Oracle CDD/Repository™

Architecture Manual

Version 5.0

Reprinted 1995

Part No. A24879-2

ORACLE®

Oracle CDD/Repository Architecture Manual

Version 5.0

Part No. A24879-2

Contents

3 Element Names

4 Data Integration and Repository Extension

5 Schema Definition and Modification

6 Properties

7 Messages and Methods

8 Relations and Relationships

9 Configuration Management

10 Modeling Dependencies

Index

Figures

Tables

Send Us Your Comments

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

You can send comments to us in the following ways:

- electronic mail — nedc_doc@us.oracle.com
- FAX — 603-897-3334 Attn: Oracle CDD/Repository Documentation
- postal service

  Oracle Corporation
  Oracle CDD/Repository Documentation
  One Oracle Drive
  Nashua, NH 03062
  USA

If you like, you can use the following questionnaire to give us feedback.

Name _____ Title _____

Company _____ Department _____

Mailing Address _____ Telephone Number _____

Book Title _____ Version Number _____

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?

- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available).

Preface

This manual describes the concepts and capabilities of the Oracle CDD/Repository object-oriented architecture. This information should allow you to more easily understand and use the other manuals in the Oracle CDD/Repository document set.

The Oracle CDD/Repository architecture provides full access to the capabilities of previous versions of Oracle CDD/Repository and includes the following capabilities:

- extensible object-oriented information model

  The information model is implemented as an element type hierarchy that specifies both the relationship of elements and their definitions. You can extend the element type hierarchy to incorporate new schema information as needed.

- project support

  Oracle CDD/Repository includes functionality that supports the following integrated project environments:

  - configuration management
  - project management
  - file management

Intended Audience

This manual is intended for anyone who wants to understand the general architecture and capabilities of Oracle CDD/Repository. In particular, those who want to integrate applications with Oracle CDD/Repository should read this manual as preparation for designing the integration.

This manual contains a description of Oracle CDD/Repository configuration management capabilities. This information may be of interest to anyone who sets up or uses an integrated project support environment based on

Oracle CDD/Repository, or in some other way makes use of its configuration management facilities.

Document Structure

This manual contains the following chapters and appendixes:

- Chapter 1 describes in general terms the capabilities of Oracle CDD/Repository, illustrates typical uses, and introduces the architecture.
- Chapter 2 describes the object-oriented model and how Oracle CDD/Repository implements it.
- Chapter 3 describes how element names are formed, resolved, and manipulated.
- Chapter 4 describes how to design element types, properties, and methods.
- Chapter 5 shows how the element type hierarchy is defined and how element types define properties and methods.
- Chapter 6 describes how properties are implemented in Oracle CDD/Repository. Properties on repository elements store various types of data, including data whose value results from traversing relations.
- Chapter 7 describes how methods are invoked and the various method types.
- Chapter 8 describes relations and relationships. Relations store data in Oracle CDD/Repository by associating repository elements in specified ways. You may need to create new relation types in order to represent application data.
- Chapter 9 describes Oracle CDD/Repository version and configuration management models.
- Chapter 10 describes Oracle CDD/Repository support for automatic system building applications.

Related Documents

Documents related to Oracle CDD/Repository include the following:

- *Oracle CDD/Repository CDO Reference Manual*
- *Using Oracle CDD/Repository on OpenVMS Systems*
- *Oracle CDD/Repository Architecture Manual*
- *Oracle CDD/Repository Callable Interface Manual*
- *Oracle CDD/Repository Information Model Volume I*
- *Oracle CDD/Repository Information Model Volume II*
- *Installing Oracle CDD/Repository on OpenVMS Systems*
- *Read Before Installing or Using Oracle CDD/Repository on OpenVMS VAX Systems* or, depending on your system, *Read Before Installing or Using Oracle CDD/Repository on OpenVMS Alpha Systems*

See online help for a glossary of defined terms.

Conventions

This manual uses the name Oracle CDD/Repository to refer to all versions of this product. Prior to Version 5.0, this product was known as CDD and CDD/Plus.

Table 1 shows the other conventions used in this manual.

Table 1 Documentation Conventions

Convention	Description
{ }	In format descriptions, braces indicate required elements. You must choose one of the elements.
[ ]	In format descriptions, brackets indicate optional elements. You can choose none, one, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification.)
( )	In format descriptions, parentheses delimit the parameter or argument list.

(continued on next page)

Table 1 (Cont.) Documentation Conventions

Convention	Description
. . .	In format descriptions, horizontal ellipsis points indicate one of the following: • An item that is repeated • An omission, such as additional optional arguments • Additional parameters, values, or other information that you can enter
. . .	Vertical ellipsis points indicate the omission of information from an example or command format. The information is omitted because it is not important to the topic being discussed.
italic type	Italic type emphasizes important information, indicates variables, and indicates complete titles of manuals.
boldface type	Boldface type in examples indicates user input. Boldface type in text indicates the first instance of terms defined either in the text, in the glossary, or both.
n.nn	A period in numerals signals the decimal point indicator. For example, *1.75* equals *one and three-fourths*.
UPPERCASE	Words in uppercase indicate a command, the name of a file, the name of a file protection code, or an abbreviation for a system privilege.
lowercase	In format descriptions, words in lowercase indicate parameters or arguments to be specified by the user.
$	A dollar sign ($) represents the OpenVMS DCL system prompt.
monospaced	This typeface indicates the name of a command, routine, partition, pathname, directory, or file. This typeface is also used in interactive examples and other screen displays.
NAMED_ELEMENT	The names of element types are set in small capitals.
MERGE **hasChildren**	The names of messages and properties are set in bold type.
mod_1(2:a:3)	The names of repository elements are set in bold type.
list_value	References to values you supply, such as arguments, are set in italics.
0 ELEMENT 1 NAMED_ELEMENT 2 VERSION	The supertype–subtype relationship between element types is represented by numbers and by indentation. The example indicates that NAMED_ELEMENT is a supertype of VERSION and a subtype of ELEMENT.

1
Overview

This chapter provides an overview of Oracle CDD/Repository, including its capabilities in the following areas:

- data dictionary
- object-oriented data modeling
- project support

The chapter also describes the following support facilities:

- transaction processing
- distributed access
- security
- utility routines

The *Using Oracle CDD/Repository on OpenVMS Systems* manual explains how to use the command line interface. Refer to the *Oracle CDD/Repository Callable Interface Manual* for information on the callable interface.

1.1 Data Dictionary

Oracle CDD/Repository evolved from a data dictionary product and provides a full range of data dictionary capabilities. A data dictionary is a central location for shared data definitions. Storing data definitions in a data dictionary makes it easier for applications to exchange data, since they are using a common definition for the data. Without a data dictionary, application writers would have to write their own data definitions. Keeping these definitions consistent between applications would be difficult or impossible, especially when the definitions change.

Data administrators store and modify data definitions in the dictionary. Users of these definitions can be notified automatically when a definition has changed. This allows necessary changes to be made in an orderly fashion.

The enhanced version control features included with Oracle CDD/Repository enhance its use as a data dictionary by allowing more precise control of change in definitions.

1.2 Object-Oriented Information Model

Oracle CDD/Repository implements an information model in the form of a type hierarchy. The **type hierarchy** defines element types and places them in a supertype/subtype relationship. **Supertypes** are the most general element types and they are at the top (or root) of the type hierarchy. **Subtypes** are more specific types of objects. At each level of the type hierarchy, you only need to define those characteristics that make a subtype more specialized than its supertype.

The information model is self-defining, which means it is implemented as a set of metadata elements whose definitions are part of the type hierarchy.

You can extend the information model as needed to support tool and application integration. Typically, the model is extended for the following purposes:

- to model business-specific information, which allows data to be accessed naturally and efficiently
- to include new software tools

In both cases, the object-oriented model allows reuse of **methods** (code defined in the repository that implements common functionality). This means you only need to write code specific to the operation of the application or tool.

Refer to Chapter 2 through Chapter 8 for additional information on the object-oriented information model.

1.3 Project Support

Oracle CDD/Repository includes functionality that supports an integrated project environment in the following areas:

- configuration management

  Configuration management is tailored to the needs of large groups working on complex development efforts. It provides for multiple versions of system elements, multiple versions of system configurations, and variant lines of development.

- project management

  Project management allows you to switch between activities equated to project tasks, and to maintain the current state of each activity.

- user-designed approval structures

These structures allow you to promote system elements to higher levels (and wider visibility) as they become more stable or meet specified criteria.

- built-in file management

  Oracle CDD/Repository provides functions that you can use to automatically manage project files.

These capabilities are derived from the needs of project support environments, but they also are applicable to data dictionary and enterprise integration applications.

Refer to Chapter 9 and Chapter 10 for additional information on the project support functions provided by Oracle CDD/Repository.

1.4 Transaction Processing

Oracle CDD/Repository allows users to declare that several operations in sequence form a **transaction**, which is a unit of work that either succeeds completely or fails completely. Transaction control is necessary to assure that the repository is left in a consistent state in the event of a failure. Oracle CDD/Repository also provides journaled file operations that allow applications to keep files outside the repository consistent with the repository. (See the *Oracle CDD/Repository Callable Interface Manual* for more information on transaction processing and journaled file operations.)

1.4.1 Transactions

A transaction consists of a start point, a number of operations, and an end point. At the end, the transaction either commits (the changes are made) or aborts (the repository is rolled back to its state when the transaction started). If a system failure occurs during a transaction, the repository is automatically rolled back when the system restarts.

An application also can explicitly abort a transaction if it detects an error during the transaction. A common example is transferring funds from one account to another in a banking system. This requires two operations: debiting one account and crediting another. If the debit operation succeeds but the credit operation does not, it is the application's responsibility to detect the error and abort the transaction. If both operations succeed, the application should commit the transaction.

A transaction can be designated as read-only, which means that no elements will be modified during the transaction. Designating a transaction as read-only allows concurrent access to elements by other transactions that need only read them.

1.4.2 Journaled File Operations

The transaction mechanism provides for internal consistency in the repository, even in the event of system failure. However, an application that manages files in the native file system in parallel with elements in the repository needs some means of keeping the files and the repository consistent with one another. Oracle CDD/Repository extends the transaction mechanism with a set of journaled file operations that allow you to place operations on the native file system under transaction control.

The journaled file operation routines work by letting you carry out operations through Oracle CDD/Repository, instead of through calls to the native file system. The operations are recorded in the Oracle CDD/Repository transaction journal. If the transaction is aborted or the system fails, Oracle CDD/Repository automatically rolls back the file operations along with repository modifications that took place during the transaction. This has the additional advantage of providing a platform-independent interface to common file operations.

Oracle CDD/Repository journaled file operations allow you to:

- create and delete files
- control the point to which the file can be rolled back
- rename files
- create and delete symbolic links to files
- create and delete file-system directories

Oracle CDD/Repository performs these operations when you call the appropriate routine and undoes them if you roll back the transaction. (See the *Oracle CDD/Repository Callable Interface Manual* for more information on transaction processing.)

1.5 Distributed Access

Oracle CDD/Repository provides distributed access to elements through the specification of a full element name. A full element name can include the node, device, and file-system directory that contains the repository. (See Section 3.1 for more information on element names).

This mechanism allows you to access elements from any repository in your network. You can use element IDs returned from remote nodes in all local operations, with the exception of subtypes of BINARY, which cannot be accessed in a distributed environment.

Distributed operations include:

- reserve, replace, and unreserve

  The master copy of a component can be maintained in a given repository, but be used in systems maintained in other repositories. It is possible for you to create a new version of a component at a remote location and replace the version in the master repository with the new version.

- remaster a component

  You can change the location of the master copy of any component.

- read and write properties

  You can access and modify the properties of a remote component without accessing its contents.

- copy a component

  You can get a version of the contents of a remote component for read-only access.

- configuration membership

  A component in one repository can be part of a configuration defined in another repository.

- create relationships

  You can create relationships between objects in different repositories; for example, to permit a component to be part of a remote collection. This capability is implied by distributed configuration membership.

Both the Oracle CDD/Repository callable interface and CDO provide distribution mechanisms. For example, in the callable interface, several routines, including *MCS_DB_new* and *MCS_initiate_database*; the messages, **import** and **export**; and several properties, including **filePath, importedFrom, storedIn,** and **storeType**, are related to distributed processing (see the *Oracle CDD/Repository Callable Interface Manual* and the *Oracle CDD/Repository Information Model Volume I* manuals). (See the *Using Oracle CDD/Repository on OpenVMS Systems* manual for information on how CDO facilities manage distributed repositories.)

Distribution across heterogenous platforms is not supported.

1.6 Security

Oracle CDD/Repository implements security using access control lists (ACLs). Every named element in a repository has an access control list.

If you do not specify an ACL when you create an element, Oracle CDD/Repository supplies a default ACL for the element. You can specify a default ACL for all new elements by setting the **defaultAccess** property on CONTEXT or DATABASE.

To modify the ACL of an existing element, set the **MCS_access** property of that element using a dispatch call with a **setProp** message. (For more information on security, see the description of the access property in the *Oracle CDD/Repository Information Model Volume I* manual and the chapter on protecting your repository in the *Using Oracle CDD/Repository on OpenVMS Systems* manual.)

1.7 Utility Routines

Oracle CDD/Repository provides several utility routines you can use to perform the following functions:

- translate directory names

  CDD$TRANSLATE applies Oracle CDD/Repository name translation rules to a set of user-supplied directory names, returning fully translated names.

- verify the integrity of a repository

  CDD$VERIFY confirms and optionally restores the integrity of a repository or directory.

- return the version of a repository

  CDD$VERSION returns a formatted buffer of the versions of all protocols in the attached repository.

- check for notices

  CDO$CHECK_NOTICES checks to see if there are notices on any of the named entities.

- call CDO from a program

  CDO$INTERPRET allows you to call CDO from a program.

For more information on these utility routines, refer to the *Oracle CDD/Repository Callable Interface Manual*.

2
Object-Oriented Data Model

The Oracle CDD/Repository callable interface presents an object-oriented view of the repository contents. The objects in the repository (called **elements**) represent the building blocks (data definitions, source and result files, designs, project management artifacts) of systems, as well as configurations consisting of selections of these elements. Applications perform operations on these elements by sending messages to them.

This chapter explains the fundamentals of the Oracle CDD/Repository object-oriented interface.

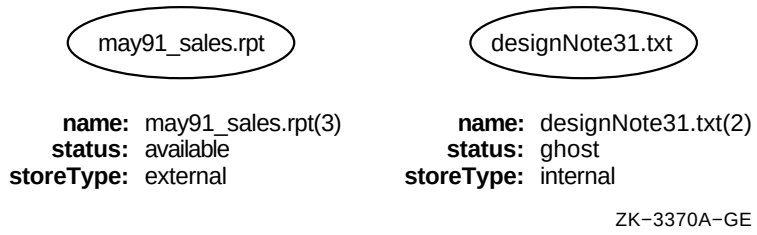
2.1 Elements, Properties, and Methods

The elements in a repository are of different **element types**, such as EVENT, NAMED_ELEMENT, and RELATION. Each element is an **instance** (occurrence) of its type, and is distinguished from other elements of that type by the values of various attributes that make it unique.

These attributes are called **properties**. Properties provide information about the element or relate the element to one or more other elements. An example of a property is the name of an element. Elements of the same type have the same set of properties, but these properties have different values from one element to the next.

For example, consider two elements of type TEXT. TEXT elements represent files that contain text. Figure 2–1 shows how these two elements possess the same properties, but the properties have different values. (Not all properties are shown.)

Figure 2–1 How Property Values Distinguish Elements



You manipulate elements by sending **messages** to them. A message requests an action, but does not specify the form of the action. For example, a message might request an element to return information about itself. The message does not supply a procedure for obtaining that information. Instead, the element has available to it a set of **methods**, one for each message that it recognizes. The methods are defined by the element type. The definition of an element includes information on how to respond to messages it recognizes. When an element receives a message, Oracle CDD/Repository invokes the appropriate method for that combination of element type and message to perform the action requested by the message.

You do not need to define a complete set of properties and methods for each element type. To avoid repeating definitions, element types **inherit** methods and properties from other element types. Each element type can have a single supertype from which it inherits properties and methods. (An object-oriented system with this characteristic is called a **single-inheritance system**.) Similarly, each element type can have zero, one, or many subtypes that inherit properties and methods from it. This inheritance structure forms an **element type hierarchy**.

An element type inherits the methods and properties possessed by its supertype. Therefore, unless an element type is modified, it responds to the same messages as the supertype, and in the same way. Element type modifications include:

- the addition of new properties to the element type
- the recognition of new messages
- the modification of the type's inherited methods, known as method refinement

Method refinement allows the element type to respond differently to a message than does its supertype, usually by providing additional or more specialized processing.

2.1.1 Elements and Project Support Environments

How do Oracle CDD/Repository element types support implementation of an integrated project support environment? Elements in a project repository correspond to various entities involved in the project. These entities include files of various types, tools such as editors and compilers, and system configurations.

The properties attached to each element contain information about that element and how it relates to other elements. For example, an element representing a source file has one property that indicates where you can find the file, and another property that gives the current status of the file. These properties contain information about the file itself. The element might also have a property indicating the previous version of the file, another indicating the system configuration(s) it belongs to, and a third indicating the specification for the code it contains. These properties relate the element to other elements.

Together, elements and their properties can represent the state of a project. Project members (through the tools that they use) query and modify this state by sending messages to elements. Each message represents a request. The request is expressed in general terms, such as "Get me this object to modify" (to paraphrase the RESERVE message). The sender of the message need not know how to "get" the object, only that it wants it. The element representing the "object" invokes the appropriate method to perform the general request made by the sender.

Elements, properties, and methods supply the building blocks for a project support environment, but they must be arranged appropriately if the environment is to be effective. Oracle CDD/Repository specifies this arrangement for the fundamental project support operations that are common to all environments. These arrangements are described in Chapter 9 and Chapter 10.

2.1.2 Access to Elements

All elements in a repository have a unique element ID. An **element ID** is a handle to the element. You always access elements by specifying their element IDs. Many Oracle CDD/Repository operations involve element IDs as arguments, return values, or both.

Most elements also have a name. If you know an element's name, you can obtain its element ID for use in further operations. (See Chapter 3 for more information on element names.)

2.1.3 Properties

Element properties have names, data types, and **implementation types** that describe how the property value is stored. You use the name to specify the property in the argument lists of routines that deal with properties. The data type determines the form of the property value; data types include the familiar scalar types as well as some types specific to Oracle CDD/Repository. (See Section 6.1 for information on property data types.)

A property's implementation type is of interest primarily when you design and implement a new property. However, knowing a property's implementation type also gives you some insight into the type of information contained in the property. The implementation types are the following:

- **normal**—data is stored in the property. Normal Oracle CDD/Repository properties are equivalent to "attributes."
- **computed**—value of the property is computed at access time by invoking a method (see Section 2.1.4).
- **relation**—value of the property is determined by traversing the relationship between two elements (see Chapter 8).
- **closure**—similar to a relation property, except that the value is determined by traversing a relationship recursively between all associated elements.

You can directly manipulate properties by means of two messages: SETPROP and GETPROP. As their names indicate, these messages set or return the value of a property. Both messages are sent to an element with a list of property/value pairs as arguments. SETPROP sets the properties on the target element to the new property value; GETPROP returns the values. The element responds to the message by invoking the method that is defined to set or retrieve property values in that element type.

The access type of a property specifies how and when you can change the value of that property, as described in the following list:

- Some properties can be explicitly changed using SETPROP.
- Some properties that cannot be changed with SETPROP can be indirectly changed using Oracle CDD/Repository messages.
- Some properties cannot be changed.

The property access types are described in the following list:

– **read only**—you only can read the property, not write it. This access type is usually found on properties that are set and changed by Oracle CDD/Repository and not directly by you, as on computed properties.

- **read/write**—you can read and change (using SETPROP) the property.
- **write once**—you can set the property once but you cannot change it (using SETPROP) thereafter.
- **write once at creation**—you can set the property only when you create the element that possesses it.

Chapter 4 contains information about how properties implement various aspects of a data model. Chapter 6 shows how property definitions are implemented in the repository.

2.1.4 Messages and Methods

You must know the difference between messages and methods. Messages request that an element perform some action, but say nothing about *how* to perform that action. The same message can be sent to elements of any type that recognizes the message.

The methods attached to element types provide linkage to the procedures that implement messages the element recognizes. For example, two elements of different types may respond to the same message by invoking different methods. Because both methods are tailored to the requirements of their element types, they provide customized responses to the message.

For example, the description of the RESERVE message (see the *Oracle CDD/Repository Information Model Volume I* manual) states that it "checks out an element to which it is sent by creating a new version of the element." You can send the RESERVE message to any element. The element type of the receiving element determines the actual method (if any) that it invokes to perform the operation. There are three general cases:

- The element type defines elements that are not intended to exist as successive versions. These element types reject the RESERVE message.
- The element type defines the most general elements that are intended to exist as successive versions (called **versionable elements**). These elements respond to the RESERVE message by carrying out the basic processing needed to create a new version.
- The element type defines versionable elements that are more specialized; for example, the elements may represent files. These elements respond to the RESERVE message by carrying out processing that is specific to that type of object (for example, reserving a file from a delta mechanism) and calling on the general RESERVE method to carry out the basic reserve processing. This is called **method refinement**.

2.1.5 Inheritance

Inheritance in Oracle CDD/Repository means that a new element type includes the property and method definitions possessed by its supertype. For example, you may want to describe C source files. If you have already described the characteristics of text files, you do not have to include those characteristics to describe C source files. First state that all C source files are text files, then describe those characteristics that are unique to C source files.

If you need to make the subtype more specialized than its supertype, you can refine the type. **Type refinement** consists of some combination of the following activities:

- adding new properties
- defining methods to respond to messages not recognized by the supertype
- refining methods inherited from the supertype

Method refinement consists of making the inherited method more specific and appropriate for the subtype. In refinement, the element type invokes the inherited method but performs some additional processing before and/or afterwards. In this way, the response to a message can incorporate the supertype's response, yet provide more specialized and appropriate action.

For example, a BINARY element uses the versioning mechanisms defined by VERSION by invoking VERSION's methods. BINARY then provides additional processing needed to operate the delta mechanism that stores files.

A refinement also can disallow the message to which the method is a response. Sometimes it is not appropriate for the subtype to respond to all messages its supertype responds to.

The inheritance structure of a supertype and its subtype results in an element type hierarchy, where each element type may have several subtypes that represent specializations of that type. It is crucial that you understand the element type hierarchy because a type's position in the hierarchy determines the properties and methods that it inherits. To completely understand an element type, you must examine and understand all its supertypes as well.

The *Oracle CDD/Repository Information Model Volume I* manual describes the core of the Oracle CDD/Repository element type hierarchy. The types described in that manual form a *minimal* hierarchy. Only those element types needed for fundamental project support and schema definition operations are represented. The remainder of the element type hierarchy is described in the *Oracle CDD/Repository Information Model Volume II* manual. The types described in that manual include the protocols supplied with Oracle CDD/Repository.

You can extend the element type hierarchy as necessary to support specific applications.

The following example shows part of the Oracle CDD/Repository element type hierarchy.

```
0 ELEMENT
    1 NAMED_ELEMENT
        2 VERSION
            3 AGGREGATE
                4 BINARY
                    5 TEXT
```

This example illustrates that the element type TEXT has the supertype BINARY, and that BINARY has the supertype AGGREGATE. This type of relationship continues up to ELEMENT, which has no supertype. The supertype/subtype relationship is shown by indentation and by the level numbers. This example illustrates:

- All element types are subtypes of type ELEMENT.
- Some elements are of type NAMED_ELEMENT. These elements have an additional property: name.
- Some named elements are of type VERSION; they possess additional properties and methods that allow creation and tracking of multiple versions of the entities represented by these elements, for example, multiple versions of a file.
- Some version elements are of type AGGREGATE; they represent compound objects.
- Some aggregate elements are of type BINARY; they represent files.
- Some binary elements are of type TEXT; they represent text files.

The element type hierarchy diagram in this example shows only TEXT and its direct ancestors, since these are all that matter when tracing the inheritance of TEXT. However, there can be multiple types defined at each level; for example, level 2 has several types other than VERSION.

To list the properties inherited by TEXT, you should include all the properties defined by ELEMENT, NAMED_ELEMENT, VERSION, AGGREGATE, and BINARY. Add to these the properties defined by TEXT and you have the complete set of properties for TEXT elements.

To list the methods inherited by TEXT is more involved. To see exactly how TEXT responds to a given message, you must look at the response of its supertype, then the supertype's supertype, on up the type hierarchy until the response to a particular message is first defined or replaced.

There may be a method defined that allows an element type to respond to messages that its supertype does not recognize. For example, VERSION defines a method to respond to the RESERVE message. NAMED_ELEMENT, the supertype of VERSION, does not recognize RESERVE because this operation is not appropriate for all types of named elements, only those that are under version control.

2.2 Defining and Extending the Repository

An Oracle CDD/Repository schema is self-defining. This means the repository contains the definition of its own structure. As a result, you can modify and extend a repository structure by using the same interface used to manipulate the objects it contains.

Elements that define the repository schema are called **metadata elements**. Metadata elements include elements that represent the following:

- Element types. These elements control the creation of new elements.
- Relation types. These elements control the creation of relationships, which are a specialized type of element. Relationships associate other elements in the repository.
- Property definitions. These elements define property characteristics such as data type and access type.
- Messages. These elements define the arguments taken by a message.
- Message arguments. These elements define the data type of an argument taken by a message.
- Methods. These elements define the processing carried out by elements of a particular type in response to a message.

By adding new metadata elements or relationships between existing metadata elements you can extend the schema. Oracle CDD/Repository uses the metadata elements to control its operations, such as creating new elements, setting and retrieving property values, and invoking methods in response to messages.

Chapter 4 through Chapter 8 describe in detail how Oracle CDD/Repository defines its schema. By understanding this mechanism, you are better able extend the Oracle CDD/Repository schema to meet your needs.

3
Element Names

This chapter describes Oracle CDD/Repository element names. It covers the following topics:

- syntax of element names
- uniqueness requirements for element names
- use of defaults and wildcards

Only elements whose type is a subtype of NAMED_ELEMENT have names. Of these element types, those that are also subtypes of VERSION include additional information in the name, and have additional rules associated with locating them by name. (Note that some subtypes of VERSION have no names; for elements of these types, the value of the **name** property is a null string.)

Note

The name syntax for Oracle CDD/Repository version 5.0 and higher is somewhat different from the syntax of prior versions, but the prior names are still supported.

3.1 Name Syntax

A full element name consists of the following parts:

repository directory-path / *element*

Note

The period character (.) and the slash character (/) are interchangeable.

These parts perform the following functions in identifying the element:

- **repository**—Specifies the node, device, and file-system directory containing the repository. A repository also is referred to as an **anchor** or **root.**
- **directory-path**—Specifies the path in the repository directory tree to the repository directory containing the element.
- **element**—Specifies the name of the element, including branch information for versioned elements.

The following example shows a full element name:

```
NODE::DISK1$:[SMITH.PROJECT]/BUILD/SOURCES/XXX.C(3:A:2)
```

The parts of this element name are as follows:

- NODE::DISK1$:[SMITH.PROJECT] is the repository. This specification identifies a node, device, and directory in the native file system.
- /BUILD/SOURCES is the directory-path. It specifies that the element is contained in the repository directory BUILD/SOURCES. If this part of the name is omitted, the element is in the repository's top-level directory.
- XXX.C(3:A:2) is the element name or **simple name**. It specifies the element named XXX.C, version 3:A:2. The division of the name into two parts, separated by a period, has no significance to Oracle CDD/Repository; however, it allows users who are used to files to be more comfortable with element names.

The sections that follow describe the parts of a full element name in more detail.

3.1.1 Repository Specification

The first part of a full element name specifies the node, device, and native file system directory. The following example shows the syntax of the repository specification:

```
DISK1$:[SMITH.PROJECT]
```

You need specify the node only if the repository is on a remote node or cluster.

If Oracle CDD/Repository returns a full element name, the name includes the repository specification. However, you can frequently omit the repository specification when you specify a name to Oracle CDD/Repository by using defaults (see Section 3.3 for more information).

3.1.2 Directory Path Specification

An Oracle CDD/Repository database contains a hierarchical directory structure, similar to (but independent of) the directory structure in the native file system. You can use the repository directory structure in the same way you use a file system directory structure:

- to organize objects in the repository in a way that is meaningful and helpful
- to partition the name space, allowing elements with identical simple names to exist in the same repository but in different directories

Dividing a large number of elements among several directories also may aid performance.

The syntax of the directory path is as follows:

[/ *directory-name*]*

This indicates that *directory-name*, preceded by a slash (/), can be omitted, written once, or written many times. For example:

```
/BUILD/SOURCES/PARSER/OLD
```

The *directory-name* must start with a letter and can include letters, numbers, underscores (_), and dollar signs ($). The length of the *directory-name* cannot exceed 31 characters.

If the directory path is omitted from a full element name, the element is located in the repository's top-level directory. However, if the directory path is omitted from an element name that also omits the repository specification, then the element is located in the current default directory. See Section 3.3 for information about using defaults.

3.1.3 Element Specification

The name of an element (typically referred to as the **simple name**), exclusive of repository and directory path information, consists of the following parts:

simple name [(*branch-version-information*)]

The *simple name* must start with a letter and can include letters, numbers, underscores (_), dollar signs ($), and one period (.). The length of any part of the simple name preceding or trailing a period cannot exceed 31 characters.

The *branch-version-information* part and its surrounding parentheses are present only if the element is of type VERSION or one of its subtypes. This part identifies the element's branch and version number. (See Section 9.1 for

information about this value.) Branch names follow the same rules as for the *simple name* part, except that they cannot include periods.

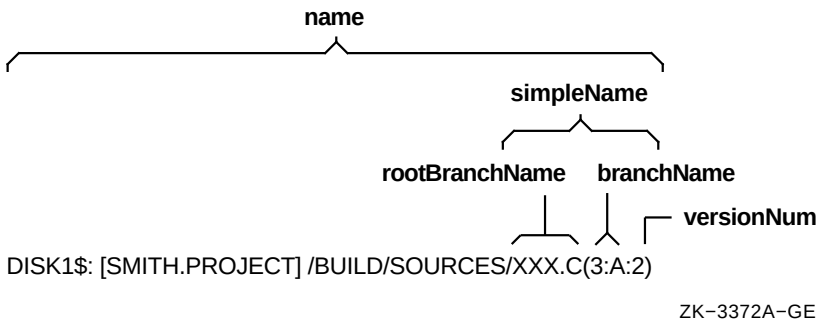
3.1.4 Case Sensitivity

Element names follow the same rules for case sensitivity as the operating system, which means that names are not case sensitive. All names are converted to uppercase for purposes of comparison with other names.

3.1.5 Name Properties

A number of properties contain all or part of an element's name. Figure 3–1 summarizes these properties. Note that all properties in Figure 3–1 except **name** and **simpleName** are defined by the VERSION element type. The value of the **versionNum** property is an integer; all other values are character strings.

Figure 3–1 Name Properties



3.2 Requirements for Unique Names

Fully-qualified element names must be unique within the repository. However, elements in different directories in the same repository can have the same simple name, as shown in the following example:

```
DISK1$:[SMITH.PROJECT]/BUILD/SOURCES/XXX
DISK1$:[SMITH.PROJECT]/BUILD/SOURCES/OLD/XXX
```

In this example, different path names make each element name unique.

The following two objects cannot exist in the same repository, even though they have different element types:

```
DISK1$:[SMITH.PROJECT]/BUILD/XXX (CONTEXT)
DISK1$:[SMITH.PROJECT]/BUILD/XXX (PARTITION)
```

A repository directory cannot contain an element whose name is the same as the name of one of the directory's subdirectories, as shown in the following example:

```
DISK1$:[SMITH.PROJECT]/BUILD/XXX
DISK1$:[SMITH.PROJECT]/BUILD/XXX/YYY
```

In the first line of this example, XXX is an element name in directory BUILD. In the second line, XXX is a subdirectory of BUILD, and YYY is an element name.

A directory also cannot contain an element whose name contains a period if the part of the name preceding the period is the same as the name of one of the directory's subdirectories. For example:

```
DISK1$:[SMITH.PROJECT]/BUILD/XXX.Z
DISK1$:[SMITH.PROJECT]/BUILD/XXX/YYY.Z
```

3.3 Element Name Defaults

Oracle CDD/Repository applies defaults for the following parts of an element name if you do not supply them:

- repository specification and directory path
- branch and version information for versioned elements

Oracle CDD/Repository also applies special defaults for metadata elements.

3.3.1 Repository and Directory Defaults

If you supply a simple element name without preceding repository or directory information, Oracle CDD/Repository applies a default directory path, including the repository specification. This default is established and modified as follows:

- When you start Oracle CDD/Repository, you can define a default directory path using the OpenVMS logical name CDD$DEFAULT.
- When you call the initiate_database routine, the *directory_name* argument establishes the default repository and directory path.
- Any time you call the MCS_set_default routine, the *directory_name* argument establishes a new default directory path.

If you supply a directory-path and simple-name, Oracle CDD/Repository tries to resolve the full name using the following process:

- Oracle CDD/Repository checks the first component of the name. If it is a repository name, Oracle CDD/Repository uses it.
- If it is not a repository name, Oracle CDD/Repository uses the current default (set using *MCS_initinate_database* or *MCS_set_default*). Oracle CDD/Repository prefixes the default path to the name you supplied.
- If the path still does not contain a repository name, Oracle CDD/Repository prefixes the translation of CDD$COMPATIBILITY.

For example, you supply the directory-path and simple-name:

```
/BUILD/XXX/YYY.Z
```

The default is defined as:

```
DISK1$:[SMITH.PROJECT]
```

Oracle CDD/Repository provides the full name:

```
DISK1$:[SMITH.PROJECT]/BUILD/XXX/YYY.Z
```

3.3.2 Branch and Version Defaults

If you want to locate a versioned element by name, specify all branch and version information. If you omit branch and version information, and there is a current context, Oracle CDD/Repository returns the version under the default path (if one exists). If none exists, Oracle CDD/Repository returns the most recently replaced version on the main line of descent.

3.3.3 Special Rules for Metadata Elements

Instances of the following element types and their subtypes are considered metadata elements:

TYPE
MESSAGE
METHOD
MSGARG

Taken together, these elements define a repository's operation: how elements are created, how property values are formed, and so on. References to names of metadata elements are treated differently, using special rules. When the element type you specify with MCS_element_getByName is one of the metadata types, the following default rules are in effect:

- If you omit a directory path, the directory CDD$PROTOCOLS is used, not your current default directory.
- If you omit branch and version information, the element ID of the element that is currently being used in the repository schema is returned, even if it is not the most recently replaced on the main line of descent.

