CA Repository Universal XML Exchange

Product Guide

r7.2

CA Product References

This document references the following CA products:

- CA Repository for z/OS

Contact CA

Contact Technical Support

For your convenience, CA provides one site where you can access the information you need for your Home Office, Small Business, and Enterprise CA products. At http://ca.com/support, you can access the following:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Provide Feedback

If you have comments or questions about CA product documentation, you can send a message to techpubs@ca.com.

If you would like to provide feedback about CA product documentation, complete our short customer survey, which is also available on the CA Support website, found at http://ca.com/docs.

Contents

Chapter 1: Introduction

This chapter introduces CA Repository Universal XML Exchange. The CA Repository Universal XML Exchange lets you extract data from XML files and load it into the repository work tables. The data in the work tables is used as an input to the Repository to store and maintain your metadata.

Note: The metadata in the XML file from CA Repository Universal XML Exchange is written to the database worktables using ODBC.

All XML file do not store XML in standard formats. The CA Repository Universal XML Exchange reads a control file that maps objects in the XML document to objects in the repository. The CA Repository Universal XML Exchange can handle CWM, XML, and other XML documents that contain metadata.

The CA Repository Universal XML Exchange consists of the following four components:

- Control File Builder (see page 8) (run once for each metadata element)
- Transformation Utility (see page 8) (run every time you perform a scan)
- Parser (see page 9) (run every time you perform a scan)
- Loader (see page 9) (run every time you perform a scan)

This section contains the following topics:

CA Repository Universal XML Exchange Control File Builder

A control file is an XML document that contains the rules about how to map an object in the XML document to an object in the repository.

The CA Repository Universal XML Exchange Control File Builder builds the control file that maps XML objects to repository entities, XML elements to repository relationships, and performs simple conversions of the data.

You can also build the control file using any XML editor, Notepad, or any other file editor. Irrespective of the method used to build or edit the control file, the XML control file must validate the control file Document Type Definition (DTD) before it can be used.

Note: You can use the CA Repository Universal XML Exchange Control File Builder to validate a control file.

More Information:

Create a Control File (see page 27)
CA Repository Universal XML Exchange Control File Builder Dialog (see page 17)
Validate a Control File (see page 28)

CA Repository Universal XML ExchangeTransformation Utility

The CA Repository Universal XML Exchange Transformation Utility transforms an XML document from one form to another using Extensible Stylesheet Language (XSL). The utility accepts XML documents and XSL documents as input. The output is a transformed XML document.

For example, the CA Repository Universal XML Exchange requires an ID attribute in the XML file to build relationships. If you encounter an XML document that does not contain an ID attribute, you can use the provided XSL document to create an ID attribute for every XML element that does not already contain one.

Note: At the time of installing CA Repository Universal XML Exchange, sample XSL files are copied to this location - Program Files\CA\ReposzOS\Universal XML Exchange\cdata. However, you can also build your own XSL documents for different scenarios.

More Information:

Using the CA Repository Universal XML Exchange Transformation Utility (see page 31)
Transform a File Using the GUI (see page 32)
How to Transform Files in Batch Mode (see page 33)

CA Repository Universal XML Exchange Parser

The CA Repository Universal XML Exchange Parser parses the XML document based on information found in the control file and loads the information to the repository work tables. All rows loaded into the work tables are inserted with a work unit. The work unit is specified when the user runs the CA Repository Universal XML Exchange Parser. The work unit groups the data so that you can run the CA Repository Universal XML Exchange Parser multiple times before a load is needed.

More Information:

Using the CA Repository Universal XML Exchange Parser (see page 37)
How the CA Repository Universal XML Exchange Parser Works (see page 37)

CA Repository Universal XML Exchange Loader

The CA Repository Universal XML Exchange Loader (load program) reads the data from the work tables and loads it to the repository. The configurable reuse rules (PCR files) instruct the loader when to reuse, update, or insert a row.

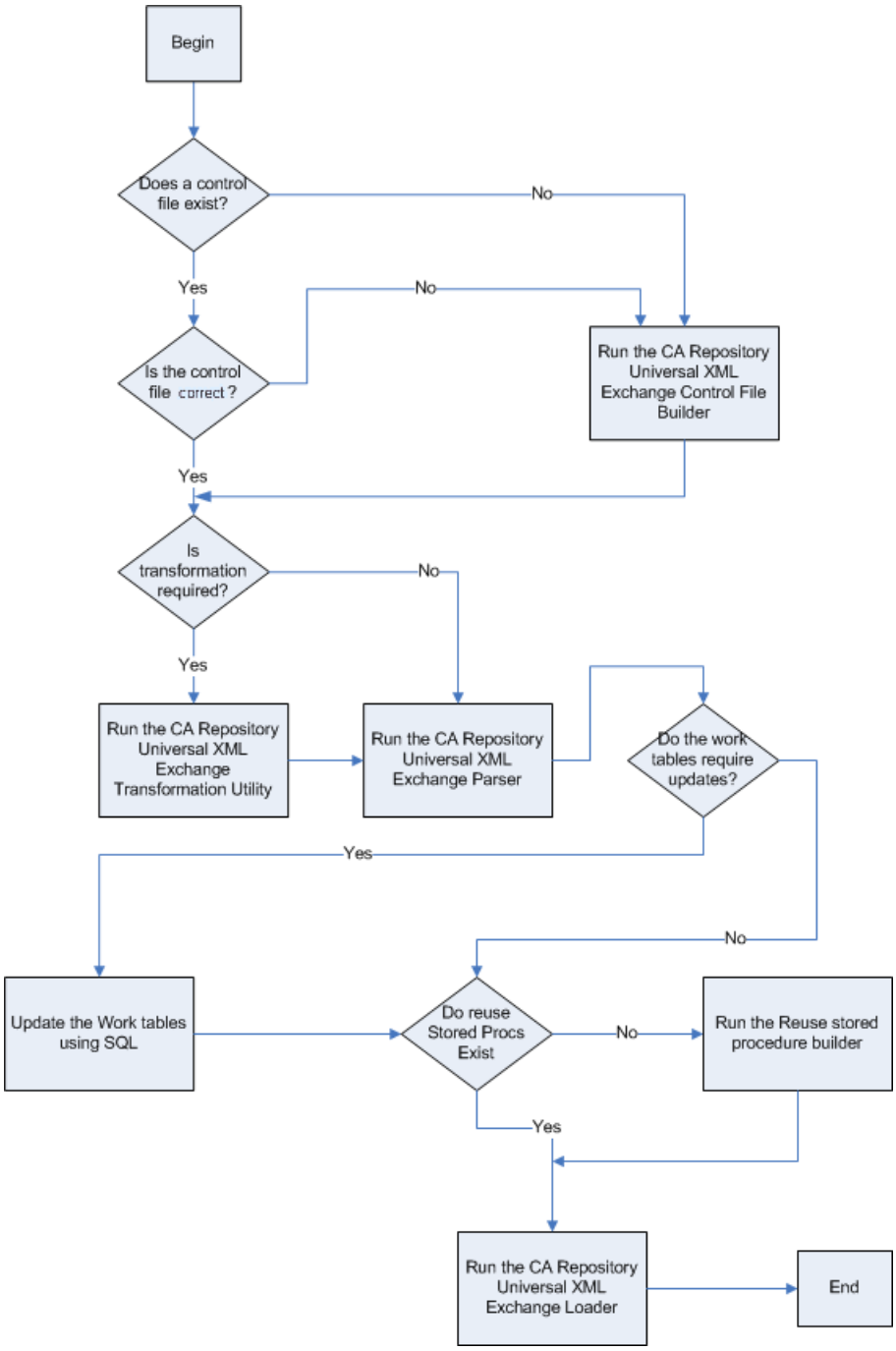
More Information:

Using the CA Repository Universal XML Exchange Loader (see page 45)

How the CA Repository Universal XML Exchange Processes Data

The following illustration shows how the CA Repository Universal XML Exchange components process your data starting at the control file level and ending with the uploading of data to the CA Repository.

Note: You must run the CA Repository Universal XML Exchange Control File Builder only once unless you change the XML Rule file. The CA Repository Universal XML Exchange Control File Builder must be used either if a control file does not exist or if a control file is incorrect and needs to be updated. You must run the CA Repository Universal XML Exchange Transformation Utility only once unless you change the input XML file. The CA Repository Universal XML Exchange Transformation Utility must be used only if the input XML file needs to be transformed.


Begin
Does a control file exist?
No
Yes
Is the control file correct?
No
Run the CA Repository Universal XML Exchange Control File Builder
Yes
Is transformation required?
No
Yes
Run the CA Repository Universal XML Exchange Transformation Utility
Run the CA Repository Universal XML Exchange Parser
Do the work tables require updates?
Yes
No
Update the Work tables using SQL
Do reuse Stored Procs Exist
No
Run the Reuse stored procedure builder
Yes
Run the CA Repository Universal XML Exchange Loader
End

Chapter 2: Installation

This chapter describes how to install the Windows component of CA Repository Universal XML Exchange. For information on installing the mainframe component, see the Installation Guide.

Note: You must use CA Repository Universal XML Exchange with CA Repository for z/OS r7.2 or later.

This section contains the following topics:

Installing the Windows Component (see page 13)

Installing the Windows Component

The CA Repository Universal XML Exchange runs on the following Windows based operating systems:

- XP
- 2003
- Vista

To install and setup the CA Repository Universal XML Exchange, follow these steps:

1. From the CA Repository Universal XML Exchange directory on the CD, launch the file named setup.exe

   The installation begins and performs the following tasks:

   - Extracts the relevant files from the CD to your computer
   - Prompts you to install JRE 1.6.0 in Program Files\CA\SharedComponents\JRE, if you do no have JRE 1.6.0 or a higher version
   - Installs the programs named XMLEXCHANGE.exe, rzos_ControlFileBuilder.jar, rzos_TransformationUtility.jar and their dependent files in Program Files\CA\ReposzOS\Universal XML Exchange
   - Installs the Control files and XSL files in Program Files\CA\ReposzOS\Universal XML Exchange\cdata

2. After the installation process has been successfully completed, you can access the Control File Builder, Scanner, or Transformation Utility from Start>Programs>CA>Repository for zOS>Universal XML Exchange.

Chapter 3: Using the CA Repository Universal XML Exchange Control File Builder

The CA Repository Universal XML Exchange Control File Builder is a component of the CA Repository Universal XML Exchange. It is a graphical user interface for creating or editing a control file that maps the metadata in an XML document to objects in the CA Repository.

