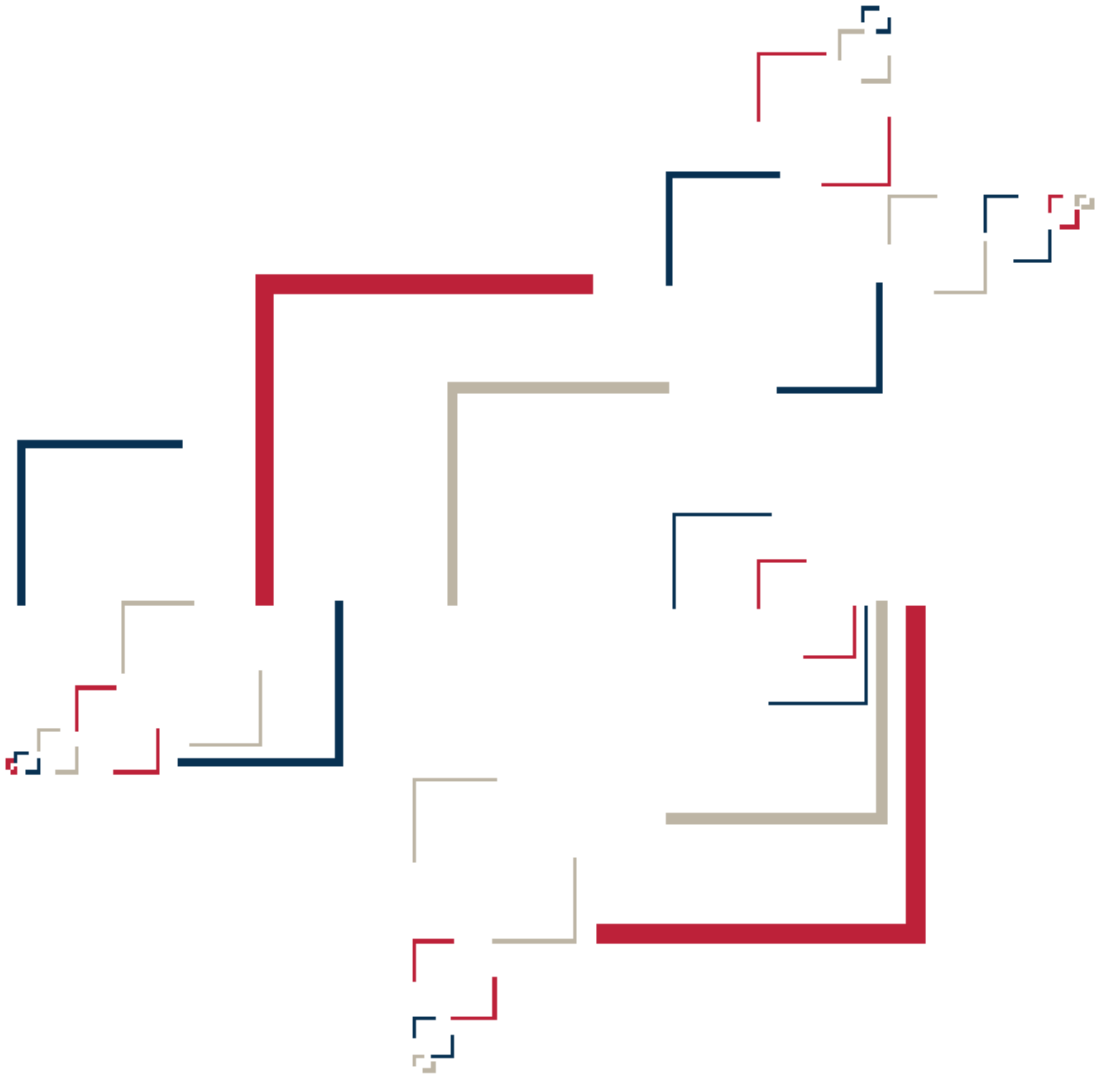


Micro Focus®

Modernization Workbench™

Preparing Projects

MICRO FOCUS®

Micro Focus (IP) Ltd.
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

Contents

Registering Source Files

Before you can analyze application source files in Modernization Workbench, you need to load, or *register*, the source files in a workspace.

