "Event declaration" on page 85.
- A problem declaration is defined by the ‹*symptom-list*› parameter (lists the basic events, symptoms, propagated symptoms and other problems that it can cause). You can specify a probability for each member in the ‹*symptom-list*› because the causal relationship between the problem and its symptoms may be probabilistic. "Problem declaration" on page 87 describes how to declare a problem in MODEL.
- A symptom declaration groups a set of basic events, symptoms, propagated symptoms, and other problems that it can cause into a single event by conjunction. The event defined using the symptom declaration is then used as a member of the ‹*symptom-list*› parameter in a problem declaration. "Symptom declaration" on page 92 describes how to declare a symptom in MODEL.
- 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 one of the members of the ‹*symptom-list*› parameter in a problem declaration. "Propagate symptom declaration" on page 95 describes how to declare a propagate symptom in MODEL.

- An aggregate declaration groups one or more basic events, problems, aggregates, or propagated aggregates declared for the same class into a single abstract event by disjunction. “Aggregate declaration” on page 96 describes how to declare an aggregate.
- 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. “Propagate aggregate declaration” on page 98 describes how to declare a propagate aggregate.

The table, “Types of events that can be exported” on page 100, lists the types of events that may be exported to EMC Smarts clients that request to be notified when the event occurs.

Event declaration

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

Basic events declared by an event declaration are intended to be used as a member of the *‹symptom-list›* parameter in problem declarations. The Domain Manager creates a data structure called a codebook that lists the problems to which EMC Smarts clients have subscribed. The codebook also includes the basic events, specified by event declarations, that each problem can cause. These basic 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> ]
                               <event_expression>
<event_guard>            ::=   if
                               <expression>
                               check
```

The *‹opt_loss›* parameter, indicated by the keyword **loss**, specifies the probability that this observable basic 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 basic event 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 basic 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 basic event and, optionally, whether the event is guarded.

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

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 122. There is one notable constraint on event expressions: they must return a Boolean value.

When the basic 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 a basic 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 basic 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 basic event is recognized.

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.

In the following example, LowInboundUtilization is a guarded event. The expression (CurrentInboundUtilization ‹ LowInboundThreshold) will not be evaluated unless ifOperStatus == 1. When ifOperStatus does not equal 1, then the two computed attributes will not be evaluated. This also saves on polling because the computed attributes are based on instrumented attributes.

```
stored attribute int LowInThreshold
   “Threshold for incoming traffic ” = 5;

computed attribute float ifInOctetsRate
   "Rate that interface is receiving octets."
          = rate(ifInOctets, PollingInterval);

computed attribute float CurrentInUtilization
   "The total number of input bits transmitted to this
   interface in seconds expressed as a “
   “ percentage of total bandwidth. "
   = ((ifInOctetsRate * 8) /  (ifSpeed + 0.01)) * 100;

event LowInboundUtilization
   "Traffic flowing into this interface is below the
   utilization threshold. "
   = if ifOperStatus = = 1
       check (CurrentInUtilization < LowInThreshold);
```

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 EMC Smarts 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> ]
                         [<apriori>]
                         [ <symptoms_list> ]
                         ";"
                   |     refine
                         problem
                         <problem_name>
                         [ <problem_description> ]
                         [<apriori>]
                         <symptoms_list>
                         ";"
<apriori>          ::=   apriori"(" <expression> ")"
<symptoms_list>    ::=   "=>"
                         <symptoms>
<symptoms>         ::=   <symptom>
                         ["," <symptoms>]
<symptom>          ::=   <problem_name> | <event_name> |
                               <propagated_symptom_name> |
                               <symptom_name>
                         [<probability> | explains ]
                         [<condition>]
<condition>        ::=   if
                         <expression>
```

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

The *‹symptoms_list›* parameter lists the basic events, problems, symptoms and propagated symptoms that this problem causes. Each member in the *‹symptoms_list›* can include optional parameters:

- A *‹probability›* parameter or **explains** keyword
- A *‹condition›* parameter

The *‹probability›* parameter 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 item in its *‹symptoms_list›*.

apriori keyword

The **apriori** keyword indicates the probability of the problem occurring before looking at any of the symptoms as evidence. The apriori value must be a floating point number greater than or equal to zero but less than or equal to one, and it can be an attribute that varies in value.

Setting the apriori to 0.0 removes the problem from the codebook. Setting the apriori to 1.0 indicates that the problem is always active, as long as there is one active symptom, and any required symptoms are active. A symptom is required when its *‹probability›* parameter is set to 1.0 and the *‹opt_loss›* parameter for the underlying basic event(s) that comprise the member is set to 0.0. Symptoms that are declared as required must be active for the problem to be active.

If the **apriori** keyword is not included in the problem declaration, an apriori of 0.05 is applied.