These default rules make it easy for you to find the element ID of a metadata element that currently governs operations in the repository.

If you supply a full directory path, Oracle CDD/Repository looks in the directory you specify, not in CDD$PROTOCOLS. Similarly, if you supply full version and branch information, Oracle CDD/Repository finds the specified version, not necessarily the currently active version.

3.4 Wildcards

If you want to find a number of elements with similar names, use wildcard characters to specify a pattern that the names must match. Oracle CDD/Repository recognizes the wildcard characters appropriate to the operating system. You can use wildcards to search for the following:

- branches
- directory paths
- element names
- versions

4

Data Integration and Repository Extension

You can add data models to a repository by extending the Oracle CDD/Repository base type hierarchy. Reasons for adding data models include the following:

- **enterprise modeling**

  Organizations may want to model the way their business operates by adding types and relationships to Oracle CDD/Repository. These types and relationships define a subschema in the repository that allows business data to be stored there in a natural way.

- **software tool integration**

  Software tools (CASE tools, for example) integrate into Oracle CDD/Repository by adding types that represent the objects upon which they operate. Depending on the tool, one or two types may suffice, or many types and relationships may be needed.

This chapter introduces the process by which you decide how to extend Oracle CDD/Repository to meet your needs. It shows how to characterize the data you need to model, then translate those characterizations into designs for property definitions. It also shows how to characterize the operations you need to perform on your data, then translate those characterizations into designs for method definitions and refinements. Implementing these definitions in Oracle CDD/Repository is called **data integration**.

Before you extend Oracle CDD/Repository, you must know:

- what objects your application manipulates in the repository and the characteristics of the types you need to create to represent them
- which standard repository element types you should use as supertypes for your types

This chapter addresses these needs. However, element design is not a strictly serial process. You must increase your understanding of your application's objects and the repository elements in parallel until you arrive at a set of detailed object descriptions that can be implemented by subtypes of existing repository element types.

4.1 Objects

Before you can map the objects used in your application to repository elements, you must know the following about each object:

- The nature of data that it stores. For example, does the object have a name? Is there numeric or string-valued information associated with it? Does it contain structured data, as a database would? Is the value of data stored by the object fixed or variable? If the value is variable, should users be able to re-create previous versions of the object?
- How objects of each type relate to other objects of the same type and of other types. You must characterize relationships in terms of:
  - The *meaning* of the relationship. For example, a relationship between an employee and the employee's manager means "has-manager" from the employee's point of view.
  - The *types of elements* that participate in a relationship. For example, employees and their managers are both persons, so the "has-manager" relationship can be characterized as associating one person (the employee) with another person (the manager).
  - The *number of elements* that participate at each end of a relationship. One manager can usually have many employees, but an employee may not be able to have more than one manager.

  You must fully characterize these relationships *before* you implement them in the repository.
- The types of operations that can be performed on it. Most objects can be created and destroyed, but there are frequently constraints on these operations. (For example, an object cannot be created that has the same name as another object, or cannot meaningfully exist unless its stored data falls within certain ranges.) Objects also can be accessed, manipulated, or transformed into other objects. Refer to the list of Oracle CDD/Repository messages (see the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals) to determine which operations they represent, how they apply to your objects, and how your objects should respond.

The sections that follow examine each of these element characteristics in more detail.

4.2 Data Associated with Objects

Multiple objects of the same type (for example, source files written in the same language or customers in an accounts receivable system) are distinguished by unique names, by numeric identifiers, or by some other combination of information associated with them. In addition to these distinguishing characteristics, objects carry other useful information. A source file, for example, contains lines of source code, which in turn provide a module identification, variable and routine declarations, and so on. An object that stands for a customer contains information about the customer's address, account balance, credit rating, and any other information required by the application.

An important part of element design is to decide what information an object definition must include and how to map these requirements into property definitions for the new element types. If the object is a file that contains data, this process is simple, since the element type will represent an entire file. While the file may implement a complex data schema, the data that must be represented *in the repository* is straightforward: the file, and some operations on it. On the other hand, an application may represent more (or all) of its data directly in the repository. When integrating these applications, mapping object data requirements into repository property definitions is more complicated.

To map object data to element properties, you must characterize the data along the following lines:

- source of the data

  Is the data an inherent attribute of the object, such as a name or identification number? Does it derive from the relationship of the object to other objects? Does it summarize, through a computation, the state of a subset of the repository at a particular time?

- type of the data

  Is the data numeric, string, a structured type, or a reference to one or more other objects?

- volatility of the data

  Is the data static or volatile? Some data, for example a unique identification number, is static. It must be specified when the object is created, and it cannot change. Other data, an account balance for example, must be present but is expected to change over time. Also, some

data may not be present in all instances of an object. For example, an employee might not have a middle name.

Data derived from the relationship of an object to other objects is described in Section 4.3. The remainder of this section describes how to map data from the other sources to Oracle CDD/Repository property definitions and how to specify data value volatility and data type.

4.2.1 Inherent Object Data

Inherent object data is data that characterizes the object independent of the existence of other objects in the system. A person's height, weight, and social security number are examples. They characterize the person, not the person's relationship with another person. If these values change, they do so independently of other objects.

Inherent data such as this is implemented using "normal" properties on elements. A **normal property** is one in which the data is stored with the element. A normal property is analogous to a numeric or string-valued field in a record structure.

When you specify a normal property, you must also specify its access type (see Section 4.2.3) and its data type (see Section 4.2.4). (Section 6.3.1 describes how normal properties are implemented.)

4.2.2 Computed Object Data

Some objects are characterized by data that can be computed from other data rather than being stored with the object. An example is a remaining-credit field for a customer. The object representing the customer can store values for credit limit and current account balance. The remaining credit can be calculated from these two values. Calculating the remaining credit amount when it is needed, rather than storing it, saves space and reduces the difficulty of keeping the values consistent. However, it takes more time to calculate the value than to retrieve it from a normal property.

For another example of a calculated value, consider an object that represents a salesperson that includes pointers to other objects that represent the salesperson's accounts. You need to know how many accounts the salesperson has. This number can be stored with the object, but then it has to be updated as a side-effect of adding or removing pointers to the accounts. Setting the value directly, independent of adding or removing accounts, has no meaning. It makes more sense to count the accounts to determine how many accounts the salesperson has.

These examples show how you can derive computed values algorithmically from other information in the system, even if the computed values are not inherent attributes of an object. Because computed values are calculated, they cannot be set, any more than you can write "A+B=6" in a programming language. These values can only be retrieved, and they must be calculated when they are retrieved.

Calculated object data is implemented by means of a "computed" property. The definition of a **computed property** is associated with a method. A request for the value of the property (via the GETPROP message) invokes the method, which computes and returns the value of the property. (Section 6.3.2 describes how to implement a computed property.)

4.2.3 Data Integrity

An object's data integrity requirements dictate whether certain object data must have a value for all instances of the object and whether that value can be explicitly changed. In many cases, the consistency of the repository depends on strict control of these values, so it is important to consider data integrity early in the design process.

There are several considerations when defining data integrity requirements:

- Can a data value be changed explicitly by users?

  "Changing explicitly" means changing the value by sending the SETPROP message.

  Some values, once set, must never be changed because to do so would compromise the consistency of the repository. For example, a value that uniquely identifies an object (a primary key) must not be changed, because other objects may refer to that value. Changing the value would invalidate these references and in effect corrupt the repository. Other values cannot be changed explicitly by users but may change as a side-effect of other operations. The computed values described in Section 4.2.2 are examples. Still other values (for example, an account balance or a nickname) can be changed freely.

- Must a data value be present when an object is created?

  Frequently, an object is not complete unless some or all of its data is defined; for example, an object referred to by an identifier is not complete without the identifier. In other cases, it is possible to supply a default value for missing data, or the *absence* of data may have meaning in itself.

- Must a data value meet specified criteria?

These criteria protect the integrity of the repository in various ways:

- ensuring that unique identifiers or names are actually unique
- ensuring that data values that are supposed to refer to other objects by an identifier actually refer to existing identifiers (the concept of "referential integrity")
- making any other checks dictated by the purpose of the data; for example, ensuring that an account balance is not allowed to exceed a credit limit

Oracle CDD/Repository provides the following ways of implementing data integrity requirements:

- property access types
- required properties
- method refinements
- validations

The relation property data types of SCAN and ELEMENTID also provide inherent referential integrity, since these values always refer to valid repository elements.

The property access type controls how a property value can be changed with SETPROP. The four access types are:

Access Type	Meaning
Read-Only	The property value can be retrieved but cannot be set. Note, however, that a read-only property value can change as the result of other operations.
Read/Write	The property value can be set at any time with SETPROP as well as retrieved.
Write Once	The property value can be set once, either when the element is created (with NEW) or afterwards (with SETPROP).
Write Once at Creation	The property value can be set once when the element is created (with NEW).

The access type is sufficient to prevent users from changing values that must not be changed, but it is not sufficient to ensure that required values are specified or that values satisfy constraints. You can further specify that a property value must be included with a new object by making the property

required for that type. When you do this, an attempt to create a new instance of the element without that property specified will fail.

When a value must satisfy a constraint, you should write a method that checks the value and, if the value is not acceptable, rejects the operation with a suitable error code. If the value is acceptable, the operation is performed. This type of method is called a **validation**. (Refer to the *Oracle CDD/Repository Callable Interface Manual* for information about writing methods that perform validations.)

Table 4–1 summarizes the mechanisms for implementing data integrity requirements.

Table 4–1 Implementing Data Integrity Requirements

Requirement	Access Type	Required?	Method Refinements
Data item must be present, must uniquely identify the object, and must not change (primary key).	Write once at creation	Yes	NEW
Data item must be present, must refer to a valid object (or satisfy some other integrity criteria), and can be changed by the user.	Read/Write	Yes	NEW and SETPROP
Data item need not be present, but if present must satisfy integrity criteria.	Read/Write	No	NEW and SETPROP
Data item must be present, can be changed by user, and has no value constraints other than those imposed by its data type.	Read/Write	Yes	None (values of incompatible type will be rejected)
Data item need not be present, can be changed by user, and has no value constraints other than those imposed by its data type.	Read/Write	No	None (values of incompatible type will be rejected)

4.2.4 Data Type

Oracle CDD/Repository provides a variety of scalar and structured data types for data that is stored with elements. The *Oracle CDD/Repository Information Model Volume I* manual describes these types and the operations you can perform on them. This section makes some general observations about what data types are appropriate for various types of properties. (See Section 6.1 for a complete list of the data types.)

A normal property can be any data type but SCAN or ELEMENTID. (However, you can only store a value in a property of the type defined for that property.) The following data types are frequently used for normal properties:

- BOOLEAN—represents a true–false value
- SMALLINT—contains a predefined value (a symbolic constant) from a restricted range (a SMALLINT is an integer stored in 16 bits)
- LONGINT—contains a predefined value (a symbolic constant) from a restricted range (a LONGINT is an integer stored in a longword)
- STRING—stores a name
- MEMBLOCK—stores a block of memory whose structure and use is determined solely by the application that stores and retrieves it
- LIST—stores an ordered sequence of values, possibly of disparate types
- DATETIME—represents time values

A computed property can be of any data type. The method you write to compute the property must compute and return the value in the correct data type.

A relation property can be of type SCAN or ELEMENTID, depending on the definition of the relation. Closure properties (closely related to relation properties) are always of type SCAN. (Section 4.3 describes relation and closure properties.)

Note

You cannot add data types to the respository.

4.3 Relations with Other Objects

Defining how objects relate to other objects is an important part of the analysis of your application's operation. Once you have defined these relations, you can create **relation properties** whose value consists of other objects that relate to the object owning the property in specified ways. You can design these properties so that their values are constrained very precisely to meet your requirements; but you must completely understand those requirements.

There are many entity-relation (E-R) modeling techniques available. If you know an E-R approach, you can use its techniques to produce a design model of objects and relationships that satisfy the functional requirements of your tool. Use the design document as a checklist of requirements that need to be supported by the repository type hierarchy.

Although you can use E-R models for design, they cannot be implemented directly in the object type hierarchy for the following reasons:

- Many E-R techniques do not include concepts of subtyping and inheritance.
- Relationships are implemented as properties, and one relation type can translate to as many as four unique relation property types.

4.3.1 Relation Characteristics

Because one object can relate to many different objects, of many different types, in many different ways, you should concentrate on one type of relation at a time. For example, a person can relate to one person as a spouse, to another person as a sibling, to a third person as a parent, and to a fourth as a child. Each of these relationships (spouse, sibling, parent, child) has a meaning separate from the others. By concentrating on one and ignoring the others, you break the definition problem into smaller, more easily understood pieces.

After you sort the relations according to their meaning, define how each relation is organized by listing how many objects can participate in the relation. There are three basic options:

- **1-to-1 relation**

  In this type of relationship, one object relates to no more than one other object. The second object cannot relate to other objects through a 1-to-1 relation. If you were modeling a family in a monogamous society, the "spouse" relation would be one-to-one, since a person can have only one spouse and the spouse cannot have additional spouses.

- **1-to-many** (or **1-to-*n***) **relation**

  An object in this type of relationship can relate to more than one object, but those related objects cannot relate to other objects using a 1-to-many relation. In a family, the "is-father-of" relation is a one-to-many relation, since a father can have more than one child but a child can have only one father.

- **many-to-many** (or ***n*-to-*m*) relation**

  This type of relationship allows objects from one set to relate in an arbitrary fashion to objects from another set, as described:

  - An object from the first set can relate to more than one object from the second set via the relation.
  - An object from the second set can relate to more than one object from the first set via the same type of relation.

  An example of a many-to-many relation taken from the family is "is-parent-of." Two objects from one set (the parents) relate to multiple objects from another set (the children) via this relation, and each object from the set of children can relate to two parent objects. In more familiar terms, a parent can have more than one child, and a child can have more than one parent.

After you have defined the meaning of the relation and its organization, you must define what types of objects can participate in the relation. For example, in the "is-spouse-of" relation, each participant is a human being. The same is true in the "is-parent-of", since both parents and children are humans. However, for the "is-father-of" relation, it is possible to characterize one participant more precisely: the father is a male human.

4.3.2 Relation Property Definitions

After you identify the various types of relations and define the organization and participating types of each, you can translate those definitions to property definitions. To do this, you must first understand how relation properties model relations between objects. (For more information on this subject, refer to Chapter 8 and Chapter 6.)

A relation property (or a closure property, which is similar) is one whose value results from following all the relationships (instances of RELATION or a subtype) of a specified type that associate one element (the element possessing the property) with other elements. For example, an element that modeled a human could have a **spouse** property whose value was found by following the "is-spouse-of" relationship to another human element. The actual value of the property is the element ID of this element.

If there are multiple relation participants, the value of the relation property is a scan property. A **scan property** is a scan of element IDs rather than a single element ID. For example, there can be more than one "is-parent-of" relationship emanating from a human element. If the **children** property is evaluated by following all such relationships, the resulting value of the property can consist of more than one element ID. The scan data type is designed for this purpose.

After you obtain the value of a scan property, access the element IDs in the scan sequentially, and remove or add element IDs as necessary. To add a relationship between the element possessing the property and the element corresponding to the added element ID, add an element ID to a scan, then change the property value. Conversely, to remove the corresponding relationship, remove an element ID from a scan, then change the property value. (See the *Oracle CDD/Repository Callable Interface Manual* for more information.)

To define a relation property between two elements, you need to specify the characteristics of the relation type you want to use. To understand how to specify relation characteristics, you must understand that a relationship has two participants, which are different and must be distinguished from each other. In Oracle CDD/Repository, relations are defined as having *owner* and *member* participant elements.

If you need to decide if a relation participant should be an owner or a member, consider the following:

- An element that is a member of a relationship cannot be deleted unless the relationship is first deleted.
- Deleting an element that is an owner of one or more relationships automatically deletes those relationships as well (but *not* the member elements).
- For dependency relation types, if the owner of the relationship is a versionable element, the relationship member cannot be changed unless the owner is reserved. (See Section 4.3.3 for information about dependency relation types.) This restriction does not apply for nondependency relation types or metadata.

Use these semantics to implement some types of data integrity constraints by making sure no element is deleted if another element refers to it.

A relation property follows the specified relationship either from the owner element to the member element, or from the member element to the owner element. This means that you can use a single relation in the definition of more than one property. For example, in the relation "is-parent-of," the relationship owner is the parent and the relationship member is the child. In this relation:

- On the element representing the parent, the **children** property follows "is-parent-of" from owner to member.

- On the element representing the child, the **parents** property follows "is-parent-of" from member to owner. (This is an example of inverse properties, which follow the same relation but in different directions.)

To define a relation property, you must specify the type of relation and the direction. To define a relation, specify what types of elements it can have as owner and as member. For the relation, also specify the types of organization in which it can participate.

To complete the example of family relations, define the relation types and the properties that follow them, as shown in the following tables. The first one provides the relation specifications and the second one shows how the properties use the relations.

Relation	Owner Type	Member Type	Multiple Relationships per Owner?[1]	Multiple Relationships per Member?[1]
is-spouse-of	WOMAN[2]	MAN	No	No
is-parent-of	HUMAN	HUMAN	Yes	Yes
is-father-of	MAN	HUMAN	Yes	No

[1]This information is required only if you define relation properties. (It is either an MCS_datatype_elementid or MCS_datatype_scan data type.)

[2]The choice of owner and member is arbitrary. Because "is-spouse-of" should be a symmetrical relation, the use of a relation having asymmetrical properties to model it is not entirely appropriate.

Property	Defined By	Relation Followed	Direction	Data Type
husband	WOMAN	is-spouse-of	To member	ELEMENTID
wife	MAN	is-spouse-of	To owner	ELEMENTID
children	HUMAN	is-parent-of	To member	SCAN
parents	HUMAN	is-parent-of	To owner	SCAN
father	HUMAN	is-father-of	To owner	ELEMENTID

4.3.3 Relation Characteristics Implementation

After you have specified the relations and associated properties, the final step in design is to translate these specifications into actual relation type definitions and property definitions.

Define a relation type as an instance of RELATION or one of its subtypes. RELATION and its subtypes are instances of RELATION_TYPE. To create a relation type, create a new instance of RELATION_TYPE and choose an existing relation type (as its supertype) to refine. Typically, you would choose between RELATION and its subtype DEPENDS_ON.

Any instance of DEPENDS_ON or one of its subtypes is a dependency relationship. In a dependency relationship, the owner of the relation is notified if the member is modified or deleted. A tool can use this notification mechanism to determine if a change has taken place and to act accordingly. If the owner of a dependency relationship is a versionable element, the relationship member cannot be modified unless the owner is reserved.

When you create a relation type, you supply values for several properties that specify its characteristics:

- **name** property

  Although individual relationships are unnamed elements, the relation *type* has a name.

- **legalOwners** and **legalMembers** properties

  These properties specify which element types can participate in the relation as owner and member, respectively. When you specify that a type can be a legal owner or legal member, you also are saying that any of its subtypes can participate in the relationship that way.

The following table shows how the family relations characterized in Section 4.3.2 translate into actual relation types:

Relation	Relation Type	legalOwners	legalMembers	Dependency	Supertype
is-spouse-of	IS_SPOUSE_OF	WOMAN	MAN	No	Relation
is-parent-of	IS_PARENT_OF	HUMAN	HUMAN	Yes	Depends On
is-father-of	IS_FATHER_OF	MAN	HUMAN	Yes	Depends On

4.3.4 Property Definitions for Relationships

Relationships are elements. Because they are elements, they can have properties. Define a property that contains information that pertains to the relationship on the relationship, not on the participants in the relationship.

For example, consider the "is-spouse-of" relationship outlined in preceding sections. It would be the logical place to define a **marriageDate** property. The marriage date does not describe the husband or the wife, but rather their relationship with each other.

There is no restriction on the types of properties that can be defined on relationships. Any type, including relation properties, is permissible.

4.4 Operations on Objects

To fully define an object type, you must list the operations that can be performed on instances of the type and describe the effects of each operation. In the object-oriented scheme, operations are very general in nature and can be applied to a broad range of object types. Each object type defines how it will respond to each operation.

4.4.1 Messages and Operations

To evaluate the operations required by your object types, you need to separate the operation from its implementation. Imagine that you are asking a skilled craftsman to build a table. You do not tell the woodworker *how* to build the table; he or she already knows the details. You simply make a general request and supply as much information as needed about the desired characteristics of the finished table.

Approach your listing of operations the same way. Find a verb that describes the operation in the most general way, and identify the information you need to supply to allow a "skilled craftsman" to carry out the request.

After you have listed the basic operations, see how they match up against Oracle CDD/Repository messages. (It is, of course, helpful to have some knowledge of the messages before you start this procedure.) The *Oracle CDD/Repository Information Model Volume I* manual contains a description of each message. Ideally, you should be able to match an existing message to each operation required by your object types. If no existing message meets your requirements, you can define a new message.

To understand a message, you need to know:

- Its general purpose, as stated in the beginning of the message description.
- Its required and optional arguments. (You can add arguments to existing messages.) The description lists the arguments recognized by Oracle CDD/Repository.
- Its precise effect on existing Oracle CDD/Repository element types. You must read the method descriptions contained in each message description for this information. You must also examine the inheritance structure for the methods, since many methods refine the method defined by the supertype.

How your required operations match up against existing messages can help you determine where to place new element types in the type hierarchy. Many messages are first recognized by an element type part way down the type hierarchy. Subtypes of this type then refine the message. For example, the messages associated with version control (RESERVE, REPLACE, and others) are first recognized by VERSION. Any object that must respond to these messages should be a subtype of VERSION or one of its subtypes.

4.4.2 Operation Arguments

If you find that an existing message matches with an operation you have identified, but that your operation requires more arguments than are defined for the message, you can add arguments. An argument specification consists of a unique name for the argument, a data type, and a general usage type (either "in," "out," or "in/out"). When you have decided upon these characteristics, implementation consists of the following steps:

1. Create a MSGARG element, specifying the name and the data type.
2. Add the MSGARG element to the value of the **argSpec** property on the MESSAGE element that represents the message. You specify the usage type as you do so. (The usage type is an attribute of an argument *when used with a particular message.* For example, a particular argument might be used as an in/out argument with one message but as input-only with another.)

4.4.3 New Operations

If, after some preliminary design cycles, you find that one of your operations does not match up with any existing message, you can create a new message to specify the operation. A message specification consists of a name for the message and a list of arguments that can be passed with the message. These arguments may already be defined, or you can create them as described in Section 4.4.2.

When you define an argument for a message that you create, you define both its usage type (as noted in Section 4.4.2) and whether or not it is a required argument when used with this message. You do not have the second option when adding arguments to messages defined by Oracle CDD/Repository. If you add a required argument, you disrupt operation of other applications that use that message.

4.4.4 Operation Implementation

When you have identified the messages that match up with operations on your object type, you next plan how to implement each operation. Before you can do this, you need to position the element types that will represent your objects in the type hierarchy. This allows you to determine how much of the implementation can be inherited from the supertype, and how much you will have to provide yourself.

To correctly place an element type in the type hierarchy and properly implement related operations, you must know how behavior is inherited by an element type. (Chapter 7 describes method inheritance, and the various types of methods, in detail.) (Refer to the *Oracle CDD/Repository Callable Interface Manual* for information about writing methods.) (Refer to the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals for descriptions of the messages and methods provided with Oracle CDD/Repository.)

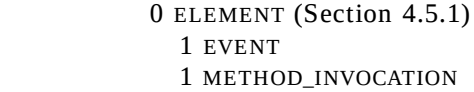
4.5 Type Hierarchy

To select a supertype for a new element type, you need to understand the purpose and the characteristics of the existing element types defined by Oracle CDD/Repository. Because your new type inherits characteristics from its supertype, the design of your new type is influenced by the range of supertypes from which you can select. (In other words, it is easier to work with the repository type hierarchy than against it.)

This section describes the predefined element types. The descriptions in this section are intended to give you a general idea of where each type fits in the overall repository scheme of things. Use these descriptions to begin placing new element types. For complete information about each type, see the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals.

Figure 4–1 illustrates the element type hierarchy, showing each type's supertype and subtypes. The number in parentheses indicates where in this section that type is described.

Figure 4–1 Element Type Hierarchy



(continued on next page)

Figure 4–1 (Cont.) Element Type Hierarchy

1 NAMED_ELEMENT (Section 4.5.2)
  2 CONTEXT (Section 4.5.3)
  2 PARTITION (Section 4.5.3)
  2 PERSISTENT_PROCESS (Section 4.5.3)
  2 VERSION (Section 4.5.4)
    3 AGGREGATE
      4 BINARY (Section 4.5.5)
        5 BINARY_TOOL
        5 TEXT (Section 4.5.5)
          6 TEXT_TOOL
      4 COMPOSITE (Section 4.5.6)
        5 COLLECTION (Section 4.5.6)
    3 MESSAGE
    3 MSGARG
    3 TOOL
      4 METHOD
    3 TYPE (Section 4.5.7)
      4 DATA_TYPE
      4 ELEMENT_TYPE (Section 4.5.7)
        5 RELATION_TYPE (Section 4.5.7)
      4 PROPERTY_TYPE (Section 4.5.7)
1 RELATION (Section 4.5.8)
  2 DEPENDS_ON (Section 4.5.8)
    3 COMPOSITE_PART
    3 HAS_DEFAULT_METHOD
    3 HAS_MSGARG
    3 HAS_PROPERTY
      4 HAS_COMPUTED_PROPERTY
      4 HAS_RELATION_PROPERTY
    3 HAS_RELATION
    3 HAS_SUPERTYPE
    3 IMPLEMENTS_METHOD
    3 IMPLEMENTS_RELATION
    3 INVOKES_TOOL
    3 RELATION_MEMBER
  2 HAS_CONTEXT
  2 HAS_CURR_COLLECTION
  2 HAS_DATATYPE
  2 HAS_METHOD

(continued on next page)

Figure 4–1 (Cont.) Element Type Hierarchy

2 HAS_PARENT
2 HAS_POSTAMBLE
2 HAS_PREAMBLE
2 HAS_RELATED_PARTITION
2 HAS_TOP_COLLECTION
2 OPENED_BY
2 RESERVED_BY

4.5.1 Elements

ELEMENT is the root of the element type hierarchy. Every new element type you define has ELEMENT as its ultimate supertype.

ELEMENT is an appropriate supertype for an element that is not a relation, but that represents objects that are not named or versioned.

4.5.2 Named Elements

NAMED_ELEMENT heads one of the major branches of the type hierarchy; it defines elements that have names. Named elements are used to represent "real" objects such as files, users, task definitions, tools, and so on. If you can give it a name, and if the user will want to refer to it by that name, then it should be a named element.

Element names must be unique in a repository, when fully qualified by the repository directory path. You need not check that a new named element has a unique name; just check for the appropriate error status when you create the element.

4.5.3 Contexts, Partitions, and Persistent Processes

Contexts, persistent processes, and partitions are elements that help you implement a configuration management system using Oracle CDD/Repository. Depending on the style of integration, your application may or may not need access to these types of elements. You are unlikely to create a subtype of one of these types, unless you are building a specialized repository user interface.

CONTEXT elements represent a user's view of the system under development. They identify a current configuration, limit the view of system elements to those that meet specified stability criteria, and provide a place for files currently being worked on by the user.

PARTITION elements represent levels of stability for replaced elements. A user can see a replaced element only if it is at or above the level of stability specified in the user's context.

PERSISTENT_PROCESS elements represent the user's current working environment. They identify a context and also store operating system information.

4.5.4 Versioned Elements

Versioned elements (the many subtypes of VERSION) represent objects that can exist in a succession of versions. All the versioned element types share some basic semantics, which are defined by the VERSION element type. These are the semantics of the Oracle CDD/Repository version and configuration management facilities as described in Chapter 9.

VERSION methods manage the basic arrangement of versioned elements in the repository in response to these messages. Any subtype of VERSION that refines these methods must include the definition of their supertype to ensure that version and configuration management takes place.

The decision of whether to make a new element type a versioned element is a basic one. If any of the following are true, you probably want a versioned element:

- The element represents an object that is part of a system.
- You need to be able to re-create the object as it existed at previous points in time.
- Several people may need to modify the object simultaneously without interfering with each other, then merge their modifications afterwards.
- The element represents a file (see Section 4.5.5).

Most of the remaining sections in this chapter describe versioned elements. Versioned elements share the basic version and configuration control semantics defined by VERSION.

4.5.5 Binary

The subtype BINARY represents files. These elements are versioned elements. BINARY refines the general version and configuration control methods defined by VERSION, and provides methods for additional messages appropriate to operations on files. If your application manipulates files, the elements that represent those files should be subtypes of either BINARY or TEXT, one of BINARY's subtypes.

Oracle CDD/Repository provides file management functions that relieve you of the task of managing files and file directories manually. Section 9.5 describes the file management functions.

TEXT elements represent ASCII text files. TEXT is an appropriate supertype for any file that is intended to be read and edited directly by users. For example, language source files should be subtypes of TEXT. TEXT is not a good choice for files that require special processing for viewing, editing, and printing. For example, you would not make a database file a subtype of TEXT, or a file containing a compound document. These files should be made direct subtypes of BINARY.

4.5.6 Composites and Collections

If an object is composed of other objects, such that if one of the included objects changes, the including object also changes, the including object is a composite object. Composites can have cycles in a graph formed by the composite's children and their children, but collections cannot.

COMPOSITE is the immediate supertype of COLLECTION. COLLECTION elements are used by Oracle CDD/Repository to represent system configurations. A collection is an element to which you can attach or detach versioned elements. A collection models your system by having the components of that system as its attached children. (Section 9.2 describes the uses and characteristics of collections.)

4.5.7 Types

Instances of TYPE and its subtypes represent repository types. Instances of these types are templates, which means they can be instantiated and control the characteristics of their instances. Because TYPE is a subtype of VERSION, repository types can exist in multiple versions, only one of which is in use (in the active schema) at a time.

It is not likely that you will need to create a subtype of TYPE or its subtypes. However, you create *instances* of these types when you extend the type hierarchy in the course of tool integration, as described in the following list:

- You create an instance of ELEMENT_TYPE to create a new element type.
- You create an instance of RELATION_TYPE to create a new relation type.
- You create an instance of PROPERTY_TYPE to create a new property.

4.5.8 Relations

Instances of RELATION and its subtypes, called relationships, associate repository elements. Properties that traverse specified relations are called relation properties. (Section 4.3 describes relations and relation properties.)

To define a new relation type to implement the desired characteristics of one or more relation properties, choose RELATION as a supertype. If you want to create a dependency relationship, choose DEPENDS_ON.

There are several subtypes of DEPENDS_ON that you can use to implement properties defined by Oracle CDD/Repository. You also might want to make a relation type a subtype of one of these.

For example, you want to define a VERSION subtype called MY_V and a COLLECTION subtype called MY_C. You want your MY_C instances to have a relation property that includes the attached instances of MY_V but not attached instances of other types. You also want the **hasChildren** property defined by Oracle CDD/Repository to include MY_V instances along with other attached versions.

To implement this example:

1. Determine how the **hasChildren** property is implemented. The immediate supertype of COLLECTION, COMPOSITE, defines the property. It is a relation property that traverses a relation type called COMPOSITE_PART.
2. Implement your property:
    a. Define a new relation type, MY_CP, that has MY_V as owner and MY_C as member *and that is a subtype of* COMPOSITE_PART.
    b. Define your property on MY_C to follow MY_CP relationships from owner to member. By virtue of the fact that MY_CP is a subtype of COMPOSITE_PART, the **hasChildren** properties (and other relation properties that follow COMPOSITE_PART) will also follow MY_CP relationships.

See Chapter 6 and Chapter 8 for complete information on relations and relation properties.

5

Schema Definition and Modification

Some elements in a repository define user data; Oracle CDD/Repository uses others (metadata elements) to control the following repository operations:

- creating new elements
- reading and setting property values
- invocating methods

This chapter provides an overview of how the repository specifies how elements of the various types relate to each other through its schema and how to modify that schema. The chapters that follow go into more detail about property definition, method invocation, and relation types.

5.1 Metadata Collection

The metadata elements that implement the repository's schema are members of a collection named CDD$METADATA. The collection resides in a repository directory named CDD$PROTOCOLS and in a partition named CDD$METADATA_PARTITION.

Use the following general procedure to implement a change in the schema for a particular repository:

1. Reserve the CDD$METADATA collection. This requires that you use a context whose **top** property is set to this collection. The context's **basePartition** property should be set to CDD$METADATA_PARTITION.
2. If you are creating new metadata elements, they will be attached to the reserved CDD$METADATA collection automatically. If you are modifying existing metadata elements, reserve them, modify them, and replace them.
3. Replace the CDD$METADATA collection into the CDD$METADATA_PARTITION partition. Your changes do not become effective until you do this.

For more information on modifying the repository schema, see the *Oracle CDD/Repository Callable Interface Manual*.

Note

If you create subtypes of metadata objects, you cannot use the new subtype as metadata as you would its supertype. For example, you cannot dispatch to a validation even though validation is a subtype of method.

5.2 Element Type Definition

Every repository stores the definition of its own element type hierarchy. Each element type is represented by an element whose type is ELEMENT_TYPE. An ELEMENT_TYPE element contains the "template" for new instances of that type. For example, to get a new TEXT instance, an application sends the NEW message to the ELEMENT_TYPE element whose name is **TEXT**.

As an analogy, consider a PASCAL record type definition (analogous to an ELEMENT_TYPE element) and the language's NEW procedure (analogous to the NEW message). Calling NEW creates a record, which can then be initialized, read, and modified according to its type. Similarly, sending NEW to a type creates a new instance of that type.

An ELEMENT_TYPE element must carry the following information:

- The name of the element type.
- The immediate supertype of the element type. Since the element type inherits methods and properties from its supertype, it must be able to identify the supertype.
- Properties defined by this type. These properties are joined with those inherited from the supertype.
- Methods defined or refined by this type.

Figure 5–1 illustrates how some ELEMENT_TYPE elements are arranged in the repository to define part of the type hierarchy. Each element has a pointer to its supertype, with the exception of **ELEMENT**, which has no supertype. These ELEMENT_TYPE elements and their supertype pointers make up the definition of the type hierarchy.

Note

Figure 5–1 shows the actual names of the ELEMENT_TYPE elements, which all begin with "MCS_". The names of metadata elements in the text of this manual omit the "MCS_" prefix.

Users that have sufficient privileges can manipulate ELEMENT_TYPE elements to extend the type hierarchy.