Note: In a multiuser environment, only a master user can register source files.

The workbench creates copies of the registered files on the server machine (or locally in a single-user environment) in the Sources folder for the workspace. These are the files you view and edit in the workbench tools. You can restore a file to its original state, update it to its current state, or export it as necessary.

Source files must have recognized DOS file extensions before they can be registered. You can view and add to the recognized extensions in the Workspace Registration options window. Files without extensions are checked for content, and if the content is recognized, the files are registered with appropriate extensions appended.

The workbench assumes that input source files are ASCII files in DOS format. Occasionally, files may be converted incorrectly from other formats to DOS-based ASCII with an extra special character (like “M”) at the end of each line. While Modernization Workbench accepts these files as input, some workbench tools may not work correctly with them. Make sure all source files are in valid ASCII format.

You can register source files in compressed formats (ZIP or RAR), as well as uncompressed formats. Modernization Workbench automatically unpacks the compressed file and registers its contents.

Note: The workbench extracts compressed source files using the command line syntax for archiver versions most widely in use. If you use newer archiver versions, specify the command line syntax in the Archivers tab of the User Preferences window.

Workspace Registration options determine registration behavior. The default values for these options are preset based on your configuration and should be appropriate for most installations.

1. In the Repository Browser, create a project for the source files you want to register, or use the default project. To create a project, choose **Project > New Project**. The Create Project dialog opens. Enter the name of the new project and click **OK**. The new project is displayed in the Repository Browser.
2. Select the project in the Repository Browser, then drag-and-drop the file or folder for the source files you want to register onto the Repository Browser.
   You are notified that you have registered the files successfully and are prompted to verify the files. Click **Close**. The Repository Browser displays the contents of the new workspace, organized by file type.

   Note: In the notification dialog, select **Never ask again** if you do not want to be prompted to verify files. On the Environment tab of the User Preferences window, select **Ask user about verification** if you want to be prompted again.

Setting Registration Options: Extensions Tab

Source files must have recognized DOS file extensions before they can be registered. Files without extensions are checked for content, and if the content is recognized, the files are registered with appropriate extensions appended.

Files with unknown extensions are flagged, provided that you uncheck **Ignore Unknown and Overloaded Extensions** on the Extensions tab of the Workspace Registration options window. If a file fails to register because its extension is unknown, simply add the extension to the list of recognized extensions on the Extensions tab and register the file again.

1. Choose **Tools > Workspace Options**. The Workspace Options window opens. Click the Registration tab, then the Extensions tab.
2. In the Source Type pane, select the source file type whose extensions you want to view. The extensions for the file type are listed in the Extensions pane. Select each extension you want the system to recognize. Add extensions as necessary.

   Note: If a source file does not specify an extension when it references an included file, the verification process assumes that the included file has one of the recognized extensions. If multiple included files have the same name but different extensions, the system registers the file with the first extension in the list.
3. Select **Ignore Unknown and Overloaded Extensions** if you do not want the registration process to issue warnings about unrecognized and overloaded extensions. An overloaded extension is one assigned to more than one file type.
4. For Cobol programs and copybooks, select **Remove Sequence Numbers** if you want the system to replace preceding enumeration characters, or *sequence numbers*, with blanks. Sequence numbers are removed only from the source file versions maintained by the workbench.
5. For Copybook, C, C++, PowerBuilder, or Java files, select **Preserve Folder Structure** if you want to register files with names derived from their locations in the folder structure rather than simple base names or, in the case of Java files, names based on the package declaration. As long as you drag-and-drop the folders (or a parent folder) onto the Repository Browser when you register the files, the folder structure is reflected in the name. Choose this option when:

   - Copybook, C, C++, or PowerBuilder files with identical names reside in different folders. Your application might use the same copybook in different Partitioned Data Sets on the mainframe, for example, in which case the copybook will reside in different folders on your PC.
   - Java files with the same name and package declaration are used in different applications. So Id.java in the app1 folder is registered as "app1\com\company\Id.java", for example, while Id.java in the app2 folder is registered as "app2\com\company\Id.java".