For example, a light bulb can be either on or off. The light bulb can be off because the switch controlling the light is turned off or because of faulty wiring. The following classes, events, relationships, and problems are defined to describe this behavior:

```
interface Bulb {

   event LightOff = LightStatus == FALSE;

   problem Out apriori(0.01) =>
      LightOff;

   relationship ControlledBy, Switch, Controls;

   relationship SuppliedBy, Wiring, Supplies;
}

interface Switch {

   relationship Controls, Bulb, ControlledBy;

   propagate symptom LightOff =>
      Bulb,
      Controls;

   problem Off apriori(0.5) =>
      LightOff;
}

interface Wiring {

   relationship Supplies, Bulb, SuppliedBy;

   propagate symptom LightOff =>
      Bulb,
      Supplies;

   problem Bad apriori(0.000001) =>
      LightOff;
}
```

The **apriori** values indicate that the probability that the switch is turned off is much higher than the probability that the wiring is faulty. For the switch, the apriori is set to 0.5 because it is off about half the time.

The apriori value can also be an attribute:

```
interface Bulb {

   event LightOff = LightStatus == FALSE;

   problem Out apriori(out_apriori) =>
      LightOff;

   attribute float out_apriori = (outPossible ? 0.01 : 0.0);

   attribute boolean outPossible = TRUE;
}
```

explains keyword

When determining the most likely problem set that causes the currently observed symptoms, a problem becomes more unlikely as it has more and more missing symptoms. Because of that, you want to model the important causal symptoms, and not all of the consequential symptoms.

The causal symptoms are those symptoms that, when active, indicate that the problem is active. These symptoms are part of the codebook and play a role in correlation. This is the causality and these symptoms are considered when determining whether the problem is the root-cause.

However, you still want the system to relate the problem to the consequential symptoms and the **explains** keyword in the problem declaration is used to provide this linkage. The **explains** keyword indicates that the problem can explain (impact) the associated symptom, but that this explained symptom is not part of the codebook and is not used when determining whether the problem is active or the root-cause.

For example, an Unresponsive problem is declared:

```
problem Unresponsive  =>
        UnresponsiveActivatorEvent,
        AgentUnresponsive explains,
        HostedServicesImpact explains;

export UnresponsiveActivatorEvent, AgentUnresponsive;

export HostedServicesImpact;
```

In this case, only the UnresponsiveActivatorEvent is added to the codebook as one of the causal symptoms of the Unresponsive problem.

The AgentUnresponsive and HostServicesImpact events have the **explains** keyword after them. This indicates that the Unresponsive problem explains (impacts) the AgentUnresponsive and HostServicesImpact events, but these two events are not added to the codebook or considered when determining whether the Unresponsive problem is active. However, the AgentUnresponsive and HostServicesImpact events will be displayed on the Impacts tab of the Unresponsive notification in the Global Console.

You can also have a symptom in a problem be both a causal symptom and an explained symptom. In this example, the AgentUnresponsive event is added to the codebook as one of the core symptoms. However, it is also linked to the problem as a consequential symptom.

```
problem Unresponsive  =>
      UnresponsiveActivatorEvent,
      AgentUnresponsive,
      AgentUnresponsive explains,
      HostedServicesImpact explains;

export UnresponsiveActivatorEvent, AgentUnresponsive;

export HostedServicesImpact;
```

Whenever the problem Unresponsive is active, if the AgentUnresponsive and HostedServicesImpact events are active, they will appear under the Impacts tab of the Notification Properties dialog for the Unresponsive problem.

The UnresponsiveActivatorEvent problem will not appear under the Impacts tab, but it is added to the codebook as one of the causal symptoms of the Unresponsive problem.

Symptom declaration

A symptom declaration groups a set of basic events, problems, symptoms, and propagated symptoms under a single name to be used in a problem declaration. This is especially useful if the same set of basic events, problems, symptoms, and propagated symptoms are referenced in multiple problem declarations.

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 by subscription to EMC Smarts 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>         ::=   <problem_name> | <event_name> |
                            <propagated_symptom_name> |
                            <symptom_name>
                        [<probability> | explains ]
                        [ <condition> ]
<condition>       ::=   if
                        <expression>
```

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

The *‹symptoms_list›* parameter lists the basic events, problems, symptoms and propagated symptoms that this symptom declaration causes. Each member of the *‹symptoms_list›* can include optional parameters:

- A *‹probability›* parameter or **explains** keyword
- A *‹condition›* parameter

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. "Syntax for expressions" on page 122 provides information about the correct syntax for an expression.

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.

explains keyword

The **explains** keyword is used for display purposes and does not affect the analysis. It indicates that this member of the *‹symptoms_list›* is explained by (impacted by) the symptom being declared. When this member of the *‹symptoms_list›* is exported and active, it is displayed in the Global Console under the Impacts tab of the Notification Properties dialog for the problem that references this symptom declaration. However, it is not added to the codebook as one of the symptoms of the problem that references the symptom declaration, and therefore, is not considered when determining whether the problem is a root cause. Table 7 on page 100 provides a list of which types of events can be exported.

For example, a NoFuelSupply problem and a NoFuelSupplySymptom symptom are declared:

```
problem NoFuelSupply =>
        NoFuelSupplySymptom;

symptom NoFuelSupplySymptom =>
      FuelTanksEmpty,
      EngineDown explains;

event FuelTanksEmpty = FuelTanksLevel < 0.01 == FALSE;

event EngineDown == FALSE;

export FuelTanksEmpty, EngineDown, NoFuelSupply;
```

In this case, the FuelTanksEmpty event is added to the codebook as one of the symptoms of the NoFuelSupply problem.

The EngineDown basic event has the **explains** keyword after it. This indicates that the NoFuelSupply problem explains (impacts) the EngineDown basic event, and this event will be displayed on the Impacts tab of the NoFuelSupply notification in the Global Console. However, the EngineDown event is not considered when determining whether the NoFuelSupply problem is a root cause. In this example, only the FuelTanksEmpty event is added to the codebook and considered when determining a root cause.

You can also have a member of the *‹symptoms_list›* be both caused and explained by this symptom.

```
problem NoFuelSupply =>
        NoFuelSupplySymptom;

symptom NoFuelSupplySymptom =>
      FuelTanksEmpty,
      EngineDown,
      EngineDown explains;

event FuelTanksEmpty = FuelTanksLevel < 0.01 == FALSE;

event EngineDown == FALSE;

export FuelTanksEmpty, EngineDown, NoFuelSupply;
```

In this example, both the FuelTanksEmpty and EngineDown events are added to the codebook as symptoms of the NoFuelSupply problem. But the EngineDown event is also included with the **explains** keyword after it. In this case, if the NoFuelSupply problem is active and the EngineDown event is active, the EngineDown event will be displayed on the Impacts tab of the NoFuelSupply notification in the Global Console.

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 basic 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 from the basic 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.

An event created with a propagate symptom declaration cannot be exported and is not available by subscription to EMC Smarts clients.

The following example creates two classes: one class representing a light bulb, and a second class that represents a light switch. When the light switch is turned off, the light bulb is off.

The Bulb class has a basic event, LightOff, that is active when the light bulb is off. The Switch class propagates the value of the LightOff basic event from the Bulb Class, over the Controls relationship. The LightOff propagated symptom is then used to determine whether the Off problem is the root cause.

```
interface Bulb {

   event LightOff = LightStatus == FALSE;

   problem Out => LightOff;

   relationship ControlledBy, Switch, Controls;
}

interface Switch {

   relationship Controls, Bulb, ControlledBy;

   propagate symptom LightOff => Bulb, Controls;

   problem Off => LightOff;
}
```

Aggregate declaration

An aggregate declaration groups one or more basic events, problems, aggregates or propagated aggregates 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 basic event, problem, aggregate or propagate aggregate declarations in the same class.

Aggregates are not used in codebook correlation but they can be exported for subscription by other EMC Smarts clients.

```
<aggregate_dcl>  ::=    aggregate
                        <aggregate_name>
                        [ <aggregate_description> ]
                        [ "<=" <events_list> ]
                        ";"
                 |      refine
                        aggregate
                        <aggregate_name>
                        [ <aggregate_description> ]
                        "<="
                        <events_list>
                        ";"
<events_list>    ::=    <event>
                        [ "," <event_list> ]
<event>          ::=    <event_name> | <problem_name> |
                               <propagated_aggregate_name>
                          |              <aggregate_name>
```

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

The *‹events_list›* parameter lists the basic events, problems, propagated aggregates, and aggregates 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.

The following example declares an aggregate, FuelSupplyCondition, that groups together the different types of events that indicate an error condition with the fuel supply. A problem with the fuel supply is indicated whenever the FuelTanksEmpty basic event or FuelSupplyBlocked problem is active.

```
event FuelTanksEmpty  "The fuel tanks are out of propellant."
        = TotalPropellantLevel <  0.01;

problem FuelSupplyBlocked
   "No propellant can reach the engine group."
   => EnginesDownSymptom;

propagate symptom EnginesDownSymptom
        => Engine, ConsistsOf, EngineDown;