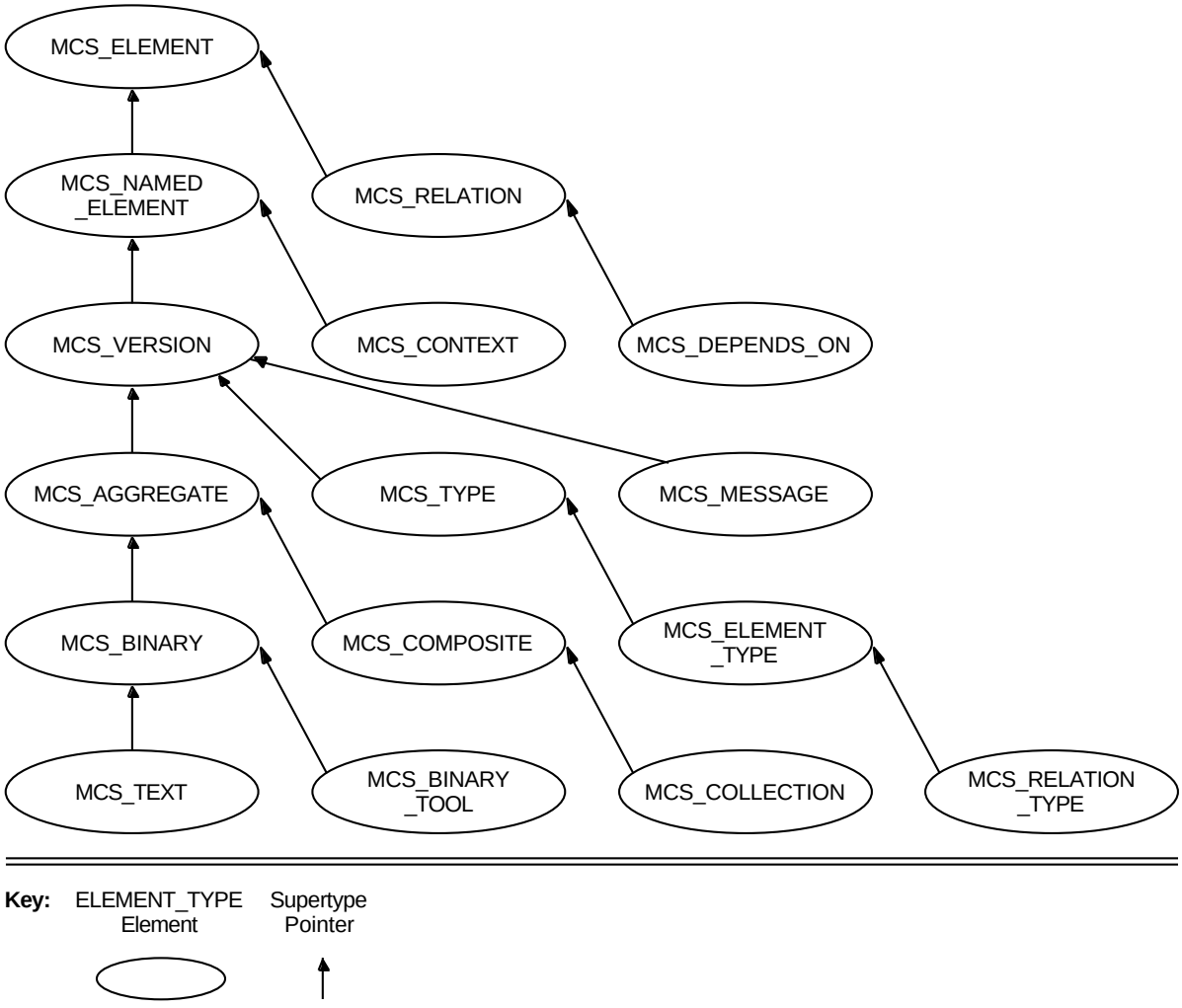
One ELEMENT_TYPE element is named **ELEMENT_TYPE**. It contains the template for elements that represent element types. To create a new element type, an integration procedure sends the NEW message to this element. The procedure must specify the new type's supertype, as well as other information that establishes the properties and methods particular to that element type. The resulting element represents the new element type in the extended type hierarchy. Sending NEW to it creates new elements of the type.

For example, you need to create an element type (MY_TEXT) that represents a specific type of text file. The integration procedure sends NEW to the ELEMENT_TYPE element named **ELEMENT_TYPE**, specifying:

- The name **MY_TEXT**. The resulting element then defines the element type MY_TEXT.
- The element type TEXT as the supertype. MY_TEXT source files are a specialization of text files.
- Any new properties that are specific to MY_TEXT files.
- Method refinements specific to MY_TEXT files.

After this new element type has been created, users can create new MY_TEXT elements to represent files by sending NEW to the ELEMENT_TYPE element named **MY_TEXT**. Because these elements are a subtype of TEXT, they will inherit file management capabilities (actually defined by BINARY) and version control semantics (defined by VERSION).

Figure 5–1 ELEMENT_TYPE **Elements in the Repository (Partial)**



5.3 Property and Method Definitions

If you create a new element type that identifies its supertype but does not identify any new properties or methods, you have defined a type that is functionally identical to its supertype. Instances of the new type have the same properties as instances of its supertype, and respond to the same messages in exactly the same way. By adding properties and methods, you make the subtype more specialized than the supertype.

Property definitions are represented by elements of type PROPERTY_TYPE. (Chapter 6 explains how these elements define properties.) Depending on the implementation type of the property, you insert the element ID of the PROPERTY_TYPE element into one of the following properties on the ELEMENT_TYPE element:

- For a normal property, modify the **propDef** property.
- For a relation or closure property, modify the **relPropDef** property.
- For a computed property, modify the **compPropDef** property.

Methods are represented by METHOD elements. (Chapter 7 shows how METHOD elements are associated with messages and describes the various types of method.) To add a method definition to an element type, you insert the element ID of the METHOD element into the value of the element type's **methods** property.

5.4 Relation Definitions

In addition to inheriting properties and methods from its supertype, an element type inherits ownership of or membership in relations. If an element type owns a relation type, then instances of that element type can own instances of the corresponding relation. Similarly, if an element type is a member of a relation type, then instances of the element type can be members of instances of the corresponding relation.

For an element type to have a relation property, it must either own, or be a member of, the relation that implements the property. An element type has relation ownership or membership either explicitly or through inheritance. To establish ownership of a relation, an element type includes the relation in its **ownsRelations** property. To establish membership in a relation, an element type includes the relation in its **relationMember** property. (See Chapter 8 for information about relations and relationships.) (See Chapter 6 for information about relation properties.)

6
Properties

This chapter describes the characteristics of properties and how they are defined by Oracle CDD/Repository.

Properties are named characteristics of elements. All elements of a given type have the same set of properties. The differing values you assign to an element's properties distinguish it from others of its type. For example, the value of an element's **name** property contains the element's name, which is different for each element.

The set of properties owned by an element type includes the properties it defines and the properties it inherits from its supertype. An element type needs to define only those properties that make it more specialized than its supertype.

The **name** property is an example of an inherited property. The NAMED_ELEMENT element type defines **name**. All direct and indirect subtypes of NAMED_ELEMENT inherit **name**.

A property's data type determines the data type of the property's values.

6.1 Property Data Types

The data stored in a property must have a type. Oracle CDD/Repository defines a number of data types, as listed in Table 6–1.

Table 6–1 Oracle CDD/Repository Data Types

Symbolic Name	Description
MCS_BOOLEAN	Boolean value
MCS_DOUBLE	Double-precision floating-point value
MCS_ELEMENTID	Element ID (elmID) of an element

(continued on next page)

Table 6–1 (Cont.) Oracle CDD/Repository Data Types

Symbolic Name	Description
MCS_FLOAT	Single-precision floating-point value
MCS_LIST	An ordered collection of data items
MCS_LONGINT	A 32-bit integer value
MCS_MEMBLOCK	A block of memory stored as a pointer to a memory block descriptor
MCS_SCAN	An unordered collection of element IDs (the first item in a scan sometimes has a special significance)
MCS_SMALLINT	A 16-bit integer value
MCS_STRING	A null-terminated character string
MCS_STRINGDSC	A character-string descriptor
MCS_VMSTIME	64-bit date-time value

Some of the data types allow storage of conventional data, such as string or numeric information. Other data types are specific to the needs of Oracle CDD/Repository. Two of these are ELEMENTID and SCAN. Properties with ELEMENTID values identify one other element. Properties with SCAN values identify a number of other elements.

6.2 Property Access Types

The access type of a property determines how and when you can use SETPROP to change the value of that property. A property that cannot be changed with SETPROP may still change as the result of another operation. For example, the value of the **numChildren** property changes as the result of ATTACH and DETACH messages sent to the COMPOSITE element that owns the property. Table 6–2 describes the property access types.

Table 6–2 Property Access Types

Symbolic Name	Description
MCS_ACCESS_READONLY	Read only
MCS_ACCESS_READWRITE	Read/write
MCS_ACCESS_WRITEONCE	Write one time only
MCS_ACCESS_WRITECREATE	Write once at creation

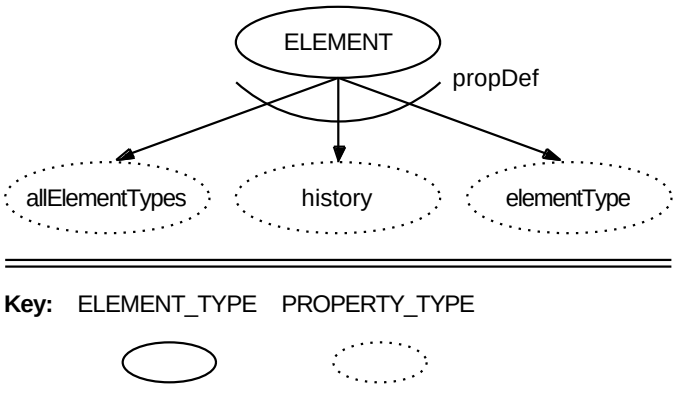
6.3 Property Definitions

You can define properties for elements and relations in your repository. An application can manipulate property definitions using the same object-oriented interface that it uses for other repository operations.

A property definition is represented by an instance of the PROPERTY_TYPE element. Properties on the instance determine basic characteristics of the defined property, such as its data type and access type. Other characteristics of the property are determined by the arrangement of the PROPERTY_TYPE instance in the repository.

You can determine all possible properties of a particular element or relation through the **propDef** property, which is defined by the element or relation's type. For example, the ELEMENT element type defines the **allElementTypes**, **history**, and **elementType** property types. The value of the ELEMENT **propDef** property is a scan of the element IDs for these property types, as shown in Figure 6–1.

Figure 6–1 The Value of the propDef Property



Properties are implemented in one of three ways, depending on the type of data stored in the property. Table 6–3 lists the property implementation types.

Table 6–3 Property Implementation Types

Implementation Type	Description
Normal	Actual data (or a pointer to actual data) is stored directly in the property.
Computed	The value of the property is computed at access time by invoking a method (see Section 2.1.4).
Relation	The value of the property is determined by traversing relationships between elements. Closure properties are similar to relation properties, except that the value is determined by traversing relationships recursively.

The remainder of this section shows how the three types of properties are defined in the repository.

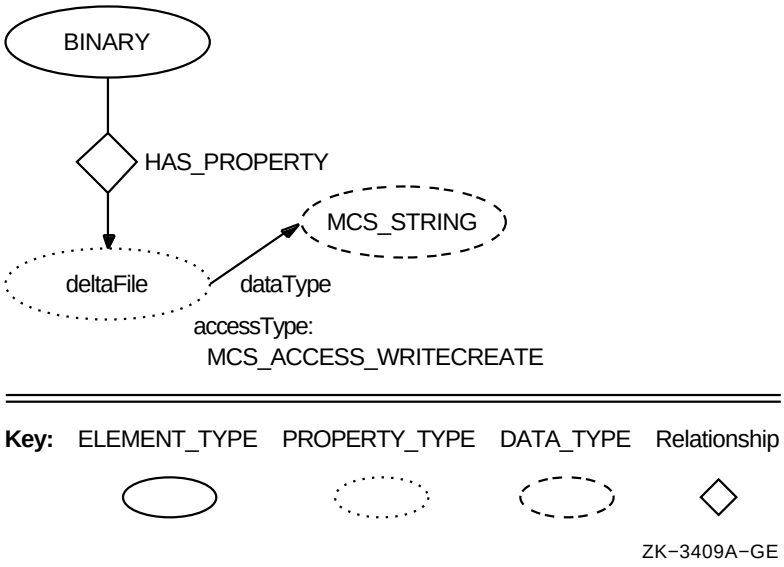
6.3.1 Normal Property Definitions

A normal property is one in which the data is stored with the element in the repository. A property whose value is a string, a number, or some other straightforward data type is generally a normal property. (An exception is a property whose value must be computed at the time it is requested. See Section 6.3.2 for an example.)

Figure 6–2 shows how a normal property, **deltaFile**, is defined on a BINARY element. A HAS_PROPERTY relationship connects **BINARY** to **deltaFile**.

Figure 6–2 introduces a new type of relation: HAS_PROPERTY. Relationships of this type and its subtypes connect ELEMENT_TYPE elements to PROPERTY_TYPE elements. (In fact, the value of **propDef** is formed by traversing HAS_PROPERTY relationships.)

Figure 6–2 Defining a Normal Property



The method invoked by a SETPROP or GETPROP message for a BINARY element's **deltaFile** property finds the definition of **deltaFile** associated with **BINARY**. Because the relationship connecting **BINARY** to **deltaFile** is a HAS_PROPERTY relationship, the method knows that the property is a normal property. It can then access the data directly, either to read it or (if allowed) to update it.

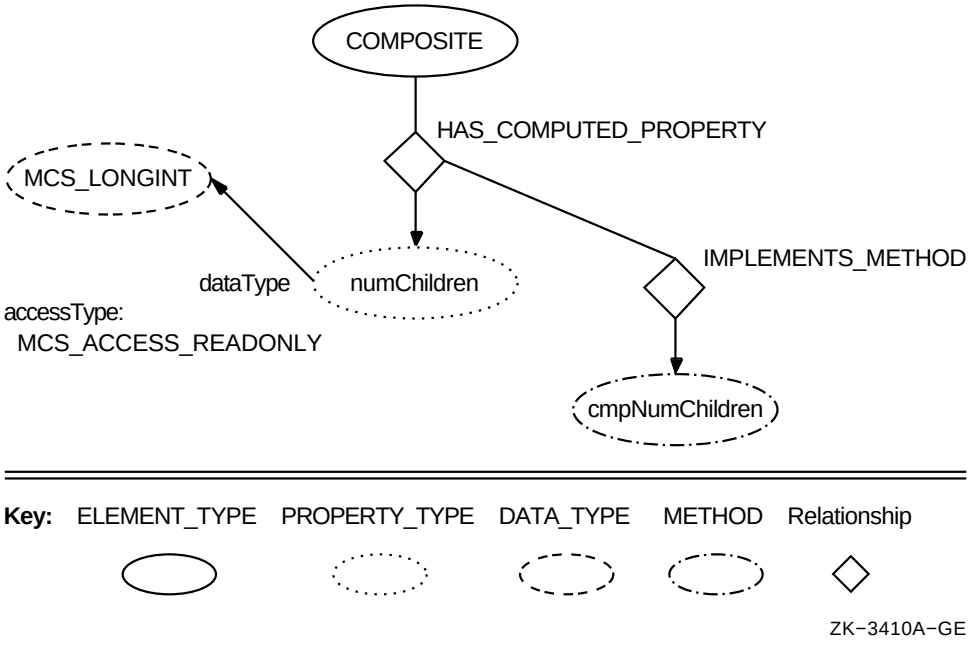
The **dataType** and **accessType** properties on the PROPERTY_TYPE element **deltaFile** determine the data type and access type of the property. The value of **dataType** is the element ID of the DATA_TYPE element that represents the data type, in this case **MCS_STRING**. The value of the **accessType** property is an integer constant that indicates that the property can be written only when you create the element.

6.3.2 Computed Property Definitions

Some property values must be computed at the time they are requested. For these properties, a computed property is used. If the value of a computed property is accessed, Oracle CDD/Repository invokes a method associated with the property. The method can do anything necessary to determine the value of the property.

numChildren on COMPOSITE elements is an example of a computed property. Figure 6–3 shows how this property is defined.

Figure 6–3 Defining a Computed Property

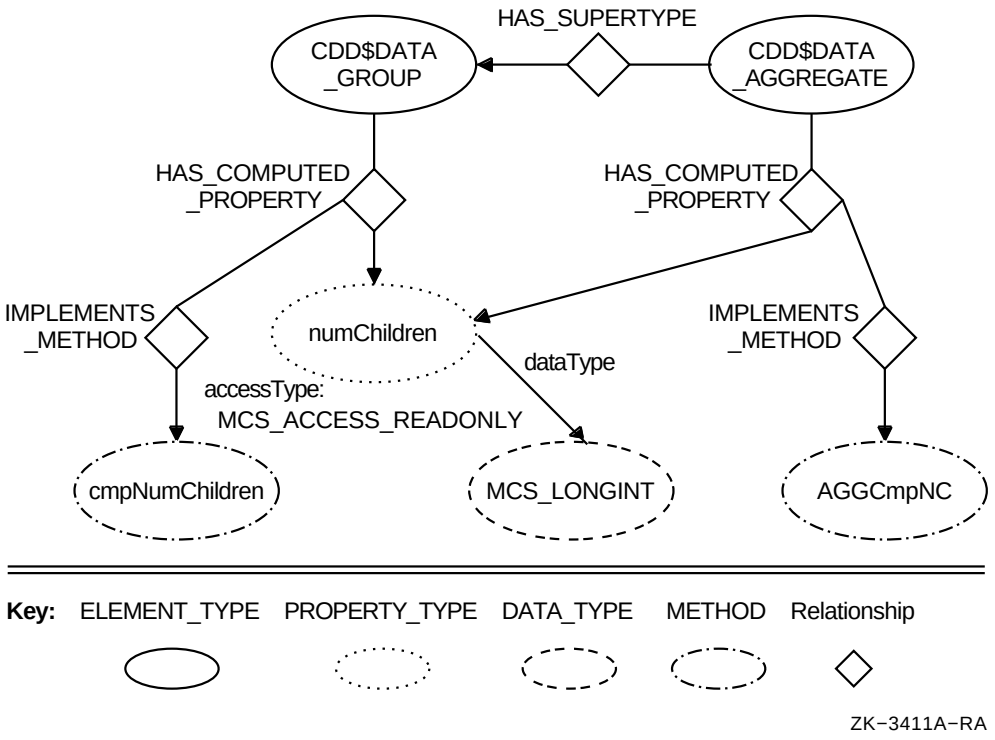


When a COMPOSITE element receives a GETPROP message for its **numChildren** property, the GETPROP method finds the definition of **numChildren** associated with **COMPOSITE**. Since the relationship connecting **COMPOSITE** to **numChildren** is of type HAS_COMPUTED_PROPERTY, the method function knows that this is a computed property. It therefore invokes the method attached to the HAS_COMPUTED_PROPERTY relationship. This method computes the number of versions attached to the composite and returns this value, which becomes the value of **numChildren**.

The **dataType** and **accessType** properties on the PROPERTY_TYPE element **numChildren** determine the data type and access type of the property. Note, however, that the method used to compute the property's value is attached to the HAS_COMPUTED_PROPERTY relationship, not to the PROPERTY_TYPE element. Therefore, the method that computes the property's value is independent of the property definition itself, except that the method must calculate a value of the correct data type.

For example, assume that the CDD$DATA_AGGREGATE element type (a subtype of CDD$DATA_GROUP) needed a different computation for **numChildren**. Instead of simply inheriting the property as defined by CDD$DATA_GROUP, CDD$DATA_AGGREGATE would establish its own connection to **numChildren** using a HAS_COMPUTED_PROPERTY relationship. The method attached to this relationship would be the method that computed **numChildren** for CDD$DATA_AGGREGATE elements, as shown in Figure 6–4.

Figure 6–4 Modification of a Computed Property by a Subtype



6.3.3 Relation Property Definitions

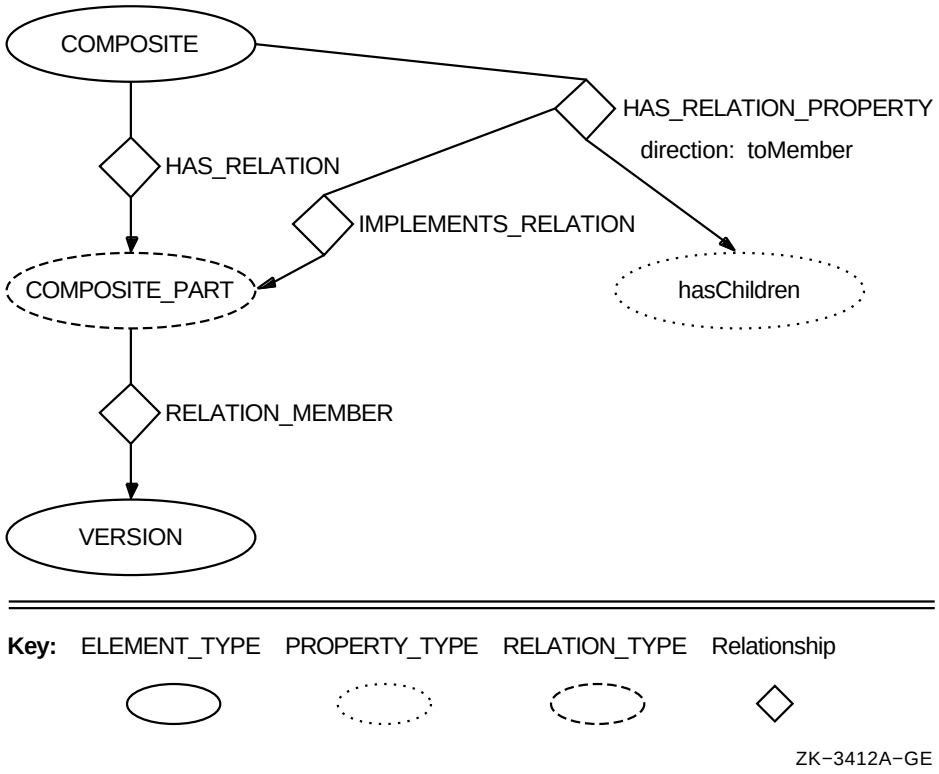
To understand how relation properties are defined, you must first understand how relations are defined (see Chapter 8). The value of a relation property (a scan or element ID) is formed by traversing all relationships of a certain type in a specified direction. Thus, a relation property must specify the type of relation to traverse (automatically including the relation's subtypes) and the direction in which to traverse.

To show how relation properties are defined, this section uses as an example the implementation of the various properties associated with composites. It uses Figure 8–4 as a starting point and adds the elements to the schema that implement the properties.

6.3.3.1 Relation Property Specification

This section shows how the composite subschema specifies the **hasChildren** property, which is a scan of the element IDs of all the immediate children of a COMPOSITE element. Figure 6–5 shows the composite subschema with the addition of the part that specifies **hasChildren**. (All properties shown in this and the following figures have the data type MCS_SCAN and are read-only. These characteristics, defined by properties on the PROPERTY_TYPE elements, are not shown in the figures.)

Figure 6–5 Composite Subschema: Specifying a Property



In a fashion similar to a normal or computed property, a HAS_RELATION_PROPERTY relationship links **COMPOSITE** to the PROPERTY_TYPE element **hasChildren**. The type of relationship used indicates that **hasChildren** is a relation property.

The HAS_RELATION_PROPERTY relationship has a property named **implementsRelation**, which points to the RELATION_TYPE element that defines the implementing relationship.

The HAS_RELATION_PROPERTY relationship has another property, **direction**, that defines the direction of traversal (see Section 8.3). The value of this property is either "toOwner" or "toMember." For **hasChildren**, the value is "toMember" since the composite owns the relationships and the children are their members.

A relation property is specified by giving the type of relationship to traverse (via the **implementsRelation** property) and the direction in which to traverse it (via the **direction** property). This information, along with the starting point for the traversal, is sufficient for Oracle CDD/Repository to form the scan that is the property's value.

A relation (such as COMPOSITE_PART) can take part in relation property definitions that traverse the relationship from owner to member and from member to owner. In the composite subschema, the **hasParents** property on VERSION elements traverses COMPOSITE_PART relationships from member to owner. (A third property, **allChildren**, also traverses COMPOSITE_PART. See Section 6.3.3.2 for details.) Figure 6–6 adds the elements needed to define the **hasParents** property. Note that the value of the **direction** property for the new HAS_RELATION_PROPERTY relationship is "toOwner," indicating that **hasParents** is a scan of the owners of the relationships.

6.3.3.2 Closure Property Specification

A **closure property** is similar to a relation property except that the relationships are traversed recursively, not once as with a relation property. The value of the property is thus the transitive closure of the elements found when traversing a relationship. **allChildren** is an example of a closure property. Its value is the immediate children of a composite, their children (if any), and so on until all the leaf nodes have been found.

The implementation of closure properties is identical to the implementation of relation properties, with one exception: the **direction** property on the instance of HAS_RELATION_PROPERTY is set to "toAllOwners" or "toAllMembers" instead of "toOwner" or "toMember." This setting forces the recursive traversal of the relation.

Figure 6–6 Composite Subschema: Adding Another Property

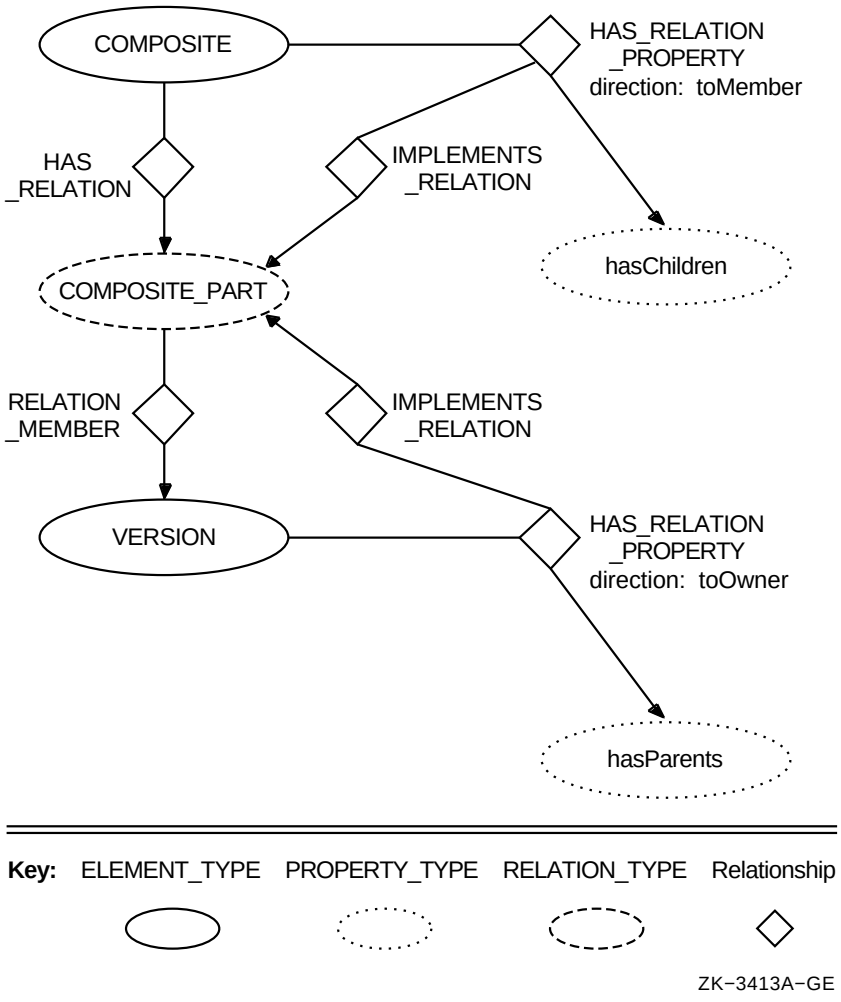
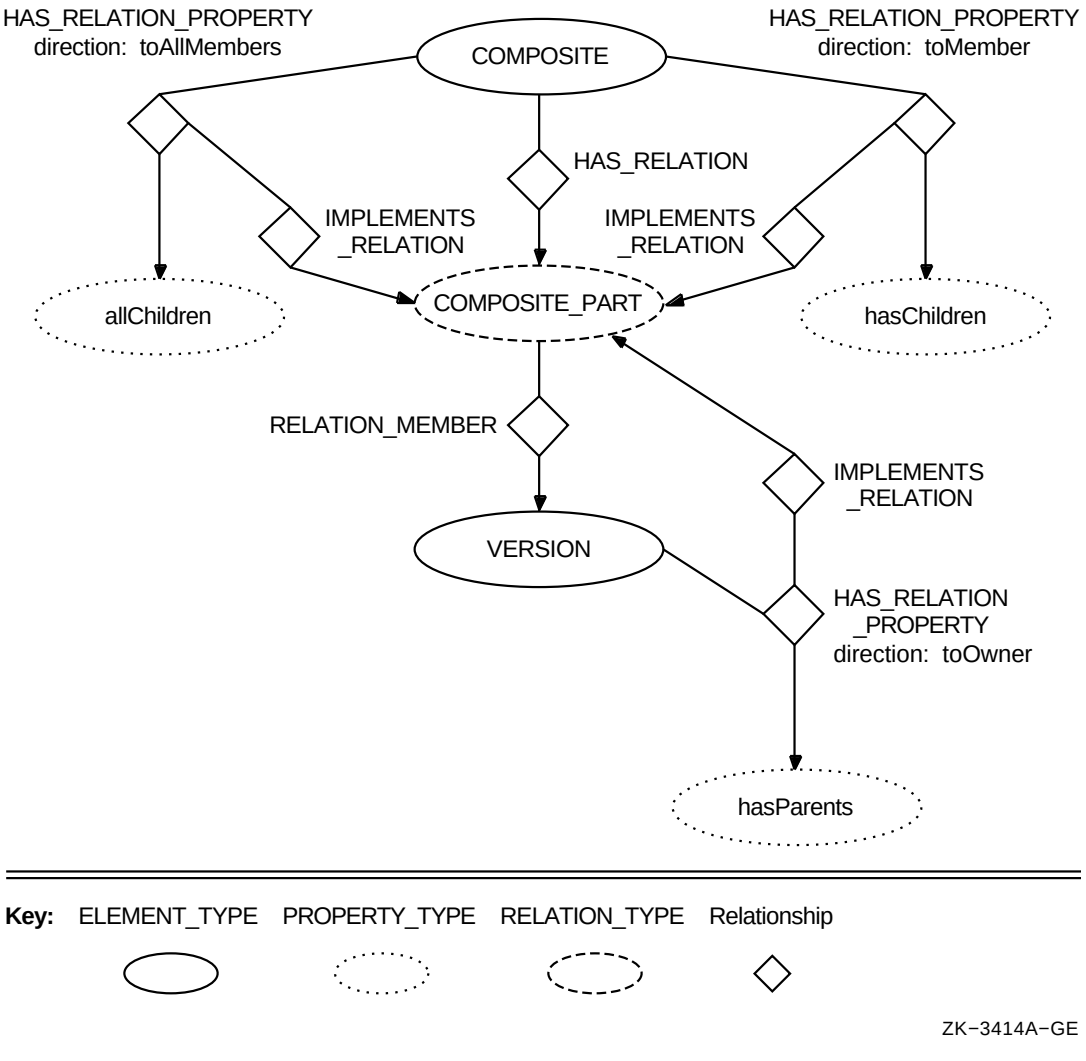


Figure 6–7 shows the composite subschema with the addition of **allChildren**.

Figure 6–7 Composite Subschema: Adding a Closure Property



ZK–3414A–GE

6.3.3.3 Example: The Value of hasChildren

Use the composite subschema whenever you examine (using GETPROP) or modify the value of the **hasChildren** property. (You cannot modify the value of **hasChildren** directly with SETPROP, but you can modify it indirectly by attaching an element to a composite.) When you attach a version to a composite, the composite subschema provides the following general information:

- It indicates the type of relationship that is used to attach the version to the composite.
- It indicates that the COMPOSITE element is the owner, and the VERSION element is the member.
- It ensures that only elements of the correct type (or their subtypes) participate in the relationship. If the types are incorrect, the modification attempt will fail.

Modifying the value of **hasChildren** by adding a new member to the composite therefore has the effect of checking the element types of the elements involved, then creating a new relationship linking the elements.

For a detailed example of how Oracle CDD/Repository uses the composite subschema, suppose that a COLLECTION element named **coll_1** receives the GETPROP message for its **hasChildren** property.

1. The method dispatcher first follows **coll_1**'s **elementType** property to the ELEMENT_TYPE element named **COLLECTION**. Through this element's supertype, COMPOSITE, the method dispatcher has access to the elements that make up the composite subschema. The method dispatcher invokes the method function that implements GETPROP for COMPOSITE elements.
2. The **propDef** property of **COMPOSITE** is a scan of the properties defined by COMPOSITE elements. The method function finds the property definition with the name **hasChildren**.
3. Since the relationship linking **COMPOSITE** to **hasChildren** is a HAS_RELATION_PROPERTY relationship, the method function knows that **hasChildren** is a relation property. The method function therefore follows the IMPLEMENTS_RELATION relation indicated by the **implementsRelation** property to determine what relation implements the property.
4. The RELATION_TYPE element named **COMPOSITE_PART** indicates that COMPOSITE_PART relationships implement the **hasChildren** property. The value of the **direction** property on the HAS_RELATION_PROPERTY relationship is "toMember," indicating that the direction of the relationship traversal is from owner to member.

5. The method function now knows enough to ask the database to form a scan of the element IDs of all members of COMPOSITE_PART relationships owned by **coll_1**. This is the value of **hasChildren** that is returned in response to the message.

Note

The scan contains not only the results of following COMPOSITE_PART relationships but also any relationships that are subtypes of COMPOSITE_PART.