   Note: For all but Java files, the folder structure for the application is preserved in the display names of the files in the Repository Browser. For Java files, check Properties > General for the full repository name.

Setting Registration Options: Source Files Tab

If your legacy application executes on a mainframe, it's usually best to convert the application source to workstation encoding. If that's not practical, you can have Modernization Workbench convert it for you, using the options on the Registration > Source Files tab of the Workspace Options window.

1. Choose **Tools > Workspace Options**. The Workspace Options window opens. Click the Registration tab, then the Source Files tab.
2. In the Legacy Source Encoding group box, choose:

- **Workstation** if the source is workstation-encoded. For DBCS configurations, if Japanese-language source files were downloaded in workstation (text) mode, specify how DBCS escape control characters were handled.
- **Mainframe** if the source is mainframe-encoded. When this option is selected, the registration process automatically converts source files to workstation-encoding. Only the source files maintained by the workbench are converted.

3. In the Object System Encoding group box, choose:

   - **English - US (ANSI MS-1252)** if the original source was U.S. English ANSI-encoded (Unisys 2200 and HP3000 Cobol).
   - **English - US (EBCDIC-CCSID-37)** if the original source was U.S. English EBCDIC-encoded (IBM Cobol).
   - **Japanese (EBCDIC-CCSID-930, 5026)** if the original source was Japanese EBCDIC-encoded, CCSID-930, 5026 (DBCS configurations only).
   - **Japanese (EBCDIC-CCSID-939, 5035)** if the original source was Japanese EBCDIC-encoded, CCSID-939, 5035 (DBCS configurations only).

   During analysis and transformation, hexadecimal literals in Cobol programs and BMS files are translated into character literals according to this setting.

   Note: Do not change these settings after source files are registered in a workspace.

4. Select **Strip trailing numeration** if you want the system to strip trailing numeration characters (columns 73 through 80) from source lines. Trailing numeration characters are removed only from the source files maintained by the workbench.
5. Select **Expand tabulation symbols** if you want the system to replace tabulation symbols with a corresponding number of spaces. Tabulation symbols are replaced only in the source files maintained by the workbench. You must select this option if you want to view HyperView information for C or C++ programs.
6. In the **Default Source Directory** field, enter the root folder on your PC from which the system should refresh unresolved files. You can type over the path in the text box or click the button to the right of the text box to browse for a new location.

Creating New Source Files

To create a new source file, select the project for the source file in the Repository Browser and choose **File > New**. A dialog box opens, where you can specify the file name (with extension) and source file type. To create a new source file with the same content as an existing file, select the file and choose **File > Save As**. The system automatically registers the created files and stores them in the appropriate folder.

Refreshing Source Files

Use the Modernization Workbench *refresh* feature to update registered source files to their current state. You can refresh all of the objects in a project or folder, or only selected objects.

The refresh looks for updated legacy source in the original location of the file or, for unresolved source, the location you specified in the Registration > Source Files tab of the Workspace Options window. Once it finds the source, it overwrites the version of the source file maintained by the system. Reverify the file after the refresh.

Note: If the refreshed file is unresolved, the referring source is not invalidated. To resolve the refreshed file, reverify both the unresolved file and the referring source.

1. In the Repository Browser, select the project, folder, or file you want to refresh and choose **File > Refresh Sources from Disk**.

   Note: If you are licensed to use the Batch Refresh feature, you can perform the refresh in batch mode. Contact support services for more information. The behavior you will see when refreshing source from disk on an unresolved object, the file that refers to that object is not invalidated. If the referring file is reverified, then the missing object that was refreshed will be resolved.
2. You are prompted to confirm that you want to refresh the selected files. Click **Yes**.

The system overwrites the workspace source files.