You can perform the following tasks using the CA Repository Universal XML Exchange Control File Builder:

- Create a new control file or update an existing control file by editing its contents
- Save an updated control file at a specified location
- View object, relationship, and mapping objects using tree controls
- Validate newly created or updated control file data using DTD
- View lists of existing objects, relationships, and mappings that appear in the window to assist a user while adding, editing, or deleting XML elements from the control file
- Search existing objects and relationships while adding, editing, or deleting them
- Save Object Map nodes in a temporary list before you add them to the control file
- Save Relationship nodes in a temporary list before you add them to the control file
- Save Attr_Map nodes in a temporary list for an XML object and relationship before they are added to Object Map and Relationship respectively

This section contains the following topics:

Input Sources for the CA Repository Universal XML Exchange Control File Builder

The CA Repository Universal XML Exchange Control File Builder uses the following input sources to create a control file that maps XML objects to repository objects:

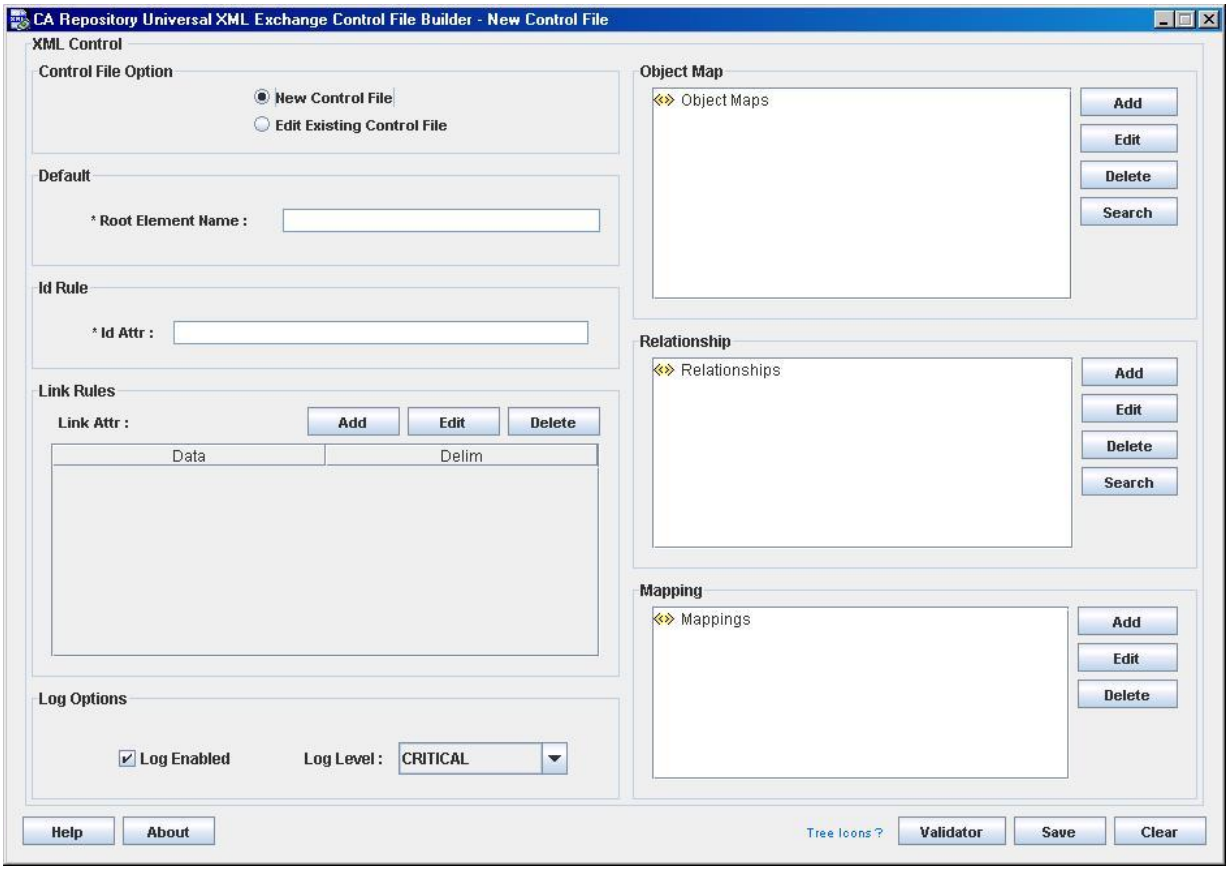
- An existing control file
- A new control file

The CA Repository Universal XML Exchange uses XPATH to navigate through the XML document. You must use the XPATH names in the control file to resolve ambiguous element or attribute names. XPATH names in the CA Repository Universal XML Exchange Control File Builder help improve performance of the CA Repository Universal XML Exchange Parser.

If you build a control file outside of the CA Repository Universal XML Exchange Control File Builder you must ensure that the control file is valid. To check the validity, you must validate the XML file against the DTD using a validating parser or browser. You can use the validation facility within the CA Repository Universal XML Exchange Control File Builder to validate your control file. You can use the CA Repository Universal XML Exchange Parser to populate the work tables that the CA Repository Universal XML Exchange Loader loads to the CA Repository.

CA Repository Universal XML Exchange Control File Builder Dialog

The CA Repository Universal XML Exchange Control File Builder dialog lets you perform tasks like creating a new control file or modifying an existing control file.



The following list explains each area of the CA Repository Universal XML Exchange Control File Builder dialog:

Control File Option

Creates a new control file or lets you edit an existing one.

Default: New Control File

Default

Defines the root element name. The root element name is the name of the root element in the input XML file.

Note: A control file can have only one XML root element per input XML file.

ID Rule

Defines the name of the XML attribute that contains the unique value of an XML object in the XML file.

IdAttr

IdAttr identifies an XML attribute in the input file that is used to uniquely identify an XML element. In XML terminology, these attributes are known as ID attribute types. The value of the ID attribute must not appear more than once in an XML document. The CA Repository Universal XML Exchange requires that the elements within an XML document contain an ID attribute to build relationships between XML elements.

When you create a new control file, the ID attribute must contain a single XML element describing this unique ID. However, when you modify an existing control file, the ID attribute shows the text of IdAttr of the existing control file.

Note: The ID can be different for each XML object in the input XML file.

The name of the ID attribute must meet the XML standards for an XML name.

Link Rules

Adds or edits link attributes.

Link Attr

A link attribute describes an attribute that is used in the XML document to reference another XML element in the document. In XML terminology these attributes are usually known as IDREF attribute types. The value of the IDREF attribute must match the value of an ID attribute on an element elsewhere in the XML document. Attributes defined as IDREFS contain a list of ID values separated by whitespace. Link attributes must be established for IDREFS type attributes. The control file allows a delimiter other than spaces.

Note: You can add multiple link attributes.

Log Options

Indicates whether the log is enabled and the level of the log.

Values: Critical, Error, Warning, Info

Default: Critical

Object Map

Adds a new object map, modifies or deletes an existing object map, and searches for the objects, repository tables, or attributes you want to view or edit.

If you are creating a new control file, this area is empty. After you Map XML Objects to the repository entities, data appears in this area.

The Object Map area contains the following information about entities:

- XML object
- Repository object
- Properties
- Rules for stored objects
- Corresponding text

Note: If you loaded an existing control file, you can expand the selection list display by clicking the nodes.

Relationship

Creates, changes, or deletes definitions that specify how objects in the repository are associated.

From this area you can add a new relationship, modify or delete a relationship, and search for the relationship.

You can perform the following tasks with the tree structure of relationships:

- View previously defined relationships in a control file
- Expand Relationship nodes to add, modify, or delete relationships

Note: You can use the Search button to locate objects, tables, and attributes.

Mapping

Adds mapping for an object_map or a relationship to the control file. This mapping enables the conversion of input XML file data into values for entities of relationships or object_maps in preparation for uploading the file to the Repository.

The Mapping area contains a selection list of existing maps that contain the conversion values from the input XML file to the repository.

Using this area, you can select an element from which you create and add a new map, change the placement of an input XML control file's object in the repository, remove the map object from the control file.

The following controls are provided in the CA Repository Universal XML Exchange Control File Builder dialog:

Help

Displays the CA Repository Universal XML Exchange Help - CA window.

About

Opens the About CA Repository Universal XML Exchange Control File Builder – CA dialog displaying the release number, copyright and license agreement information.

Tree Icons

Displays a list of various tree icons and their description.

Validator

Opens the Validator dialog for control file.

Save

Saves your new or modified control file.

Clear

Clears the data entered in the dialog.

Tree Icons

Several of the CA Repository Universal XML Exchange Control File Builder window areas use a tree structure to display existing definitions. Each icon in the tree has a specific meaning.

The following list explains each icon:

A tag that has at least one subtag. Optionally, it can contain character data and attributes.

Attribute

Empty tag

A tag containing only character data (untagged text)

A tag that has character data and at least one attribute

A tag containing more than one attribute and no content

A tag containing exactly one attribute and no content

The character data (untagged text) contained in a tag

Note: You can view the description for each icon by clicking the Tree Icons link at the bottom of the CA Repository Universal XML Exchange Control File Builder dialog.

Control File Tags

The following section lists the control file tags and their usage:

CARepository_Control

Defines the Control File document. All other tags must be enclosed between <CARepository_Control> and </CARepository_Control>;

Object_Map

Defines the mapping of XML objects to their repository tables. Each object in the XML document can map to multiple tables in the repository so the Repository_Table tag can appear multiple times for the same object.

This groups the processing of an XML object, which is defined as any item that has an attribute defined as ID**="..."** providing a unique identifier to the XML document.

Object

Describes the ID and location of XML objects.

Attribute:

key—The *key* attribute in an XML object defines the ID for that particular object. If the key attribute is not present in the XML object, the ID (IdAttr) specified at the top of the XML file is taken as the default.

Repository_Table

Groups the information to be recorded in one of the repository tables. Each object can be recorded in one or more tables in the repository.

Table

Provides the name of the repository table that is to receive the data.

Attribute:

output—Determines whether this entry is always created, or is dependent on the attribute data found inside the XML object. The *output* attribute has the following values:

- Mandatory—Always records in this table.
- Optional—Records if applicable. A <Rule> indicates when this applies.
- Choice—Records in any one of the tables used. Rule tags determine which one applies. Only one table (without rules) can be the default to use if no rules match.

Attr_Map

Groups the recording and rule checking for a single tag in the XML stream.

Attr

Provides the tag name to search for in the XML stream. Do not include the < and > characters.

Note: Do not include the < and > characters.

Attributes:

- check—Determines whether this XML element or a parent XML element contains the tagname in the XML document. The *check* attribute has the following values:
  - Parent—Searches the tagname in a parent XML element.
  - None—Searches the tagname in the XML element currently being processed.
- default—An attribute default = "…." can exist inside Attr. If this attribute does not exist in the input XML file, this default value is uploaded to the work tables. Otherwise, the attribute value from the input XML file is uploaded to the work tables.

Column

Defines the name of the column in the repository that receives the data.

Note: Text such as descriptions, comments, and so forth are stored separately and must be specified using <TextCol>.

TextCol

Specifies that the attribute contents are stored as Text. The required attribute *type* determines which repository text table receives the data.

Rule

Determines whether the rule for the current table (optional or choice) applies.

Attributes:

- action—Includes or excludes attribute from processing. The action attribute has the following values:
  - Include—Includes the attribute if the rule is satisfied.
  - Exclude—Excludes the attribute if the rule is satisfied.
- type—The *type* is mandatory. The *type* attribute has the following values:
  - Exists—Records in the current table if this tag is found.
  - Equals—Records whether the tag in the XML file has the same contents as in the control file.
  - LT—Records whether the contents are less than this value. Operators LE, GE, GT, and NE are also supported.

Relationship

Groups the tags defining when an association or relationship is recorded in the repository. The Relationship tag encloses the definitions of the containment and split-entity associations and relationships.

RelName

Defines the name of the repository association or relationship. For relationships, you can specify that the relationship be recorded only if a corresponding Table, typically having output=optional, is being recorded.

Attributes:

- output—Determines whether this entry is always created, or whether this is dependent on attribute data of the source or target. The output attribute has the following values:
  - mandatory—Always records in this.
  - optional—Records if applicable. A <Rule> indicates when this applies.
- type—Determines whether relationship is dependent or not on another object existing in the XML document for the repository relationship. The type attribute has the following values:
  - standard—Makes this a nonconditional relationship
  - conditional—Makes this a conditional relationship.

  Default: Standard
- test—Determines which entity controls the recording of this relationship. The test attribute has the following values:
  - Source
  - Target

Source

Defines the name of the repository entity-type (Table) that is the source end of the relationship.

Attribute:

key—The attribute key="..." is written in the source only if it is different from ID (IdAttr). Otherwise, the ID (IdAttr) is taken as the key.

Target

Defines the name of the repository target end of the relationship.

Attribute:

key—The attribute key="..." is written in the target only if it is different from ID (IdAttr). Otherwise, the ID (IdAttr) is taken as the key.

Type

Determines the type of relationship. The *Type* tag contains the following values:

- CONTAINS—Used when the target entity is found inside the tags for the source entity in the XML file (or conversely).
- SIBLINGS—Used when an XML object is recorded in more than one Repository table. This relationship is created when the source and the target entity types derive from the same origin.

REFERENCE—This relationship is recorded if the XML attribute or XML element points to the ID attribute elsewhere in the document and a relationship is to be created between these XML objects. Enter the XML attribute or XML element that references the ID attribute if the type is Reference. Select SrcAsRefAttr check box if source is the reference attribute in the Reference relationship. If target is the reference attribute in this relationship then do not select SrcAsRefAttr.

- ATPAR—This relationship is used when a relationship needs to be created between two elements that are at the same level in an XML file. This relationship has the following additional tags:
  - SourceRef—Describes the path for source ID
  - TargetRef—Describes the path for target ID
  - RelidRef—Describes the path for relationship ID

  Note: For the relationship types - CONTAINS, SIBLINGS and REFERENCE - the KEY_GUID generated in the work tables, is either from source_id or target_id. However, for the ATPAR relationship, the KEY_GUID is generated from the RelidRef attribute.