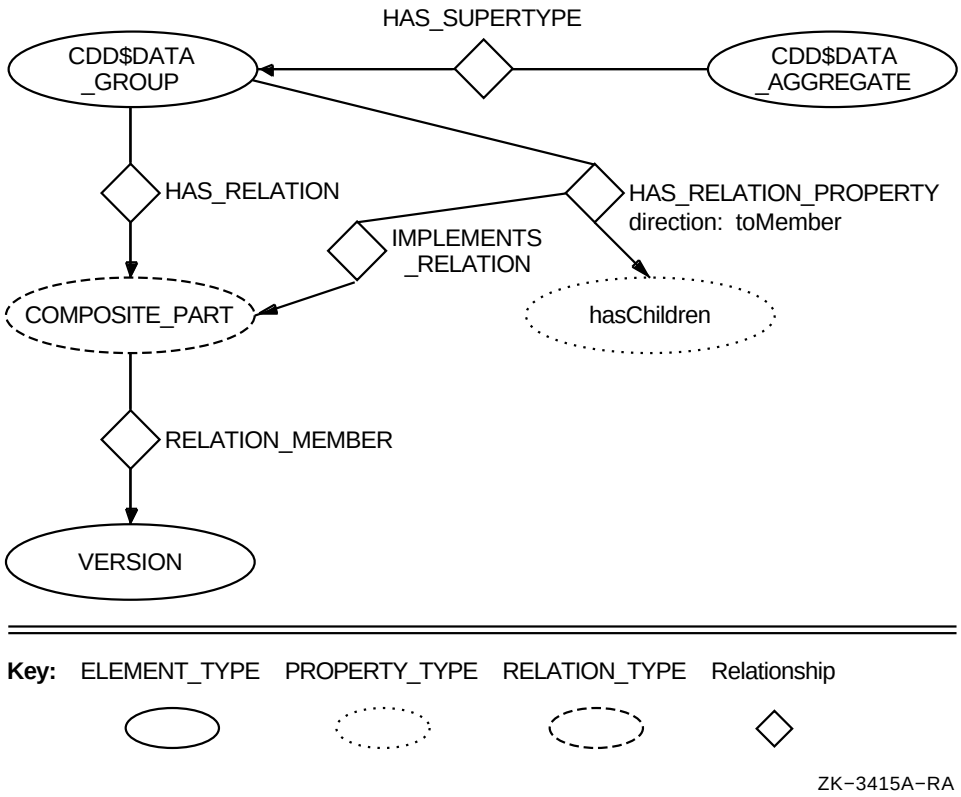
6.3.3.4 Relation Property Inheritance

Element subtypes inherit the properties owned by their supertypes, including relation properties. However, with relation properties, a subtype can constrain the element types that may appear in a relation property more strictly than the supertype.

As an example, consider CDD$DATA_AGGREGATE, which is a subtype of CDD$DATA_GROUP and which represents a record definition. While CDD$DATA_GROUP can have any subtypes of VERSION as its members, a record definition should contain only certain VERSION subtypes, such as field definitions and other record definitions. Field definitions are represented by elements of type CDD$DATA_ELEMENT and subtypes. This example shows how CDD$DATA_AGGREGATE can be constrained to contain only CDD$DATA_ELEMENT and subtype elements.

Figure 6–8 shows the schema that Oracle CDD/Repository would use for the **hasParents** property on CDD$DATA_AGGREGATE if it did *not* want to constrain the members of record definitions to be fields. The ELEMENT_TYPE element **CDD$DATA_AGGREGATE** is linked to **CDD$DATA_GROUP** by a HAS_SUPERTYPE relationship. This indicates that CDD$DATA_AGGREGATE is a subtype of CDD$DATA_GROUP.

Figure 6–8 Inheriting hasChildren Without Constraints



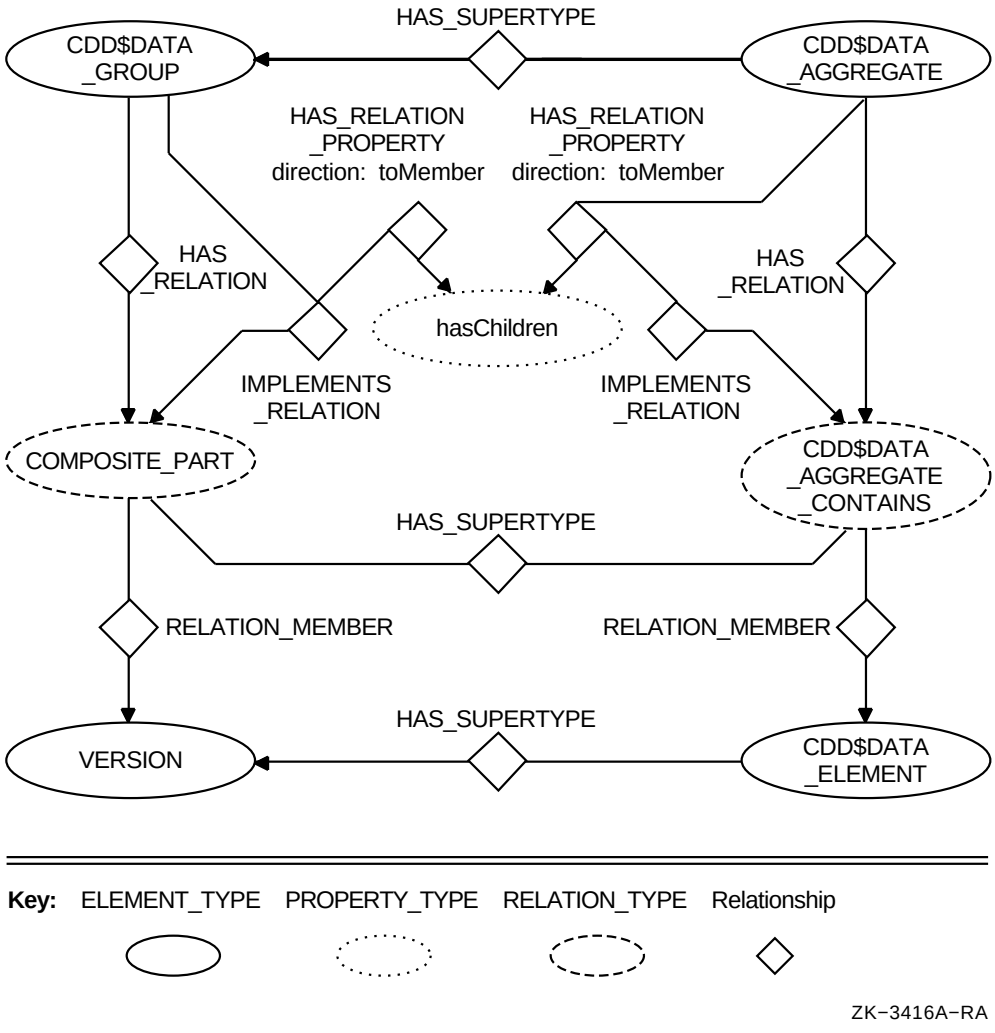
A GETPROP message to a CDD$DATA_AGGREGATE element finds no **hasChildren** property associated directly with CDD$DATA_AGGREGATE, so it follows the HAS_SUPERTYPE relationship to **CDD$DATA_GROUP**. CDD$DATA_GROUP does define the **hasChildren** property, and it is this definition that is used. However, CDD$DATA_GROUP's definition of **hasChildren** allows VERSION elements to be included. This means that an attempt to attach an arbitrary VERSION element to a record definition would succeed, given the schema in Figure 6–8. Such an attempt should fail.

To constrain record definitions to contain only CDD$DATA_ELEMENT elements and subtypes, the schema is modified to look like Figure 6–9, as described in the following list:

- A new relation type, CDD$DATA_AGGREGATE_CONTAINS, is introduced to link CDD$DATA_AGGREGATE elements to CDD$DATA_ELEMENT elements. CDD$DATA_AGGREGATE_CONTAINS is a subtype of COMPOSITE_PART.

- The CDD$DATA_AGGREGATE element type is associated with the **hasChildren** property through a HAS_RELATION_PROPERTY relationship.

Figure 6–9 Inheriting hasChildren with Constraints



ZK–3416A–RA

A GETPROP, DETACH, or ATTACH operation on a CDD$DATA_AGGREGATE element finds the definition of **hasChildren** that constrains the members of record definitions to be CDD$DATA_ELEMENT elements or subtypes. An attempt to attach an element that is not a field definition (or subtype) to a record definition fails.

You also can include COMPOSITE_PART as one of the relations of CDD$DATA_AGGREGATE by creating an explicit HAS_RELATION connection to it.

7
Messages and Methods

If you extend the type hierarchy, you often need to define new methods. These methods respond to messages sent to instances of the new type and perform specialized processing required by the type. You also may need to define new messages to represent operations on the new element types that are not adequately described by existing messages.

This chapter first describes how a message sent to an element in the repository invokes a particular method, then explains the various types of method functions.

7.1 Method Invocation

This section shows how Oracle CDD/Repository selects the correct method to invoke in response to a message. You must understand this mechanism to understand how to make your methods available to Oracle CDD/Repository.

There are three element types involved in the invocation of a method:

- An instance of MESSAGE, which represents the interface (message name and a set of arguments) to a method.
- An instance of METHOD, which represents the method body (the action to be executed in response to a message). The action may be code that you write or one of a set of predefined actions.
- An instance of ELEMENT_TYPE, which represents an element type.

Methods are invoked by sending a message to an element. To send a message, a tool calls the dispatch routine MCS_dispatch_op, as shown in the following example:

```
MCS_dispatch_op(element, message, arglist)
```

The call to a dispatch routine does not directly identify the method to be executed. Rather, it identifies the message (by its name) and the element to which the message is sent (by its element ID). The call also contains arguments whose number, type, and meaning are defined by the message being sent.

Every element has an **elementType** property whose value is the element ID of the ELEMENT_TYPE element that represents the element's type. The dispatch routine reads the **elementType** property to determine the element's type.

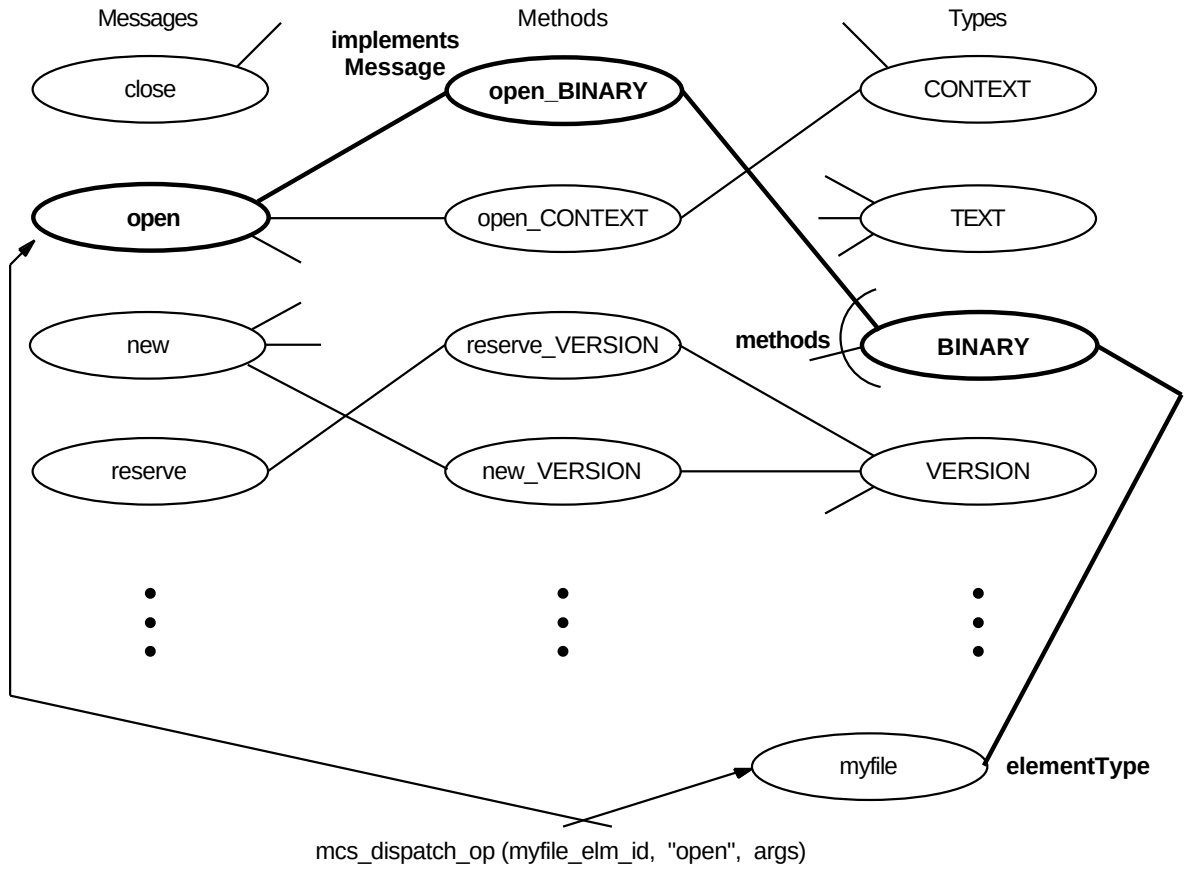
Given the type and the message, the dispatch routine attempts to find a method connected to both the MESSAGE instance and the ELEMENT_TYPE instance. Figure 7–1 illustrates this procedure for the OPEN message invoked on an element called **myfile**. The dispatch routine first follows the **elementType** property of **myfile** to find its type, which is BINARY. The dispatch routine knows that the message is OPEN. To determine which method to invoke, the dispatch routine checks the **methods** property of the ELEMENT_TYPE element **BINARY**, looking for the method that implements the OPEN message. The method that connects with both **BINARY** and **open** is named **open_BINARY**, and this is the method that the dispatch routine invokes.

Note

The name of the method, **open_BINARY** in this case, is by convention derived from the names of the message and the element type. The choice of name is arbitrary. The name has no influence on method selection.

Because there is no reference to the method in the calling code, you can replace the method without any required changes to the calling code. Also, the calling code does not need to know anything more about an element than its ability to respond to the message. The calling interface is the same, no matter what the type of the element (see Figure 7–1).

Figure 7–1 Method Selection



ZK–3417A–GE

Element type CONTEXT implements its own method (**open_CONTEXT**) for the OPEN message. Thus, the OPEN message sent to an element of type CONTEXT invokes the OPEN_CONTEXT method, while the same message sent to an element of type BINARY invokes the OPEN_BINARY method. However, *the form of the message* — the call to MCS_dispatch_op — *is exactly the same* in either case. There is only one instance of the MESSAGE element type named **open**, and it is this element that determines the interface presented by the OPEN message.

Type inheritance also influences the method to execute. With type inheritance, the application designer needs to implement a method for a message/type combination only when the required action differs from, or adds to, the method supplied by the supertype. Type inheritance offers the following advantages:

- Unnecessary duplication of code is eliminated.
- If the method for a type changes, the change is automatically inherited by the type's subtypes.

If the type of an element passed as a parameter to the dispatch routine does not have a method that implements the specified message, Oracle CDD/Repository examines the type's supertype. In Figure 7–2, element type TEXT does not implement a method for the OPEN message. Because of this, the system determines TEXT's supertype (BINARY), and checks if it implements the message. Since it does, sending the OPEN message to an element of type TEXT selects the OPEN_BINARY method. Again, the interface presented by the OPEN message is the same for TEXT elements as for BINARY elements. If the TEXT element type implements its own method for OPEN at some later time, the calling code does not have to change.

Figure 7–2 Method Selection and Inheritance

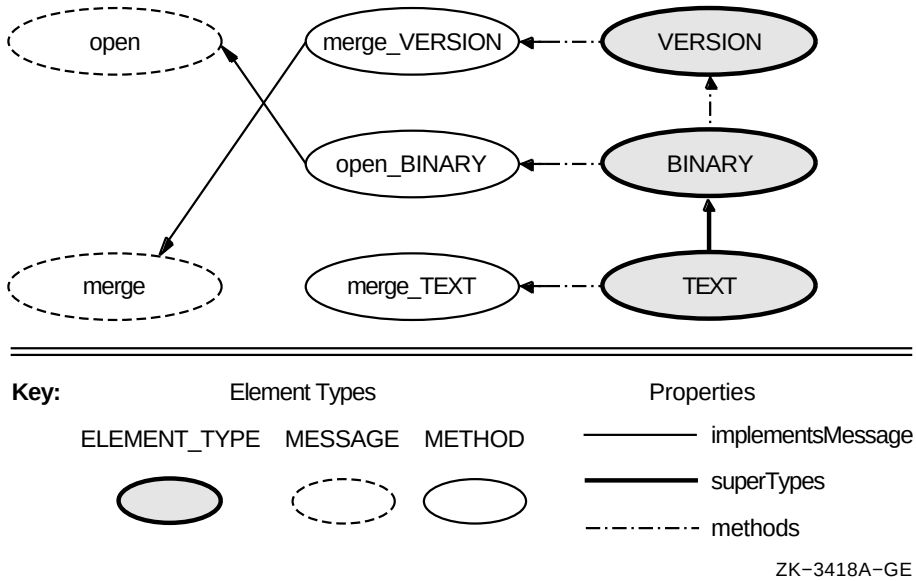


Figure 7–2 shows another message, MERGE, which is implemented by methods **merge_TEXT** and **merge_VERSION**. The method selected for a MERGE message depends on the type of element to which the message is sent, as described in the following list:

- If the message is sent to a TEXT element, the method **merge_TEXT** is selected.
- If the message is sent to a BINARY or VERSION element, the method **merge_VERSION** is selected.

The **merge_TEXT** method can invoke the **merge_VERSION** method at some point during its processing by calling the MCS_dispatch_superOp routine. If it does not, the **merge_VERSION** method will not execute when the MERGE message is sent to a TEXT element. Since any merge operation must include the processing provided by **merge_VERSION**, the routine that implements **merge_TEXT** does in fact call MCS_dispatch_superOp. In Oracle CDD/Repository, almost all refined methods invoke their supertype methods in this way.

Some messages are not implemented for an element type or any of its supertypes. An example in Figure 7–2 is the OPEN message sent to a VERSION element. Since neither VERSION nor any of its supertypes provides a method for OPEN, the message fails.

7.2 Message Arguments

The third argument to the MCS_dispatch_op routine contains the **dispatch list**. The dispatch list is an argument list, which includes arguments that accompany the message and provide additional information. Arguments in the dispatch list can also be used by Oracle CDD/Repository to return information to the sender of the message. For example, the NEW message returns the element ID of the created element to the sender in an output argument.

A single argument can be used by more than one message. For example, the *new_inst_elmID* argument is used by the NEW and RESERVE messages to return the element ID of the new element.

There are two characteristics associated with a particular message's use of an argument:

- whether or not the message requires that argument
- how the message uses the argument: as input, as output, or as both input and output

The following example explains how messages can make different uses of the same argument. The ATTACH message requires the *collection_elmID* argument indicating the collection to which to attach. The NEW message accepts the same argument, but it is optional. (Section 7.2.2 shows how this differentiation is accomplished.)

7.2.1 Dispatch List Format and Use

In a dispatch list each entry identifies one argument for the message. The entry, called an argument specifier, contains the following information:

- the name of the argument (Oracle CDD/Repository provides symbolic identifiers for argument names)
- a value or pointer to a variable
- a status field that the caller and Oracle CDD/Repository can use to exchange status information

There are special routines to create, modify, and read the contents of any argument list, including a dispatch list. (See the *Oracle CDD/Repository Callable Interface Manual* for information on using these routines.)

The sender of a message can put any arguments in the dispatch list. However, the method(s) that implement the message only use arguments that are required or optional for that message. If any required arguments are omitted, the dispatch fails.

One dispatch list argument, *arglist*, is also an argument list. It is used with the NEW and SETPROP messages to supply property values to set, and with the GETPROP message to return property values. The format of the *arglist* argument is the same as the format of a dispatch list, except that the argument specifiers in the list contain the names of properties, not message arguments. Using a list allows multiple property values to be set or read in one operation.

In the *arglist* argument, Oracle CDD/Repository makes the following use of the status field in each entry:

- If a problem occurs setting or reading a property value, Oracle CDD/Repository places the status code indicating the reason for the failure in the status field of the corresponding entry. This allows the message sender to determine which property caused the problem. (A GETPROP message may succeed even though one or more of the property values was not read successfully. A SETPROP or NEW message is considered to have failed unless all the property values are set successfully.)

- If the property value has never been set, the status field is filled in with the status MISSING. This is different from the property having been set to a null or zero value; those values may have meaning. If the sender of SETPROP sets the status field to MISSING, the corresponding property is made to seem as if it had never been set.

7.2.2 Message Argument Implementation

Message arguments are represented in the repository by elements of type MSGARG. The MSGARG element contains the following information about the message argument:

- The name of the argument. The elment's **name** property holds the argument name.
- The data type of the argument. The element's **dataType** property identifies the argument's data type.

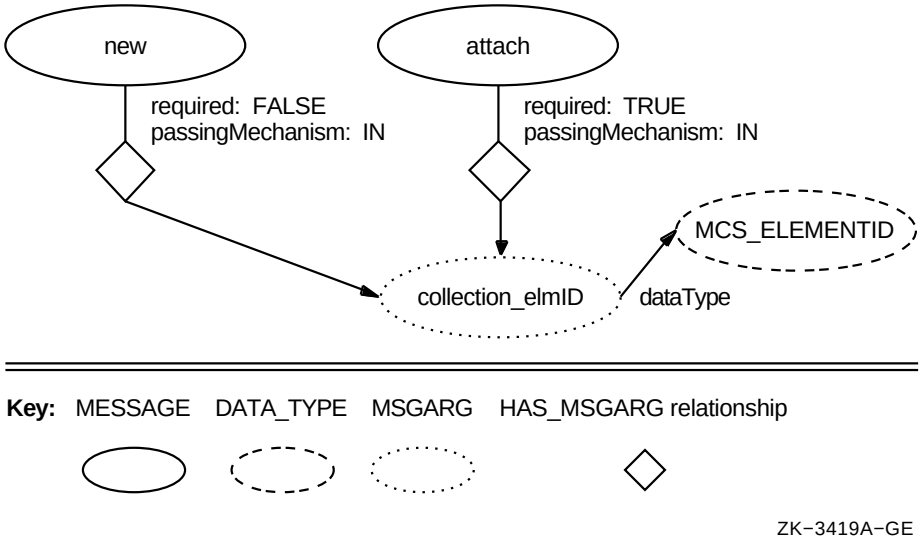
A MSGARG element is associated with the MESSAGE element that represents a message by a HAS_MSGARG relationship. The relationship carries information about the use of the message argument by that message as follows:

- The relationship's **required** property indicates whether the argument is required by that message.
- The relationship's **passingMechanism** property indicates whether the message uses the argument as an input argument, an output argument, or both.

A MESSAGE element has a scan property, **argSpec**, whose value is the MSGARG elements representing the message's arguments. The value of this property is formed by following the HAS_MSGARG relationships. If you store the information (previously described) on the relationships, rather than on the MSGARG elements, you can have different messages use the same argument in different ways.

Figure 7–3 shows how the NEW and ATTACH messages share the *collection_elmID* argument. Properties on the HAS_MSGARG relationships indicate that ATTACH requires the argument, while NEW does not. Both messages use the argument as input.

Figure 7–3 Associating Messages with Message Arguments



7.3 METHOD Elements and Method Functions

Up to this point, *method* has referred to elements of type METHOD. These are the elements selected in the process described in Section 7.1. These elements *represent* a desired action. The term **method function** describes the actual *behavior* of a method. It is important to distinguish between the method (an element) and the method function (a behavior invoked when the method is selected).

For example, the OPEN message sent to a BINARY element results in the selection of a METHOD element named **open_BINARY**. The corresponding method function is a routine named open_BINARY. (The name of the routine need not match the name of the element.) The element **open_BINARY** represents the method function open_BINARY. The routine open_BINARY specifies the action taken when you select **open_BINARY**.

There are several different types of method functions. Some of these provide access to code; others specify a certain action and do not require that code be written. The value of the **funcType** property on METHOD elements indicates which type of method function is used. The following sections describe each type and the general means by which they produce an action.

7.3.1 External Code Methods

External code methods are routines that are called directly by the message dispatch mechanism. External code methods include the following characteristics:

- They are called with a standard sequence of arguments:
  1. the element ID of the element to which the message was sent
  2. the dispatch list that was sent with the message
- They can use element IDs that you pass in; for example, one of the arguments they receive is the element ID of the element that received the message.
- They can call the MCS_dispatch_superOp routine to invoke a method defined by a supertype of the element type with which they are associated. You can use external code methods to refine a supertype's method.

External code methods reside in shareable images that are activated the first time the method dispatcher calls a method routine they contain. The methods provided with Oracle CDD/Repository are internal and are included in the same shareable image, CDDSHR, as are the method dispatcher and the routines that constitute the callable interface. Your application links with this shareable image to gain access to the Oracle CDD/Repository routines and methods. Your application also provides a shareable image to make its own methods available to other applications. Because these methods are in a shareable image other than CDDSHR, they are external code methods.

The interface between a METHOD element and an external code method is a BINARY_TOOL element. The METHOD element's **invokes** property contains the element ID of the appropriate BINARY_TOOL element. The METHOD element's **invocationString** property contains the routine name. The message's argument list describes the calling sequence for the routine, and the arguments supplied with the message form the arguments for the call to the routine.

7.3.2 External Programs

External programs are invoked as separate processes through a mechanism appropriate for the host platform. The external program can be an executable image or a command script.

External program methods differ from external code methods in that they do not execute in the same repository session as applications sending messages that start them. (In fact, an external program need not start a repository session at all.) Therefore, an external program cannot share element IDs with

the invoking application. This also means that an external program does not execute in the same transaction as its invoker.

The interface between a METHOD element and an external program is either a BINARY_TOOL element (for an executable image) or a TEXT_TOOL element (for a command script). The METHOD element's **invokes** property contains the element ID of the appropriate BINARY_TOOL or TEXT_TOOL element. The METHOD element's **invocationString** property contains a template for the command line used to invoke the program. The method dispatcher fills in the template from information contained in the message, and uses the resulting command line to start the application in a separate process.

7.3.3 Null Functions

Use **Null functions** to define METHOD elements that have no function body; in essence, they do nothing and do not invoke the supertype's method. These can serve as placeholders for preambles or postambles (see Section 7.5).

7.3.4 Superop Functions

Superop functions automatically invoke the method defined for an element type's supertype. A superop function is equivalent to an external code method function that simply calls MCS_dispatch_superOp and exits. Defining the method as a superop method saves you the trouble of writing the routine and is more efficient to execute. Superop functions are convenient placeholders for preambles and postambles.

7.3.5 Illegal Functions

Use **Illegal functions** to disallow a message for an element type and its subtypes. Illegal functions result in an error indicating that no method function was specified for the message.

7.4 Method Refinement

Because a subtype is a specialization of its supertype, methods and tools that operate on the supertype can also operate on the subtype, although not with the degree of specialized service that one might want. For example, consider the method function that implements the REPLACE message for VERSION elements. This method function manipulates elements and properties in the repository to maintain the correct relationships of versions, branches, and collections. This procedure is the same for all the subtypes of VERSION, although the subtypes may require additional processing to implement REPLACE correctly. For example, the BINARY element type refines VERSION's REPLACE method by storing the modified file in a delta format.

There is no need for a type's method functions to reimplement procedures provided by the method functions of its supertypes. Instead, a type can build on its supertypes' procedures to provide the proper level of specialized service for messages sent to elements of that type. This process is called method refinement.

The MCS_dispatch_superOp routine makes the method of a type's supertype(s) available to the type's method function. A method function can perform work that is specific to its type, then call MCS_dispatch_superOp to invoke the supertype's method for that message. When MCS_dispatch_superOp returns, indicating that the supertype's method has done its job, the method function can perform additional specialized work, if necessary, before it returns. Because the method of a supertype can call MCS_dispatch_superOp, a method belonging to a type lower in the type hierarchy can indirectly invoke methods of its supertypes several levels up.

Of course, a method function is not required to use its supertype's method. In some cases that may not be appropriate. Possibilities in addition to method refinement include the following:

- The message may be newly defined for this element type. In this case the supertype does not recognize the message corresponding to the method, so trying to call MCS_dispatch_superOp returns an error. The method function must fully define the method.
- The method function may *redefine* the method, providing all the service needed without using the supertype's method.
- The method function may *disallow* the message. In this case, elements of the type respond to the message with a failure code indicating that no method is defined for the message.

Finally, no method may exist for a given message/type combination. In this case, the message dispatch routine examines the type's supertype for a method to invoke, as described in Section 7.1.

7.5 Preambles and Postambles

A METHOD element can define a set of **preambles** and a set of **postambles**. Preambles are methods that are invoked *before* the method to which they are attached. Postambles are methods invoked *after* the method to which they are attached. The order in which preamble methods and postamble methods are invoked (relative to other preamble and postamble methods attached to the same method) is undefined.

Preambles and postambles are implemented by properties (**preamble** and **postamble**) on METHOD elements. By using preambles and postambles, you can modify the action of a method. This capability is useful when a user or application needs to change the behavior of a method that is already built into Oracle CDD/Repository.

For example, a software development group can modify a programming environment based on Oracle CDD/Repository so that mail is sent to the project leader every time a source file is replaced. A method that sends the mail is devised, and its element ID added to the **postamble** properties on the REPLACE methods for the various source file element types. Now, whenever a source file is replaced, the mail-sending method is invoked as a result of postamble execution.

One use of a preamble is to validate information in the dispatch list before the main body of the method is executed. For example, a preamble can check that a comment is supplied with every RESERVE message for a particular type. If the comment is not supplied, the preamble can place an error status on the error stack and cause the method to fail by returning an error code.

The METHOD elements contained in preambles and postambles are invoked with the same arguments as the method to which the preambles and postambles are attached.

7.6 Methods and Version Control

To reproduce a system configuration, you must not only use the original components (source files and definitions) but also the tools that were originally used. For example, a newer version of a compiler may produce slightly different output than an older version. To ensure complete reproducibility, an environment must be able to identify the correct tools.

This is possible because METHOD is a subtype of VERSION, which places method elements under version control. When a newer version of a tool becomes available, you can reserve the method element that invoked the older version, associate it with the new tool, and replace it, just as with a version element representing a file. Dependency information that refers to the older version of the tool remains valid, which allows you to locate the older version if needed.

8
Relations and Relationships

This chapter describes how Oracle CDD/Repository establishes and maintains relationships between elements in the repository. Oracle CDD/Repository uses relationships to implement some scan properties, called relation properties. You can define your own relation properties.

A relationship is an element in the repository that links other elements. For example, in a collection, a COMPOSITE_PART relationship links the COLLECTION element to each of the VERSION elements in the collection. Scan properties implemented by these relationships include the **hasChildren** property on the COLLECTION element and the **hasParents** property on the VERSION element. To return or alter the value of either of these properties, Oracle CDD/Repository traverses the relationships and forms a list of the element IDs of the related elements.

You can use relationships with special characteristics to create scan properties. Because relationships are elements, you also can create relation types with properties defined on them. This allows you to store information on the relationship.

8.1 Relationship Versus Relation

A **relationship** is an element in the repository that links other elements together. These elements are instances of RELATION or one of its subtypes. Oracle CDD/Repository uses relationships to implement relation properties.

A **relation** is a template for a relationship. Specifically, relations are instances of RELATION_TYPE. As such, they are analogous to elements of type ELEMENT_TYPE, which provide templates for elements.

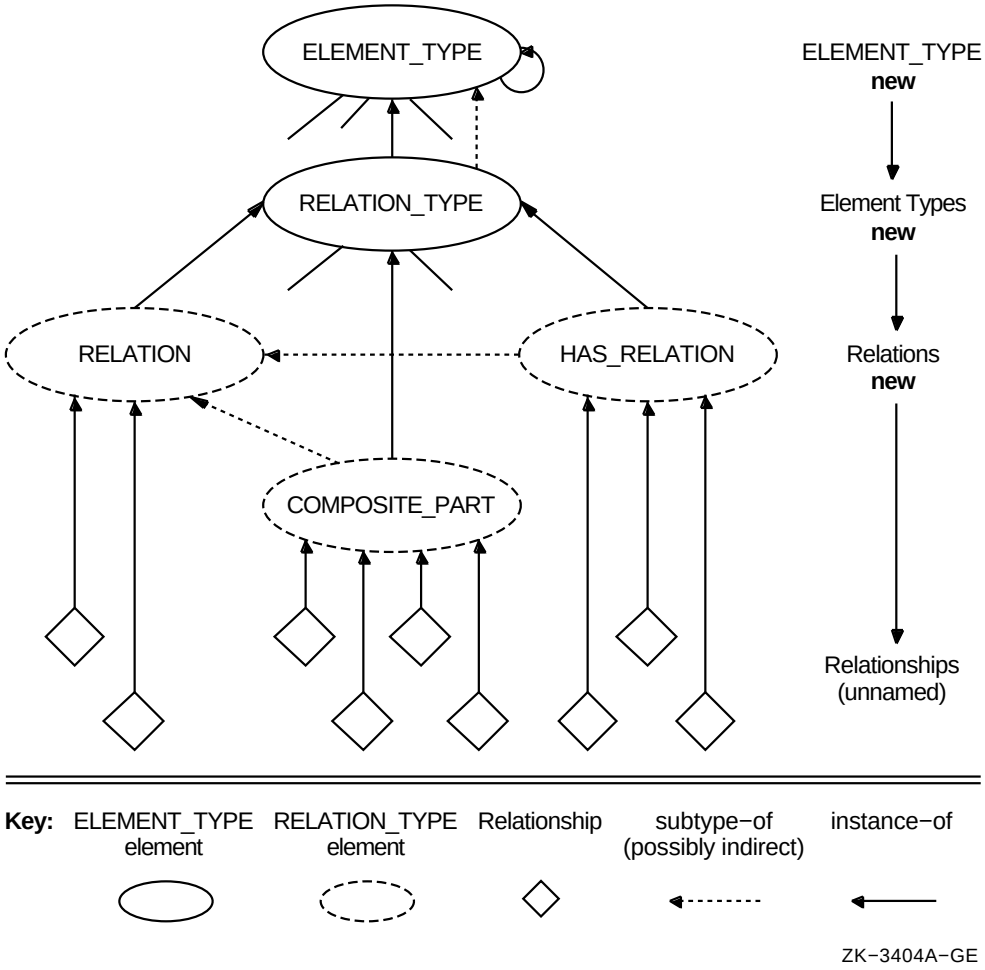
Figure 8–1 is an instantiation diagram for relations and relationships. This diagram shows the following:

- The ELEMENT_TYPE element named **ELEMENT_TYPE** is the template for elements that are templates for other elements (instances of ELEMENT_TYPE can themselves be instantiated to form new elements).

- One instance of ELEMENT_TYPE is named **RELATION_TYPE**. Unlike other element types, **RELATION_TYPE** is a *subtype* of ELEMENT_TYPE as well as an *instance* of ELEMENT_TYPE. This means that instances of **RELATION_TYPE** can *themselves* be instantiated to make new types.
- The relations shown in Figure 8–1 are some of those defined by Oracle CDD/Repository. Application integrators can define their own relations.