Exporting Source Files from a Workspace

To export the workspace source for a project or file to a new location, select the project or file in the Repository Browser and click **File > Export Sources**. A dialog box opens, where you can specify the location.

Deleting Objects from a Workspace

To delete an object from a workspace, select it and choose **File > Delete from Workspace**. To delete a folder and all its contents from a workspace, select it and choose **File > Delete Contents from Workspace**.

Deleting a Workspace

To delete a workspace, choose **File > Delete Workspace** in the Modernization Workbench Administration tool. A Delete workspace dialog opens, where you can select the workspace you want to delete.

Note: Only a master user can delete a workspace in a multiuser environment.

Japanese Language Support

Modernization Workbench provides full support for mainframe-based Cobol or PL/I Japanese-language applications. Make sure to set Windows system and user locales to Japanese before registering source files.

You can register Japanese source files downloaded in text or binary mode:

- Source files downloaded in text (workstation) mode must be in Shift-JIS encoding. If Shift-Out and Shift-In delimiters were replaced with spaces or removed during downloading, Modernization Workbench restores them at registration.
- Source files downloaded in binary (mainframe) mode are recoded by Modernization Workbench from EBCDIC to Shift-JIS encoding at registration.

Use the options on the Registration > Source Files tab of the Workspace Options window to specify how DBCS escape control characters were handled in source file downloaded in text mode:

- **Replaced with spaces** if DBCS escape control characters were replaced with spaces.
- **Removed** if DBCS escape control characters were removed.
- **Retained or not used** if DBCS escape control characters were left as is or were not used.

Preserving delimiters during download is recommended. Replacing delimiters with spaces during download generally yields better restoration results than removing them.

In all workbench tools that offer search and replace facilities, you can insert Shift-Out and Shift-In delimiters into patterns using Ctrl-Shift-O and Ctrl-Shift-I, respectively. You need only insert the delimiters if you are entering mixed strings.

Setting Up Projects

Workspace *projects* typically represent different portions of the application modeled in the workspace. You might have a project for the batch portion of the application and another project for the online portion. You can also use a project to collect items for discrete tasks: all the source files affected by a change request, for example.

Creating Projects

When you set up a workspace in Modernization Workbench, the system creates a default project with the same name as the workspace. Create projects in addition to the default project when you need to analyze subsystems separately or organize source files in more manageable groupings.

1. Choose **Project > New** project. The Create Project dialog opens.
2. Enter the name of the new project and click **OK**.
   The new project is displayed in the Repository Browser. The project is selected by default.

Sharing Projects

In a multiuser environment, the user who creates a project is referred to as its *owner*. Only the owner can *share* the project with other users.

A shared, or *public*, project is visible to other members of your team. A *private* project is not. If the project is not protected, these team members can delete the project, add source files, or remove source files.

Projects are private by default. Turn on sharing by choosing **Project > Toggle Sharing**. Choose **Project > Toggle Sharing** again to turn it off. A  symbol indicates that the project is shared.

Protecting Projects

By default, projects are *unprotected*: any user to whom the project is visible can delete the project, add source files, or remove source files.

The project owner or master user can designate a project as *protected*, in which case *no* user can delete or modify the project, including the project owner or master user: the project is read-only, until the project owner or master user turns protection off.

Turn on protection by selecting the project in the Repository pane and choosing **Project > Toggle Protection**. Choose **Project > Toggle Protection** again to turn it off. Look for a symbol like this one to indicate that a project is protected.

Moving or Copying Files into Projects

Copy the contents of a project, folder, or file to a different project by selecting it and dragging and dropping the selection onto the project, or by using the **Edit** menu choices to copy and paste the selection. Use the **Project** menu choices described below to move selections, or to include referenced or referencing objects in a move or copy.

Note: In other workbench tools, use the right-click menu or the **File** menu to include files into projects.