aggregate FuelSupplyCondition
   "Problem with the engine group's fuel supply"
   <= FuelTanksEmpty, FuelSupplyBlocked;
export FuelSupplyBlocked,FuelTanksEmpty,FuelSupplyCondition;
```

Logic for aggregate processing

The value of an aggregate is determined as follows:

- If all underlying events are inactive, then the aggregate is inactive.
- If any underlying event has an error, then this event overrides all inactive events. Therefore, if one underlying event is an error and the rest are inactive, the aggregate is an error (the same error that the underlying event is). If more than one underlying event has an error, then the error of the aggregate will be any one of those. An error value of pending takes precedence over all other error values (in the case of multiple underlying events with errors).
- An active event overrides errors. Therefore, if one underlying event is active, the aggregate is active, regardless of the rest of the underlying events.

Propagate aggregate declaration

A propagate aggregate is an event or group of events that are 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. Depending on how many symptoms of a propagate aggregate occurs, the codebook calculates a certainty (percentage value) that the propagate aggregate has occurred.

The events that comprise a propagate aggregate must be defined as basic event, problem, aggregate or propagated aggregate 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.

Logic for propagate aggregate processing

The value of a propagate aggregate is determined as follows:

- If all underlying events are inactive, then the propagate aggregate is inactive.
- If any underlying event has an error, then this event overrides all inactive events. Therefore, if one underlying event is an error and the rest are inactive, the propagate aggregate is an error (the same error that the underlying event is). If more than one underlying event has an error, then the error of the propagate aggregate will be any one of those. An error value of pending takes precedence over all other error values (in the case of multiple underlying events with errors).
- An active event overrides errors. Therefore, if one underlying event is active, the propagate aggregate is active, regardless of the rest of the underlying events.

Export declaration

An export declaration is required to make the event-driven behavior declared for a class visible outside of the Repository. EMC Smarts clients, such as the Global Console and adapters, can subscribe to exported event-driven behavior and receive notifications when they occur. Table 7 on page 100 describes which types of event-driven behavior can be exported.

```
<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.

The following example exports the FuelTanksEmpty event:

```
event FuelTanksEmpty = FuelTanksLevel < 0.01 == FALSE;

export FuelTanksEmpty;
```

Table 7 on page 100 lists the types of event-driven behavior that can be exported.

Table 7 Types of events that can be exported

Type of event	Can It be exported
Basic Event	Yes
Problem	Yes
Symptom	No
Propagate Symptom	No
Aggregate	Yes
Propagate Aggregate	Yes

Imported events

Imported events allows you to specify events for which the state reflects the state of the event with the same name, on an instance of the same name and class in another Manager. In addition to reflecting the state of the remote event, whenever the connection between the two Managers is severed, the imported event will be suspended, and automatically resumed as soon as the connection is resumed.

There are two steps you need take to ensure the correct operation of imported events.

1. Make sure the interface names for the classes are the same, and that the instance names of the instance you want to import the events are the same.
2. Assign the remote Manager name to the ServiceName attribute in the Manager that will be importing the events.

Specifying imported events in MODEL

The following example shows how to specify imported events in MODEL. Note that the interface names for the classes are the same in both Manager MODEL files, as are the event names that are being imported.

```
interface TestIF : MR_ManagedObject
{
    event e1 "This event is imported " imported;
    export e1;

    event e2 "This one also"  imported;
    export e2;

    event a1 "This event is really an aggregate in the remote domain
   manager"  imported;
    export a1;

    aggregate a2
        "I can use all these events normally in this MODEL"
        = e1, e2, a1;
    export a2;
}
```

The remote Domain Manager MODEL could look like:

```
interface TestIF : MR_ManagedObject
{

    attribute boolean p1 = FALSE;
    event e1 = p1;
    export e1;

    attribute boolean p2 = FALSE;
     event e2 = p2;
    export e2;

    aggregate a1 = e1, e2;
    export a1;
}
```

CHAPTER 9
Operation Declarations

This chapter consists of the following sections:

Overview

Operations provide a method for manipulating objects and their properties. You can retrieve data based on attributes and topology, and you can also Get or Set attribute values (including internal attributes). Operations also allow you to perform computations, traverse relationships, and call other operations.

Operations may be called by client programs, such as:

- dmctl
- ASL scripts
- Java API Client
- PERL API Client

In addition, get operations (readonly) may appear in computed attribute expressions.

Declaring an operation

An operation declaration includes the name of the operation, the type of its arguments, and the type of any returned values.

Note: There is no operation keyword.

```
<operation_dcl>          ::=  [internal]
                              [static]
                              [const | readonly | idempotent]
                              <return_type>
                              <operation_name>
                              "(" [ <arguments> ] ")"
                              [ <operation_description> ]
                              <operation_definition>
                              ";"
                         |    refine
                              [ instrumented | stored ]
                              <operation_name>
                              [ <operation_description> ]
                              <op_definition_refinement>
<return_type>            ::=  void
                         |    error_code
                         |    <value_type>
                         |    set "(" <value_type> ")"
                         |    <class_name>
                         |    set "(" <class_name> ")"
<arguments>              ::=  <argument>
                              [","<arguments>]
<argument>               ::=  <in_argument>
                         |    <inout_argument>
                         |    <out_argument>
```

```
<in_argument>                ::=   in
                                   <argument_type>
                                   <argument_name>
                                   ["="<expression>]
<inout_argument>             ::=   inout
                                   <argument_type>
                                   <argument_name>
<out_argument>               ::=   out
                                   <argument_type>
                                   <argument_name>
<argument_type>              ::=   <value_type>
                             |     set "(" <value_type> ")"
                             |     <class_name>
                             |     set "(" <class_name> ")"
<operation_definition>       ::=   definition: <assignments>
<assignments>                ::=   <assignment>”,”<assignments>
                             |     <assignment>
                             |     <return_expression>
<assignment>                 ::=   <identifier>"="<expression>
<return_expression>          ::=   return <expression>
<op_definition_refinement>   ::=   <operation_definition>
                             |     "=" <expression>
```

The **internal** keyword indicates that this operation is only visible within the generated C++ code. The result of the operation is not visible through an external interface, and therefore it cannot be viewed in a console.