Attribute:

SrcAsRefAttr= true /false—Determines whether the source or the target is the reference attribute. Must be used with Type – tagname when source and target are same objects

Mapping

Contains the specifications for translating the content of an XML tag to the content of a repository column.

Example: true->Y; false->N.

Allows replacement of string pattern using tags FromPattern and ToPattern

For example: replace dashes on underscores

<FromPattern>-</FromPattern>

<ToPattern>_</ToPattern>

FromSource

Groups a combination of Object and Attr tags to declare the source tag in the XML file.

ToRepository

Correspondingly groups the Table and the Column to receive the translated content.

Convert

Groups the before and after content of the mapping.

FromValue

Defines the value to find in the original XML tag.

ToValue

Defines the value to record in the repository column.

FromPattern

Defines the value of the replaced string in the original XML tag.

ToPattern

Defines the value of the replacing string in the repository column.

Create a Control File

Control files let you map objects in XML document to objects in the repository. You can create a control file in multiple ways. One of them is using the CA Repository Universal XML Exchange Control File Builder. You can also manually create a control file using any XML editor.

The processes, task, and examples discussed here assume that you are using the CA Repository Universal XML Exchange Control File Builder to create a control file.

To create a control file

1. Click Start, Programs, CA, Repository for zOS, Universal XML Exchange, and Control File Builder.

   The CA Repository Universal XML Exchange Control File Builder-CA dialog opens.

2. Complete the following fields in the Control File Option area:

   New Control File

   Creates a new control file.

   Edit Existing Control File

   Edits an existing control file.

3. Enter the following information in the Default window area of the CA Repository Universal XML Exchange Control File Builder dialog:

   Root Element Name

   Defines the root element name of the input XML file.

   Note: For more information, see Specify the Root Element Name and Id Attr in the Control File Builder help.

4. Enter the name of the XML attribute that contains the unique value of an XML object in the XML file in the Id Attr field.

   Note: For more information, see Specify the Root Element Name and Id Attr in the CA Repository Universal XML Exchange Control File Builder help.

5. Click Add in the Link Rules area to add a new link attribute.

   Note: For more information about adding a new link, see Add a New Link Attribute in the CA Repository Universal XML Exchange Control File Builder help.

6. Map the input XML file objects to repository entities by using the Object Map, Relationship, and Mapping area.

   Note: For more information about mapping input XML file objects to repository entities, see Map XML Objects to Repository Entities in the CA Repository Universal XML Exchange Control File Builder help.

7. Complete the following fields in the Log Option area:

   Enable Logging

   Enables CA Repository Universal XML Exchange Control File Builder to log your activities.

   Default: Yes

   Log Level

   Specifies the level at which you want the activities logged.

   Default: Yes

   Values:

   - CRITICAL—Only critical messages are written to the log.
   - ERROR—All errors including critical errors are written to the log.
   - WARNING—Both error and warning messages are written to the log.
   - INFO—All messages are written to the log.

8. Click Save.

   Your new control file is created.

Validate a Control File

You must validate the XML control file to the control file DTD before it can be used. You can use the CA Repository Universal XML Exchange Control File Builder dialog to validate your control file with the DTD.

To validate a control file

1. Click Start, Programs, CA, Repository for zOS, Universal XML Exchange, and Control File Builder.

   The CA Repository Universal XML Exchange Control File Builder-CA dialog opens.

2. Enter the Control File name to load, and click Validate.

   The CA Repository Universal XML Exchange Control File Validator dialog opens and provides the details of parsing and shows whether the XML file is valid.

How the CA Repository Universal XML Exchange Control File Builder Edits an Existing Control File

The following procedure explains how the CA Repository Universal XML Exchange Control File Builder edits an existing control file and uses different objects to store different XML elements.

1. An existing control file is parsed using a Xerces implementation. The Xerces module loads all the information of the existing control file into the primary memory for modifying (adding, editing or deleting) the elements.
2. After specifying an input control file, the CA Repository Universal XML Exchange Control File Builder parses the XML file and retrieves data using tag names that are stored in the objects. It validates the control file with the DTD. If there are any errors in the control file, the CA Repository Universal XML Exchange Control File Builder displays the errors in a dialog.
3. The CA Repository Universal XML Exchange Control File Builder uses different objects to store different XML elements:
   - IdRule value is stored in IdRuleObject.
   - LinkRules are stored in LinkRuleObject, which in turn holds a list of LinkRule.
   - ObjectMaps are stored in ObjectMapsObject, which holds the list of Object Maps.
   - Relationships are stored in RelationShipsObject, which holds the list of relationships.
   - Mappings are stored in MappingsObject, which holds the list of Mappings.

     Note: The CA Repository Universal XML Exchange Control File Builder sets all the objects and their respective values specified in the input XML control file. If no values are specified in the input XML control file, objects are loaded with their default values.

Chapter 4: Using the CA Repository Universal XML Exchange Transformation Utility

The CA Repository Universal XML Exchange Transformation Utility is a component of the CA Repository Universal XML Exchange. It is provided as a tool to assist users in generating and formatting source XML files for input into the scanner.

The CA Repository Universal XML Exchange Transformation Utility lets you perform the following tasks:

- Transform your input XML document using an XSL (see page 86) stylesheet.
- Create a transformed or updated XML file that you can use as input to the scanner.

The CA Repository Universal XML Exchange Transformation Utility transforms your input XML file based on the syntax in the XSL stylesheet and creates an output XML file.

You can use the CA Repository Universal XML Exchange Transformation Utility when an input XML file does not contain everything that a scanner needs. For example, the scanner needs an ID attribute. If the input XML document does not have an XML attribute defined as ID, an XSL stylesheet can be created to add an ID attribute. The CA Repository Universal XML Exchange Transformation Utility executes the transformation and creates an output XML file that is recognized by the scanner (see page 86).

This section contains the following topics:

XML File Transformation

Transformation is a process that takes your input XML document and creates an XML output file that you can use as an input to the scanner. You can transform a file in two ways:

- Using the Transformation GUI Utility. For more information, see Transform a file using GUI (see page 32).
- Using Batch mode. For more information, see How to Transform files in Batch mode (see page 33).

Transform a File Using the GUI

You can generate and format source XML files for input into the scanner using the CA Repository Universal XML Exchange Transformation utility. When you use the CA Repository Universal XML Exchange Transformation Utility, a dialog prompts you to enter the information specified in the following procedure.

Note: All procedures and examples assume that you are familiar with XML.

To transform a file using the CA Repository Universal XML Exchange Transformation Utility GUI

1. Specify the name and location of the following files using the Browse button located next to each field:
   - Source input XML file
   - XSL file to use to transform the input file
2. Enter the name and browse for the location where you want the transformed file placed, and click Transform.

   Messages appear when the file is being validated and when the transformation is complete.

   Note: If you get an error message stating that the input file is too large, you must start the CA Repository Universal XML Exchange Transformation Utility from a command prompt window and specify additional memory.

Increase Available Memory to Transform Large Files

If you receive an error message stating that the input XML file is too large to process, you must increase the amount of heap space used by the Java Virtual machine (JVM). The amount of heap space needed depends on the size of the XML file. A 50 MB XML file may need an initial allocation of 32 MB and a maximum allocation of 512 MB.

To increase the amount of heap space (see page 85) for large files

1. Open a command line prompt window and change directories to the directory that contains the CARepositoryUniversalXMLTransformer.jar file.
2. Run the following Java command from the command prompt window to increase the heap allocation of the JVM:

   ```
   java -Xms32m -Xmx512m -jar CARepositoryUniversalXMLTransformer.jar
   ```

   The CA Repository Universal XML Exchange Transformation Utility GUI starts so that you can transform your file.

   Xms32m

   Indicates that 32 MB is the initial allocation.

   Xmx512m

   Indicates that 512 MB is the maximum allocation.

   You can adjust these parameters depending on the size of the XML file.

How to Transform Files in Batch Mode

Some users may prefer using batch mode to transform their input XML files. Batch mode is useful in the following situations:

- You have a large input XML file.
- You want to transform multiple files.
- You prefer batch execution.

To transform a file in batch mode, do the following:

1. Open a text editor of your choice.
2. Enter the commands to transform your file.
3. Save the file as a BAT file.
4. Run this BAT file from a command prompt window.

Transform Multiple Input Files

The following is a sample batch file that transforms three input files:

```
java -jar "D:\CARepositoryUniversalXMLTransformer.jar" -IN C:\input1.xml -XSL
C:\idgenerator.xsl -OUT C:\output1.xml
java -jar "D:\CARepositoryUniversalXMLTransformer.jar" -IN C:\input2.xml -XSL
C:\idgenerator.xsl -OUT C:\output2.xml
java -jar "D:\CARepositoryUniversalXMLTransformer.jar" -IN C:\input3.xml -XSL
C:\idgenerator.xsl -OUT C:\output3.xml
```

Transform Large Files

The following is a sample batch file that transforms large files:

```
java -Xms32m -Xmx512m -jar "D:\CARepositoryUniversalXMLTransformer.jar" -IN
C:\input4.xml -XSL C:\idgenerator.xsl -OUT C:\output4.xml
```

Transformation Example

The following example shows a sample transformation that provides the needed attributes so that the transformed file can be used as input to the scanner.

Note: It is assumed that you understand XML.

Sample Input File

The transformation uses an XML file for Visual C++ from Visual Studio 2005. It shows you how to relate the function in the RelativePath attribute of the File XML element to the project in the Name attribute of the VisualStudioProject XML element that you need and an ID type attribute on the following XML elements:

```
<VisualStudioProject ProjectType="Visual C++" Version="8.00" Name="catalog2"
ProjectGUID="{5D09DB25-ABEB-43AC-8650-4E90534B84E9}" RootNamespace="catalog2"
Keyword="MFCProj">
   <Files>
      <Filter Name="Source Files" Filter="cpp;c;cxx;rc;def;r;odl;hpj;bat;for;f90">
         <File RelativePath="AboutDlg.cpp">
```

Sample XSL File

This is the XSL file that generates IDs for all XML elements (complete code):

```
<?xml version='1.0' encoding='utf-8' ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:template match="*">
    <xsl:copy>
    <xsl:if test="not(@id)">
      <xsl:attribute name="id">
        <xsl:value-of select="generate-id()"/>
      </xsl:attribute>
    </xsl:if>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="@*|processing-instruction()|comment()">
    <xsl:copy>
      <xsl:apply-templates select="node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Transformed XML File

This is a portion of the *transformed* XML file:

```
<VisualStudioProject id="N10001" ProjectType="Visual C++" Version="8.00"
Name="catalog2" ProjectGUID="{5D09DB25-ABEB-43AC-8650-4E90534B84E9}"
RootNamespace="catalog2" Keyword="MFCProj">
  <Files id="N100D8">
    <Filter id="N100DA" Name="Source Files"
Filter="cpp;c;cxx;rc;def;r;odl;hpj;bat;for;f90">
       <File id="N100DE" RelativePath="AboutDlg.cpp">
```

View Transformation Output

To view the transformation output, open the transformed XML file in a browser or XML editing tool.

Chapter 5: Using the CA Repository Universal XML Exchange Parser

This chapter describes how the CA Repository Universal XML Exchange Parser works and how you can load data to the repository using the CA Repository Universal XML Exchange Parser.

This section contains the following topics:

How the CA Repository Universal XML Exchange Parser Works

The CA Repository Universal XML Exchange Parser reads an XML file and loads it into the repository work tables based on mapping rules that are specified in the control file.

When you input an XML file, the CA Repository Universal XML Exchange Parser uses the following process to load the data into the repository work tables:

- The CA Repository Universal XML Exchange Parser (extractor) reads an XML file and parses the data according to the instructions provided in the control file.
- You must specify what you want to use to process the metadata in the control file.
- The control file governs the translation.

  Note: For more information about the control file, see the CA Repository Universal XML Exchange Control File Builder help.
- The data is stored in the CA Repository work tables.

Required Files

The CA Repository Universal XML Exchange Parser uses the following XML documents as input:

- An XML file containing data that represents the metadata to be stored in the repository. This file can be from a variety of sources. It can be an XML file that is generated from an application such as Visual Studio 2005 or one that is written to describe an in-house system.
- A control file that maps the input XML file to repository objects.

  This XML control file contains rules about how to map objects in the XML file to objects in the repository.

Perform a Direct Load to Mainframe—PC Tasks

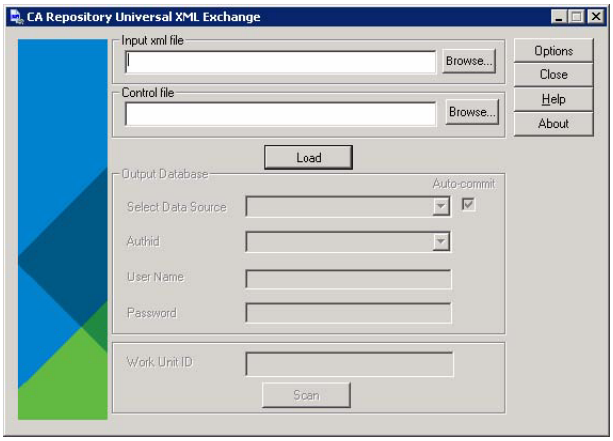
You can use CA Repository Universal XML Exchange to load data to the repository by specifying an input XML file that the CA Repository Universal XML Exchange Parser uses to validate and load to the repository.

You must perform tasks on the system (using either the GUI or the batch processor) and on the mainframe to complete the scan process.

To run the Parser (PC tasks)

1. Click Start, Programs, CA, Repository for zOS, Universal XML Exchange, Scanner.

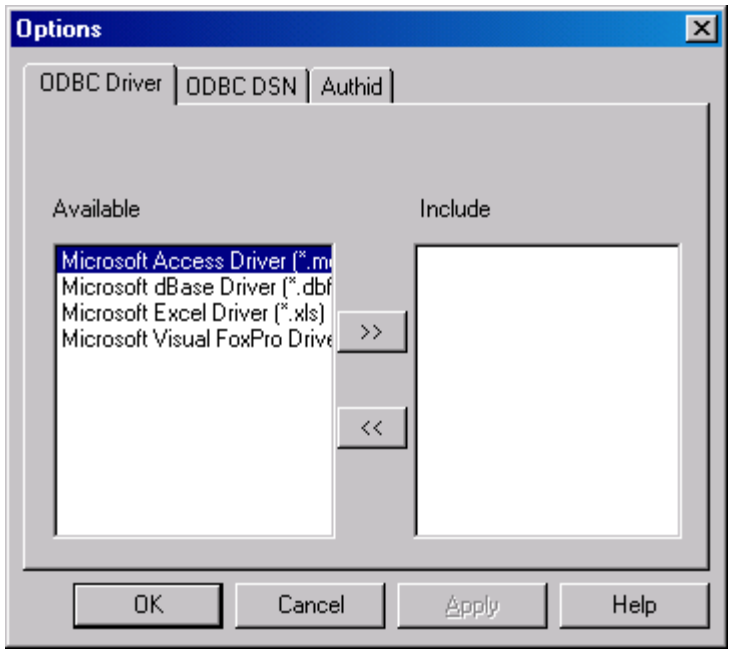
   The CA Repository Universal XML Exchange window opens.



2. Specify the path and name of the XML file you want to process in the Input XML File field.
3. Specify the path and name of the control file to use with your XML document.

4. (Optional) Click Options if this is the first time you are uploading this XML document or if you are making changes.

   The Options dialog opens.



5. Review or select the ODBC driver.

6. Select ODBC DSN in the ODBC DSN tab, and click Load.

   The selected XML control file is read and parsed.

   The beginning of the CA Repository Universal XML Exchange file is read to validate if the root XML element tag matches the root tag specified in the control file.

   The other dialog fields are enabled and the XML file and control file are in the memory.

7. Enter values in the following Output Database fields, and click Scan.

   Select Data Source

   Specifies the name of the data source. Select the DSN to use from the drop-down list (in Step 6, on the DSN tab, you placed these selections on the Include list). This is the ODBC source name of the database (DB2) where the work tables reside.

   Authid

   Specifies the schema or the owner name of the work tables.

User Name and Password

Defines the user name and the password. You must enter the values if required, based on the security requirements of your site.

Work Unit ID

Defines a unique ID for the load. This might be your TSO ID, any other prespecified value, or a value you create to enable you to uniquely identify this unit of work. This value must also be supplied to the mainframe when the model is scanned into the CA Repository.

The file is loaded into the following work tables:

- PRMXML_OI Work Table Objects (see page 75)
- PRMXML_PI Properties Work Table (see page 77)
- PRMXML_AI- Associations Objects Work Table (see page 78)
- PRMXML_TI Text Objects (see page 79)

A progress dialog displays the upload activity as the file is loaded.

Note: The CA Repository Universal XML Exchange Scanner creates a log file with the same name and same path as of the input XML file with an extension .log. The log file will be placed in the folder where the input XML file exists. The log file will have the same name as the input XML file. However, the .xml extension will be removed and replaced with a .log extension. For example, if the input XML file is located at C:\InputData\InputFile.xml, then the log file will be created at C:\InputData\InputFile.log

Execute the CA Repository Universal XML Exchange Parser from the Command Line

You can also execute the CA Repository Universal XML Exchange Parser from command line without using the GUI.

To execute the CA Repository Universal XML Exchange Parser from command line

1. Open the command prompt.
2. Change the location to the folder where the Parser EXE exists, and type the following.

```
XMLEXCHANGE.exe

/i="C:\XML_Exchange\AION\AionAuto.xml"

/c="C:\XML_Exchange\AION\Aion_OO_new_rel_in_oi.xml"

/d="Worktables"

/u="sa"

/p="sa"
```

```
/w="nagja01"
/t="true"
/l="C:\XML_Exchange\AION\logFile.log"
```

The command has the following format

/i

Defines the input xml file path.

/c

Defines the control file path.

/d

Defines the data source name.

/u

Defines the user Id.

/p

Defines the password.

/w

Defines the work unit id.

/t

Specifies whether to set Auto Commit.

/l

Defines the log file name

The last two items(/t and /l) are optional. If they are not provided then Auto commit by default is false and a log file is created with the same name and same path as of the input xml file with an extension .log.

Perform a Direct Load to Mainframe—Mainframe Tasks

After the data is loaded into the work tables, you must start the mainframe process.

To start the mainframe process

1. Click Import and XML from the File menu.

   A panel similar to the following is displayed:

```
--------------------------- CA Repository for z/OS --------------------------

---------------------- Enter XML document load parameters --------------------

COMMAND ===>


More:     +

Import Options:
 Dialog for Repository Model    ===>  OBJECTO
 Status for New Objects         ===>  XMLAION
 First Search Status            ===> XMLAION
 Second Search Status           ===>
 Third Search Status            ===>
 Version                        ===> 0

 Work Unit ID                   ===> KALAB01

 Commit Mode                    ===>  A    (E, A)
      (E - Commit after each XML element definition is processed
       A - Commit at the end of the job after all xml element definitions)
 Translate Case as defined
      on the screen attributes? ===>   Y  (Y-Yes,N-No)
 Workstation Id                 ===>

   Reuse Rule PCR File Name     ===>

Job Options:
   Edit JCL?                    ===>  Y
```

2. Enter the XML document load parameters in the fields that appear on the CA Repository panel, as follows:

   Dialog for Repository Model

   Defines the repository model to be loaded.

   Status for New Objects

   Indicates the status for inserting objects. This status is also used to store new objects when a reusable object cannot be found.

   First Search Status, Second Search Status, Third Search Status

   Defines three statuses for search objects in default reuse rules.

Version

Defines version for objects from PCR file with version as reuse attribute. This version is not present in the PI table in the XML file.

Work Unit ID

Defines the work unit ID from the CA Repository Universal XML Exchange Parser.

Commit Mode

Indicates the commit mode.

Values:

- E—Commits after each repository object is processed.
- A—Commits at the end of the job after all the repository objects are processed.

Translate Case as defined on the screen attributes?

Translates character attribute based on case flag attribute from DBX_SCREEN_ATTR table or keep original case.

Workstation ID

(Optional) Defines the Workstation Id in which all new, updated and reused objects will be recorded.

Reuse Rule PCR File Name

(Optional) Defines the PCR data set.

3. Enter **Y** to edit and submit the generated JCL.

Chapter 6: Using the CA Repository Universal XML Exchange Loader

This chapter describes the CA Repository Universal XML Exchange Loader and the load tasks.

The CA Repository Universal XML Exchange Loader performs the following tasks:

- Reading parameters from the XML Import Panel
- Validating and expanding values in the PI table
- Loading objects from working tables into the repository

This section contains the following topics:

How the CA Repository Universal XML Exchange Loader Validates and Expands Value in the PI Table

The CA Repository Universal XML Exchange Loader performs the following tasks to validate and expand values in the PI table:

- Stores all values in the PI table as characters.

  The corresponding attributes in the repository can be in the following formats: character, numeric, and so on.

- Validates the PROP_VALUE of the PI table with corresponding numeric date or time repository format described in the DBX_IO_MAP_ATTR.
- Expands the PROP_VALUE is in the PI table.

- Checks the DBX_SCREEN_ATTR table is for corresponding Map ID.
- Generates the attribute using existing attributes from PI table and stored into PI tabl if an attribute has a default value and this attribute is not present in the PI table.

  Example:

  ```
  COLUMN has COLUMN_NAME default value as      S.TB_NAME||.||T.ELEMENT_NAME.
  ```

The loader checks whether the COLUMN_NAME exists in the PI table. If it does not exist, the loader selects the source TB_NAME (from Table) and the Element_name (from element) and concatenates them.

How Objects Are Loaded from Working Tables into the CA Repository

The CA Repository Universal XML Exchange Loader reads the four intermediate tables PRMXML_OI, PRMXML_AI, PRMXML_PI, and PRMXML_TI and loads the repository metadata tables.

The loading process is based on the Reuse Rules. There are two types of reusability rules:

- Dedicated reuse rules (see page 53) designed for a repository object and described in the PCR file
- A default reuse rule of Name and Status for any object not defined in the PCR file

Dedicated Reuse Rules (DRR) (see page 53) exist in the repository as DB2 Stored Procedures and are built by Reuse Rule Stored Procedure Builder (RRSPB) (see page 58) based on the input information from PCR file.

Default Reuse Rule (DefRR) does not use Stored Procedures to determine Reuse Rules. The Name and Status attributes are searched by the load program to determine if the row already exists.

The CA Repository Universal XML Exchange Loader processes all objects that exist in the PCR file with DRR rules and then processes the rest of objects from OI and AI tables with DefRR.

How Dedicated Reuse Rules Work

The CA Repository Universal XML Exchange Loader runs DRR in the following sequence:

1. Reads reuse rule information from the PCR file on current entity.
2. Builds the RRSP name and a list of reuse attributes.
3. Processes corresponding objects from the OI or AI tables depending on the object type-Entity, Relationship, or Association.
4. Selects attribute values from the PI table with respect to the reuse attribute list.
5. Calls RRSP and passes reuse attribute values as input parameters.

The Reuse Rule Stored Procedure sends back EntId, return code, message qualifier, message ID, and the reason and action code. If the return code is 0, the process continues. If the return code is 4 (not found), it tries the next search status for entities and relationships (if it exists); otherwise it stops and produces an error message.

The text processes for entities and relationships with respect to the #Text_Process rule in the PCR file if it exists, or with the default rule, which is O, override.

The CA Repository Universal XML Exchange Loader processes the current object depending on the DRR result and the action in PCR, as described in the following table:

Reuse Rule	Action in PCR	Stored Procedure Result F - Found N - Not Found	Action in CA Repository Universal XML Exchange	STORED Column after Action	Comments
#REUSE_OBJECT		N	Insert	I	
		F	Reuse	R	
	#ANCHOR	N	Skip	S	
		F	Reuse	R	
	#UPDATE	F	Update	U	
		N	Insert	I	
	#PURGE	N	Insert	I	

Reuse Rule	Action in PCR	Stored Procedure Result F - Found N - Not Found	Action in CA Repository Universal XML Exchange	STORED Column after Action	Comments
		F	Delete Insert	I	Existing Object will be deleted and new object will be inserted
#DEPENDENT_RELATED		N	Insert	I	
		F	Reuse	R	
	$UPDATE	F	Update	U	
#DRAIN_RELATES		N	Insert	I	
		F	Delete Insert	I	Existing Object will be deleted and new object will be inserted

How the Default Reuse Rule for Entity and Relationship Works

The CA Repository Universal XML Exchange Loader reads all the entities first and then the relationships from OI and AI tables. The Loader performs the following actions:

1. Retrieves Column Name for attribute with attr_code *N* NAME attribute in the DBX_SCREEN_ATTR table.
2. Selects NAME property from the PI table.
3. Sets SourceId and TargetId if it is a Relationship.
4. Sets Name and Status to check for DefRR. If object is not found, the CA Repository Universal XML Exchange Loader runs the next status.
5. Runs DefRR against the repository.

If the object is found, the CA Repository Universal XML Exchange Loader updates the object into the repository using DBXVLCX module. The CA Repository Universal XML Exchange Loader performs the following actions:

1. Retrieves all the attributes from the PI table for this object.
2. Retrieves the corresponding text from the TI table.

3. Updates the object into the repository using DBXVLCX.
4. If the option for workstation exists, the CA Repository Universal XML Exchange Loader adds EntId received from DBXVLCX to the reuse list.
5. Sets STORED column with *U* of OI, AI tables or both.
6. Sets ENT_ID and ENT_TYPE columns of OI, AI tables or both.
7. Updates OI, AI working tables or both.
8. Adds items to the Loading Report.

If the object is not found, the CA Repository Universal XML Exchange Loader inserts objects into the repository using DBXVLCX. The CA Repository Universal XML Exchange Loader performs the following actions:

1. Retrieves all the attributes from the PI table for this object.
2. Sets SourceId and TargetId if it is a Relationship.
3. Sets Status on Insert Status.
4. Retrieves the corresponding text from the TI table.
5. Inserts the object into the repository using DBXVLCX.
6. Sets STORED column with *I* of OI, AI tables or both.
7. Sets ENT_ID and ENT_TYPE columns of OI, AI tables or both.
8. Updates OI, AI working tables or both.
9. Adds items to the Loading Report.

How the Default Reuse Rule for Association Works

The CA Repository Universal XML Exchange Loader reads all unprocessed associations from AI table.

It performs the following actions:

1. Sets SourceId and TargetId.
2. Runs DefRR against repository.

If the object is found, the Loader reuses the object in the repository. It performs the following actions:

1. Sets STORED column with *R* of AI table.
2. Set ENT_ID and ENT_TYPE columns of AI table.
3. Updates AI working table.
4. Adds items to the Loading Report.

If the object is not found, the Loader inserts the objects into the CA Repository using DBXVLCX. It performs the following actions:

1. Sets SourceId and TargetId.
2. Sets STORED column with *I* of AI table.
3. Sets ENT_ID and ENT_TYPE columns of AI tables.
4. Updates AI working table.
5. Adds items to the Loading Report.
6. If the option for workstation exists, the CA Repository Universal XML Exchange Loader adds EntId received from DBXVLCX to the reuse list.

How the CA Repository Universal XML Exchange Loader Inserts Objects into the Repository

The CA Repository Universal XML Exchange Loader performs the following actions when inserting objects into the repository:

1. Selects all attributes for the current object from the PI table if it is an entity or a relationship.
2. Sets SourceId and TargetId if it is a relationship or an association.
3. Sets Status with Insert Status if it is an entity or a relationship.
4. Sets Version on 0; if it exists, assigns the next number.
5. Retrieves corresponding text from TI table if it is an entity or a relationship.
6. Inserts objects into the repository using DBXVLCX module.
7. Sets STORED column to I of OI, AI tables or both.
8. Sets ENT_ID and ENT_TYPE columns of the OI, AI tables or both.
9. Updates the OI, AI working tables or both.
10. Adds items to the Loading Report.

How the CA Repository Universal XML Exchange Loader Updates an Entity or a Relationship

The CA Repository Universal XML Exchange Loader performs the following actions to update an entity or a relationship:

1. Selects all Updated attributes (described in the PCR file) from the PI table.
2. Retrieves corresponding text from the TI table.

3. Sets EntId received from RRSP.
4. Updates objects in the repository using DBXVLCX.
5. Sets STORED column with U of OI, AI tables or both.
6. Sets ENT_ID and ENT_TYPE columns of the OI, AI tables or both.
7. Updates the OI, AI working tables or both.
8. Adds items to the Loading Report.

How the CA Repository Universal XML Exchange Loader Deletes Objects from the Repository

The CA Repository Universal XML Exchange Loader performs the following actions to delete objects from the repository:

1. Sets EntId received from RRSP.
2. Checks privileges for deleting this object.
3. Deletes object from the repository using the DBXIOD stored procedure.

How the CA Repository Universal XML Exchange Loader Reuses Objects

The CA Repository Universal XML Exchange Loader performs the following actions to reuse objects in the repository:

1. Retrieves corresponding text from the TI table and appends or overrides text depending upon the text flag in the PCR file if it is an Entity or a Relationship.
2. Adds EntId received from RRSP to the reuse list, if the option for workstation exists.
3. Sets STORED column with *R* of OI, AI tables or both.
4. Sets ENT_ID and ENT_TYPE columns of the OI, AI tables or both.
5. Updates the OI, AI working tables or both.
6. Adds items to the Loading Report.

Chapter 7: Understanding Dedicated Reuse Rules and PCR Files

Reuse rules enable you to reuse metadata that exists in the CA Repository rather than duplicating all the data in the database. You must specify the reuse rules in the PCAF Reuse file (PCR file). The Repository's Reuse Rules Stored Procedure Builder (RRSPB) reads the PCR file, and generates and builds Reuse Rule Stored Procedures (RRSPs), which in turn are used to select data for uploading to the CA Repository.

Note: The Reuse Rules stored procedure builder is documented in the CA Repository for z/OS Administration guide.

This section contains the following topics:

Reusability

Reusability is an important factor in the functionality of a repository as it gives the loader the ability to load data that is changed in the database instead of duplicating all the data in the repository. A good reuse scheme avoids data duplication and makes data rationalization possible. Without reusability, the repository becomes unusable from a data administration point of view.

Dedicated Reusability Using PCR File

This section explains the processing logic employed to reuse the existing object instances when new instances are imported into the repository. It also explains the PCR file format.

Reuse Terminology

This section defines the specialized meanings of the following terms that are used throughout the GUI:

Objects

Single entities, relationships and associations, or groups of entities, relationships and associations, which represent some external construct (for example, Relational Table, JCL Proc, and so on).

Source objects

Objects to be inserted into the repository and for which you are seeking reuse matches.

Functional Key

The set of attributes specifying reusability criteria for the source object.

Note: These attributes can span multiple repository entities and relationships.

Candidates

Any objects that meet the reusability criteria (match the functional key) for the source objects.

Types of Reuse

The majority of reuse cases can be broken down into the following categories:

- Simple Functional Key reuse
- Single-level, one-to-one entity dependency
- Single-level, one-to-many entities dependency
- N-level, one-to-many entities dependency

The following sections discuss each type of reuse case and the PCAF reuse rules (*.pcr) file that are used to control each case.

Simple Functional Key Reuse

This is the simplest case of reusability. If the exchange is ready to insert an entity into the repository, a functional key is a set of attributes that defines the entity in the context of the tool that is inserting it.

For example, you want to insert a data ELEMENT entity into the repository. The functional key for an ELEMENT is simple:

- The ELEMENT entity in a COBOL scan context has a functional key consisting of the main COBOL attributes (NAME, DATA_TYPE, and so on).
- In the context of a Sybase catalog import, the functional key is SYBASE_NAME, SYBASE_DATA_TYPE, and so on.

The NAME of the ELEMENT is not a part of the functional key, but the functional key changes for different contexts. This allows the same ELEMENT instance to be used in different contexts.

Other objects have functional keys that span multiple entities and relationships. For a Sybase TABLE object, two tables cannot be considered the same logical tables just because they have the same name. The two tables must have the same columns with the same data types in the same order to be considered equivalent.

You can expand the definition of reuse to include indexes, foreign keys, and so on. The functional keys can span simple repository entities and relates.

Single-Level, One-to-One Reuse

A functional key check lets you check the attributes of a relation between a specific entity and its children and parents to decide if there is a match. In this case, the definition of the functional key expands to include the relevant attributes of the root entity and the relevant attributes of the relationship and entities that also describe a reuse match.

To prevent multiple relationship instances from being created, not only are the relationship instances' attributes used during the reuse process, but the source and the target instances are also used. For example, COLUMN (Relational Column) is the relationship between TABLE and ELEMENT; an incoming COLUMN instance is compared to the existing COLUMN instances. If a COLUMN instance exists with the same attributes specified, and the same source and target, it is reused. The functional key consists of relevant attributes of COLUMN and the source and the target instance.

This case is characterized as a single-level, one-to-one reuse because there must only be a one-to-one relationship from TABLE to ELEMENT.

Single-Level, One-to-Many Reuse

This case is similar to the single-level, one-to-one case except that instead of having a one-to-one relationship, there can be one-to-many relationships.

The TABLE entity example discussed in Simple Functional Key Reuse (see page 54) states that matching the table name is not enough for the tables to be considered equivalent. It is possible for two tables to have the same name by coincidence, but to be different data objects.

In this case, you can expand the functional key of TABLE to be the TB_NAME, and all the COLUMNs that make up the table. Because a TABLE can have many COLUMNs, this is a case of single-level, one-to-many reuse.

Note: For the COLUMNs to be the same, they must relate to the same ELEMENT objects. Thus, two different TABLE entities are considered the same if they have the same TB_NAME and the same COLUMNs.

N-Level, One-to-Many Reuse

This is the most complicated reuse case. In this case, an entity contains other entities that again contain entities. This can go to *n* levels. A match occurs only in the case when all *n* levels are the same for load and candidate.

For example, the GROUP entity (from the COBOL and C models).

A GROUP in this case can contain many elements as well as other GROUPS. These underlying GROUPS can themselves contain other ELEMENTS, GROUPS, and so on. All the levels must be checked and an exact match must be found between source object and candidates.

PCR File Reference

The PCR file is complex because of the information it captures. There are default PCR files for each exchange supplied with the repository. However, if you do not want the repository objects reused in the way they are defined in the PCR file, you can change the reuse rules to make the load and reuse of objects behave differently.

PCR File Format

A PCR file consists of labels that indicate the type of information to follow, and the records that follow the labels containing the information. This combination of labels and associated information makes up the reuse rules. You can use a combination of labels and their associated information to create a block of related rules describing object blocks or dependent relate blocks.

All labels are prefixed by a # or $ character in the first byte. Some of the important considerations are as follows:

- Use the # identifier for a rule that is part of the original object block.
- Use the $ identifier for a rule that is included as part of a dependent relate block.

  During Loader processing, leading and trailing spaces and tabs are ignored. This is useful for specifying dependent objects. For more information, see #DEPENDENT_RELATE.
- Use a carriage return at the end of each line.

The PCR file must end with one carriage return after the last line of the file as specified in the syntax description. For more information, see the appendix "Syntax Diagrams."

The following sample PCR file shows a dependent relate block:

```
 #REUSE_RULE_SET
 PRO_EXAMPLE
 #REUSE_OBJECT
 1,TABLE,1
 #DRAIN_RELATES
 COLUMN,>
 #PROC_PREFIX
 PRO_EXAMPLE
 #GLOSSARY
 FUNCTIONAL_KEY_NAME,GLOSSARY_NAME,B
 #DEPENDENT_RELATE
    2,TBL COLS,>
    $FUNCTIONAL_KEY_NAME
    COLUMN
    $ATTRIBUTE_INFORMATION
    SOURCE_ID,L,4
    TARGET_ID,L,4