1. In the Repository Browser, select the project, folder, or file you want to move or copy, then choose **Project > Copy Project Contents** (if you selected a project) or **Project > Include into Project** (if you selected a folder or file). The Select Project window opens.
2. In the Select Project window, select the target project. Click **New** to create a new project.
3. Select:
   - **Include All Referenced Objects** if you want to include objects referenced by the selected object (the Cobol copybooks included in a Cobol program file, for example).
   - Select **Include All Referencing Objects** if you want to include objects that reference the selected object.

   Note: This feature is available only for verified files.
4. Select:
   - **Copy** to copy the selection to the target project.
   - **Move From Current Project** to move the selection to the target project.
   - **Move From All Projects** to move the selection from all projects to the target project.
5. Click **OK** to move or copy the selection.

Including Referenced and Referencing Objects in a Project

After verification, you can include referenced or referencing objects in a project to ensure a closed system. You can include all referencing objects or only “directly referencing” objects: If program A calls program B, and program B calls program C, A is said to directly reference B and indirectly reference C. You can also remove unused support objects.

To include in a project:

- Every object referenced by the objects in the project (including indirectly referenced objects), select the project in the Repository Browser and choose **Project > Include All Referenced Objects**.
- Every object that references the objects in the project (including indirectly referencing objects), select the project in the Repository Browser and choose **Project > Include All Referencing Objects**.
- Every object that directly references the objects in the project, select the project in the Repository Browser and choose **Project > Include Directly Referencing Objects**.

Removing Unused Support Objects from a Project

To move unused support objects (Cobol copybooks, JCL procedures, PL/I include files, and so forth) from a project to the workspace, select the project in the Repository Browser and choose **Project > Compact Project**.

Emptying a Project

To empty a project (without deleting the project or its contents from the workspace), select the project and choose **Project > Empty Project Contents**.

Deleting a Project

To delete a project from a workspace (without deleting its source files from the workspace), select it and choose either **File > Delete from Workspace** or **Project > Delete Project**.

Note: Only the owner of a project can delete it.

Verifying Source Files

Parsing, or *verifying*, an application source file generates the object model for the file. Only a master user can verify source files.

You can verify a single file, a group of files, all the files in a folder, or all the files in a project. It's usually best to verify an entire project. Modernization Workbench parses the files in appropriate order, taking account of likely dependencies between file types.

Note: You need not verify copybooks. Copybooks are parsed when the including source file is verified.

If your RPG or AS/400 Cobol application uses copy statements that reference Database Description or Device Description files, or if your MCP Cobol application uses copy statements that reference DMSII DASDL files, you need to generate copybooks for the application before you verify program files.

Workspace and Project Verification options determine verification behavior. The default values for these options are preset based on your configuration and should be appropriate for most installations.

1. In the Repository Browser, select the project, folder, or files you want to verify and choose **Prepare > Verify**.
2. You are prompted to drop repository indexes to improve verification performance. Click **Yes**. You will be prompted to restore the indexes when you analyze the files.

The parser builds an object model for each successfully verified file. For an unsuccessfully verified file, the parser builds an object model for as much of the file as it understands.

Enabling Parallel Verification

Parallel verification typically improves verification performance for very large workspaces by using multiple execution agents, called *HyperCode Converters*, to process source files concurrently. You can start any number of converters on the local machine, remote machines, or some combination of local and remote machines. You can run parallel verification online in the Modernization Workbench or in batch mode with the Batch Refresh Process (BRP).

Important: When you run parallel verification on more than one machine, you need to make sure that workspace and project verification options are set identically on each machine. The easiest way to do this is to log in as the same Windows user on each machine. Alternatively, you can define a default option set that is automatically assigned to every user in the environment who has not explicitly defined a custom option set. See the related topics for more information on option sets.

You enable parallel verification in three steps:

- Select the parallel verification method and the minimum number of concurrent converters on the Verification > Parallel Verification tab of the Workspace Options.
- Start the converters on the local and/or remote machines. If you start fewer than the minimum number of converters specified on the Parallel Verification tab, the verification process starts the needed converters automatically on the local machine.
- Verify the workspace online in the Modernization Workbench or in batch mode using the Batch Refresh Process (BRP).