Note: You can view the results of an internal operation in an external interface if you set the value of a computed attribute (not an internal computed attribute) equal to the results of the internal operation.

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

The **const** keyword indicates that the operation does not change the instance in the repository, so it cannot assign values to attributes of the class. The **const** keyword is also used to obtain an instance-only read lock. “Instance-only read lock” on page 110 provides additional information on this type of repository lock.

The **readonly** keyword is used to obtain a repository-wide read lock. It indicates that the operation leaves the repository unchanged. A readonly operation may not assign to attributes of a class. “Repository-wide read lock” on page 109 provides additional information on this type of repository lock.

The **idempotent** keyword indicates that the operation, if called multiple times without an intervening put, will return the same value each call.

The *‹value_type›* parameter can be one of the supported types listed in “Data types” on page 55. The *‹class_name›* parameter can be the name of any class in the repository, such as Engine, Host or Router. Examples are described in “Return_type parameter” on page 106.

The *‹expression›* parameter has to be an EMC Smarts expression; it cannot be procedural code. “Writing Expressions in MODEL” on page 111 provides additional information.

Return_type parameter

The *‹return_type›* parameter declares the type of value returned by the operation. The return value can either be one of the supported types listed in “Data types” on page 55 or it can be a class type.

The following example defines the getCelsiusTemperature operation, which has a ‹*return_type*› of float.

```
readonly float getCelsiusTemperature()
   definition:
   // Convert from Fahrenheit to Celsius
   return (Temperature - 32) / 1.8;
```

This operation would be called on a class instance that contained the attribute Temperature. This operation gets the value of the local attribute Temperature (in degrees Fahrenheit) and returns the equivalent celsius temperature.

If the operation has no return value, use the keyword **void**. In this example, the operation doesn’t return any value, but it does set the Temperature attribute of the instance to the results of the expression .

```
void setCelsiusTemp (in float CTemp)
    definition:
// Convert from input Celsius to Fahrenheit
   Temperature =  CTemp *  1.8 + 32;
```

An operation can also return a set of values or class objects. If this is the case, the keyword **set** should precede the ‹*value_type*› or ‹*class_name*› parameter. Sets are supported for every data type in Table 4 on page 55, except boolean and enumerations.

For example, an Engine class has a ManeuveredBy relationshipset to a ServiceModule class (each ServiceModule can maneuver one or more Engines). In this example, the getEngines operation returns the set of Engine instances that share the ManeuveredBy relationship with this instance of ServiceModule class.

```
readonly set(Engine) getEngines()
   definition:
   return Engine(ManeuveredBy);
```

Refine keyword

The keyword **refine** declares this operation as a refinement of an operation declared in a superclass of this class.

You can change the access to instrumented using the **instrumented** keyword or to non-instrumented using the **stored** keyword.

If you do not specify the access type in the **refine** statement, the access type of the operation defaults to the access type defined in the original declaration of this operation (the first definition going up the hierarchy class, in case the operation is successively refined.)

You can provide multiple assignments and/or a return expression by specifying an *‹operation_definition›* for the *‹op_definition_refinement›*.

If you specify an *‹expression›* for the *‹op_definition_refinement›* all assignments are removed and the return expression is replaced with the expression that has been provided.

Arguments

An operation declaration takes a list of arguments. Each argument requires one of the following keywords:

- in
- inout
- out

These keywords indicate the direction in which information passes between the caller and the operation.

The *‹argument_type›* parameter specifies the argument's type, The argument's type can either be one of the supported types listed in "Data types" on page 55 or it can be a class type. You can declare a set expression with the ‹*argument_type*› parameter, in which case the keyword **set** precedes the ‹*value_type*›. Sets are supported for every data type in Table 4 on page 55, except boolean and enumerations. The syntax for set expressions is the same as that for logical or arithmetic expressions. However, a special group of operators must be used. "Syntax for expressions" on page 122 describes the syntax of expressions, and "Operators for set expressions" on page 116 describes set operators.

in keyword

The **in** keyword indicates that the information passes from the caller to the operation. 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 122. If you declare an argument with a default value, any successive arguments must also have a default value.

In this example, the operation setCelsiusTemp is passed the value of the variable CTemp (whose data type is float). This operation doesn't return any value, but it does set the Temperature attribute of the instance to the results of the expression (CTemp * 1.8 + 32).

```
void setCelsiusTemp (in float CTemp)
    definition:
    // Convert from input Celsius to Fahrenheit
   Temperature =  CTemp *  1.8 + 32;
```

inout keyword

The **inout** keyword indicates that the parameter has an initial value that is passed into the operation, but the value of the argument is then set by the operation and returned to the caller. The value of an inout parameter may be assigned to in the operation definition. You cannot define a default value for an inout argument.