  One relation defined by Oracle CDD/Repository is named **RELATION**; all other relations are subtypes of **RELATION**.
- Instances of relations are called relationships. Relationships are unnamed. It is relationships that form the actual links between elements. Relationships of different types have different characteristics, depending on the relation that defines them.

Figure 8–1 Relation and Relationship Instantiation



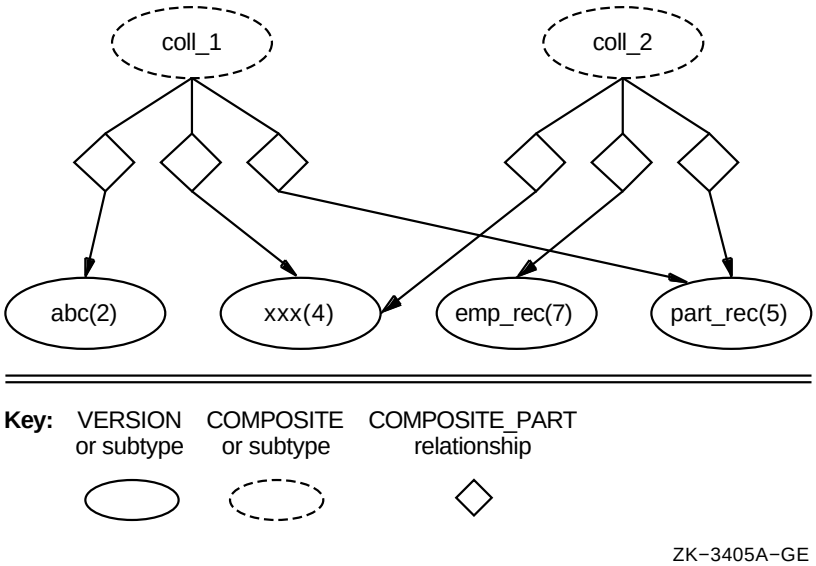
8.2 Relationship Characteristics

Each type of relationship has characteristics that distinguish it from other types. These characteristics are tailored to fit the need that the relationship fills in the repository. This section describes the characteristics. Section 8.4 shows how the characteristics are defined in the repository.

A relation type must specify the two element types (or groups of element types) that the relationship links. Every relationship links an **owner** to a **member**. For example, in the relationship that links a COLLECTION element to one of the versions in the collection, the COLLECTION element is the owner of the relationship, and the version is the member.

The owner must be of type COMPOSITE (the direct supertype of COLLECTION) or one of its subtypes. The member must be of type VERSION or one of its subtypes. Figure 8–2 shows how elements and relationships are arranged in a collection. Each relationship links only two elements, but the assembly of elements and relationships associates two COLLECTION elements with four VERSION subtypes in an *n*-to-*m* fashion.

Figure 8–2 A Collection Implemented by Relationships



By convention, relationships are represented as diamonds. The line from the relationship to the member of the relationship is drawn with an arrowhead; the line from the relationship to its owner, is drawn without an arrowhead. Lines that emerge from the relationship along the sides of the diamond, rather than from the points, represent properties on the relationship.

8.3 Relationship Traversal

To return the value of a scan property that is implemented by a relationship, Oracle CDD/Repository can traverse the relationship either from the element that owns the property to the elements that are members or from a member to its owner. For example, to find the value of the **hasChildren** property, Oracle CDD/Repository traverses each COMPOSITE_PART relationship from the COMPOSITE element to the VERSION element. The element IDs of all the VERSION elements found is the value of the **hasChildren** property.

Oracle CDD/Repository needs three pieces of information to traverse relationships and return scan property values:

- type of relationships to traverse

  This information is represented by a relation (an element of type RELATION or one of its subtypes). (Oracle CDD/Repository automatically includes any subtypes of the specified relation type in the traversal.)

- starting point for the traversal

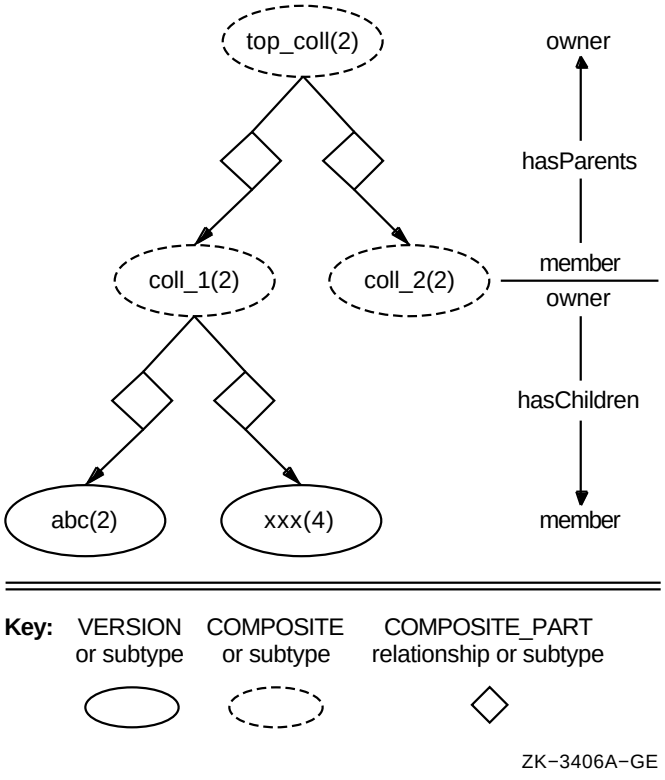
  this is an element ID.

- traversal direction

  There are two possible directions: from owner to member, and from member to owner.

To see why traversal direction is important, consider the collection hierarchy illustrated in Figure 8–3. All of the relationships shown are COMPOSITE_PART relationships. **coll_1(2)** is the *owner* of the relationships that link it to **abc(2)** and **xxx(4)**, but it is the *member* in the relationship that links it to **top_coll(2)**. Given **coll_1(2)** as the starting point, and COMPOSITE_PART as the relation to traverse, Oracle CDD/Repository still needs to know whether to traverse towards the owner (**top_coll(2)**) or the members (**abc(2)** and **xxx(4)**) of the relationships. For the **hasChildren** property, the direction of traversal is to the members. A different property, **hasParents**, traverses COLLECTION_PART relationships from member to owner.

Figure 8–3 Relationship Traversal Direction



In addition to relation properties, Oracle CDD/Repository lets you traverse relationships by calling the MCS_scan_query routine. This routine provides a generalized query capability. You can specify an arbitrary combination of relation types to traverse, filter out element types, and traverse a chain of relationships of different types. The results of this traversal are returned in the form of a scan, which you can examine as you would the value of a scan property.

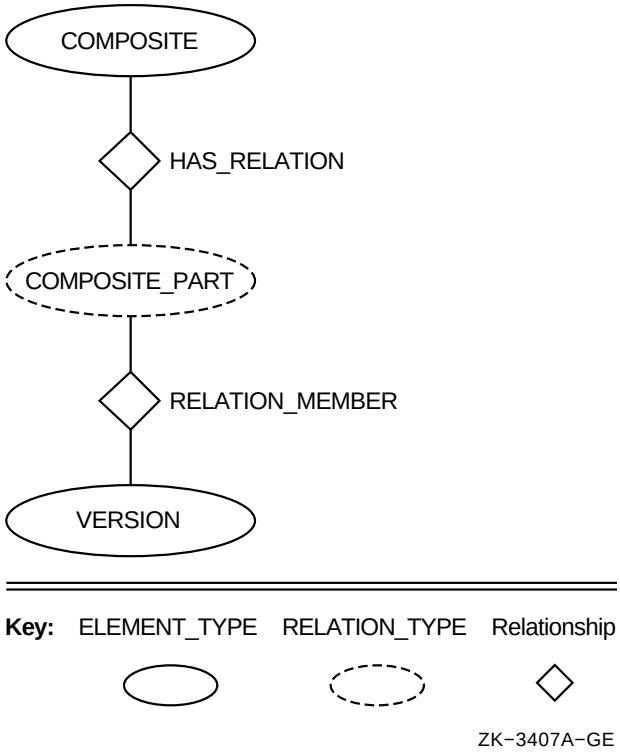
8.4 Relation Definitions

This section shows how relation characteristics are specified in the repository by examining the implementation of collections and demonstrating how the COMPOSITE_PART relation is defined. (Chapter 6 describes how this relation is used in the definition of collection-related properties, such as **hasChildren**.)

The repository schema not only includes how elements of the various types relate to each other, but also what properties they possess. A new repository contains the elements that make up its schema, including subschemas for each of the various models present in the repository. The remainder of this section describes the subschema that defines how collections work.

Figure 8–4 illustrates the elements and relationships needed to specify COMPOSITE_PART relationships and their characteristics. The RELATION_TYPE element named **COMPOSITE_PART** is the template for COMPOSITE_PART relationships.

Figure 8–4 Collection Subschema: Specifying the Relationship



HAS_RELATION relationships specify what relation types an element type owns. In the collection subschema, the HAS_RELATION relationship that links **COMPOSITE** to **COMPOSITE_PART** indicates that COMPOSITE elements (and subtypes, such as COLLECTION) can own COMPOSITE_PART relationships. If there were other types that could own these relationships, their ELEMENT_TYPE

elements would be linked to **COMPOSITE_PART** by additional HAS_RELATION relationships.

Similarly, RELATION_MEMBER relationships specify what relationships an instance of the type can be a member of. The collection subschema shows that VERSION elements and their subtypes can be members of COMPOSITE_PART relationships. Other valid members would be shown by additional RELATION_MEMBER relationships.

8.4.1 Relation Type Hierarchy

The element type RELATION is a peer of NAMED_ELEMENT in the Oracle CDD/Repository type hierarchy. All relations are descendents of RELATION, and therefore inherit certain basic characteristics from it; for example, all relations define elements (relationships) that are unnamed.

The relation type hierarchy plays an important part in relationship traversal. A relation property definition consists of "instructions" to follow relationships of a certain type *and its subtypes* and return a scan of all the elements at the far ends of those relationships. (Full details are given in Chapter 6.) For example, to find the value of **hasChildren**, Oracle CDD/Repository follows COMPOSITE_PART relationships to their members. Oracle CDD/Repository also follows any relationships whose types are subtypes of COMPOSITE_PART. This mechanism preserves the notion that an instance of a subtype can be treated the same as an instance of its supertype.

8.4.2 Relation Type Inheritance

Element types (including relation types) inherit the values of certain properties from their supertypes. For example, every element type possesses a property, **propDef**, that is a scan of all that type's property definitions, both those it defines itself and those it inherits. When you define an element type and specify its supertype, the new type inherits the value of **propDef** from the supertype, and may add to it if you define new properties on the new type.

Element types inherit the values of two properties having to do with relation definition from their supertypes. The properties are the following:

- The **ownsRelation** property is a scan of all the relation types that an element type owns. The value is formed by following HAS_RELATION relationships from owner to member.
- The **relationMember** property is a scan of all the relation types for which an element type can be the member. The value is formed by following RELATION_MEMBER relationships from member to owner.

These property values are inherited from the supertype, with the following exception: If a subtype of an element type owns a relation type that is a subtype of a relation type owned by the element type's supertype, then the subtype relation type replaces the supertype relation type in the **ownsRelation** scan. Similarly, if a subtype of an element type can be a member of a relation type that is a subtype of a relation type that can be a member of the element type's supertype, then the subtype relation type replaces the supertype relation type in the **relationMember** scan.

Using Figure 8–4 as an example, you can see that the COMPOSITE type's **ownsRelation** scan has a value of COMPOSITE_PART, because COMPOSITE owns a HAS_RELATION relationship to COMPOSITE_PART. The actual value of **ownsRelation** can include several other relation types, as described in the following list:

- If a subtype of COMPOSITE called MY_COMP is added, it inherits the value of **ownsRelation** from COMPOSITE.
- If a relation type, MY_RELTYPE, is created and added to MY_COMP's **ownsRelation** scan, the scan now contains COMPOSITE_PART (inherited from COMPOSITE) and MY_RELTYPE, *so long as* MY_RELTYPE *is not a subtype of* COMPOSITE_PART.
- If a relation type, MY_CP, a subtype of COMPOSITE_PART, is created and added to MY_COMP's **ownsRelation** scan, MY_CP *replaces* COMPOSITE_PART in the **ownsRelation** scan.

Relation types also possess properties that show what types can be their owners and members, including:

- The **legalOwners** property is a scan of the types that can own a relation type. Its value is formed by following HAS_RELATION relationships from member to owner.
- The **legalMembers** property is a scan of the types that can be members of a relation type. Its value is formed by following RELATION_MEMBER relationships from owner to member.

Unlike the values of **ownsRelation** and **relationMember** on element types, the values of **legalOwners** and **legalMembers** are *not* inherited by subtypes of a relation type. This is because relation subtypes should be more specific, and therefore more restrictive, than their supertypes. Inheriting these values would defeat that purpose. For example, the top of the relation type hierarchy, RELATION, can have ELEMENT elements (including subtypes) as its owners and members. This hierarchy includes all the elements in the repository because all element types are subtypes of ELEMENT. Subtypes of RELATION need to be

able to restrict legal ownership and membership. This means that the property values of **legalOwners** and **legalMembers** are not inherited.

8.5 Dependency Relationships

One important subtype of RELATION is DEPENDS_ON. This relation defines a **dependency relationship**, one in which the owner is notified if the member is modified. For example, a dependency relationship can express a build dependency in which a derived object (such as an object file) depends on a source object (such as a source file). In this situation, the derived object owns the dependency relationship and the source object is the member.

COMPOSITE_PART is a subtype of DEPENDS_ON and is therefore a dependency relationship. This means that a COLLECTION element is notified of changes in the elements in the collection. (See Section 10.1 for more information about the characteristics and uses of dependency relationships.)

8.6 Operations on Relationships

This section describes how you can manipulate relationships using either implicit or explicit techniques.

8.6.1 Implicit Relationship Manipulation

Most operations on relationships are transparent to Oracle CDD/Repository programmers. For example:

- The ATTACH message creates COMPOSITE_PART relationships from the COMPOSITE element to the version to be attached.
- The DETACH message deletes the COMPOSITE_PART relationship between the COMPOSITE element and the version to be detached.

Relationships may also impose additional semantics on some operations. For example, the FREE message fails if its target is a member of a relationship. Conversely, sending FREE to an element that owns relationships deletes the element and its relationships (but not the members of those relationships).

When a scan property is implemented by relationships, the various MCS_scan calls implicitly manipulate the relationships. For example, MCS_scan_insert( ) followed by MCS_dispatch_setProp( ) creates a new relationship to implement insertion of the element in the scan.

8.6.2 Explicit Relationship Manipulation

Although it is generally unnecessary to do so, relationships can also be manipulated explicitly. Many of the MCS_scan calls have an added argument, ***rID***, that returns the element ID of the relationship affected by the call. For example, the *rID* argument to MCS_scan_getNext( ) receives the element ID of the relationship that connects the element possessing the property to the element returned by the call. With this element ID, you can operate directly on the relationship. You can use this technique to get or set the value of a property on a relationship.

Relationships respond to the following messages:

NEW
FREE
GETPROP
SETPROP

Although you can send these messages to relationships, you cannot modify them unless they are mutable. Relationships are not versionable, so they cannot be reserved. Their mutability depends on the reservation status of elements that own them, as follows:

- You can modify a relationship that is owned only by reserved or uncontrolled elements.
- You cannot modify a relationship (except for **access** and **history**) that is owned by a checked-in (replaced) element.
- You can modify a relationship that is owned by a nonversionable element (such as another relationship) if you can modify its owner.

9

Configuration Management

This chapter describes the Oracle CDD/Repository configuration management models. Each model description begins the concept behind the model. Then the Oracle CDD/Repository implementation of the model is described in terms of its element types, properties, and messages.

The models described in this chapter support the management of orderly system development in large or small engineering organizations. The models allow for systems with the following characteristics:

- evolves over time
- includes a set of components, which also evolve over time
- exists in several variants
- developed concurrently by many engineers

The models provide a means of identifying the contents of obsolete versions of the system and of re-creating an obsolete version if necessary. The models also allow system components that have reached specified levels of stability to be made available to increasingly broad groups of outside users.

The implementation descriptions are conceptual. No attempt is made to give complete implementation details or programming information. For reference information, see the *Oracle CDD/Repository Callable Interface Manual*.

9.1 Version Management

Version management seeks to solve problems that arise when several people work on the same project. Two people often need access to the same system object at the same time. For example, one person may be modifying a source file at the same time another needs that file to build a system. Worse yet, two people may try to modify the same file at the same time.

In small projects, it may be possible to manage such conflicts verbally. But accidents may still occur; and as project size grows, managing access conflicts informally becomes impossible. Some formal, automated means is required. The Oracle CDD/Repository version management model provides this means.

Beyond its role in managing simultaneous access to individual system objects, version management provides the foundation for management of change in project components. Component change is represented in the repository as a succession of versions, any of which can be re-created at a later time, even after it is no longer the latest version. The ability to identify and re-create previous versions is important for most system development and is a contractual requirement for many.

As described in Section 9.2, Oracle CDD/Repository uses the same model to manage change in systems as it does for individual system components.

9.1.1 Versions

Conceptually, a **version** is a "snapshot" of the state of an evolving system object at some point in time. In the familiar case of a file, each version of the file is a modification of the previous version.

In Oracle CDD/Repository, a "version" is also an element whose type is VERSION, or any of the subtypes of VERSION. These subtypes include version elements that represent text and binary files. However, the version management model applies to many entities other than files. Any of the subtypes of VERSION can take advantage of the mechanisms described in this section.

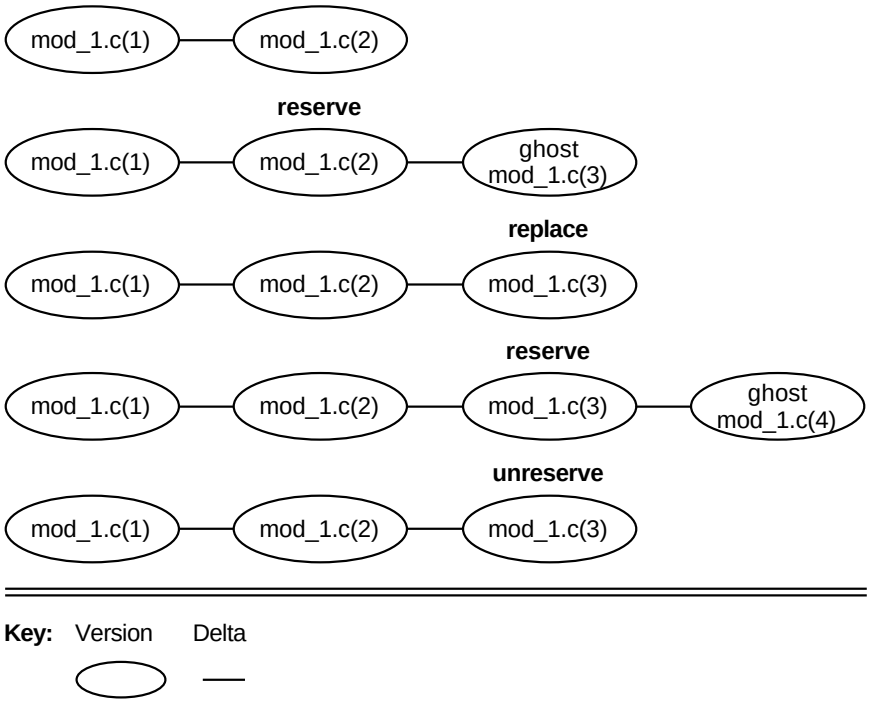
A central principle in Oracle CDD/Repository version management is that a shared version element in the repository is **immutable**; it cannot be changed. However, a version can be **reserved**. Reserving a version (also known as "checking out") creates a private, writable copy of the version element. This new version element is called a **ghost** because it is visible only to the person who reserved it. After the version element has been modified, it is **replaced** (or "checked in"). Replacing makes the (modified) private copy public and read-only.

Figure 9–1 illustrates the three basic operations on versions. At the top of the figure, **mod_1.c(2)** is a version that represents a C source file. The RESERVE message causes a ghost of that version to be created. The new version is named **mod_1.c(3)**, indicating that it is a successor to **mod_1.c(2)**. The RESERVE message copies all normal properties, all required relation properties, and all relationships whose type is composite part or its subtypes from the original version to the new version. The new version can be changed only by the person who reserved the version.

The REPLACE message transforms the private, ghost version into a public, read-only version. This version now represents the latest public version of the file mod_1.c.

Later, someone reserves this file again. This time, instead of replacing the file, the person decides to discard the modifications by **unreserving** the file. The ghost version **mod_1.c(4)** responds to the UNRESERVE message by restoring the repository to its state before the RESERVE message was issued.

Figure 9–1 Elemental Operations on Versions



ZK–3373A–GE

Figure 9–1 includes symbols to represent the VERSION subtypes and the connections between them, labeled delta. Delta indicates that each version was created by changing the previous version.

A set of versions of the same system object (in Figure 9–1, the versions are **mod_1.c(1)**, **mod_1.c(2)**, **mod_1.c(3)**) is called a **component**. A component includes all the versions and variants of an object.

9.1.2 Branches

If a component includes the variants of an object, those variants are contained in **branches** and subbranches. A branch is a sequence of versions, forming a line of development distinct from other lines of development in the same component.

A succession of versions is called a **line of descent**; each version is a descendent of the previous version. The initial series of versions (**mod_1.c(1)**, **mod_1.c(2)**, and **mod_1.c(3)** in Figure 9–1) is called the **main line of descent** because it proceeds directly from **mod_1.c(1)**, the initial version of the component.

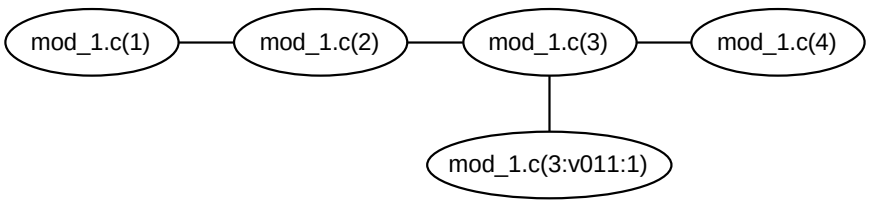
If two people need to modify the same system object simultaneously, the main line of descent must split to allow these parallel modifications. One of these paths branches from the main line of descent, while the other path continues the main line of descent. The branch line may itself branch.

Note

You cannot create metadata elements on a branch.

When a line of descent has branched, you might want to **merge** the branch back into the line of descent. For example, a branch created to fix bugs should be merged to incorporate the bug fixes into the developing code.

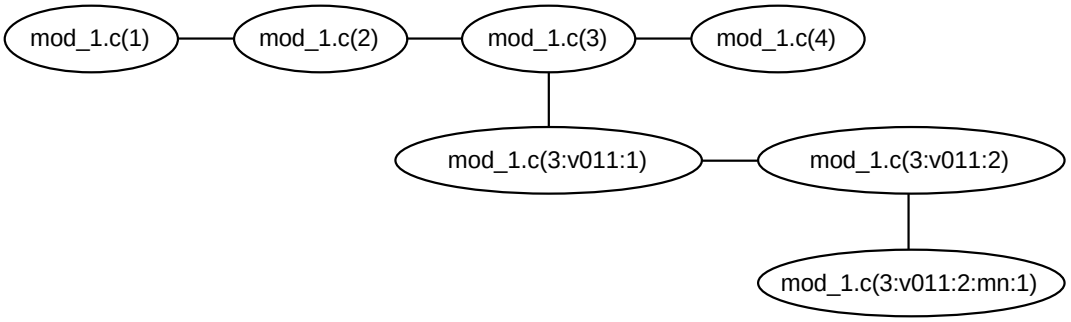
As an example, assume that development has proceeded to the point shown in Figure 9–1, and that **mod_1.c(3)** is the version used to build Version 1.0 of the product. Now the development stream branches: one branch begins work on Version 2.0 of the product, while another branch begins maintenance work for Version 1.1. The Version 2.0 work continues on the main line of descent with **mod_1.c(4)** and so on. The maintenance work branches from **mod_1.c(3)** to form the **variant branch** that begins with the VERSION subtype named **mod_1.c(3:v011:1)**. The component now appears as follows:



ZK–3374A–GE

The line from **mod_1.c(3)** to **mod_1.c(3:v011:1)** is a delta connection, indicating that the branch version was obtained by changing **mod_1.c(3)**.

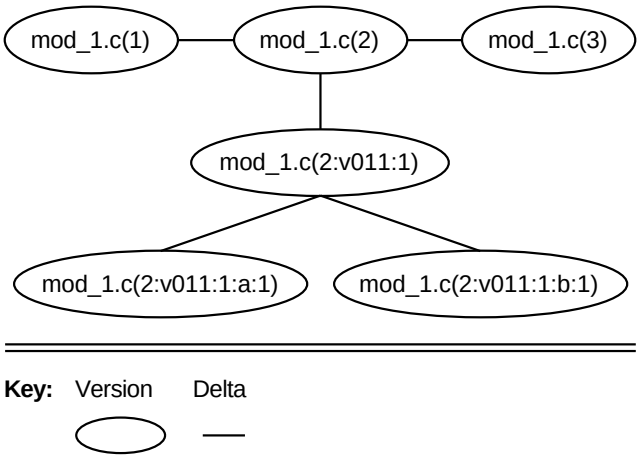
If the maintainer creates a new version on the 3:v011 branch, it is named **mod_1.c(3:v011:2)**. Later, the maintainer creates a subbranch from version **mod_1.c(3:v011:2)**, to incorporate multinational (mn) character support. This produces the following situation:



The version name indicates the position of the version in the branch tree. A version on the main line of descent contains the single version number in parentheses; for example, **mod_1.c(3)**. For a version on a branch, the contents of the parentheses indicate the path from a version on the main line of descent, finishing with the name of the branch and the version number on the branch. For example, **mod_1.c(3:v011:2)** is the second version on the branch named v011 that descends from **mod_1.c(3)**. **mod_1.c(3:v011:2:mn:1)** is the first version on the branch named mn that descends from **mod_1.c(3:v011:2)**.

Figure 9–2 illustrates multiple levels of branching (and resulting names) as well as multiple branches from a single version.

Figure 9–2 Multiple Branching

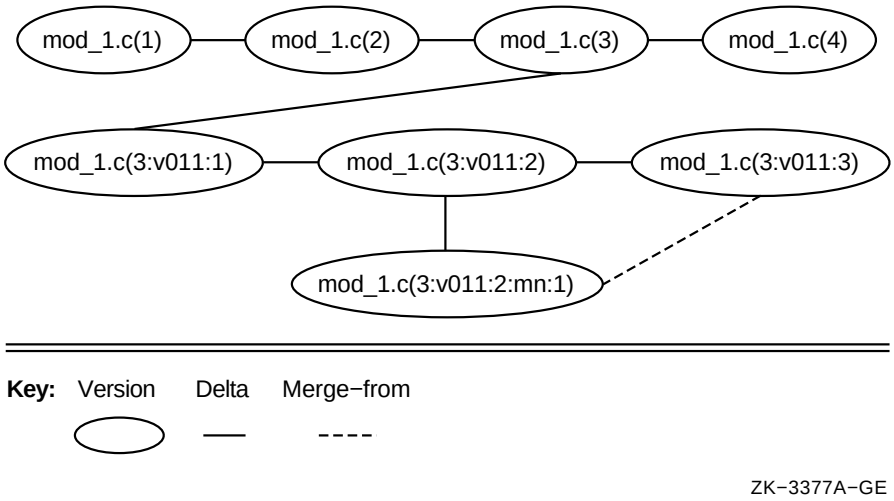


9.1.3 Merging

As Figure 9–3 shows, development of Version 1.1 is complete and you can terminate the multinational variant line by merging it into the Version 1.1 development line. Because no new version has been created on the 3:v011 branch since the 3:v011:2:mn branch was created, you do not need to merge the contents of both branches. **mod_1.c(3:v011:2)** is simply reserved, and the contents of the resulting new version are those of **mod_1.c(3:v011:2:mn:1)**. A merge-from connection (shown by a dashed line) shows that the mn branch has been merged back into the v011 branch. This type of merge is called a **genealogy merge** (or **branch merge**).

Figure 9–3 shows how 3:v011:2:mn:1 is merged into the 3:v011:3.

Figure 9–3 Merging



ZK−3377A−GE

The simplest case of a branch merge occurs when a branch has been created from a version on the parent branch, and the branch is to be merged back into its parent branch, but the version from which the branch sprouts has no successors. This might happen when a code version must be frozen for testing by a quality assurance group, but development is continuing. The continuing development is done on a branch, and, barring minor bug fixes by the quality group, the main line version is not changed before the branch is merged back into the main line.

If the version from which the branch sprouts has a successor on its own line of descent, then a content merge must be done in addition to the branch merge. In a content merge, both objects to be merged exist and contain data; the differences between the contents of the two objects must be reconciled to create a single merge result.

For example, **mod_1.c(3:v011:2)** could have already had a successor when the mn branch was created, and thus have forced the creation of the branch in order to reserve that particular version. If a new version of **mod_1.c(3:v011:2)** did exist, the content merge would be necessary to reconcile the differences in content (if any) between the two versions being merged.

To prevent further development on a branch that has been merged back into a line of descent, you can send the FREEZE message to the last version on the branch. Once frozen, the version rejects RESERVE messages.

9.1.4 Version Management Properties and Messages

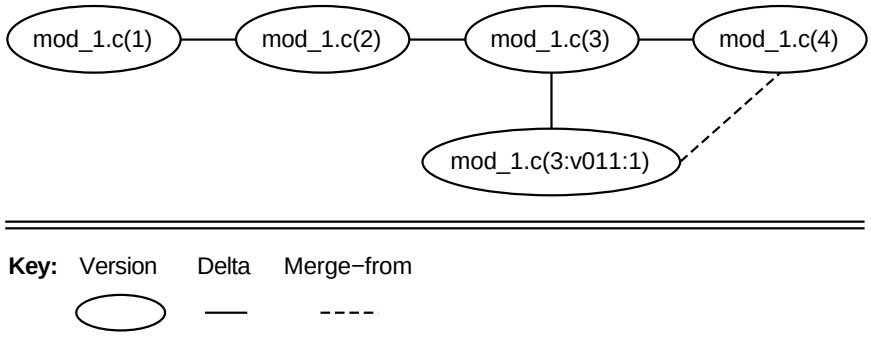
This section describes the properties related to version management. To illustrate each property description, an example shows the value of that property for one or more of the elements in the instance diagram of Figure 9–4.

Only those properties that have meaning for the concepts presented so far are described here. Later sections describe properties that implement configuration management and properties that depend on the context.

This section also describes a number of messages that request version management operations.

For more information on the properties and methods described in this section, refer to the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals.

Figure 9–4 Version Control Properties



9.1.4.1 Version Management Properties

The properties described in this section are properties of VERSION and its subtypes.

nextVersions

The value of the **nextVersions** property is a **scan** (unorder list) of the element IDs of the logical successors of this VERSION element. The logical successors include the following:

- The next VERSION element on the direct line of descent from this one. If it exists, this version is guaranteed to be the first in the scan.
- A VERSION element that this one merges into. There may be more than one.

- A VERSION element that branches from this one. There may be more than one such version.

Example For the VERSION element **mod_1.c(1)**, the value of **nextVersions** is a scan containing **mod_1.c(2)**.

For **mod_1.c(3)**, the value of **nextVersions** is a scan containing **mod_1.c(4)** (which will be first in the scan) and **mod_1.c(3:v011:1)**.

For **mod_1.c(3:v011:1)**, the value of **nextVersions** is a scan containing **mod_1.c(4)**.

For **mod_1.c(4)**, which has no logical successors, the value of **nextVersions** is null (an empty scan).

prevVersions

The value of the **prevVersions** property is a scan of the element IDs of the logical predecessors of this VERSION element. The logical predecessors include the following:

- The preceding VERSION element on the direct line of descent leading to this one. If it exists, this version is guaranteed to be the first in the scan.
- A VERSION element that merges into this one. There may be more than one such version.
- A VERSION element from which this one branches.

Example For the VERSION element **mod_1.c(2)**, the value of **prevVersions** is a scan containing the elmID for **mod_1.c(1)**.

For **mod_1.c(4)**, the value of **prevVersions** is a scan containing the elmIDs for **mod_1.c(3)** (which will be first in the scan) and **mod_1.c(3:v011:1)**.

For **mod_1.c(3:v011:1)**, the value of **prevVersions** is a scan containing the elmID for **mod_1.c(3)**.

For **mod_1.c(1)**, which has no logical predecessors, the value of **prevVersions** is null.

status

The value of the **status** property indicates the current status of this version. Values include the following:

- available—Can be reserved to create the next version on this line of descent
- read-only—Can be read but not reserved, unless the reservation specifies a branch
- ghost—Has been reserved

- frozen—Line of descent has been frozen; no new versions can be created

Symbolic constants correspond to each of these values (refer to the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals.)

Example The value of the **status** property for **mod_1.c(1)**, **mod_1.c(2)**, and **mod_1.c(3)** is "read-only." These versions have been replaced and cannot be modified. The value of **status** for **mod_1.c(4)** is "ghost" if it has not been replaced, and "available" if it has been replaced. The value of **status** for **mod_1.c(3:v011:1)** is either "available" or "frozen," depending on whether a FREEZE message has been sent to it.

availVersion

The value of the **availVersion** property is the element ID of the most recently replaced version on this branch's line of descent. This is the version that is available to others. The value of this property is the same for *any* version on a given line of descent.

Example For any of the versions **mod_1.c(*n*)**, the value of the **availVersion** property is **mod_1.c(4)**. If someone reserves **mod_1.c(4)**, a ghost version, **mod_1.c(5)**, is created, but this version is not available. Therefore, the value of **availVersion** remains the elmID of **mod_1.c(4)** until **mod_1.c(5)** is replaced.

lastVersion

The value of the **lastVersion** property is the element ID of the most recently visible version on this branch's line of descent, regardless of replacement status. (Recall that the contents of a ghost version are visible only to the person who created it by reserving the element.) For a given user, the value of this property is the same for *any* version on a given line of descent; however, the value changes depending on who sends the message. If you have a reserved version on the line of descent, the value of **lastVersion** is that reserved version; for all others, the value is the same as **availVersion**.

Example For any of the versions **mod_1.c(*n*)**, the value of the **lastVersion** property is **mod_1.c(4)**. If someone reserves **mod_1.c(4)**, a ghost version, **mod_1.c(5)**, is created, and this version becomes the value of **lastVersion** *for the person who reserved* ***mod_1.c(4)***. For others, the value of **lastVersion** remains **mod_1.c(4)**.

firstVersion

The value of the **firstVersion** property is the element ID of the first version on this branch's line of descent. The value of this property is the same for *any* version on a given line of descent.