Note: Verification results are reported in the Activity Log History window. They are not reported in the Activity Log itself (for online verification) or BRP log files (for batch verification). You can also use a Verification Report to view the results.

Follow the instructions below to launch HyperCode Converters and to specify the type of work the converters perform. You can launch multiple converters on the same machine. Once the minimum number of converters has been started, you can launch the converters at any point in the verification process.

1. In the Modernization Workbench Administration window, choose **Administer > Launch HyperCode Converter**. The Launch HyperCode Converter window opens.
2. In the **Serve workspace** combo box, specify the workspace to be processed.
3. In the Processing Mode pane, select any combination of:
   - **Conversion** to perform operations used to generate a HyperView construct model.
   - **Verification** to perform verification operations.
4. Select **Produce Log File** to generate a log file for parallel verification. The log file has a name of the form <*workspace_name*>HCC.<*random_number*>.log and is stored at the same level as the workspace (.rwp) file.
5. Click **OK**.

   The workbench launches the HyperCode Converter. Click the  button on the Windows toolbar to view the HyperCode Converter window.

   Note: Once verification has started, you can change the processing mode for a converter by selecting the appropriate choice in the **Processing** menu in the HyperCode Converter window.

How the System Refreshes the Repository

When you edit a source file in the Modernization Workbench, the system recursively checks every repository object that may be affected by the edit: *refreshes* the repository. If the edit *invalidates* the object, you need to reverify the source file that contains it. The file with the invalidated object is displayed in **bold** type in the Repository Browser.

Invalidating Objects Before Reverification

You can save time reverifying very large applications by invalidating some or all of the source files in them before you reverify. You can invalidate a single file, a group of files, all the files in a folder, or all the files in a project.

In the Repository Browser, select the project, folder, or files you want to invalidate and choose **File > Invalidate Selected Objects**. Invalidated files are displayed in **bold** type in the Repository Browser.

Setting Workspace Verification Options: Legacy Dialects Tab

Use the Verification > Legacy Dialects tab of the Workspace Options window to identify the dialect of the source files in your application.

1. Choose **Tools > Workspace Options**. The Workspace Options window opens. Click the Verification tab, then the Legacy Dialects tab.
2. In the Source Type pane, select the source file type whose dialect you want to specify, then select the dialect in the dialect pane.
3. Set verification options for the dialect. The table below shows the available options.

Option	Dialect	Description
48-character set	All PL/I	Specifies that the parser handle the 48-character set used with older hardware for logical operators.
Allow long copybook names	Cobol/390, Enterprise Cobol	Specifies that the parser allow references to copybooks with names longer than 8 characters. The reference is flagged as unresolved.  **Note:** By default, names longer than 8 characters are truncated in the parse tree.
Allow long program names	Cobol/390, Enterprise Cobol	Specifies that the parser allow references to programs with names longer than 8 characters. The reference is flagged as unresolved.  **Note:** By default, names longer than 8 characters are truncated in the parse tree.
ASCII Compatibility	Unisys 2200 UCS Cobol	Specifies that the parser ensure consistency with the ASCII version of Unisys Cobol. Emulates behavior of compiler COMPAT option.
Binary Storage Mode	ACUCOBOL-GT®, Micro Focus Cobol	Specifies the binary storage mode: Word (2, 4, or 8 bytes) or Byte (1 to 8 bytes).
COPY REPLACING Substitutes Partial Words	All Cobol	Specifies that the application was compiled with partial-word substitution enabled for COPY REPLACE operations.
COPY statements as in COBOL-68	All Cobol	Specifies that the application was compiled with the OLDCOPY option set.
Currency Sign	All Cobol, all PL/I	Specifies the national language currency symbol.
Data File Assignment	Micro Focus Cobol	Specifies the setting of the compiler ASSIGN option: Dynamic or External.
Enable MF comments	Micro Focus Cobol	Specifies that the application contains comments in the first position.
Extralingual Characters	All PL/I	Add the specified lower-case national language characters to the supported character set. Do not separate characters with a space.
Extralingual Upper Characters	All PL/I	Add the specified upper-case national language characters to the supported character set. Do not separate characters with a space.
Graphical System	ACUCOBOL-GT®	Specifies that the application was executed on a graphical rather than character-based system.
In Margins	All	Specifies the current margins for source files.
Line Number Step	All Natural	Specifies the line-numbering increment to use in restoring stripped line numbers in Natural source files: Autodetect, to use a line-numbering