In this example, an invokeDrivers operation is defined. It has two in parameters: the name of the element and the type of the element. It also includes an inout parameter: the object name.

```
#pragma Unlocked
    boolean invokeDrivers(in string elementName,
                                in string elementType,
                                inout string objectName);
```

Note: At this time, the remote API does not support the use of inout parameters.

out keyword

The **out** keyword indicates that the value of the argument is set by the operation and returned to the caller. You cannot define a default value for an out argument.

In this example, a findElement operation is defined. It has two in parameters: the name of the element and the type of the element. It also includes one out parameter, and the operation sets the value of this parameter.

```
boolean findElement(in string elementName,
             in string elementType,
             out MR_Object element);
```

Note: At this time, the remote API does not support the use of out parameters.

Assignment and return_expression parameters

For ‹*assignment*›, the ‹*identifier*› is an attribute name with ‹*expression*› described in “Syntax for expressions” on page 122, where the result of ‹*expression*› is assigned to ‹*identifier*›.

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

Repository locking states

An operation in a class instance may use one of five locking states with the repository:

- “No locking”
- “Repository-wide write lock” (the default)
- “Repository-wide read lock”
- “Instance-only write lock”
- “Instance-only read lock”

No locking

Using #pragma Unlocked before your operation results in no repository lock at all. You can lock the repository in your own code. However, even if you do your own locking, if you delete the object just as the operation or attribute is being dispatched to your code, you may get a crash.

Repository-wide write lock

A repository-wide write lock is the default lock obtained when invoking an operation.

Repository-wide read lock

The **readonly** keyword is used to obtain a repository-wide read lock.

In this example, an Engine class has a ManeuveredBy relationshipset to a ServiceModule class (each ServiceModule can maneuver one or more Engines). The getEngines operation returns the set of Engine instances that share the ManeuveredBy relationship with this instance of ServiceModule class.

```
readonly set(Engine) getEngines()
   definition:
   return Engine(ManeuveredBy);
```

The getEngines() operation doesn’t change the values of any attributes, but it does need to read all of the Engine instances that have a ManeuveredBy relationship to the instance that is calling the operation. Therefore, the **readonly** keyword is used to obtain a repository-wide read lock.

Instance-only write lock

The #pragma Local Operation locks only the object on which the operation is invoked. "#pragma Local Operation" on page 138 provides an example and further discussion.

Instance-only read lock

There are two different methods to obtain a read lock on only the instance that is invoking the operation.

The **readonly** keyword, when used in conjunction with #pragma Local Operation, locks (for reading) only the object on which the operation is invoked.

The **const** keyword, when used in conjunction with the #pragma Local Operation, also obtains a read lock on only the object on which the operation is invoked.

CHAPTER 10
Writing Expressions in MODEL

This chapter consists of the following sections:

Overview

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, operations, assignments, return 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 116. 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, as described in "Built-in functions" on page 117.

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 64 bit, floating-point number. When the trailing F is included, it is treated as a 32 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. This technique is used 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, EMC Smarts does not recommend declaring multiple enumerators with the same value.

Operators

MODEL supports the C and C++ operators listed in Table 8 on page 113. 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 117.

Table 8 Operators supported by MODEL (page 1 of 2)

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

Table 8 Operators supported by MODEL (page 2 of 2)

Operator Type	Symbol	Definition
Logic	&& ||	logical and logical or
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

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. At runtime, an error is produced if the selection expression value does not match any of the key expression values.

Keys within a case operator can only be enumeratons or integers.

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
         key UNKNOWN : 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 enumeration:

```
readonly instrumented attribute int ifAdminStatus;

refine computed AdminStatus =
   case (ifAdminStatus) {
      key 1:  UP;
      key 2:  DOWN;
      key 3:  TESTING;
      default:  OTHER;
   } else UNKNOWN;
```

Operators for set expressions

Set expressions use their own group of terms and operators, as shown in Table 9 on page 116. The precedence of set operators is at the same level as the corresponding symbol for logical and arithmetic operators. Table 10 on page 117 provides additional information.

Table 9 Operators for set expressions

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.

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 10 on page 117 lists the exact order of precedence and associativity for operators in MODEL.

Table 10 Precedence and associativity of operators

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

Built-in functions

The following terms are built-in functions supplied by MODEL. They typically appear in expressions, as described in "Syntax for expressions" on page 122. Although the syntax of MODEL permits you to use a function as a term in a complex expression, 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 at runtime by the monitoring system.