    -,NULL_INDICATOR,C,1
    SEQNUM,L,4
    $GLOSSARY
    NAME,DB2 SAMPLE,B
#FUNCTIONAL_KEY_NAME
 TABLE
 #ATTRIBUTE_INFORMATION
 -,NAME,C,35
 #TEXT_PROCESS
 0,DESCRIPTION
```

The PCR file is validated using the RRSP Builder (see page 58).

Working with RRSP Builder

The RRSP Builder builds RRSPs based on the reusability features written in the PCR file. RRSPs are stored in DB2 run library and can be later called for applying proper reuse rule for repository objects.

For applying reuse rule for <prodname> objects, PCR file reference is used.

A PCR file can be created for any <prodname> product. PCR files provide flexible tools for reusing objects inside of <prodname>.

The RRSP Builder performs the following actions:

- Reads PCR file and processes the Reuse objects one by one
- Puts text of stored procedure into output file
- Calls DB2 Stored Procedure Processor (DSNTPSMP) for building the stored procedure

Required Rules

The following rules are mandatory:

- #REUSE_OBJECT
- #REUSE_RULE_SET
- #PROC_PREFIX

Other rules are required only if they are used in conjunction with another rule.

#REUSE_OBJECT

The #REUSE_OBJECT rule is mandatory for each entity or relationship type to be loaded. It identifies the start of a block of reuse rules for an entity or relate. The sample PCR file contains only one entity type—TABLE.

For every entity type to be loaded, this block will be repeated except for the #REUSE_RULE_SET rule.

In the following example, the line following the label contains three fields, separated by commas:

```
#REUSE_OBJECT
object_type,object_name,process_order
```

The example contains the following information:

object_type

Indicates the type of the object.

Values:

- 1—Entity
- 2—Relationship
- 3—Association

object_name

Defines the name of the object to be compared.

process_order

Specifies the order in which this object must be processed.

The order is important because in a one-to-many hierarchy structure of N-levels, leaf nodes must be processed first. If you have source and target as reuse criteria, for objects that span multiple objects, both the source and the target object must be processed before the relationship is processed. For objects with only a simple functional key that does not span multiple objects, that is, does not have any downward relationships, this order is not important. They can have any value, and it will not interfere with any other object.

Note: Objects in a dependency hierarchy do not have to be adjacent as long as the leaves are processed at some point before their parent nodes.

#REUSE_RULE_SET

The #REUSE_RULE_SET rule is mandatory. It appears once for a PCR file. The line following the label specifies the name of the reuse scheme. The label must be the first non-comment line in the file.

```
#REUSE_RULE_SET
reuse_scheme_name
```

In the sample PCR file, PRO_EXAMPLE is the name of this reuse scheme.

For more information, see REUSE_RULE_SET syntax diagram (see page 82).

#PROC_PREFIX

The #PROC_PREFIX rule is mandatory.

The following example specifies the prefix for the name of the database stored procedure used during the reuse check.

```
#PROC_PREFIX
stored_proc_name
```

Object Block Rules

Use the following rules within the original object block:

- #ACTION
- #ATTRIBUTE_INFORMATION
- #COLUMN_INFORMATION
- #DEPENDENT_RELATE
- #DRAIN_RELATES
- #FUNCTIONAL_KEY_NAME
- #ORDER_ATTRS
- #TEXT_PROCESS

Note: Ensure to begin each label with a # identifier.

#ACTION

(Optional) If the #ACTION rule is used, it must follow the #PROC_PREFIX rule and precede the optional #GLOSSARY rule. If omitted, the default action is PERFECT, which means that if an instance is found in the repository that matches this reuse rule, it will be reused as it is without modifications.

The following are examples of # ACTION Rule:

```
#ACTION
action
```

Or

```
#ACTION
UPDATE
#UPDATE_ATTRS
attribute
.
.
.
attribute
```

The following are valid actions:

ANCHOR

Turns off internal referential integrity checking for the relationship objects in the working tables. The ANCHOR action is necessary when in working tables a relationship instance is used as a source or a target of another relationship instance, but this relationship's source and target are not supplied in the same working tables. The ANCHOR action will suspend referential integrity checks by using the attributes listed under the relationship's #ATTRIBUTE_INFORMATION as the reuse criteria for anchoring that incoming instance.

For more information about how to use #ACTION-ANCHOR, see Anchoring Relationship Instances.

PURGE

Deletes the existing object and adds the new one in its place.

UPDATE

Updates the list of #UPDATE_ATTRS that follows, if the object is found.

If the ACTION is UPDATE, a #UPDATE_ATTRS label must follow on the next line. After the #UPDATE_ATTRS label, a list of attribute names must follow the #UPDATE_ATTRS label. The list can include one or more attributes. When an existing instance in the repository is found to match this candidate object, each of the attributes listed in the #UPDATE_ATTRS section is set to the new value from the PI working table.

Example

The following example shows how the UPDATE action is specified, including the list of attributes to be updated.

Attributes listed for update are updated with incoming values only when the reuse criteria (all attributes listed under #ATTRIBUTE_INFORMATION) are satisfied (matched).

```
#REUSE_OBJECT
1,FILE,3
#PROC_PREFIX
PR_JCL2
#ACTION
UPDATE
#UPDATE_ATTRS
DS_ORGANIZATION
DATASET_TYPE
DIRECTORY_BLOCKS
BLOCKSIZE
LOGICAL_REC_LEN
ALLOCATION_UNIT
PRIMARY_ALLOCATION
SECONDARY_ALLOCATION
RECORD_FORMAT
RECORD_ORG
#FUNCTIONAL_KEY_NAME
FILE
#ATTRIBUTE_INFORMATION
NAME,C,60
STATUS,C,12
```

#ATTRIBUTE_INFORMATION

The #ATTRIBUTE_INFORMATION rule is optional.

For the object to be reused, the list of attributes for that object must be the same together with any dependent relates. In this example, only the NAME must match.

The line following the label contains four fields separated by commas.

```
#ATTRIBUTE_INFORMATION
case_flag,attribute_name,attribute_data_type,attribute_length
```

The preceding example contains the following parameters:

case_flag

Indicates case-insensitive reuse. To instruct the Loader to perform case-insensitive reuse, insert a hyphen (-) in this field.

Default: No hyphen. All comparisons are case-sensitive.

attribute_name

Defines the name of the attribute for the associated object.

attribute_data_type

Defines the data type of the attribute. The data type must match the corresponding entry in the repository.

Values:

- C—Character
- V—Variable-length character
- S—Short integer 2 bytes
- L—Long integer 4 bytes
- P—Decimal 8 bytes
- T—Time 8 bytes
- D—Date 8 bytes
- M—Timestamp 26 bytes

attribute_length

Defines the attribute's length, in bytes.

Example

To instruct the Loader to perform case-insensitive reuse for the NAME attribute, character format, 76 bytes long, specify the following rule in the PCR file:

```
#ATTRIBUTE_INFORMATION
-,NAME,C,76
```

The load is performed with no distinction between uppercase and lowercase versions of the same character.

#DEPENDENT_RELATE

The #DEPENDENT_RELATE rule is optional. It is used when an object's reuse criteria depends on its relationships or associations. These relationships or associations must exist in order for that object to be considered the same.

The TABLE example from the sample PCR file (see page 56), shows that a TABLE is the same only if it has the same NAME and the same COLUMN relationships, with those columns themselves having the correct attribution.

In the following example, the first line following the label indicates the beginning of group information for the relationship. This line contains three fields, separated by commas.

```
#DEPENDENT_RELATE
dependent_object_type,dependent_object_name,direction
   additional information
   .
   .
   additional information
```

The preceding example contains the following parameters:

dependent_object_type

Indicates the type of the dependent object.

Values:

- 1—Entity
- 2—Relationship
- 3—Association

dependent_object_name

Defines the name of the dependent object.

direction

Indicates the direction of the dependent object.

Values:

- >—Indicates the direction is downward (object is the source of this relationship).
- <—Indicates the direction is upward (object is the target of this relationship).

additional information

Specifies additional rules that complete the dependent relationship block. This block is repeated for each relationship that the object is dependent on. For more information about the rules used within a dependent relate block, see the DEPENDENT_RELATE syntax diagram (see page 84).

Example

The following example shows a complete dependent relate block. Note that the change of label identifier from # to $ indicates rules are associated with dependent relate:

```
#DEPENDENT_RELATE
    2,COLUMN,>
    $ACTION
    $FUNCTIONAL_KEY_NAME
    COLUMN
    $ATTRIBUTE_INFORMATION
    SOURCE_ID,L,4
    TARGET_ID,L,4
    -,NULL_INDICATOR,C,1
    SEQNUM,L,4
```

DRAIN_RELATES

The #DRAIN_RELATES rule is optional. This rule deletes the specified relationship objects when its associated object (identified in the #REUSE_OBJECT rule) is reused. This associated object can either be an entity or a relationship.

The following is an example of the #DRAIN_RELATE rule:

```
#DRAIN_RELATES
rel_object,direction
.
.
rel_object,direction
```

The example contains the following information:

rel_object

Specifies the name of the relationship or the association object. More than one relationship object can be specified for draining.

direction

Indicates the direction of drain from the associated object.

Values:

- >—Indicates the drain is downward (object is the source of this relationship).
- <—Indicates the drain is upward (object is the target of this relationship).

The direction of the drain is important because there can be relationship objects that have an object as both source and target (for example, DOM_DOM, LIKE). For example, to drain PROGRAM during a rescan of a COBOL program, drain the CSECT relationships downward, but not the CSECT that point to the PROGRAM being scanned.

The #DRAIN_RELATES rule is useful when specified relationship or association objects for the associated object are to be refreshed. If used, place the #DRAIN_RELATES rule just before the #PROC_PREFIX rule. Unlike the #ACTION – PURGE rule, the #DRAIN_RELATES rule does not delete the object itself.

For any specified relationship or association objects to be drained, every reuse criteria for the associated object must be met. That is, the associated object must be reused for the relationship or association objects to be drained. The reuse criteria include matching on every attribute listed under the object's #ATTRIBUTE_INFORMATION rule, and satisfying all of its #DEPENDENT_RELATE criteria.

If a match is not found, the object is not reused, and the specified relationship or association objects are not deleted.

Note: When the specified relationship or association objects are deleted, this deletion triggers the deletion of any relationship or association that is associated with it, and continues to cascade to delete all subsequent associated relationships.

Example

The following is an example of where and how the #DRAIN_RELATES block appears in a PCR file:

```
#REUSE_OBJECT
2,TABLE,1
#DRAIN_RELATES
COLUMN,>
PRIM_KEY,<
#PROC_PREFIX
PR_DRAIN
#FUNCTIONAL_KEY_NAME
TABLE
#ATTRIBUTE_INFORMATION
NAME,C,125
STATUS,C,12
```

The reuse rule for TABLE states that if an instance in the repository matches on NAME and STATUS of an incoming instance, use that instance in the repository and delete all COLUMN and PRIM KEY instances that are associated with this TABLE instance.

COLUMN,> specifies a downward (>) drain, which means that the source of this relationship object (ELEMENT) is from the associated object (TABLE).

PRIM KEY,< specifies an upward (<) drain, which means that the target of this relationship object (KEY) is from the associated object (TABLE).

#FUNCTIONAL_KEY_NAME

The #FUNCTIONAL_KEY_NAME rule is optional. The value supplied is used to distinguish between different Functional Keys for the same object.

The following is an example of #FUNCTIONAL_KEY_NAME rule:

```
#FUNCTIONAL_KEY_NAME
functional_key_name
```

#ORDER_ATTRS

The #ORDER_ATTRS rule is optional. It is used to process recursive objects.

The following is an example of #ORDER_ATTRS rule:

```
 #ORDER_ATTRS
relationship/association
```

The preceding example contains the following parameter:

relationship/association

Specifies the relationship or association that is to be used in the recursive method.

This rule is used for processing the recursive objects. If you have a relationship CL ASSOC as a dependent relate rule, you must process the CLASS first and then process the CLASS without a parent.

The recursive method is used for processing such type of objects.

Example:

The following is an example of where and how the #ORDER_ATTRS block appears in a PCR file:

```
#REUSE_OBJECT
1,CLASS,20
#PROC_PREFIX
OBJ
#DEPENDENT_RELATE
2,CL MB RC,>
$FUNCTIONAL_KEY_NAME
CL MB RC
$ATTRIBUTE_INFORMATION
SOURCE_ID,L,4
TARGET_ID,L,4
#DEPENDENT_RELATE
2,CL MB FU,>
```