Example For any of the versions **mod_1.c(*n*)**, the value of the **firstVersion** property is the elmID of **mod_1.c(1)**.

rootBranchName

The value of the **rootBranchName** property is the name of the element, stripped of version, branch, or directory information.

Example For any of the versions in Figure 9–4, the value of **rootBranchName** is "mod_1.c".

branchName

The value of the **branchName** property is the name of the branch containing the element, up to but not including the version number.

Example For any of the versions **mod_1.c(*n*)**, the value of **branchName** is a null string, since these versions are not on a branch.

For **mod_1.c(3:v011:1)**, the value of **branchName** is "v011".

versionNum

The value of the **versionNum** property is the version number of this version. The value is an integer, not a character string. Successive versions have increasing version numbers. Version numbers appear at the end of the branch name expression, contained in parentheses in version names.

9.1.4.2 Version Management Messages

The VERSION element type responds to messages that implement version management operations. When a VERSION or one of its subtypes receives one of these messages, the resulting method performs the specified operation. The method generally changes the state of the repository by changing property values and possibly creating or deleting elements.

This section describes only those messages that operate within the limited version management model described thus far. Messages that modify a configuration or that depend on a context are described in later sections.

These brief message descriptions cover only the effect of the message on a VERSION element. This processing is fundamental to operations on versioned elements; however, subtypes of VERSION frequently refine this processing. For example, the BINARY element type refines the RESERVE message by creating a modifiable copy of the file in the user's context (private area). (See the *Oracle CDD/Repository Information Model Volume I* and *Oracle CDD/Repository Information Model Volume II* manuals for a complete description of how each element type responds to the messages listed in this section.)

reserve

The RESERVE message creates a copy (new version) of the version to which it is sent. The new version is in the ghost state, which means it is visible only to its creator. The initial contents of the new version are identical to that of the existing version.

Subtypes of VERSION further refine the RESERVE method. For example, the BINARY element type responds to RESERVE by creating a workfile you can edit and by associating the workfile with the ghost version.

When RESERVE is sent to the last version in a line of descent, the new version has a version number that is one higher than that of its predecessor. This use of RESERVE is shown in Figure 9–1. If the version is not the last in the line of descent, RESERVE will fail unless you specified that a branch be created.

When RESERVE is sent with a branch name specified, the new VERSION element is created with a name that incorporates the branch name and is properly linked to the VERSION element that was the target of the operation. The full branch and version information of the new version is as follows:

(*existing-branch-and-version*:*branch-name*:1)

For example, if **mod_1.c(2:a:3)** is reserved on branch x, the resulting name is **mod_1.c(2:a:3:x:1)**.

Note

You cannot create metadata elements on a branch.

unreserve

Sent to a ghost version created by RESERVE, UNRESERVE deletes the version and undoes the changes made by RESERVE.

Subtypes of VERSION further refine the UNRESERVE message. For example, the BINARY subtype responds by deleting the workfile that RESERVE created.

Note

You cannot unreserve metadata elements.

replace

Sent to a "ghost" version, REPLACE changes its status to "available" and makes it immutable. A version, once replaced, can never be changed again; new versions must be made of it.

You can send the REPLACE message with a branch name specified. In this case, you can replace the version on a branch of the line of descent from which you reserved it.

Subtypes of VERSION further refine the REPLACE message. For example, the BINARY subtype can respond by using a text delta engine to record the changes made to the file.

merge

The MERGE message merges a VERSION element on a branch back into a line of descent. Section 9.1.3 describes the various types of merging.

Subtypes of VERSION refine the MERGE method. For example, TEXT elements respond to MERGE by invoking a text merge tool to compare the two text files to be merged and report the differences.

freeze

The FREEZE message prevents future RESERVE messages to the VERSION element from succeeding.

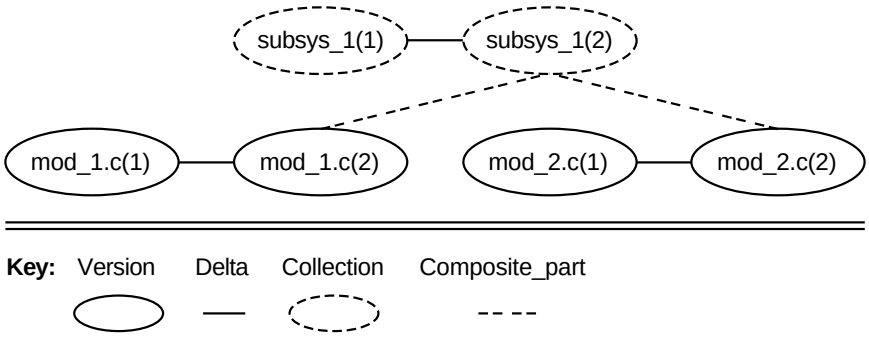
unfreeze

The UNFREEZE message undoes the effect of a previous FREEZE message.

9.2 Modeling Configurations

In the previous sections, the running example has illustrated the evolution of a single component, but your software system probably consists of many components. You can express the configuration of a software system at a point in time (such as a baselevel, a field test, or a release) as a **collection** of particular versions that make up the system at that time.

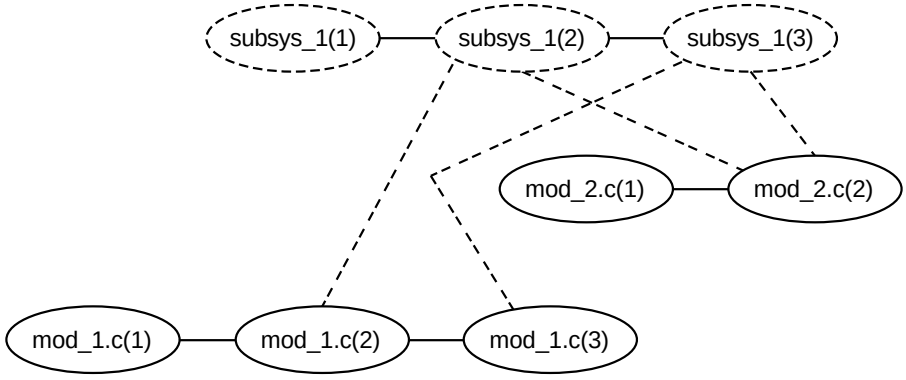
For example, a system consists of two components, **mod_1.c** and **mod_2.c**. The first versions of each of these constitute the field test configuration of the product called **subsys_1**. To represent the field test configuration, establish the following collection:



ZK−3379A−GE

The lines between **subsys_1(2)** and the versions **mod_1.c(2)** and **mod_2.c(2)** indicate that the versions are a part of the collection **subsys_1(2)**. The lines represent relationships of the type COMPOSITE_PART. The value of the **hasChildren** property on the collection **subsys_1(2)**, which is the two elements **mod_1.c(2)** and **mod_2.c(2)**, is determined by following those relationships.

During the development of the product, a new version of **mod_1.c** is created and included in the product configuration. Since the product configuration has changed, it is necessary to create a new collection to represent the changed configuration. The new element, **subsys_1(3)**, includes the current versions of **mod_1.c** and **mod_2.c**, as follows:



ZK−3380A−GE

This figure shows the similarity of the structures used to represent the configurations (nodes **subsys_1(2)** and **subsys_1(3)**) and their components. The collections **subsys_1(1)**, **subsys_1(2)**, and **subsys_1(3)** are VERSION

elements, or subtypes of VERSION. This is a crucial point: since collections are implemented using a subtype of VERSION (named COLLECTION), collections are subject to version control. As a result, configurations (represented by collections) can exist in a number of versions, and you can identify and re-create a previous system configuration by locating the collection that represents it.

The versions included in a collection need not be the latest versions, or those from the main line of descent. However, only one version from each component can be part of the collection. This constraint implements the concept that configurations represent a fixed point in system development and therefore cannot contain multiple versions of a component, since multiple versions represent multiple points in development.

9.2.1 Modeling Complex Configurations

The preceding examples have shown collections with two versions representing C source files attached to them. A collection can contain an element of type VERSION or any of its subtypes. Since COLLECTION is a subtype of VERSION, it follows that a collection can contain another collection. The result is a **collection hierarchy**.

Use a collection hierarchy to model more complex system configurations than the simple example presented so far. For example, you can divide a system into several subsystems. In a large system, the subsystems might be further divided. The collection hierarchy might go several levels deep before reaching the noncollection versions at the **leaves** (those elements representing the files and other objects that actually make up the system) of the hierarchy.

A sequence of collection versions (or line of descent) can model a series of system baselevels or releases. You can create branches in collection components. Branches can model variant configurations. For example, a branch can hold collections that represent the system implementation for an alternate hardware platform or operating system.

9.2.2 Reserving and Replacing in a Collection

If you want to work on any versioned object, you must reserve it before you can change it. In practice, an object that you want to reserve will be attached to a collection and may be several levels deep in a collection hierarchy. (Your working environment identifies the top collection of this hierarchy, as illustrated in Section 9.3.1. The collection hierarchy represents the version of the system, or subsystem, that you are currently working on.)

The contents of a collection consist of the versions that are attached to it. When you reserve one of these versions, you create a new ghost version. This ghost must be attached to a collection for you to work on it. However, if the collection to which the original version was attached is a replaced (nonghost) collection, its contents cannot change. You must reserve the collection before you reserve one of its versions.

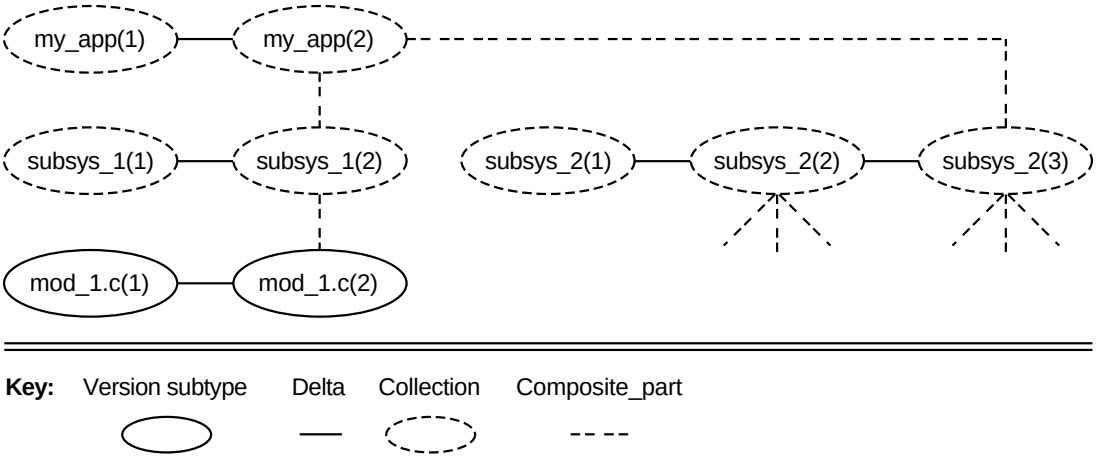
See the illustrations of collections in Section 9.2. Consider the possibilities if **mod_1.c(2)** is reserved to create **mod_1.c(3)**. (**subsys_1(2)** is a replaced collection representing a field test configuration.)

- If **mod_1.c(2)** is detached from **subsys_1(2)** and **mod_1.c(3)** is attached to it, the contents of **subsys_1(2)** change. This is not allowed because **subsys_1(2)** is a replaced version. Such a change, if permitted, would alter the field test configuration and make it impossible to re-create that configuration.
- If **mod_1.c(2)** remains attached to **subsys_1(2)**, there is no collection to which **mod_1.c(3)** can be attached. (A collection can contain only one version of any component. **subsys_1(2)** cannot contain both **mod_1.c(2)** and **mod_1.c(3)**.)
- The only remaining possibility is to reserve **subsys_1(2)** *first*, creating the ghost collection **subsys_1(3)**. The initial contents of **subsys_1(3)** are the same as those of **subsys_1(2)**. Then when **mod_1.c(2)** is reserved, it can be detached from **subsys_1(3)** and **mod_1.c(3)** can be attached in its place.

In a collection hierarchy, the effect of reserving an element ripples up through the hierarchy until the top of the hierarchy is reached. No element (including a collection) can be reserved until its containing collection has been reserved; so it is necessary to start at the top of the hierarchy and proceed downwards, reserving those collections that have not already been reserved.

Since it is so often necessary to reserve containing collections, Oracle CDD/Repository supplies a "reserve to top" option for the RESERVE message. This option causes the message to be sent recursively to all the containing collections of the target element, until the sender's top collection is reached. Then, starting from the top and working down, all unreserved collections are reserved. Finally, the original target element is reserved.

The following example shows how a reserve-to-top operation affects a collection hierarchy. Collection **my_app(2)**, the top of the collection hierarchy, contains collections **subsys_1(2)** and **subsys_2(3)**. Collection **subsys_1(2)** contains **mod_1.c(2)**. The following figure shows the initial state.

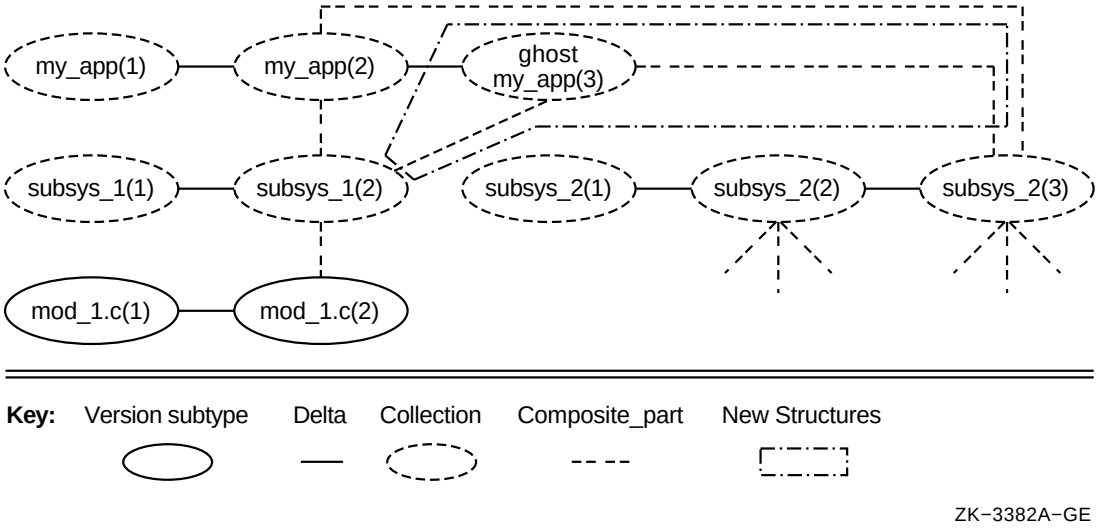


ZK–3381A–GE

Send the RESERVE message to **mod_1.c(2)** with the "reserve to top" option.

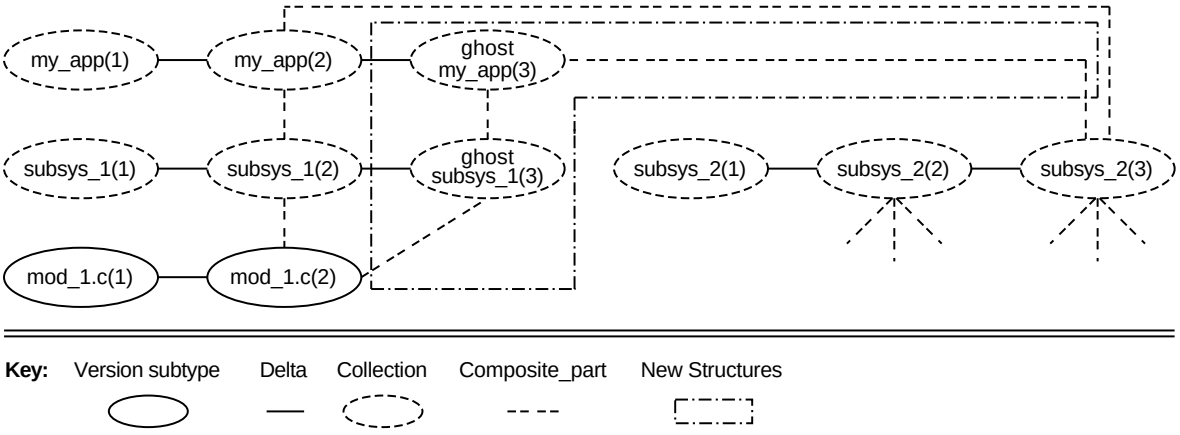
Before it can carry out the reservation of **mod_1.c(2)**, Oracle CDD/Repository looks upwards in the collection hierarchy to find out what collection(s) must be reserved first. **mod_1.c(2)** is a member of **subsys_1(2)**, so **subsys_1(2)** must be reserved. Element **subsys_1(2)** is a member of **my_app(2)**, which also must be reserved. Because **my_app(2)** is at the top of the collection hierarchy, no further reservations will be needed.

Oracle CDD/Repository starts at the top by reserving **my_app(2)**, creating the ghost version **my_app(3)**. The contents of the two collection versions, including their members, are initially identical. The next figure shows the resulting state.



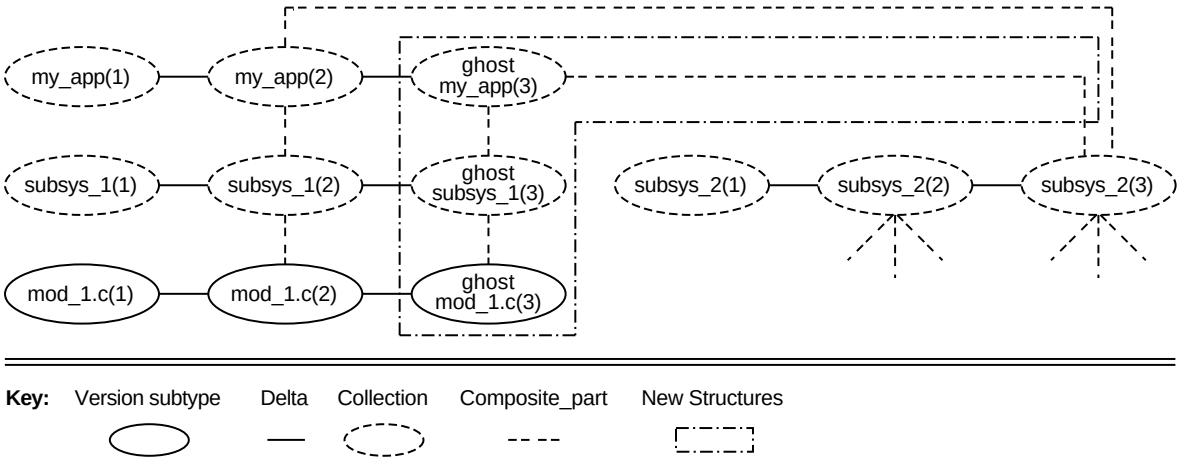
ZK–3382A–GE

Next, Oracle CDD/Repository reserves **subsys_1(2)**, creating the ghost version **subsys_1(3)**. **subsys_1(3)** replaces **subsys_1(2)** as a member of **my_app(3)**. This processing is shown in the following figure.



ZK–3383A–GE

The next figure shows how Oracle CDD/Repository reserves **mod_1.c(2)**, creating the ghost version **mod_1.c(3)**, which becomes a member of the ghost version **subsys_1(3)**:



ZK–3384A–GE

This example has shown you how to reserve a version without disturbing existing views of the system (those represented by the existing, public collections **my_app(2)** and **subsys_1(2)**).

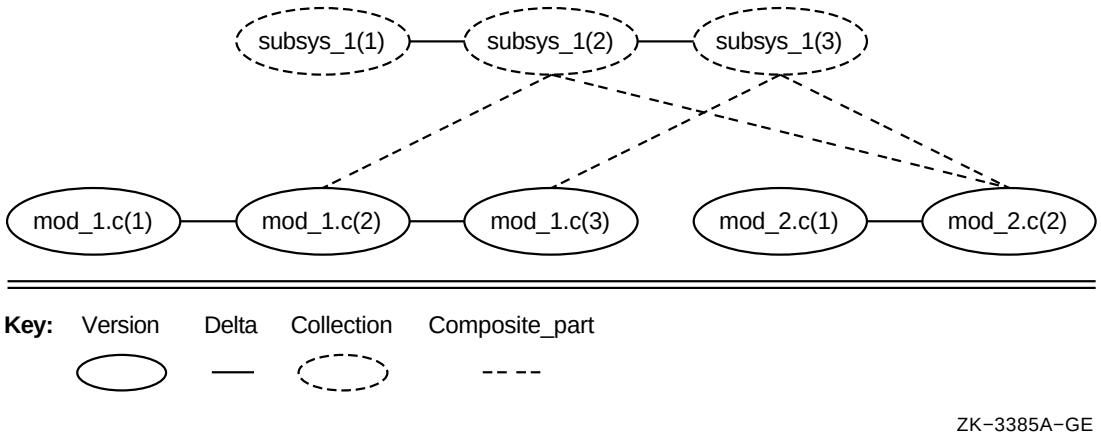
It is not necessary to replace a collection version each time one of its members is replaced. Collection versions are replaced only if you want to make your view of the system visible to others. By default, the contents of reserved versions are visible only to the context that has them reserved.

9.2.3 Collection-Related Properties and Methods

Collection-related properties and methods include VERSION and COLLECTION elements. COLLECTION is a subtype of VERSION, and so inherits all the properties and methods defined for VERSION. COLLECTION defines additional properties that implement the idea of a collection.

Examples of property values in this section refer to the instance diagram in Figure 9–5.

Figure 9–5 Collection Properties



9.2.3.1 Properties on Collections

The properties described in this section are defined on COLLECTION elements.

hasChildren

The value of the **hasChildren** property is a scan of the members of this collection.

Example For the COLLECTION **subsys_1(2)**, the value of **hasChildren** is a scan of versions **mod_1.c(2)** and **mod_2.c(2)**.

allChildren

The value of the **allChildren** property is a scan of the members of this collection and of the members' members, when the members are collections. The process continues recursively until all leaf versions in the collection hierarchy have been found.

Example Because the sample collections contain only noncollection versions, the values of the **allChildren** properties for the collection elements is the same as the values of their **hasChildren** properties.

numChildren

The value of the **numChildren** property is the number of members of this collection.

Example For both **subsys_1(2)** and **subsys_1(3)**, the value of **numChildren** is 2.

9.2.3.2 Properties on Versions

The properties described in this section are defined for VERSION elements. Note that since COLLECTION is a subtype of VERSION, these properties are also available to COLLECTION elements.

hasParents

The value of the **hasParents** property is a scan of the collections that have this version as a member of the collection. As the words imply, a version can be a member of more than one collection.

Example For **mod_1.c(2)**, the value of **hasParents** is a scan of the elmID of **subsys_1(2)**.

For **mod_2.c(2)**, the value of **hasParents** is a scan of the elmIDs of **subsys_1(2)** and **subsys_1(3)**.

9.2.3.3 Collection-Related Methods

attach

The ATTACH message attaches a version to a collection. The message fails if another version of the component is already a member of the collection. The ATTACH message is useful for modifying a configuration to include another component.

detach

The DETACH message detaches a version from a collection. The DETACH message is useful for modifying a collection to exclude a component.

update

The UPDATE message brings a reserved collection "up to date" by detaching old versions and attaching current versions of the same component. The action for each collection member can be one of the following:

- Attach the latest replaced version.
- Attach the latest version, regardless of replacement status. This can be either a replaced version or a version reserved by the sender of the UPDATE message.
- Retain the specific version currently attached to the collection.

The action to take is established by the sender's context and can be overridden on a version-by-version basis.

merge

When sent to a COLLECTION element, the MERGE message carries out some additional processing beyond that needed to merge to VERSION elements. The message attempts to form a collection that is the result of merging the two collections specified in the message. The message returns a list of the elements in the new collection and a list of reasons showing why each was chosen. In the event of a conflict, the message leaves the corresponding list entry empty and indicates that a conflict occurred.

See the description of the MERGE message in the *Oracle CDD/Repository Information Model Volume I* manual for details of this process.

9.3 Contexts and Persistent Processes

So far, this chapter has shown how Oracle CDD/Repository maintains multiple versions of a system object and how individual versions of objects can be gathered up into a collection in order to model a configuration. You also need a means to access to these mechanisms. This requires that you be able to do the following:

- Perform a variety of activities, possibly putting one aside temporarily to start another.
- Work on parts of the system at the same time others are working on those parts. You should be able to have a view of the system that is unique to your activity and that includes the objects on which you are currently working (reserved versions), as well as objects that are publicly available (replaced versions). Other users can have views that do not include your reserved objects.
- Set up a working environment that reflects your preferences.
- Set up several alternative environments.

Contexts and persistent processes combine to satisfy these requirements. A **context** contains pointers to the top of a collection hierarchy and the bottom of a partition hierarchy. (A partition hierarchy organizes and controls visibility of the replaced versions in a repository. See Section 9.4 for information.) The context also contains a pointer to a file system directory that contains the files associated with BINARY elements (and subtypes) in the collection hierarchy. A context provides you with a view of the system under development and provides access to the files that make up the system.

A **persistent process** models an activity in your development environment. It identifies the context that provides your current system view, and includes a pointer to your collection hierarchy that specifies a default collection in which elements are created. It also preserves information about your environment outside the repository, such as operating system symbols and logical names. You can set up multiple persistent processes to model different environments or activities.

Figure 9–6 illustrates the relationship between persistent processes, contexts, and collection hierarchies.

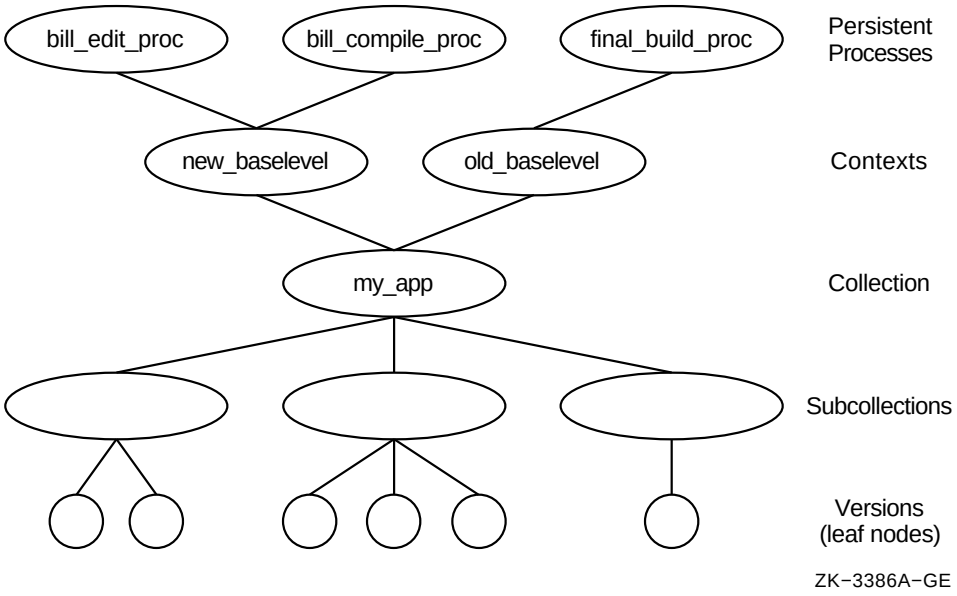
- At the top level of the illustration are persistent processes. These represent activities. The two on the left are both being used by one developer, one for editing and one for compiling. Each one reflects the personal preferences of its owner, and also (through the context) identifies what the owner is working on at the moment.

  Persistent processes can define tool options. For example, a system built from **bill_edit_proc** or **bill_compile_proc** could incorporate the debugger, while a system built from **final_build_proc** uses all available optimizations.

- Each persistent processes identifies a context; each context corresponds to a system configuration. A context identifies (through a pointer to a collection) a set of version elements.

- Each context identifies a collection version (in this example, the same collection, **my_app**). The collection stands at the top of a hierarchy that may include other collections and whose leaf nodes represent the parts that make up the product.

Figure 9–6 Context Management Overview



9.3.1 Contexts

Conceptually, a context is a pointer into a hierarchy of collection versions. The top of the hierarchy is a COLLECTION version whose members may themselves be COLLECTION versions. At the bottom level of the hierarchy are the versions that make up the system under development. Taken as a whole, the hierarchy can represent a complete system. Each level of the hierarchy is then an increasingly finer division of the system into subsystems.

A context that points to a collection limits your view of the repository to the subhierarchy topped by that collection. A context allows you to manipulate the complete system, a subsystem, or a sub-subsystem. By changing or creating contexts, you can obtain different views of the system.

A context is represented by a CONTEXT element in the repository. A CONTEXT element's **top** property points to the COLLECTION version that represents the top of the configuration hierarchy.

To use a context, send the OPEN message to it. This context then becomes your current context. You can send the RESERVE message only if you have a current context. The context owns the ghost VERSION element resulting from the reserve operation. You do not own it, even though you sent the message.

You can see the ghost version only if the reserving context is your current context.

There are several ways that you can set or change a context's **top** property:

- When you create a context, you can specify the collection version that it points to initially.
- When you reserve the collection version pointed to by a context, the context is modified to point to the newly reserved version of the collection.
- When a context owns no reservations or open files, you can change its **top** property directly by sending SETPROP to the CONTEXT element. (See Section 9.5 for information about operations on files.)

9.3.1.1 Properties on Contexts

The CONTEXT element type defines several properties, some of which are described in this section. Refer to Section 9.2.2 and Section 9.3.1.4 for examples of these properties in use.

top

The value of the **top** property is the COLLECTION element at the top of the hierarchy for this context.

checkout

The value of the **checkout** property is a scan of the versions this context currently has reserved. These versions include both collection versions and the versions that make up the leaf nodes of the hierarchy.

defaultAttachment

The value of the **defaultAttachment** property is the default **attachment** for this context. The attachment determines how a version is attached to a collection. Attachment can take one of the following values:

- "specific"—attach a specific version
- "latest checked-in"—attach the latest checked-in version
- "latest"—attach the latest version, whether checked-in or ghost (you cannot attach a ghost version created with another context; you cannot even see it)

basePartition

The value of the **basePartition** property is the base partition in the partition hierarchy. (See Section 9.4 for information about partitions.)

contextDir

The value of the **contextDir** property identifies the top of the file system directory hierarchy in which files associated with the context (under Oracle CDD/Repository control) are located. (See Section 9.5 for information about the Oracle CDD/Repository file system.)

openedFiles

The value of the **openedFiles** property is a scan of BINARY (or subtype) elements representing files that have been opened by the context. (See Section 9.5 for information about operations on files.)

9.3.1.2 Context-Dependent Properties on Version

Some properties have different values depending on the context through which the element is accessed. This section describes the properties on VERSION elements.

parentInContext

The value of the **parentInContext** property is a scan of the COLLECTION elements that immediately include this VERSION in the context. (A version can be a part of several collections.) Collections are included in the **parentInContext** scan only if there is a path from the top of the context's collection hierarchy through the collection to the version that possesses the property.

attachmentInContext

The value of the **attachmentInContext** property is the attachment for this version in the context. As the same version can be attached to different collections in different contexts, the attachment type also can vary depending on context. (See the description of **defaultAttachment** in Section 9.3.1.1 for information about the values **attachmentInContext** may have.)

Usually, the value of **attachmentInContext** will be the same as the value of **defaultAttachment** on the current context. You can override this value for each attachment of a version to a collection.

lastVersion

The value of the **lastVersion** property may differ depending on the context. Its value is the last visible version on a line of descent, regardless of reservation status. If the context that is current when you get the value of this property owns a reserved version on the line of descent, that version is the value of **lastVersion**. All other contexts get the value of that version's predecessor, the last replaced (and publicly available) version on the line of descent.

9.3.1.3 Methods on Contexts

CONTEXT elements implement two important methods: OPEN and CLOSE. Even though these messages are usually sent to contexts as a result of operations on persistent processes (see Section 9.3.2), you should understand their semantics.

Open

The OPEN message prepares a CONTEXT element for use by a user and makes the context current for the session. Subsequent OPEN messages from that same user are allowed.

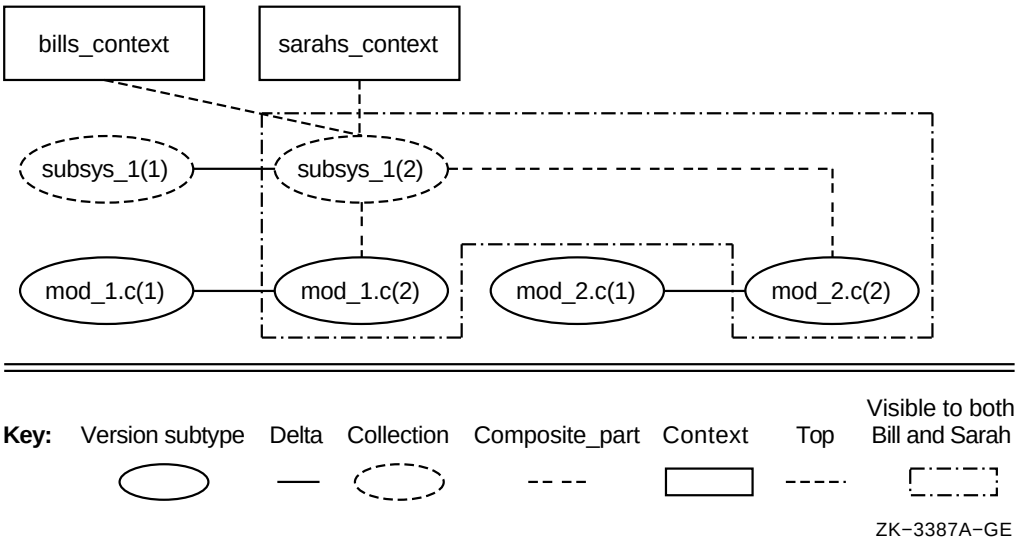
Close

The CLOSE message deletes the CONTEXT element for the current session. After a CLOSE message, the session has no current context.

9.3.1.4 Contexts with Multiple System Views

Multiple contexts allow users to have different views of the same system. Each user can reserve and replace elements as if there were no other users working on the system. Oracle CDD/Repository assures that these multiple activities do not interfere destructively with each other.