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);
};
```

Rate_last

The term rate_last(A, T) is an alternative to rate(A, T) that can be used for polled attributes in cases when rate interval T equals the polling period of A.

The term rate_last(A,T) returns the difference between the current and the previous values of A divided by the time difference between when the two values were polled.

If A was polled early, or if A was polled late on one polling cycle and polled on time on the next polling cycle, the two consecutive samples of A may not cover the rate interval T, in which case rate(A, T) will be calculated over the two polling cycles.

The term rate_last(A, T), always uses two consecutive samples.

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 that 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);
};
```

Previous

The previous(attribute_name) function returns the previous value. Therefore, `previous(A)`, where *A* is an attribute, returns the previous value of attribute *A*.

This can be used for an instrumented attribute to retain the previous value of the attribute when the value cannot be obtained from the instrumentation source.

Is_server_disconnected

The `is_server_disconnected(attribute_name)` function returns TRUE if the error number for the last get of the attribute was `MR_SUBSCRIPTION_SUSPENDED` and the reason for being suspended is that the remote server was disconnected.

```
interface XYZ {

instrumented attribute boolean IsFlapping;
computed attribute boolean X_IsFlapping =
   is_server_disconnected(IsFlapping) ? previous(IsFlapping) :
   isFlapping;
...
}
```

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 operands on either side of the operator distinguish between set expressions and mathematical or logical expressions.

```
<expression_syntax>  ::=    <term>
                     |      <term>
                            <binary_operator>
                            <expression>
<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 113. 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 112.

The *‹function_name›* and *‹arg_list›* parameters refer to operations defined in the MODEL code and “Built-in functions” on page 117.

The vertical bars around an expression, | *‹expression›* |, return a count of the number of members in a set. “Operators for set expressions” on page 116 describes the list of set operators.

The *‹identifier›* . *‹identifier›* parameters select a field in a previously declared struct.

The **self** keyword references the current instance in the expression.

Examples of set operators

You can perform operations on sets to evaluate and compare their contents. This section provides examples for the operators listed in Table 9 on page 116.

Count

The use of count is illustrated in the declaration of nPath, which is the number of members in Path.

```
computed attribute int nPath = |Path|;
```

Subset

The subclasses Target_Layer2a and Target_Layer2b and the computed relationshipsets isLayer2a and isLayer2b use the subset operator to get their values.

```
computed relationshipset isLayer2a, Target_Layer2a
   = Target_Layer2a(Path);

computed relationshipset isLayer2b, Target_Layer2b
   = Target_Layer2b(Path);
```

Intersection

The computed relationship SetIntersection uses the intersection operator to produce the intersection of the computed relationshipset isLayer2a and the relationship AltPath.

```
computed relationship SetIntersection, Target_Layer2a
   = AltPath & isLayer2a;
```

Union

The computed relationship SetUnion uses the union operator to produce the union of the computed relationshipset isLayer2a and the relationship AltPath.

```
computed relationship SetUnion, Target_Layer1
   = AltPath | isLayer2a;
```

Difference

The computed relationship SetDifference uses the difference operator to produce a set that includes the members of isLayer2a that are not members of AltPath.

```
computed relationship SetDifference, Target_Layer2a
   = isLayer2a - AltPath;
```

Unique

The UniqueInt declaration uses the unique operator to return a set of values from the vInt attribute without any duplicate values.

```
set(int) UniqueInt
   definition : return unique(Path->vInt);
```

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 11 on page 124 shows all possible outcomes for event expression E1 when the value for one of its attributes, A1 or A2, is known.

Table 11 Truth table for event E1 = A1 && A2

	Attribute A1		
Attribute A2	**True**	**False**	**Unavailable**
True	True	False	Unavailable
False	False	False	False
Unavailable	Unavailable	False	Unavailable

Table 12 on page 124 shows all of the possible outcomes for event expression E2 when the value for one of its attributes, A1 or A2, is known.

Table 12 Truth table for event E2 = A1 || A2

	Attribute A1		
Attribute A2	**True**	**False**	**Unavailable**
True	True	True	True
False	True	False	Unavailable
Unavailable	True	Unavailable	Unavailable

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.

CHAPTER 11
Constraints

This chapter consists of the following sections:

Overview

A constraint is a Boolean expression that is evaluated at the end of a write transaction or anytime a change (write) is made to an attribute or relationship. When the constraint is checked, the expression must evaluate to TRUE or the transaction fails and an error is returned.

Constraint expressions are used to enforce limits on the values of attributes or combinations of attributes. Constraints are enforced at runtime, unlike ranges. Constraints can also be applied to relationshipsets.

Syntax

The syntax for a constraint on an attribute is:

```
<constraint>        ::=    [ hard | soft ]
                           constraint
                           <constraint_name>
                           [ <constraint_description> ]
                           "="
                           [ old ]
                           <expression>
                           ";"
```

The **hard** keyword modifies a constraint to indicate that it is always enforced and fatal to violate. The **hard** keyword is the default.

The **soft** keyword is currently treated the same as the **hard** keyword.

The **old** keyword can be used before an attribute name in a constraint expression. When used, it indicates "check the attribute's old value".

The *‹expression›* parameter must evaluate to TRUE or an error is returned when the constraints are checked. The ‹expression› may not include &&, ||. ?:, or "else" operators.