```
$FUNCTIONAL_KEY_NAME
CL MB FU
$ATTRIBUTE_INFORMATION
SOURCE_ID,L,4
TARGET_ID,L,4
#FUNCTIONAL_KEY_NAME
CLASS
#ATTRIBUTE_INFORMATION
NAME,V,245
STATUS,C,8
// recursive order depend on CL ASSOC
#ORDER_ATTRS
CL ASSOC
//
```

#TEXT_PROCESS

The #TEXT_PROCESS rule is optional. It describes how the corresponding type of text associated with the current object is going to be processed if the object is going to be reused.

The following is an example of #TEXT_PROCESS rule:

```
#TEXT_PROCESS
text_process_identifier,text_type
```

This example contains the following information:

text_process_identifier

Specifies how to process the text attribute.

Values:

- A—Appends the text from the current load to the text already in the repository.
- O—Overwrites the text in the repository with the text from the current load.

text_type

Specifies the type of text 1-5.

$ACTION

The $ACTION rule is optional. If it is used, it must follow the #DEPENDENT_RELATE rule and must precede the $FUNCTIONAL_KEY_NAME rule. If omitted, the default action is PERFECT, which means that if an instance is found in the repository that matches this reuse rule, it will be reused as it is without modifications.

The following is an example of $ACTION rule:

```
$ACTION
action
```

Or

```
$ACTION
UPDATE
$UPDATE_ATTRS
attribute
.
.
.
attribute
```

This rule has the following valid actions:

PURGE

Indicates that the existing object type instance in the repository is to be deleted if the reuse criteria are satisfied, and then the incoming instance is to be added. This action replaces the existing instance with the new incoming instance. The deletion of an object type instance cascades to its relationship instances.

UPDATE

Updates the list of $UPDATE_ATTRS that follows if the object is found.

If the ACTION is UPDATE, then a $UPDATE_ATTRS label must follow immediately on the next line. After the $UPDATE_ATTRS label, a list of non-text attribute names must follow. The list may include one or more attributes. When an existing instance in the repository is found to match this candidate object, then each of the attributes listed in the $UPDATE_ATTRS section is set to the new value from the PI working table.

Example

The following example shows how the UPDATE action is specified within a dependent relate block, including the attribute to be updated:

```
#REUSE_OBJECT
1,TABLE,12
#PROC_PREFIX
PRO_ODBC
#DEPENDENT_RELATE
2,COLUMN,>
        $ACTION
        UPDATE
        $UPDATE_ATTRS
        SEQ_NUM
        $FUNCTIONAL_KEY_NAME
        TBL COL
        $ATTRIBUTE_INFORMATION
        SOURCE_ID,L,4
        TARGET_ID,L,4
        -,NULL_INDICATOR,C,1
        #FUNCTIONAL_KEY_NAME
TABLE
#ATTRIBUTE_INFORMATION
NAME,C,76
STATUS,C,12
```

The attribute SEQ_NUM will be updated if COLUMN is reused based on the attributes specified in the $ATTRIBUTE_INFORMATION (see page 71) section.

$ATTRIBUTE_INFORMATION

The $ATTRIBUTE_INFORMATION rule is required for dependent relate objects. It indicates what attributes on the relationship must be used in a reuse check for the dependent relationship.

The following is an example of $ATTRIBUTE_INFORMATION rule:

```
#ATTRIBUTE_INFORMATION
attribute_name,attribute_data_type,attribute_length
.
.
attribute_name,attribute_data_type,attribute_length
```

This example contains the following information:

attribute_name

Defines the name of an attribute for the dependent relationship.

attribute_data_type

Identifies the data type of the attribute.

Values:

- C—Character
- V—Variable-length character
- S—Small integer
- L—Integer
- P—Decimal
- T—Time
- D—Date
- M—Timestamp

attribute_length

Defines the attribute length in bytes.

In the sample PCR file (see page 56), there are six attributes for COLUMN that must match for the column to be considered the same. All the COLUMNs relating to TABLE must match in all these attributes. If any one fails to match on even one attribute, the table will not be reused.

$FUNCTIONAL_KEY_NAME

The $FUNCTIONAL_KEY_NAME rule is mandatory if there is a dependent relate block. The value supplied is used to distinguish between different Functional Keys for the same object.

The following is an example of $FUNCTIONAL_KEY_NAME rule:

```
$FUNCTIONAL_KEY_NAME
functional_key_name
```

$TEXT_PROCESS

The $TEXT_PROCESS rule is optional. It describes how the text attributes associated with the current dependent relate object are processed if the dependent relate object is reused. The reuse processing for the text attributes does not cause the text attributes themselves to version, nor does it cause any related parent objects to version.

The following is an example of $TEXT_PROCESS rule:

```
$TEXT_PROCESS
text_process_identifier, text_type
```

The example contains the following information:

text_process_identifier

Specifies how to process the text attribute.

Values:

- A—Appends text from the current load to the text already in the Repository.
- O—Overwrites text in the Repository with text from the current load.

text_type

Specifies the type of text 1-5.

Anchor Relationship Instances

The #ACTION–ANCHOR rule is used in a PCR file when a relationship instance (COLUMN) is used as a source or a target of another relationship instance (TRIG COL) in a working table, but this relationship's source and target (TABLE and ELEMENT) are not specified in the working tables.

If there are instances of relationship objects in these working tables that do not have their source or target specified, they must still be used as the source or target of an incoming relationship. For the Loader to suspend referential integrity checking on relationships used for a source or target, the #ACTION–ANCHOR rule is used in the PCR file.

In the following example, if this COLUMN instance has no source or target specified in the working tables, the source or target tokens are replaced by 0 and the #ACTION–ANCHOR rule is applied to COLUMN in the working tables.

For a relationship instance to be anchored, the instance must already exist in the repository. A relationship instance cannot be added if the source and the target instance are not specified in the same working tables at the time of the load. Using the #ACTION–ANCHOR rule in the PCR file, the PCR file tells the Loader to by-pass that rule.

If you want to load a TRIG COL instance and the COLUMN already exists in the repository, anchor it based on the specified attributes; the source and the target (TABLE an ELEMENT, respectively) are not checked during the load process.

Similar to the regular reuse, the attributes specified under that object's #ATTRIBUTE_INFORMATION section are used to find the matching instance on which to anchor. To get the best match to anchor on, specify as many attributes as there are in the working tables.

Example

The following is an example of where and how the #ACTION–ANCHOR block appears in a PCR file:

```
#REUSE_OBJECT
2,COLUMN,1
#PROC_PREFIX
PR_RC
#ACTION
ANCHOR
#FUNCTIONAL_KEY_NAME
COLUMN
#ATTRIBUTE_INFORMATION
COLUMN_NAME,C,245
STATUS,C,12
```

The previous reuse rule for COLUMN states that if an instance in the repository matches the COLUMN_NAME and STATUS of an incoming instance, use that instance in the repository. Ensure that the attributes specified in the PCR file exist in the working tables; otherwise, a match will not be found.

Appendix A: Work Tables

The work tables that hold the output from the scan that is loaded by the CA Repository Universal XML Exchange Parser are as follows:

- Object Intermediate (OI) table, PRMXML_OI
- Properties Intermediate (PI) table, PRMXML_PI
- Associations Intermediate (AI) table, PRMXML_AI
- Text Intermediate (TI) table, PRMXML_TI

During Scanner processing, the loader reads these four work tables and inserts, updates, or reuses data in the <prodname> objects.

The rules for mapping objects are in the control file starting with the XML element whose tag name is Object_Map (formerly ERWXML_Object in the ERwin control files).

This section contains the following topics:

PRMXML_OI Work Table Objects

PRMXML_OI contains a list of repository entity objects that were parsed from the XML document.

Such objects can be entities or relationships.

Storing Relationships in both OI and AI tables is optional besides the following cases:

- Relationship represents as a separate object in XML file
- Relationship stores as a source or a target for another relationship or association

The ent_name is the name of the entity in the repository. An object in the XML document is defined as an XML element with an ID attribute.

An object in the XML document can be mapped to more than one repository object. For example, an XML element with a tag of Method may map to a repository object called PROGFUNC.

Note: The rules for mapping objects are in the control file starting with the XML element whose tag name is Object_Map (formerly ERWXML_Object in the ERwin control files).

The following table describes the objects in the PRMXML_OI work table.

Column Name	Data Type	Description
KEY_GUID	VARCHAR(128)	The ID of the object in the XML file as populated by the CA Repository Universal XML Exchange Parser.
ENT_NAME	VARCHAR(60)	The repository entity name as populated by the parser. The XML control file contains this name as the target for a particular XML element.
ENT_TYPE	SMALLINT	The repository entity type. This attribute is populated by the load program.
ENT_ID	INTEGER	The unique ID of the repository row. This attribute is populated by the load program.
STORED	CHAR(01)	■ I—inserted ■ U—updated ■ R—reused ■ NULL—not processed ■ F—found in the RRSP ■ N—not found in the RRSP ■ S—skip from processing ■ E—error during processing
GENERATED_GUID	CHAR(01)	Specific to the XML scanner.
WORK_UNIT	CHAR(30)	A value that is used to separate one scan from another. This attribute is populated by the CA Repository Universal XML Exchange Parser.

PRMXML_PI Properties Work Table

The Properties Work Table, PRMXML_PI, contains a list of attributes about the repository tables that were parsed from the XML document. The control file contains rules about what constitutes an attribute in the XML file. The rules for mapping of attributes are in the control files. The XML element Attr_map (formerly known as ERWXML_Attr in the ERwin control files) is in the Object_Map XML element.

The data stored in the Column PROP_VALUE may have already gone through any required transformation process. Before storing the PRMXML_PI table, the CA Repository Universal XML Exchange Parser applies transformation rules specified in the control files. The XML Mapping element describes the simple transformation rules. For example, the value of 0 in the XML document for property FLAG in object TABLE must be stored as an N.

The following table describes the objects in the PRMXML_PI work table.

Column Name	Data Type	Description
KEY_GUID	VARCHAR(128)	The ID of the object in the XML file as populated by the CA Repository Universal XML Exchange Parser. This ID exists in the PRMXML_OI table.
ENT_NAME	VARCHAR(60)	The repository entity name as populated by the Parser. This name matches the ENT_NAME in the PRMXML_OI table for this specified KEY_GUID.
PROP_TYPE	CHAR(18)	The name of the column (attribute) of the repository object that stores the value in the XML document.
PROP_VALUE	VARCHAR(750)	The value as found in the XML document that will be populated in the specified column of the specified object.
WORK_UNIT	CHAR(30)	A value used to separate one scan from another. This attribute is populated by the CA Repository Universal XML Exchange Parser.

PRMXML_AI- Associations Objects Work Table

The Associations Objects work table, PRMXML_AI, contains a list of repository relationship and association objects that were parsed from the input XML document. The control file contains rules for what constitutes a relationship in the XML file. The section in the control file for defining relationships starts with the XML element with a tag name of Relationship.

Column Name	Data Type	Description
KEY_GUID	VARCHAR(128)	The ID of the object in the XML file as populated by the parser. This ID is in the PRMXML_OI table.
ENT_NAME	VARCHAR(60)	The repository entity name as populated by the parser. The XML control file contains this name as the target for a particular XML element.
KEY_GUID_SOURCE	VARCHAR(128)	The ID of the source of the object in the XML file as populated by the parser. This ID is in the PRMXML_OI table.
ENT_NAME_SOURCE	VARCHAR(060)	The repository entity type of the source object. This attribute is populated by the load program.
KEY_GUID_TARGET	VARCHAR(128)	The ID of the target of the object in the XML file as populated by the parser. This ID is in the PRMXML_OI table.
ENT_NAME_TARGET	VARCHAR(060)	The repository entity type of the target object. This attribute is populated by the load program.
ENT_ID	INTEGER	The unique ID of the relationship row in the repository. This is populated by the load programs.
STORED	CHAR(1)	■ I—inserted ■ U—updated ■ R—reused ■ NULL—not processed ■ F—found in the RRSP

Column Name	Data Type	Description
		■ N—not found in the RRSP ■ S—skip from processing ■ E—error during processing
WORK_UNIT	CHAR(30)	A value used to separate one scan from another. This attribute is populated by the CA Repository Universal XML Exchange Parser.
ENT_TYPE	SMALLINT	The repository entity type. This attribute is populated by the load program.

PRMXML_TI Text Objects

PRMXML_TI contains textual data for an object. The repository supports five different text tables. Therefore, each object can contain up to five text entries.

The control file contains rules for what constitutes text data in the XML file. The rules for mapping of text data are found in the control files. The XML element TextCol is contained with the Attr_Map XML element.

Column Name	Data Type	Description
KEY_GUID	VARCHAR(128)	The ID of the object in the XML file as populated by the CA Repository Universal XML Exchange Parser. This ID is in the PRMXML_OI table.
ENT_NAME	VARCHAR(60)	The repository entity name as populated by the parser. This name matches the ENT_NAME in the PRMXML_OI table for this specified KEY_GUID.
TEXT	VARCHAR(32000)	Text data that is stored in the XML document.
TYPE	CHAR(1)	The repository text type which represents which text table is to be used.
WORK_UNIT	CHAR(30)	A value that is used to separate one scan from another. This attribute is populated by the CA Repository Universal XML

Column Name	Data Type	Description
		Exchange Parser.
CREATE_TIME	TIMESTAMP	A DB2 system generated timestamp for use by the load program to sort the text.

Appendix B: Syntax Diagrams

This appendix contains the following syntax diagrams:

- REUSE_RULE_SET (see page 82)
- REUSE_OBJECT (see page 83)
- DEPENDENT_RELATE (see page 84)

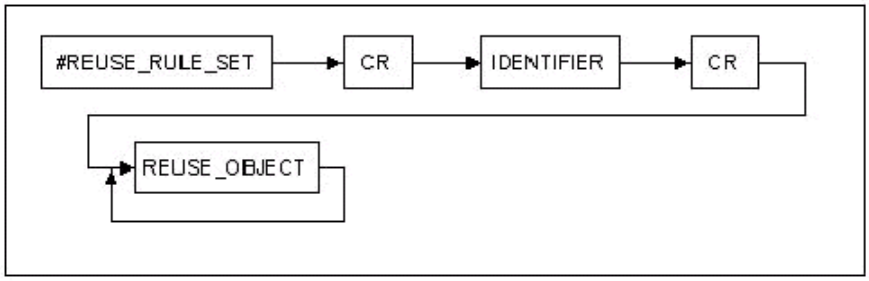
The following table describes the symbols and definitions used in the syntax diagrams:

Symbol	Definition
CR	Carriage return.
IDENTIFIER	Valid identifier.
DT	Data type, that is, CHAR, LONG.
OBJECT_NAME	The entity, relationship or association name
ORDER_NUM	An unsigned integer that specifies the order in which the object will be processed.
ATTRIBUTE	Attribute that exists in the PI table for the currently processed object.
PROCEDURE_PREFIX	Any identifier with a maximum number of characters equal to 9.
LENGTH	An unsigned integer specifying the length of the column. For L (integer) use 4, for S (smallint) use 2.
GLOSSARY_NAME	Identifier for name of glossary.

This section contains the following topics:

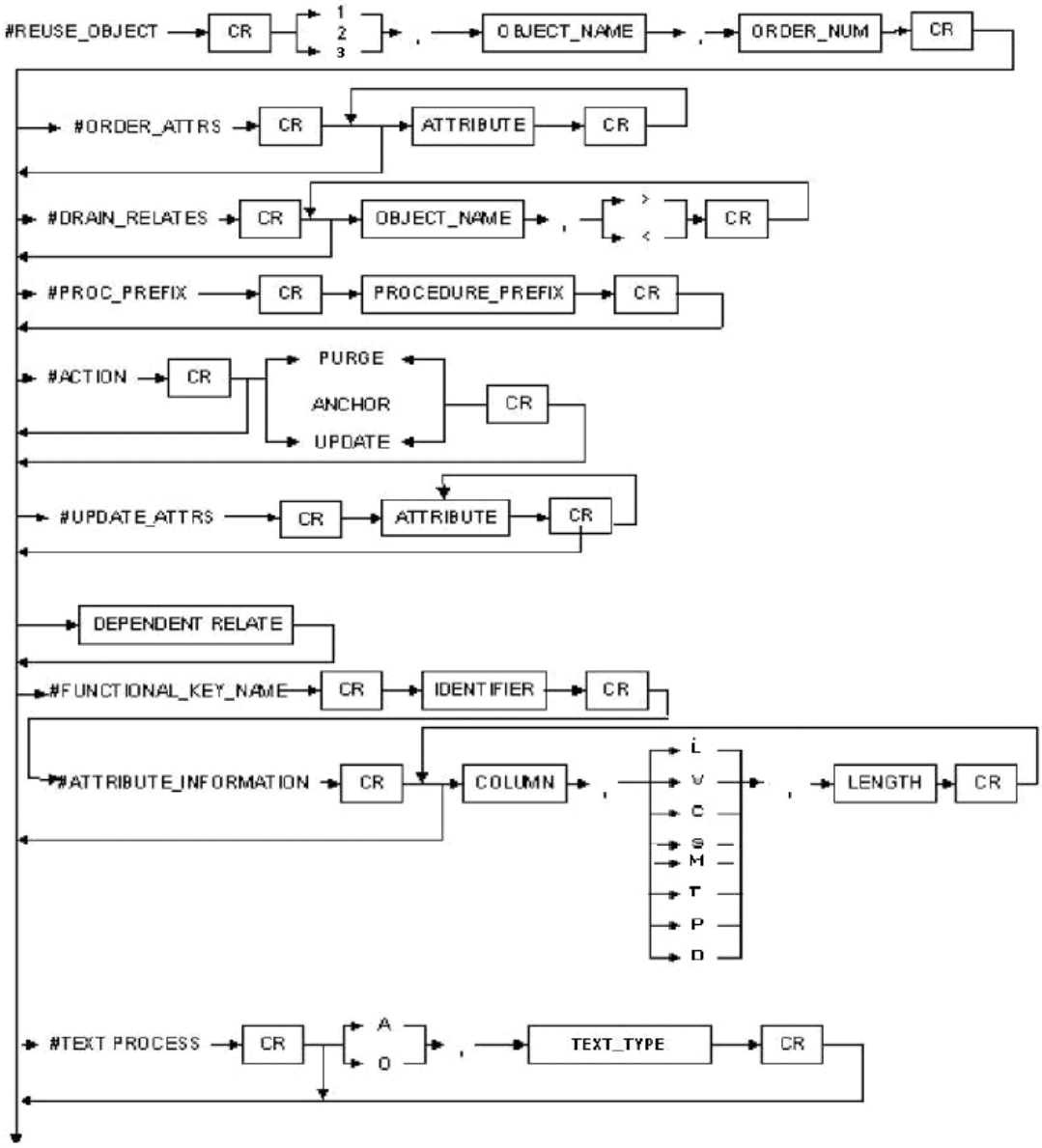
REUSE_RULE_SET

The following is the syntax diagram for REUSE_RULE_SET:



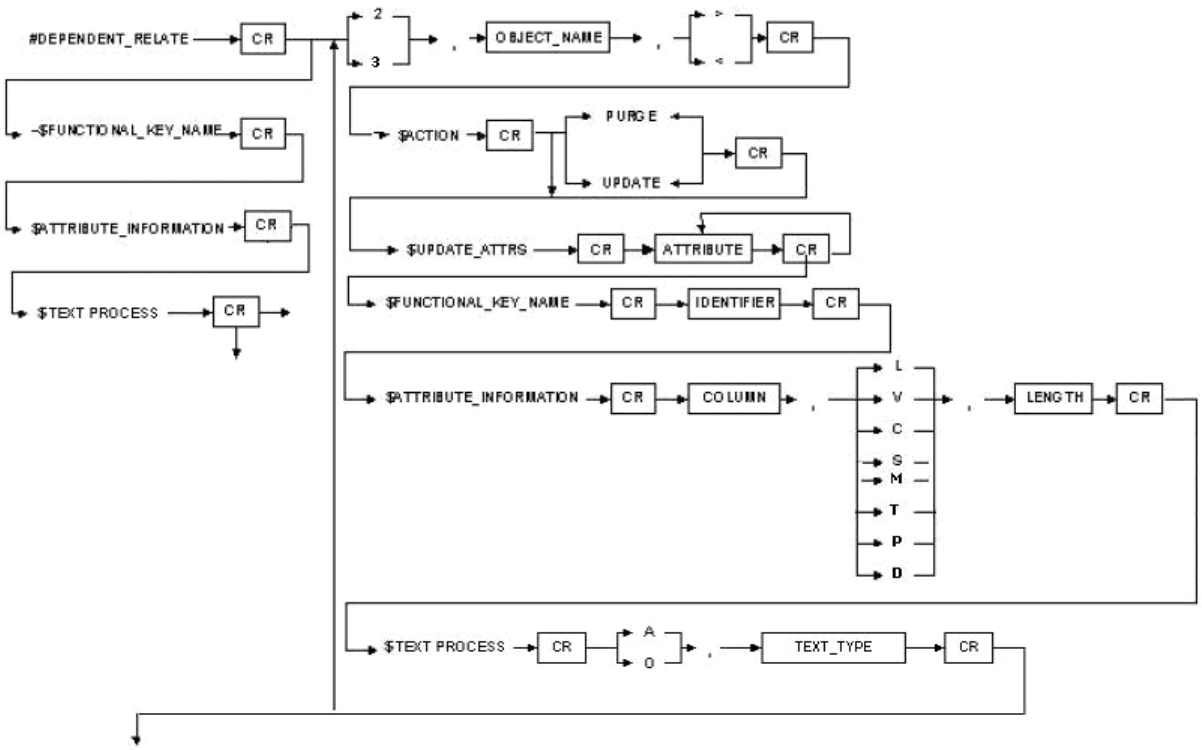
REUSE_OBJECT

The following is the syntax diagram for REUSE_OBJECT:



DEPENDENT_RELATE

The following is the syntax diagram for DEPENDENT_RELATE:



Glossary

attribute name

Attribute name specifies the name of the attribute used throughout the XML file to reference another XML element in the XML file.

control file

The *control file* contains the rules for locating the data in the input XML files and mapping the data to work tables in the repository database on the mainframe.

heap space

Heap space refers to a common pool of memory that is available in a program. Heap management is either performed by the application itself by allocating and deallocating memory as required, or by the operating system or other system program.

link attribute

A *link attribute* describes an attribute that is used in the XML document to reference another XML element in the document. In XML terminology, these attributes are usually known as IDREF attribute types.

mapping

Mapping is a mechanism for changing values in the input XML file before it is uploaded to the repository. Mapping enables the conversion of input XML file data into <prodname> values for Entities or Relationships in preparation for uploading the file to the repository.

metadata

In the repository, *metadata* is data about data, processes and any other information assets, and their interrelationships. This includes definitions about data elements, records, tables, sub-schema, schema, definitions in a data dictionary or SQL catalog as well as documentation on operations, files, programs, processes, data warehouse extraction mappings, transformation rules, and load mappings, and any other information asset an organization would like to track. On a business level, stewardship, business rules, categorization, and business terms and definitions are considered metadata.

parser

A *parser* is a component of the compiler that transforms input text into a data structure, usually a tree, which is suitable for later processing and which captures the implied hierarchy of the input.

relationship

A *relationship* consists of definitions that specify how objects in the repository are associated.

repository

A *repository* is a central point where metadata is populated, stored, and made available to users. It is the roadmap to information assets within an organization.

reuse rules

Reuse rules let you reuse metadata that exists in the CA Repository rather than duplicating all the data in the database.

root element name

The *root element name* element, RootElementName, describes the Name of the Root Element in the input XML file. A control file can have only one XML one root element per input XML file.

scanner

A *scanner* is a program that reads through (scans) program code looking for information that describes data structures (metadata). The scanner creates an output file that you can subsequently load into the <prodname> and merge with object definitions from other data sources.

transformation

Transformation is the process that takes an input XML document and creates an XML output file that you can use as input to the CA Repository Universal XML Exchange.

work unit

A *work unit* groups data so that you can run the CA Repository Universal XML Exchange Parser multiple times before you perform a load.

XSL

XSL (eXtensible Stylesheet Language) is a family of transformation languages that let you describe how files encoded in the XML standard must be transformed or formatted.

XSLT (XSL transformations)

XSLT (XSL transformations) is an XML language for transforming XML documents.

Index

S

T

W

X