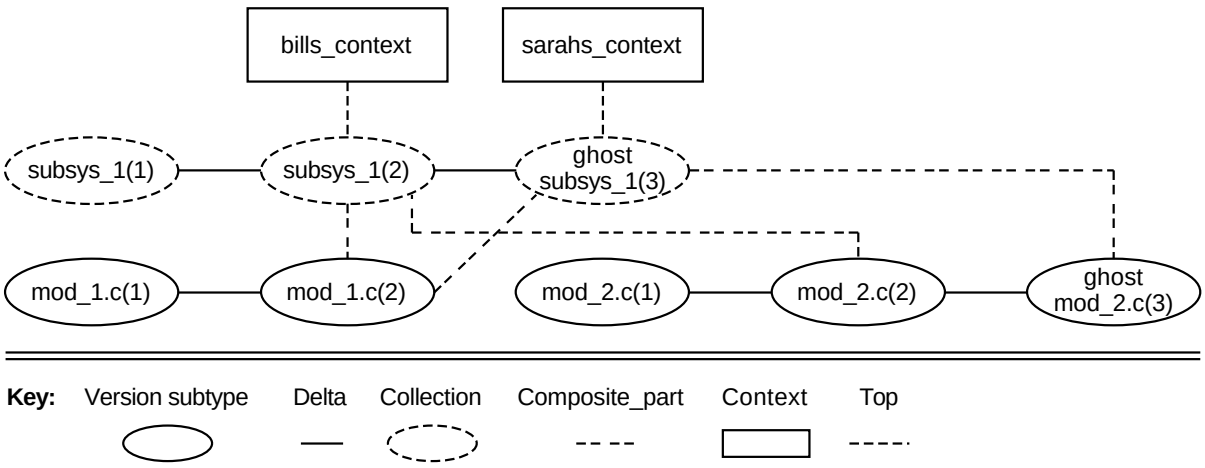
The following example shows how two users can work independently on a system. The illustration shows a very simple configuration. None of the VERSION elements have been reserved. The contexts **bills_context** and **sarahs_context** both point to the same collection version, so Bill and Sarah begin with the same view of the system.



Sarah now reserves **mod_2.c**. (She does not need to use a version number because a collection can contain only one version of a component.) This reservation causes the following changes:

- A new ghost version of **subsys_1**, **subsys_1(3)**, is created and the **top** property of **sarahs_context** is adjusted to point to it.
- A new ghost version of **mod_2.c**, **mod_2.c(3)**, is created. **subsys_1(3)** is adjusted to point to this new version. This adjustment is possible because **subsys_1(3)** is in the ghost state and is therefore mutable.

Bill and Sarah now have different views of the system. Bill's view includes **subsys_1(2)**, **mod_1.c(2)**, and **mod_2.c(2)**. Sarah's view includes **subsys_1(3)**, **mod_1.c(2)**, and **mod_2.c(3)**. The new state of the system is shown in the following diagram.



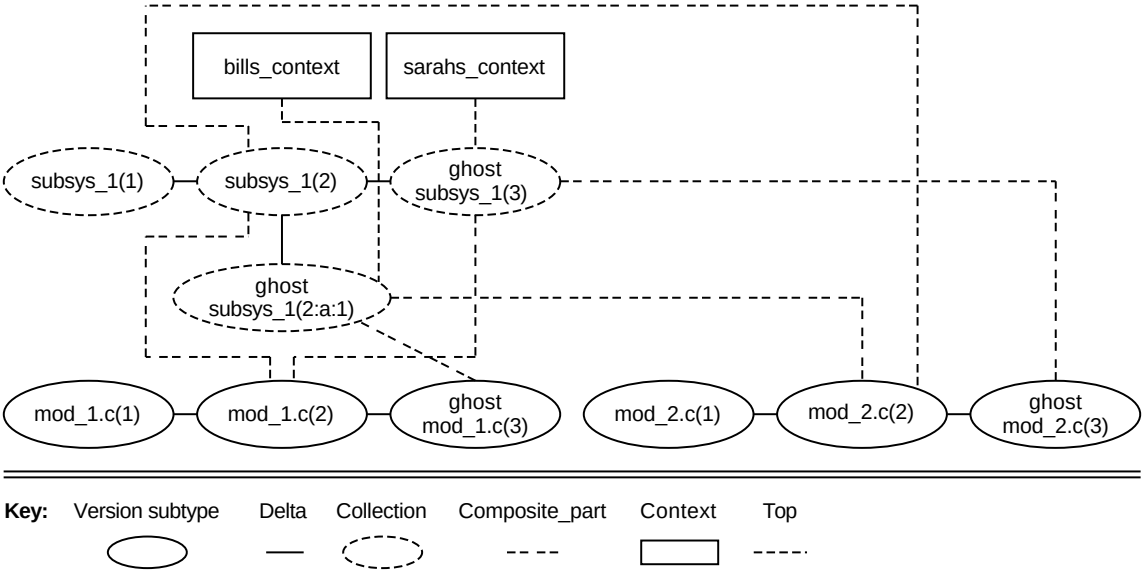
ZK–3388A–GE

Bill then reserves **mod_1.c**. This reservation causes the following changes:

- Oracle CDD/Repository attempts to reserve **subsys_1(2)**, the collection containing **mod_1.c(2)**, in preparation for creating a new version of **mod_1.c(2)**. However, **subsys_1(2)** has already been reserved by Sarah (to work on **mod_2.c**), so Oracle CDD/Repository returns an error. Bill therefore asks Oracle CDD/Repository to create a variant branch of **subsys_1**, specifying the branch name *a*. Oracle CDD/Repository creates the initial version of this branch, **subsys_1(2:a:1)**. The collection version **subsys_1(2:a:1)** is a ghost belonging to Bill, and initially includes **mod_1.c(2)** and **mod_2.c(2)**.

- The ghost version **mod_1.c(3)** can now be created. **subsys_1(2:a:1)** is adjusted to include **mod_1.c(3)** instead of **mod_1.c(2)**.

Bill's reservation forces the creation of a variant branch of the collection **subsys_1**, even though Bill thinks he has the system to himself. The next figure shows the resulting system.

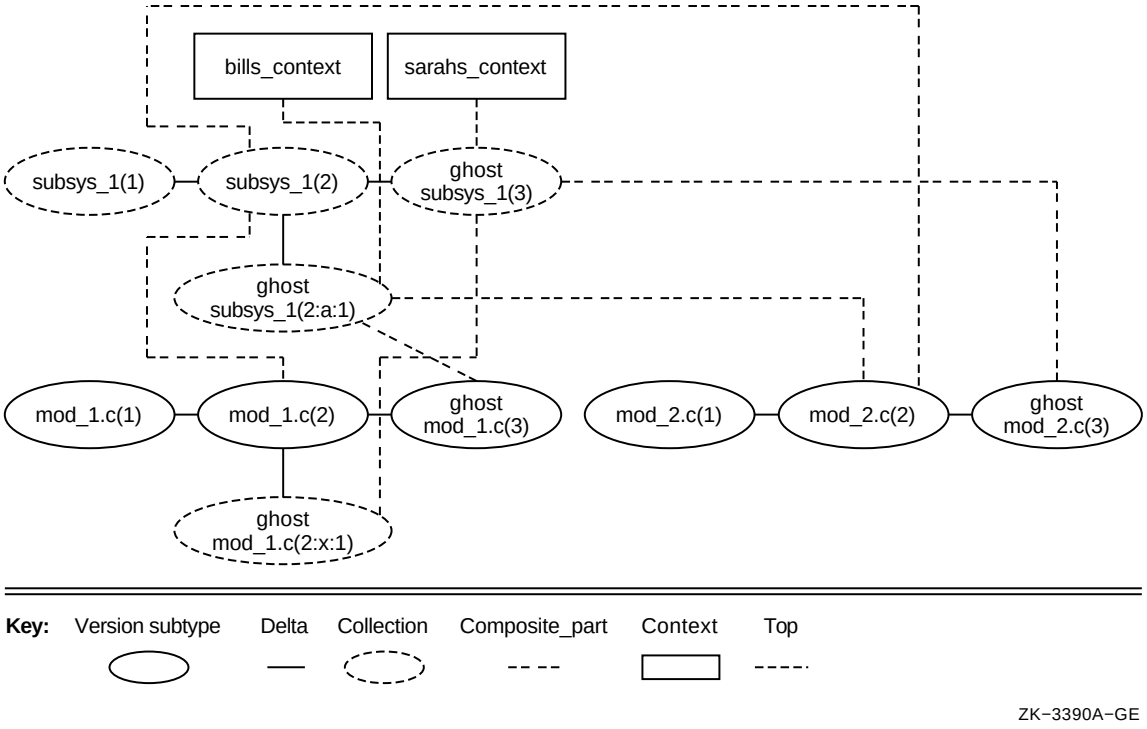


ZK–3389A–GE

Sarah then reserves **mod_1.c**. The version of **mod_1.c** in her current system view is **mod_1.c(2)**, so Oracle CDD/Repository attempts to reserve that version. Although Sarah does not know it, Bill previously reserved **mod_1.c(2)**. Therefore, Sarah must ask Oracle CDD/Repository to create a variant branch. The initial version of this branch is **mod_1.c(2:x:1)**.

Sarah's current collection, **subsys_1(3)**, is a ghost, so Oracle CDD/Repository can adjust it to include **mod_1.c(2:x:1)** without reserving it first.

The next figure shows the resulting state of the system.



In the course of this example, two variant branches have been created to allow Bill and Sarah to work on the system at the same time. At some point it will be necessary for these branches to be merged into the main lines of descent.

9.3.2 Persistent Processes

The element type PERSISTENT_PROCESS supports the Oracle CDD/Repository ability to "remember" the state of your work. You can have several persistent processes available, each representing an activity. The purpose of the persistent process is to allow you to switch easily from one activity to another. For example, you might have one persistent process for correcting errors on the released system, another for design work on the next major release, and a third for project administration. By switching from one persistent process to another, you can change activities.

A persistent process contains information about several aspects of an activity:

- The environment in which the activity takes place. The environment includes default directory or path, logical names or environment variables, command synonyms, and similar information.

- The degree to which the activity has progressed. By identifying a context, the persistent process carries information about work in progress in the form of reserved version elements in the configuration hierarchy.

The persistent process is the way you typically access CONTEXT elements. Instead of opening a context directly, you open a persistent process. The persistent process in turn opens the context associated with it. If you are finished with an activity for the time being, you close the activity's persistent process, thereby closing the context as well. You also can switch to another persistent process, leaving the first persistent process and its context open.

A persistent process can be used by only one (operating system) process at a time. However, a persistent process is not bound to a single user. Any user can open any idle persistent process, subject to access control restrictions.

9.3.2.1 Properties on Persistent Processes

currContext

The value of the **currContext** property is the element ID of the current CONTEXT element. This identifies the configuration in which you are working, as described in Section 9.3.1.

Using SETPROP to change the value of **currContext** results in the following sequence of events:

1. A CLOSE message is sent to the CONTEXT element identified by **currContext**.
2. The value of **currContext** is changed.
3. An OPEN message is sent to the new CONTEXT element. If the open should fail, the persistent process is left without an open context.

currCollection

The value of the **currCollection** property is the element ID of a COLLECTION element within the configuration identified by **top**. This is similar in concept to a default directory in your directory tree. Elements created while the persistent process is active are attached to the current collection by default.

aliases

The value of the **aliases** property is a list of command string aliases. These correspond to csh aliases or DCL symbols.

symbols

The value of the **symbols** property is a list of string symbol definitions. These correspond to OpenVMS logical names.

9.3.2.2 Methods on Persistent Processes

Open

The OPEN message, sent to a PERSISTENT_PROCESS element, performs the following functions:

1. Establishes the PERSISTENT_PROCESS as the current one for the session.
2. Opens the CONTEXT element identified by the **currContext** property on the PERSISTENT_PROCESS.

If the OPEN operation on the CONTEXT element fails, then the OPEN operation on the PERSISTENT_PROCESS also fails.

Close

The CLOSE message closes a persistent process. Closing a persistent process also causes the CLOSE message to be sent to its current context.

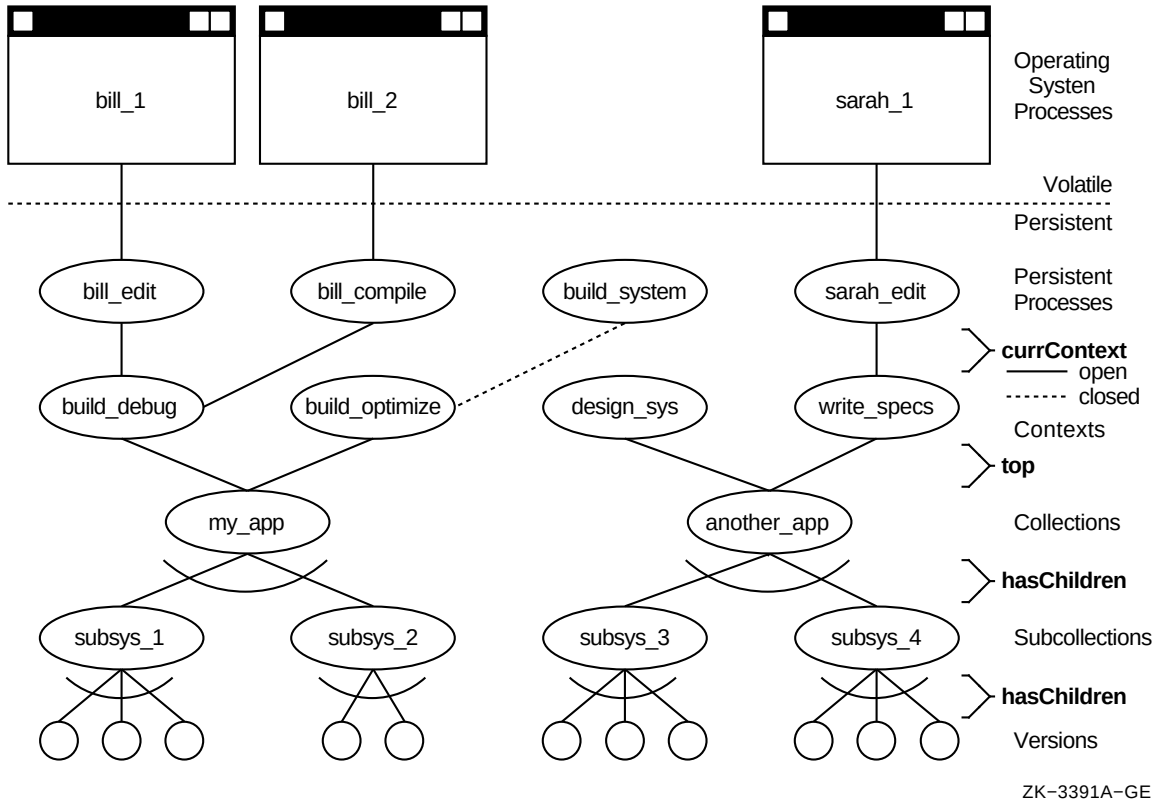
9.3.3 Persistent Processes and Contexts

This section summarizes how operating system processes, persistent processes, and contexts work together.

Figure 9–7 illustrates a situation with two users and a number of persistent processes and contexts. The following points are important:

- Bill has two open persistent processes, **bill_compile** and **bill_edit**. They both identify a single context, **build_debug**.
- No one is using the persistent process named **build_system**. Although the value of its **currContext** property is **build_optimize**, that context is not open because **build_system** is not open.

Figure 9–7 Using Persistent Processes and Contexts



9.4 Partitions

In previous sections, this chapter described two organizing principles for versioned elements in the repository:

- version control, which allows you to store successive versions of an object in the repository
- configuration modeling, in which collections gather together versions of different objects into a representation of a system

Partitions provide a third way to organize versions. With partitions, the organizing principle is the *level of stability* or *approval* that a replaced version has achieved. Partitions allow replaced versions that are more stable to be made visible to a broad range of users, while less-stable versions are visible

only to a small group of users, those who need access to the newer versions of objects at the cost of stability.

Conceptually, partitions are "containers" for replaced versions, organized in a hierarchy. Versions residing in partitions in the lower levels of the hierarchy can be promoted to higher levels as they meet stability criteria. You connect to a partition through its contexts. You can then see replaced versions in that partition and in higher partitions, *but not in lower partitions*.

You should realize that version control, configuration modeling, and partitions are overlapping organizing structures. For example, a versioned element is always subject to version control. A versioned element also is typically part of a collection. If it is replaced, it always resides in a partition. Each structure provides a different type of control: version control, configuration control, and visibility control.

9.4.1 Visibility Control

The goal of visibility control is to allow access to objects that have reached a specified level of approval. Different users have different needs. An engineer who is working on a new release of a product needs access to the latest modules, even though they may not have been thoroughly tested. On the other hand, an engineer who is building the product for release wants access to only those modules that have been thoroughly debugged and approved by a testing organization.

One engineering group may want access to modules developed by another group or to definitions maintained on a company-wide basis. Although the first group may be working with their own code in a highly unstable state, they generally want stable modules from other groups. Therefore, it should be possible to access another group's modules in such a way that only modules that have achieved a specified approval level are visible.

In some engineering environments, the process of approval and promotion to higher levels of stability may be highly formalized, or even dictated by standards or contractual requirements. A visibility control system must be flexible enough to implement these formal systems as well as those using a more casual approach.

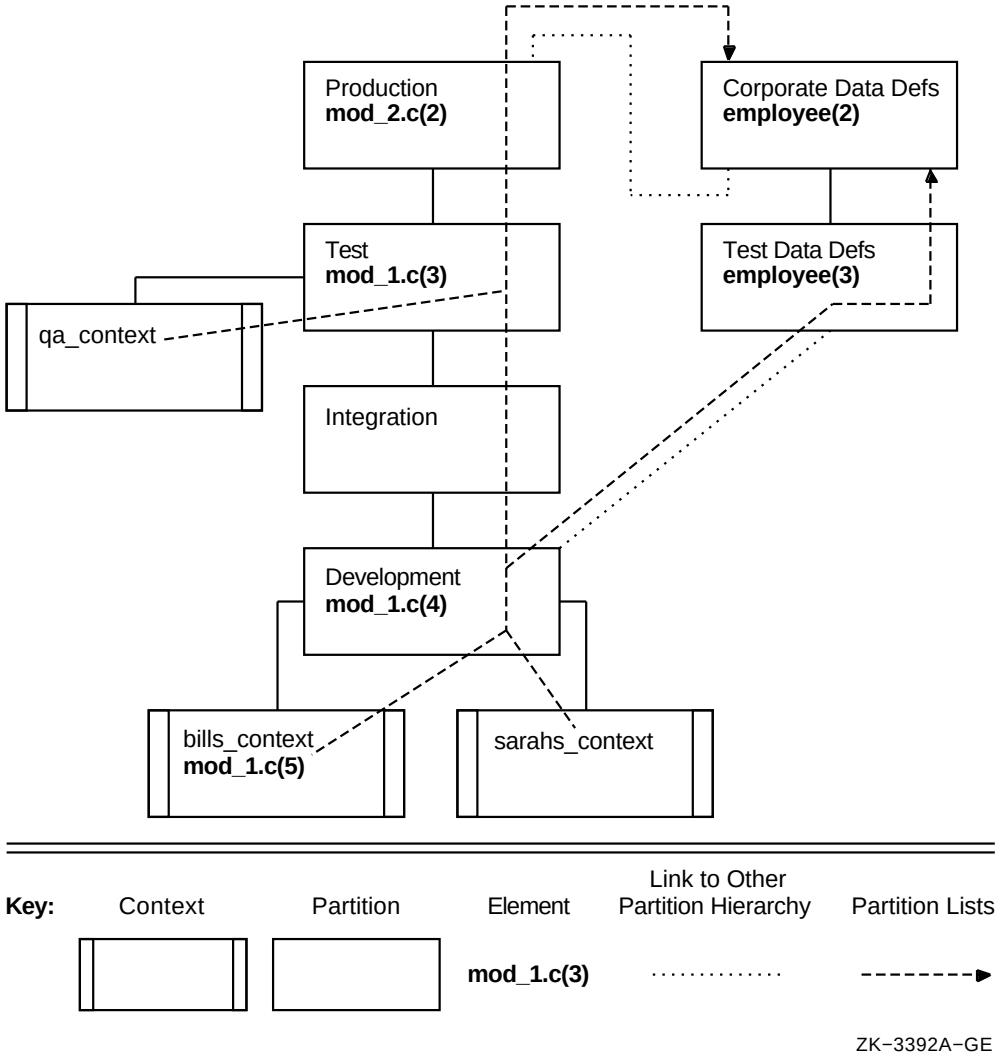
Figure 9–8 illustrates a sample visibility control system, in which partitions implement the levels of visibility. Several version elements (subtypes of VERSION) are shown in the system to illustrate the various operations that are possible in the model. The sections that follow describe these operations.

The following list outlines the important points about the system shown in Figure 9–8:

- Version elements reside either in a context or in a partition. A version element that has been reserved can exist only in a context. A version element that has been replaced can exist only in a single partition. Various versions of **mod_1.c** reside in various contexts and partitions, but no version of **mod_1.c** resides in more than one place.
- The fact that a version element resides in a specific partition implies the level of approval it has reached. Residence in partitions higher in the partition hierarchy indicates more rigid criteria for approval, and thus greater stability. Residence in lower partitions indicates looser criteria for approval, and thus less stability. The names of the partitions reflect this.
- Your view of a system object depends on your view into the partition hierarchy. This view is established by a pointer from your CONTEXT element to a partition, called the **base partition** for the context. You can see versions in your base partition, as well as versions in higher partitions.

  For example, Bill sees **mod_1.c(5)**, **mod_1.c(4)**, and **mod_1.c(3)**. Sarah sees **mod_1.c(4)** and **mod_1.c(3)**. A user of **qa_context** sees **mod_1.c(3)**. However, all three users see **mod_2.c(2)**. Figure 9–8 shows each context's view through the partition hierarchy with **partition lists** formed by the partitions visible to each user.
- Two partition hierarchies exist side by side. Pointers indicate that the hierarchy on the right is accessible from the hierarchy on the left. For example, Bill and Sarah can see both **employee(3)** and **employee(2)**. However, the user of **qa_context** sees only **employee(2)**.

Figure 9–8 A Sample Visibility Control System



9.4.2 Promotion

Pieces of a software system migrate from lower partitions to higher levels through a **promotion** process. This process requires approval of one or more members of the organization that is responsible for a particular level of approval. When a version is promoted, further changes to that component are not visible at the higher level until those changes are also promoted.

As a version is promoted through the partition hierarchy, it is assumed to be more stable. The version should have passed more acceptance testing or reviews and should be more error free. Changes at higher levels are made only through rigid approval and tracking mechanisms. There also should be fewer versions at higher levels.

Promotion must take place one level at a time. For example, if a version is currently at the "Development" level in the sample partition hierarchy, it cannot be promoted to the most stable level without first being promoted to the intermediate level. Thus, **mod_1.c(4)** can be promoted to the "Integration" level but not directly to the "Test" level.

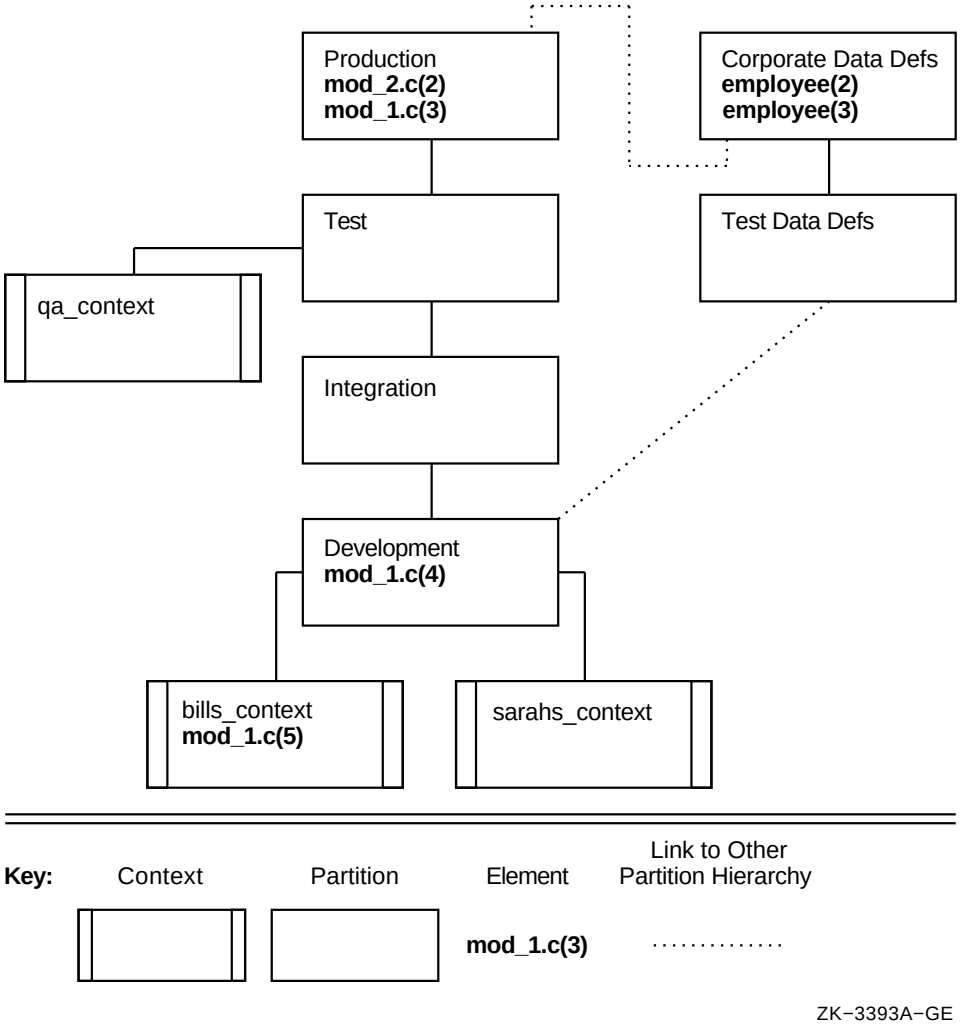
You can restrict the promote operation to specific users. Further, different sets of users can have promote access at different levels in the partition hierarchy.

Note

Do not make nonversioned objects subject to visibility control. After it is promoted, a nonversioned object can still be changed, but the approved version cannot exist simultaneously with the changed version.

Figure 9–9 illustrates promotion in the version control system shown in Figure 9–8. **employee(3)** has been promoted to the "Corporate Data Defs" partition; it now becomes the most recent version of the component at that level. **mod_1.c(3)** has been promoted to the "Production" partition.

Figure 9–9 Promotion in the Sample Visibility Control System



To promote a version to the next-higher level in the partition hierarchy, send the PROMOTE message to it. You can promote only to the parent partition, not between partition hierarchies. For example, you cannot promote **mod_1.c(4)** from the "Development" partition to the "Test Data Defs" partition because "Test Data Defs" is not the parent partition for "Development." (See Section 9.4.4 for a restriction on promotion imposed by dependency relationships.)

As new versions are promoted to higher partitions, the previous versions of the component at that level are still accessible to users so that they can continue to work against these versions and so that a previous state of the system can be accessed at a later date. For example, in Figure 9–9, **employee(2)** is still available even though **employee(3)** has been promoted to the same level. **mod_1.c(3)** is still available to Sarah even though **mod_1.c(4)** resides at a lower approval level.

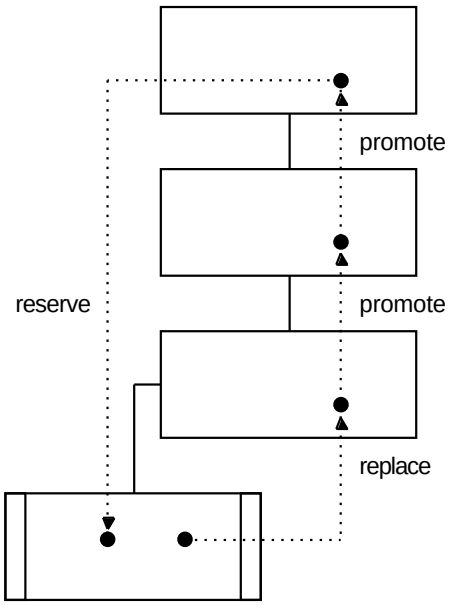
9.4.3 Reserve–Replace–Promote Cycle

Successive versions in a component follow a cycle. You create a particular version in the reserved state and it is in your context, invisible to other users. If you replace the version, it moves from your context into your base partition (the partition associated with your CONTEXT element). This makes a context a special type of partition and replacement a special type of promotion.

After entering the partition hierarchy at a low level, the replaced version can be promoted up through several levels of the hierarchy as it meets ever-stricter approval criteria. Eventually, however, you create a new version and a user reserves the new version. The old promoted version remains in its partition, available to others who still need it. The new version is created in the context of the user who reserved it to begin the cycle again.

Figure 9–10 illustrates this cycle.

Figure 9–10 Reserve–Replace–Promote Cycle



ZK–3394A–GE

9.4.4 Partitions and Dependency Relationships

The PROMOTE message enforces the general rule that *a version cannot depend on a version in a lower partition.* The message fails if the requested operation violates the rule.

An element that owns a dependency relationship to another element is said to depend on that element. (See Chapter 8 for information about dependency relationships.) The relationship is meant to express the notion that if an element changes, elements that depend on it may have to change also, or may no longer be valid.

This restriction means that an object cannot be any more stable than the least-stable object that it depends on. For example, since a derived object such as an executable file depends on its source objects, a system that is built up from many source files is only as stable as the least-stable of those source files.

The relationship between a COLLECTION element and the versions in the collection is a dependency relationship owned by the COLLECTION element; therefore, a collection depends on its members. For purposes of promotion, this means that a replaced collection must be in a partition that is no higher

than that of the lowest of its members. (You cannot reserve any members of a replaced collection. It is forbidden by the semantics of version control.)

Oracle CDD/Repository does allow you to promote collections as a unit, by using the "promote-to-bottom" option. Promoting the collection without first promoting those of its members occupying the same partition violates the dependency relationship restriction because the collection would depend on elements (its members) in a lower partition.

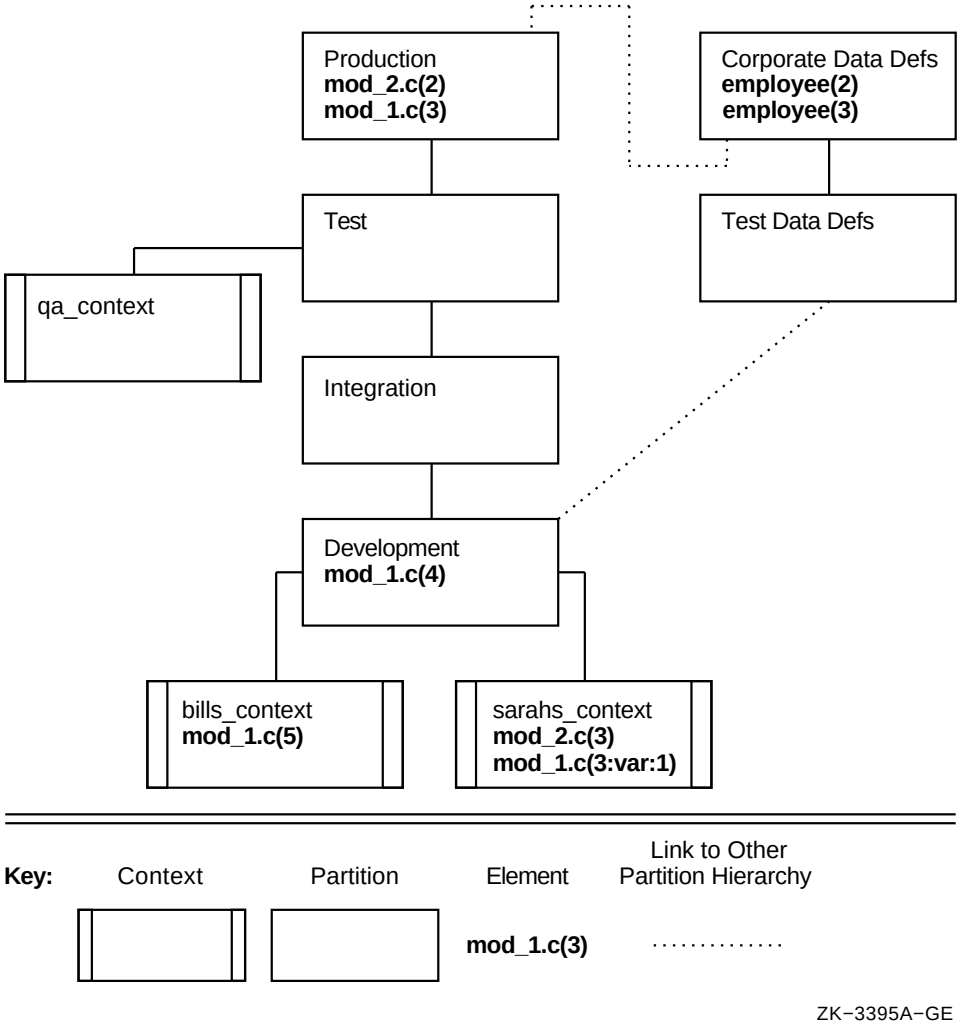
9.4.5 Intercomponent Relationship Support

If a user starts referencing a new system configuration, be sure to check if it contains a version of a component that is already reserved by that user. If it does, check to see if that version is an ancestor of the version that person reserved. If it is not, you must notify the user of the new configuration that it may contain intercomponent dependencies that will not work correctly with the versions that the user already has reserved.

This can occur when one user reserves a version, and another user reserves and replaces a version of the same component as well as a version of another component, and the changes contain intercomponent dependencies. The second user's changes are then promoted and a new configuration is created. At this point, the first user elects to use the new configuration and will get the changed version of the second component but not the first component, because a version of the first component is already reserved to the first user.