```
attribute double timeOut= 0.0;
constraint timeOutNonNegative
   = timeOut >= 0.0;
```

In this example, the constraint timeOutNonNegative was applied to the attribute timeOut. Therefore, if there is an attempt to assign a negative value to the attribute timeOut, the assignment will fail, the attribute will retain its old value, and an error will be returned.

The following example has the expression checking the old value of vChar when evaluating the constraint. In this example, the value of the attribute, vChar, must never be set to the same value twice in a row.

```
attribute int vChar;

hard constraint NotOld
   "Requires that vChar never be set to the same value twice
   in a row"
   = old vChar != vChar;
```

The syntax for a constraint on a relationshipset is:

```
<constraint>          ::=     [ hard | soft ]
                              constraint
                              <constraint_name>
                              [ <constraint_description> ]
                              "="
                              foreach
                              <iterator_name>
                              "(" <relationshipset_name> ")"
                              <expression"
                              ";"
```

For each object in the relationshipset, the *‹expression›* parameter must evaluate to TRUE or an error is returned when the constraints are checked. This form of constraint constrains the related class, therefore, you cannot use a unidirectional relationshipset; the inverse path must be available.

In the *‹expression›* parameter, you get the value of the specified attribute in the related class using:

```
<iterator_name> @ <relatedClass> :: <attribute>
```

If the related class is a subclass of the one to which the relationshipset relates, then only instances of that class in the relationshipset are examined. The attribute must be declared, not inherited, in the related class. The expression may not include &&, ||, ?:, or 'else' operators.

In this example, the constraint checks that every object in the relationshipset DrivenBy is a driver whose age is greater than 18.

```
interface Driver;

interface Car : MR_ManagedObject
{
   relationshipset DrivenBy, Driver, Drives;

   constraint age_constraint
      = foreach i (DrivenBy) i@Driver::age > 18;
}

interface Driver : MR_ManagedObject
{
   relationshipset Drives, Car, DrivenBY;

   attribute int age;
}
```

CHAPTER 12
Instrument Declarations

This chapter consists of the following sections:

Overview

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. For example, with 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 EMC Smarts 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.

Syntax

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

The keyword **instrument** indicates that this is an instrumentation declaration. The keywords **SNMP** and **REMOTE_REPOSITORY** declare the type of instrumentation, however, it is informational only.

Each optional *‹instrument_mapping›* parameter joins an instrumented attribute with the OID of the SNMP object. The syntax for declaring an instrumented attribute is described in "Instrumented attributes" on page 72.

The following example declares the instrumentation for three variables: if Descr, ifInOctets and ifOutOctets.

```
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,EthernetInterfac,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 hostname 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 EMC Smarts Framework. You may also connect an instance to the SNMP accessor through the EMC Smarts Manager's C API.

CHAPTER 13
MODEL Pragmas

This chapter consists of the following sections:

Overview

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. There are two required pragmas:

- #pragma include_c
- #pragma include_h

#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 filename 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 filename 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 Idempotent Get

This pragma is used with computed attributes, where you supply the code for the attribute put and get. It tells the monitoring system that the value of the attribute will change only if there is a put to the object. Thus, the monitoring system can treat this attribute as a stored attribute in terms of monitoring. If the user-defined attribute can change arbitrarily, then the monitoring system has to poll to detect changes.

#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 import

This pragma terminates a #pragma include block.

#pragma include

This pragma treats the input file, from this point until either the end of the file or a #pragma import, as part of the main file. Therefore, code is generated for declarations in included files. Normally, code is not generated for declarations in included files.

#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 Local Operation

This pragma locks only the object on which the operation is invoked. Normally, the MODEL compiler generates code so that invoking an operation implicitly locks the entire repository.

```
#pragma Local Operation
    readonly string HighUtilization_attributes()
   "Returns event format data for the HighUtilization event."
      definition:
          return "THRESHOLD  PCT UtilPct > MaxUtilPct"
                "PCT MaxUtilPct - -"
             "KB  StorageSize - -"
                "KB  StorageUsed - -";
```

#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. This pragma locks the entire repository, however, the lock type (read/write) will be according to the type operation called on the attribute. For example, if you are setting the value of an attribute it will be a write lock.

#pragma Unlocked

This pragma can be used for either interfaces, attributes or operations. It tells the MODEL compiler not to lock the repository at all. You can lock the repository in your own code. However, even if you do your own locking, if you delete the object just as the operation or attribute is being dispatched to your code, you may get a crash. In addition, this is inherited, meaning that once you have marked an attribute or operation as unlocked, you cannot change it back to locked, even in a derived class.

Note: You can also use this pragma for interfaces, however, it is not recommended to unlock an entire interface.

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.

```
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 that 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

Symbols

A

B

C

D

E

F

H

I

K

L

M

N

O

U

V