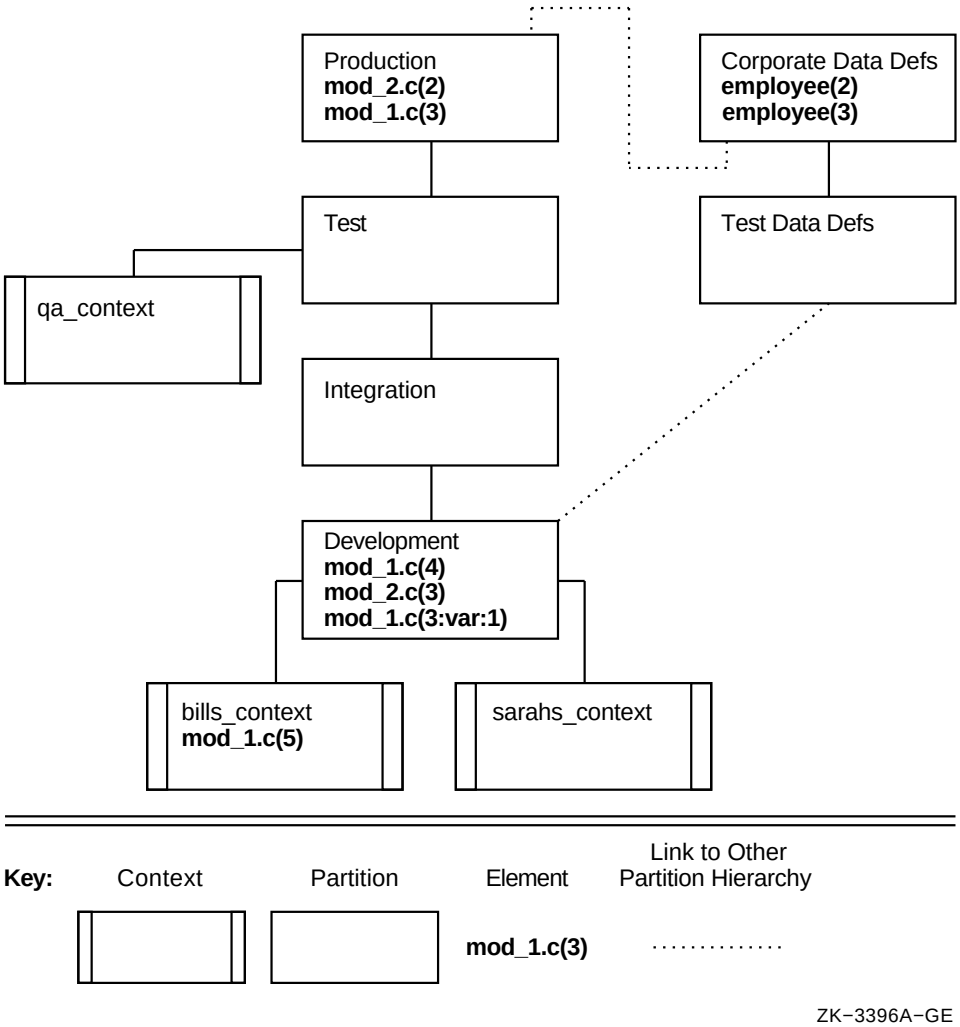
For example, starting from Figure 9–9, assume that **mod_2.c(2)** and **mod_1.c(3)** make up a configuration. In this example, Bill has already reserved a version of **mod_1.c** that is named **mod_1.c(5)**. Therefore, Bill sees **mod_2.c(2)** (from the configuration) and **mod_1.c(5)** (from his context). Now Sarah reserves **mod_2.c(2)** and **mod_1.c(3)**, creating the ghost versions **mod_2.c(3)** and **mod_1.c(3:var:1)** in her context. (The reservation of **mod_1.c(3)** creates a variant branch because higher-numbered versions already exist.) Figure 9–11 shows the resulting system state.

Figure 9–11 Reserving Versions from a Configuration



Sarah finishes her changes and replaces the two new versions; they go into the "Development" partition. She defines a configuration that includes the new versions. Figure 9–12 shows the new system state.

Figure 9–12 Creating a New Configuration



Bill decides to start using the new configuration. He now sees **mod_2.c(3)** from the new configuration, but still sees **mod_1.c(5)** from his context. If Sarah has made changes to a call interface between **mod_2.c(3)** and **mod_1.c(5)**, Bill's system breaks. This is why Bill must be notified of the changes.

9.4.6 Support for Uncontrolled Change

To support existing dictionary applications, you must be able to change a version in place, without reserving and replacing it.

A version that can be changed in place is said to be **uncontrolled**; it exists outside the normal restrictions of the version control model. Uncontrolled change is supported only for elements that are outside of any partition hierarchy. When an element is promoted to the lowest controlled level, all versions of the element become controlled. They can no longer be modified without following the reserve/replace model. Also, you cannot make a controlled element uncontrolled by removing it from a partition. After an element is controlled, its versions must always reside in a partition.

You can control a version by sending it the CONTROL message, which effectively promotes it to the sender's base partition.

Controlled versions cannot be dependent on uncontrolled elements. An error is returned when you attempt to make a version controlled if that version depends on uncontrolled elements.

9.4.7 Properties Related to Partitions

Partitions are represented by instances of the PARTITION element type.

A partition is associated with only one repository database. Therefore, the partition that a version is in implies its database. This makes it possible for promotion to control the physical location of versions.

Partitions can be replicated to other databases. Partition **A** might be associated with database DB3 but replicated to database DB2. If a version is moved to partition **A**, it is replicated in both DB2 and DB3.

Because partitions are elements, they have properties. The following properties are associated with partitions.

parentPartition

This property identifies the parent partition in the partition hierarchy. Because the partition hierarchy is a strict hierarchy, a partition can have only one parent; therefore, the value of this property is an element ID, not a scan.

childParitions

This property identifies the child partitions in the partition hierarchy.

related

This property identifies one or more related partitions. The related partitions may be in different partition hierarchies or in the same partition hierarchy. To determine a list of partitions to search, Oracle CDD/Repository starts at the base partition, then works upwards following the **parentPartition** and **related** properties at each partition it reaches, until reaching the root of all partition hierarchies.

instances

This property is a scan of element IDs of the VERSION elements contained in the partition.

autopurge

This property defines whether intermediate versions of elements are purged on replace. For example if **a(2)**, **a(3)**, and **a(4)** are contained in a partition dedicated to system testing, a REPLACE operation should not purge **a(2)** and **a(3)** to preserve the change history. If the same three versions exist in a partition intended for system development, **a(2)** and **a(3)** should be purged before replacing **a(4)** since only the final version is of interest.

basePartition

The CONTEXT element type defines this property. The property identifies a user's current base partition, which defines the user's upward view into the partition hierarchy and any related partition hierarchies.

inPartition

The VERSION element type defines this property. The property identifies the partition in which a replaced version resides.

9.5 File Management

Many of the objects that make up a system under development are files. Repository elements that are subtypes of BINARY represent files in the native file system. Messages sent to these elements may result in operations on the corresponding file.

Oracle CDD/Repository defines two different storage types for files: "internal" and "external". The value of the **storeType** property on BINARY elements (and subtypes) indicates the storage type.

Oracle CDD/Repository actively manages files whose storage type is internal. For files whose storage type is external, Oracle CDD/Repository only stores the file's location. The following sections describe how Oracle CDD/Repository deals with files of each storage type.

9.5.1 Internal-Storage Files

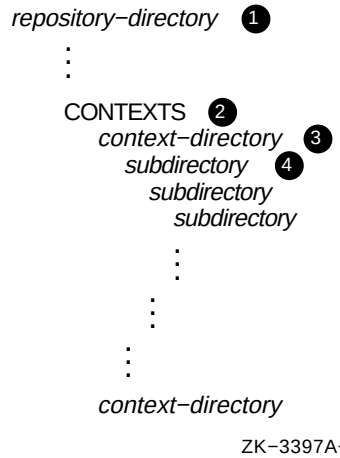
When a file's storage type is "internal", Oracle CDD/Repository manages the file in the following ways:

- It uses a delta mechanism to store successive versions of files.
- It makes files available in a way that is transparent to the user. The user does not see the delta mechanism or know where (or how) the file has been stored.
- It maintains files in a directory structure that it creates (as part of the process of creating a repository) and manages.

9.5.1.1 Directory Structure

When you create a repository, you specify a directory to contain the repository. This directory becomes the root of the directory structure that contains internal-storage files. The directory structure is depicted and described in Figure 9–13.

Figure 9–13 Directory Structure for Internal-Storage Files



❶ You specify this directory when you create the repository. It contains various files that implement the repository and also serves to contain (in subdirectories) internal-storage files.

❷ Subdirectories of this directory correspond to CONTEXT elements created in the repository. Oracle CDD/Repository creates these context directories when you creat new CONTEXT elements.

❸ There is one directory per CONTEXT element. Each context directory serves as a root for a directory hierarchy. Because context names are unique, the directory name is the same as the context name unless you supply a different name at the time you create the CONTEXT element.

❹ You can create a directory hierarchy that maps to each context's current directory hierarchy, using the context directory as the root. When a user reserves a file, Oracle CDD/Repository creates it in the file system directory that corresponds to the Oracle CDD/Repository directory that contains it. The file system directory also contains file system links corresponding to replaced BINARY elements in the collection that the user has opened; the file system links point to the actual files in partition directories.

9.5.1.2 Context and Collection Directories

An Oracle CDD/Repository directory structure contains a subdirectory for each CONTEXT element within the repository. This subdirectory serves as the root for a directory tree. The directory tree mirrors the Oracle CDD/Repository subdirectory hierarchy identified by the name of each element.

A CONTEXT element's **contextDir** property identifies the name of the context directory. The value of this property is a simple name, not a path name; a subdirectory of that name is created under the CONTEXTS directory when the context is created. When you create the CONTEXT element, you can supply a value for **contextDir**. If you do not, Oracle CDD/Repository uses the name of the context. You also can use **setProp** to change the value of **contextDir** after the context has been created. Doing so renames the corresponding context directory.

Subdirectories in the collection directory tree contain files corresponding to reserved or opened BINARY (and subtype) elements in collections within the context's current collection hierarchy. You can create private, modifiable files in these subdirectories by sending RESERVE messages to the corresponding elements.

Sending the OPEN message to a replaced element creates a file system link in the context subdirectory to a read-only file.

A REPLACE or UNRESERVE message sent to a reserved element deletes the corresponding file from the collection directory. A CLOSE message removes the link to an open file from the collection directory.

Note

Many users can open a replaced BINARY element. To avoid wasting disk space by having multiple copies of a single read-only file on the system,

Oracle CDD/Repository creates one copy of the file and establishes file-system links from context subdirectories to that file. The file is deleted after all users have closed the corresponding element. This mechanism is transparent to users; each user appears to have a read-only copy of the file in his or her collection directory.

If an element has multiple names, the element is listed once for each name in the context's repository.

Reserved files are created in the directory corresponding to the primary name of the element. If the element has more than one name, the other names are ignored. An element's **name** property identifies the name of the subdirectory for that element.

The depth of the Oracle CDD/Repository directory hierarchy is limited by the file system limit of eight on nested subdirectories. In the following example, the repository is created in directory [SMITH.REPOS]. Oracle CDD/Repository creates the subdirectory [.CONTEXTS] to contain context directories, of which [.MY_CTX] is one. The remaining four subdirectories are available to model the current Oracle CDD/Repository directory hierarchy. In this example, the element with the full name of [SMITH.REPOS]/DIR_1/DIR_2/DIR_3/DIR_4 /FOO.C is displayed in the file system as:

```
[SMITH.REPOS.CONTEXTS.MY_CTX.DIR_1.DIR_2.DIR_3.DIR_4]FOO.C
```

9.5.1.3 File-Related Properties

In addition to the **contextDir** property previously described, the following properties help implement the Oracle CDD/Repository file system. All the properties are defined by BINARY.

storeType

The value of the **storeType** property determines whether a file is stored internally or externally.

filePath

The **filePath** property always contains the native file system file specification for a file, if the file is available. This is the property a tool can use to locate a file regardless of its storage type.

For internal-storage files, the value of **filePath** is the file's location in the repository directory structure. The value of the property is computed by concatenating the following:

- repository directory
- CONTEXTS (as appropriate)

- context directory name
- collection directory name(s)
- name of the file

If the BINARY element is a replaced version that has not been opened, the file does not exist. If you attempt to get the value of **filePath** Oracle CDD/Repository returns an error.

For external-storage files, the value of **filePath** is the full file specification in the native file system.

openedBy

The value of the **openedBy** property is a scan of all the CONTEXT elements that have sent the OPEN message to the element.

9.5.2 External-Storage Files

If the storage type of a file is external, Oracle CDD/Repository does not manage the storage of that file. To create a BINARY element representing an external-storage file, specify a **storeType** value of "external" and supply the file specification as the value of the **storedIn** property.

For a BINARY subtype that represents an external-storage file, the value of the **filePath** property is the native file system file specification as contained in the **storedIn** property. It is up to the user to make sure that the value of the **storedIn** property correctly identifies the file in the native file system.

9.5.3 Importing and Exporting Files

The IMPORT and EXPORT messages copy files into and out of the Oracle CDD/Repository file system, respectively.

import

The IMPORT message creates or supersedes an internal-storage file from the contents of a specified file in the native file system. Sending this message sets the value of the BINARY element's **importedFrom** property to the file specification from which the file was imported.

export

The EXPORT message creates a specified file in the native file system from the contents of the BINARY element to which it is sent.

To create an internal-storage file from an existing file in the native file system, follow these steps:

1. Use the NEW message to create a new instance of BINARY, specifying a store type of "internal". This creates a replaced version 1.

2. Reserve the replaced version.
3. Send the IMPORT message to the reserved version, specifying the file to be imported with the message.

You also can specify a value for **importedFrom** in the argument list to the NEW message. In this case, all these steps are done automatically.

9.5.4 Example

This section contains an example of the Oracle CDD/Repository file system in use. The example shows how a series of operations on a collection hierarchy and its contents manipulates the corresponding directory structure and the relevant properties.

Figure 9–14 illustrates the starting configuration for the example.

Figure 9–14 File System Example: Initial State

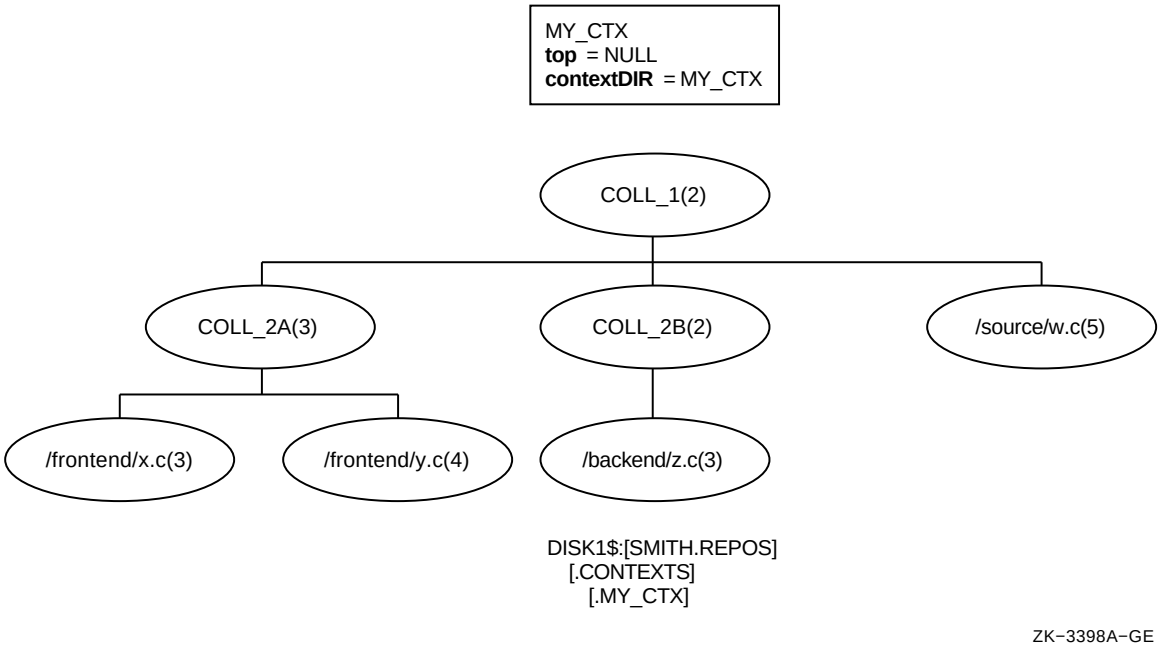


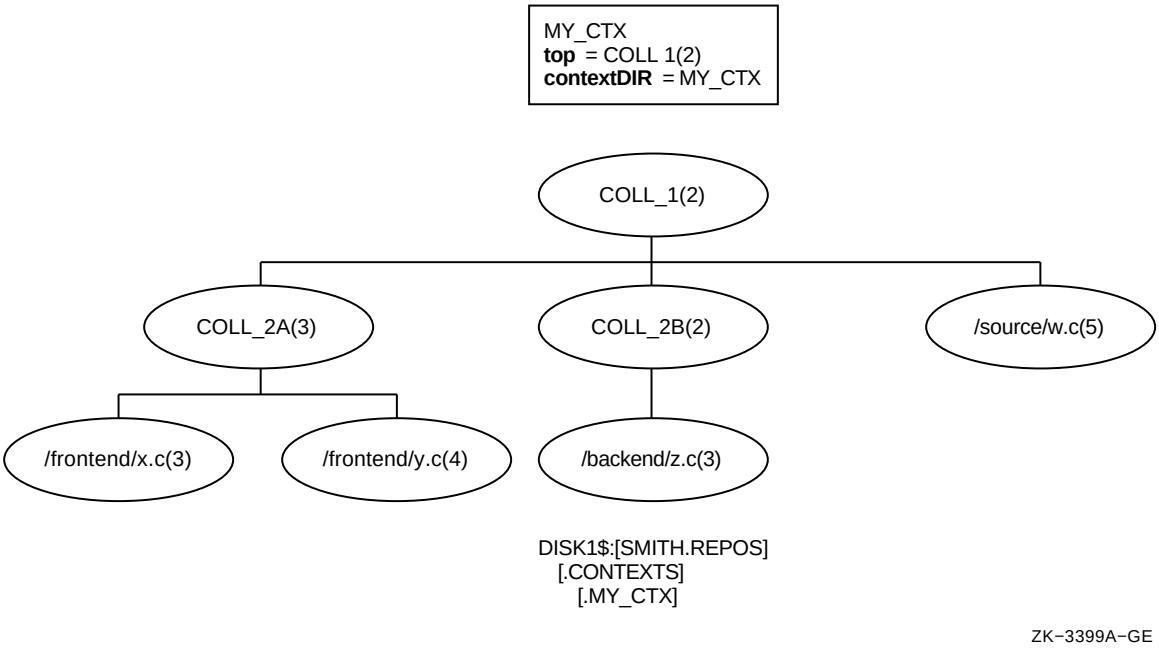
Figure 9–14 shows the following:

- The value of the **top** property for the context **MY_CTX** has not been set. Therefore, there are no directories under the corresponding context directory MY_CTX.

- The collection hierarchy headed by **COLL_1(2)** consists entirely of replaced versions. The leaves in the collection hierarchy are all BINARY subtypes.

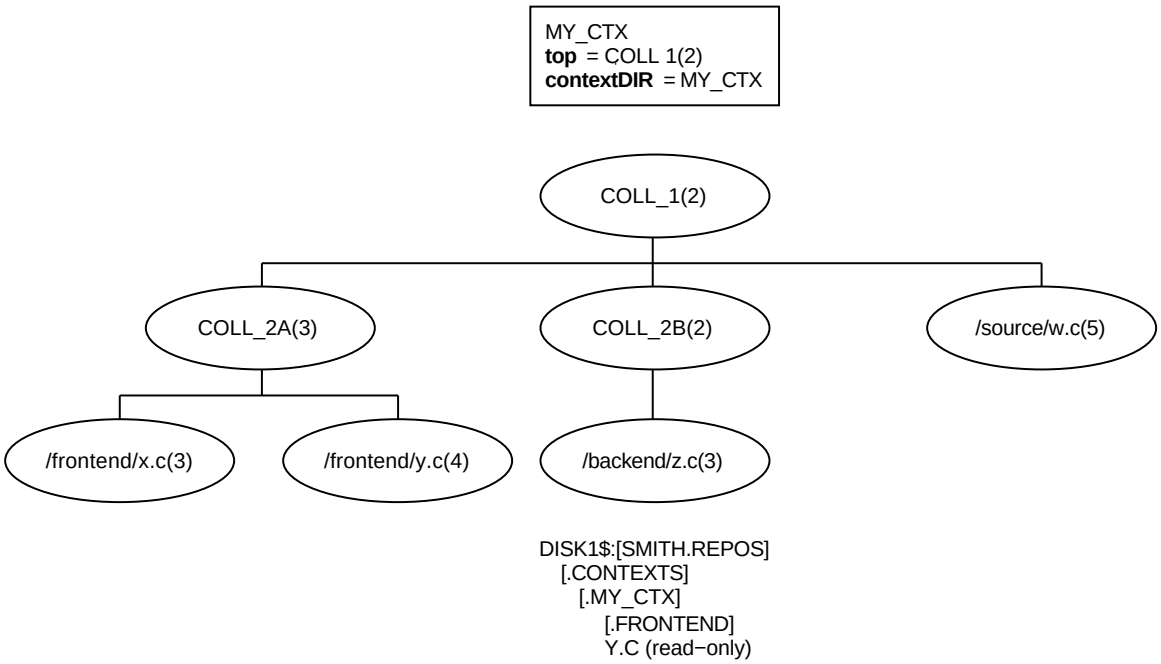
Set the context's **top** property to **COLL_1(2)**. Figure 9–15 shows the results of this action. If **MY_CTX** was previously set to a different collection hierarchy, the directory structure is deleted as a result of changing the value of **top**.

Figure 9–15 File System Example: Setting top



Oracle CDD/Repository does not create any directories until an element has been opened or reserved. At that time, Oracle CDD/Repository creates only the directories needed to contain the element.

Figure 9–16 File System Example: Opening a BINARY **Element**

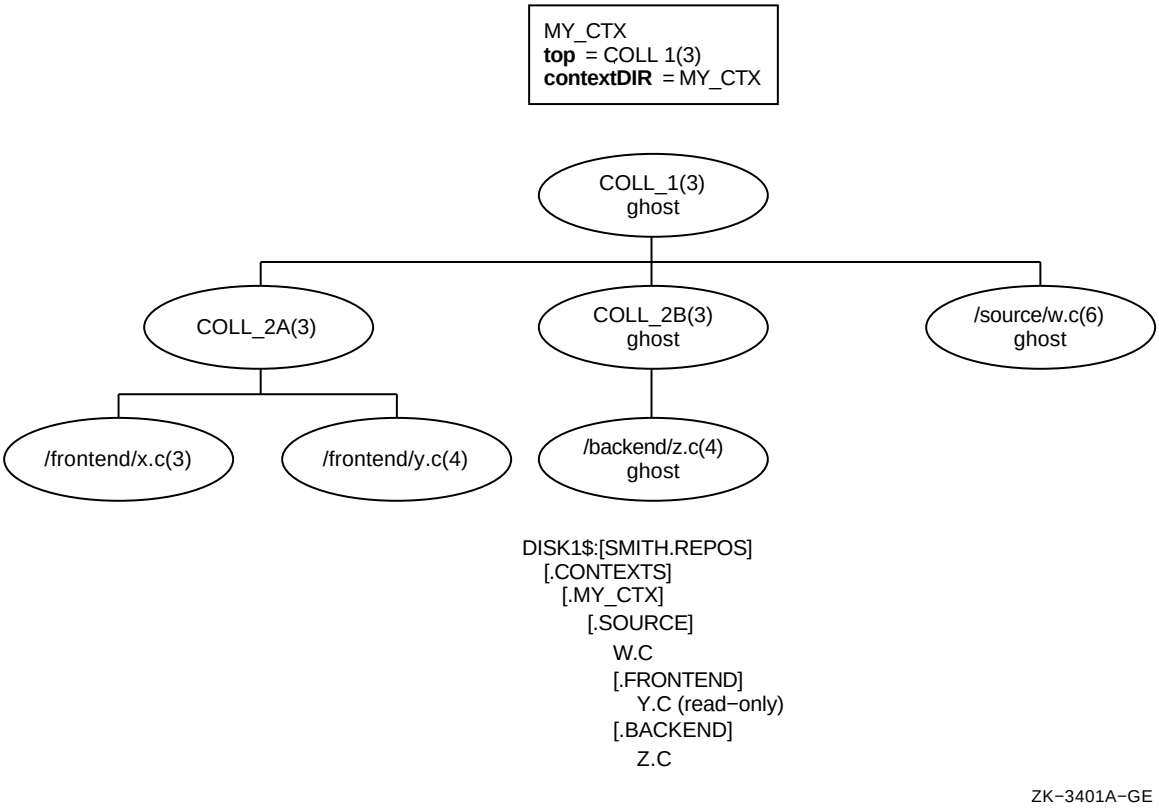


The **filePath** property for **y.c(4)** has the following values:

DISK1$:[SMITH.REPOS.CONTEXTS.MY_CTX.FRONTEND]Y.C

Reserve **w.c(5)** and **z.c(3)**. To do so, you also must reserve **COLL_1(2)** and **COLL_2B(2)**. **COLL_1(3)** becomes the new value of the context's **top** property. The files are placed in the appropriate directories. Figure 9–17 shows the resulting state.

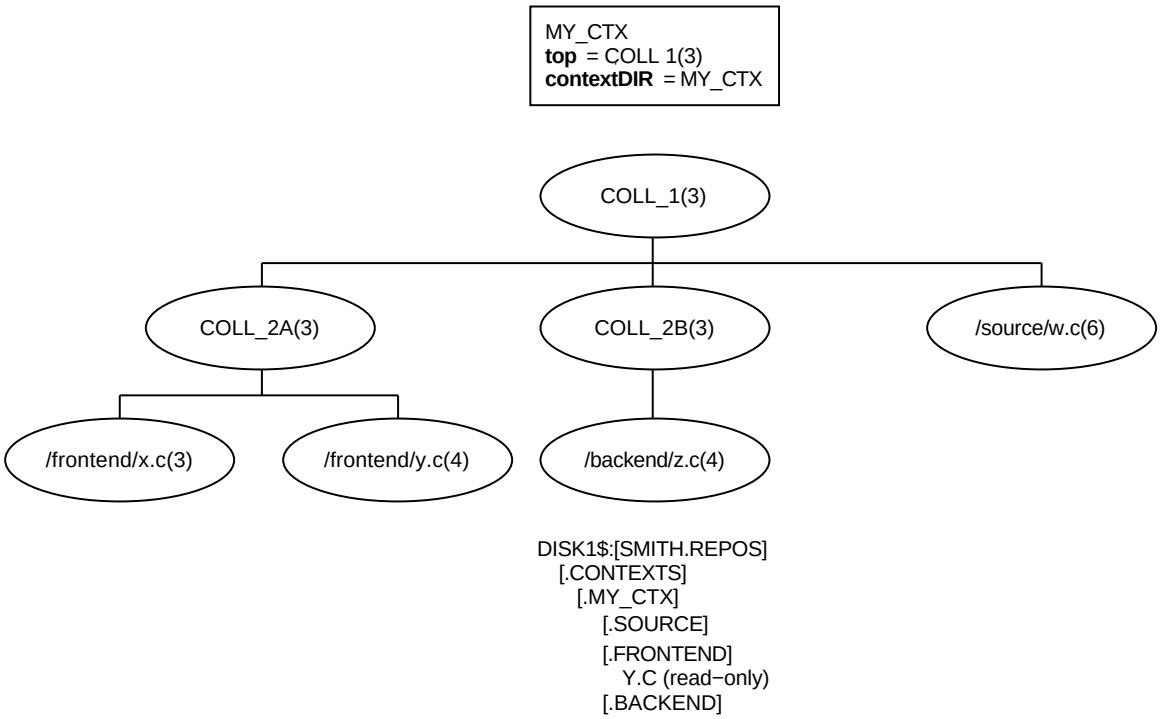
Figure 9–17 File System Example: Reserving BINARY Elements



The two files created by the reservations have the initial contents of their predecessor versions but, unlike Y.C, you can modify them.

After you modify the two reserved files and test the changes, replace them and the collections that contain them. Figure 9–18 shows the resulting state.

Figure 9–18 File System Example: Replacing BINARY Elements



Replacing the two versions deletes the files from the collection directories. They are now available for any user to open or to reserve.

At this point, you still have **y.c(4)** open for reading. Send the CLOSE message to this version to remove the file from the directory. You also can change the value of the **top** property to some other collection hierarchy, which automatically performs the following functions:

- closes open BINARY elements in the collection hierarchy
- deletes the corresponding files from the directory structure
- deletes the directory structure

10
Modeling Dependencies

In most system models, one part of the system that depends on another may become obsolete and need to be rebuilt if the other part changes. Two familiar examples of this are the following:

- An object file depends on the source files from which it is compiled. If the source files are modified, the object file becomes obsolete and must be rebuilt. Similarly, an executable file depends on the object files from which it is linked. The relationship is transitive: the executable files depend on the source files through the intermediate object files. System building software uses information about these dependencies to perform minimal rebuilds of systems.
- A software system that uses record and field definitions from a data dictionary depends on those definitions. If the definitions change, the software system must be rebuilt to cooperate with other systems that use the new definitions.

Oracle CDD/Repository provides facilities for modeling dependencies at three levels of abstraction:

- Dependency relationships, described in Section 10.1, are a general-purpose means of modeling dependencies. In a dependency relationship, the owner of the relationship is notified if the member of the relationship is modified. You can read the resulting notices on the relationship owner.
- METHOD_INVOCATION elements, described in Section 10.2, are designed to record the fact that a certain set of derived objects was built from a certain set of source objects by invoking a certain method. The resulting dependency between the source and derived objects is called a **build dependency**.

  Translation tools such as compilers and linkers can create METHOD_INVOCATION elements to record their execution. METHOD_INVOCATION elements allow system-building tools to determine when a derived object needs to be rebuilt. They also retain important

information about the translation process, such as the version of the tool that performed the translation.

- The configuration management system (described in Chapter 9) also models dependencies. A collection that represents a system configuration depends on the members of the collection. By forcing all collections between a reserved element and the top of the collection hierarchy to also be reserved, Oracle CDD/Repository ensures that the dependencies represented by the collection hierarchy are up-to-date.

10.1 Dependency Relationships

A dependency relationship is a primitive and general-purpose means of modeling a dependency of one element on another.

- It is *primitive* in the sense that it is simple. You must do some work to interpret the information provided by such a relationship.
- It is *general-purpose* because of its simplicity. More complex mechanisms, designed for more specific purposes, can be built on dependency relationships. An example is the build dependency model described in Section 10.2.

In the element type hierarchy, a dependency relationship is an instance of any relation type that is a subtype of DEPENDS_ON. Dependency relationships have the common characteristic that a modification to the relationship member may result in the delivery of a notice to the relationship owner. The notice contains the following information:

- The element ID of the element that was modified. An element can own more than one dependency relationship. You must be able to tell which of the depended-on elements was modified.
- If the notice resulted from the element being modified or deleted.

Notices are made available to applications in the form of value structures that contain data of type MCS_NOTICE. This is an opaque data structure. (See Chapter 8 for information about relationships.) (See the *Oracle CDD/Repository CDO Reference Manual* and the *Oracle CDD/Repository Callable Interface Manual* for more information on notices.)

10.2 Modeling Build Dependencies

Oracle CDD/Repository includes a model for representing the dependencies among **source objects** (such as source files and record definitions), **derived objects** (such as executable, object, listing, and diagnostic files), and the **processor objects** (such as compilers and linkers) that turn source objects into derived objects. Software tools can use this information for the following purposes:

- to answer "what-if" questions, such as "If I change this source module, what objects need to be rebuilt?" and "What modules use this record definition?"
- to do system building and minimal rebuilding

Oracle CDD/Repository provides the METHOD_INVOCATION element type to represent build information. (See the *Oracle CDD/Repository Information Model Volume I* manual for a description of the METHOD_INVOCATION element type and its subtype ATIS_METHOD_INVOC.)

A METHOD_INVOCATION instance models a single execution of a translation process such as a compile or link. The tools that implement these processes create METHOD_INVOCATION instances to record their inputs and their outputs.

You can find METHOD_INVOCATION elements in a repository in several ways:

- The value of the **allDerivedFrom** property on a VERSION element (or a subtype) identifies all the versions that contributed to a derived object, either directly or indirectly. For example, the value of the **allDerivedFrom** property on an element that represents an executable file includes the following:
  - Object files from which the executable file was linked
  - Source files (and other inputs, such as record definitions) from which the object files were compiled
  - Other inputs to the processes that resulted in the executable file
- The value of the **allDerives** property on a VERSION element identifies all the elements derived from a source element, either directly or indirectly. This value allows you to determine what needs to be rebuilt if a source object is modified.
- The METHOD_INVOCATION elements themselves contain information about the process of converting source elements into derived elements. This information includes the message that started the process, the target of the message and arguments that were sent with it, the method that was invoked to implement the message, and metrics of the method's execution.

Use this information to reconstruct the process that translated a set of source elements to a set of derived elements.

This information is present only if the translation tools have properly created METHOD_INVOCATION elements. The following sections describe the properties that tools must set when creating METHOD_INVOCATION elements. Tools such as system builders or verifiers can read these properties for information about how a particular object was constructed.

10.2.1 Association of Source and Derived Versions

A METHOD_INVOCATION element includes two properties that associate it with the source and derived versions for the translation process:

- The **derivedFrom** property is a scan of the source versions that were used in the translation.
- The **derives** property is a scan of the derived versions that resulted from the translation.

A translation tool that creates a METHOD_INVOCATION element must set these properties correctly to assure that the source and derived versions are associated correctly. Setting these properties on a METHOD_INVOCATION instance results in the correct values for the **allDerivedFrom** and **allDerives** properties mentioned earlier.

10.2.2 Representation of Process Information

Properties on the METHOD_INVOCATION element carry information about the process it represents, as follows:

- Information about the message that was sent to begin the translation process. The properties that contain this information include the following:
  - The **msgSent** property identifies the MESSAGE element.
  - The **argsSent** property is a list of the arguments that were sent with the message.
  - The **msgTarget** property identifies the target of the message.
- Information about the method resulting from sending the message. The **methodUsed** property identifies the METHOD element that was invoked.
- Information about the command line that started the translation tool. The **invocationString** property is the actual command line. The **optionsString** property is a string that shows all options that were in effect for the processor. It shows the exact values for all options, even those not explicitly specified on the command line.

- Processor statistics, included in the following individual properties with numeric values:
  - **CPUTime**
  - **elapsedTime**
  - **scalingFactor**

    Use the **scalingFactor** property to characterize the power of the CPU that executed the process. It determines the relative time that a process requires. Relative time is a function of CPU time and CPU power.
- The execution log of the process, identified by the **logFile** property.
- The operating system version on which the process was carried out, contained in the **OSVersion** property.

It is the responsibility of the translation tool to set these properties correctly when it creates the METHOD_INVOCATION element. None of this information is required; however, including it makes it easier for tools to re-create the process that created the derived versions. For example, if a tool sets the **invocationString** property, a system builder can simply use the same command line to carry out the same translation. If the property is omitted, the system builder must derive it from other information.

Index

A

B

C

D

E

F

G

H

N

O

P

R

S

T

U

V