Option	Dialect	Description
		increment based on line number references in the source code, or User defined, to use the line-numbering increment you specify. If you select User defined, enter the increment in the Value field.
Logical Operators	All PL/I	Specifies handling of logical operator characters used in source files: Autodetect, to autodetect logical operator characters, or Characters, to use the logical operator characters you specify. If you select Characters, specify the characters used for NOT and OR operations.
Out Margins	All PL/I	Specifies the margins for components to be created with the Modernization Workbench Component Maker tool.
PERFORM behavior	ACUCOBOL-GT®, Micro Focus Cobol	Specifies the setting of the compiler PERFORM-type option: Stack, to allow recursive PERFORMS, or All exits active, to disallow them.
Picture clause N-symbol	Cobol/390, Enterprise Cobol	Specifies the national language behavior of picture symbol N and N-literals: DBCS or National. Emulates behavior of compiler NSYMBOL option.
Preserve dialect for verified objects	All Cobol	Specifies that the parser reverify Cobol files with the same dialect it used when the files were verified first.
RM/Cobol compatibility	ACUCOBOL-GT®	Specifies that the parser ensure proper memory allocation for applications written for Liant RM/COBOL. Emulates behavior of -Ds compatibility option.
Support Hogan Framework	Cobol/390, Enterprise Cobol	Specifies that the parser create relationships for Hogan Cobol programs. Enter the location of Hogan Cobol configuration files in the **Hogan Files Location** field.
Treat COMP-1/COMP-2 as FLOAT/DOUBLE	ACUCOBOL-GT®	Specifies that the parser treat picture data types with COMP-1 or COMP-2 attributes as FLOAT or DOUBLE, respectively.
Unisys MCP Control Options	Unisys MCP Cobol-74, Unisys MCP Cobol-85	Specifies that the application was compiled with control options set or reset as specified. Add control options as necessary.

Setting Workspace Verification Options: Settings Tab

Use the Verification > Settings tab of the Workspace Options window to specify verification behavior for the workspace. Among other tasks, you can:

- Enable *staged parsing*, which may improve verification performance by letting you control which verification stages the parser performs.
- Enable *relaxed parsing*, which lets you verify source despite errors.

- Enable advanced data flow analysis for control language files.
- Enable sort card analysis.

1. Choose **Tools > Workspace Options**. The Workspace Options window opens. Click the Verification tab, then the Settings tab.
2. In the Source Type pane, select the source file type whose verification options you want to specify.
3. Set verification options for the source file type. The table below shows the available options.

Note: Click the **More** or **Details** button if an option listed below does not appear on the Settings tab.

Option	Source File	Description
Allow Implicit Instream Data	JCL	Specifies that a DD * statement be inserted before implicit instream data if the statement was omitted from JCL.
Allow Keywords to Be Used as Identifiers	Cobol, Copybook	Enables the parser to recognize Cobol keywords used as identifiers.
At Sign	System Definition File	Specifies the national language character for the at symbol.
C/C++ Parser Parameters	C, C++	Specifies the parameters used to compile the application. You can also specify these parameters in the Project Verification options, in which case the project parameters are used for verification.
Create Alternative Entry Point	Cobol	Specifies that an additional entry point be created with a name based on the conversion pattern you enter in the Conversion Pattern field. Supports systems in which load module names differ from program IDs. For assistance, contact support services.
Cross-reference Report	TWS Schedule	Specifies the full path of the TWS cross-reference report (.xrf).
Currency Sign	System Definition File	Specifies the national language character for the currency symbol.
Debugging Lines	Cobol	Specifies parsing of debugging lines: Off, to parse lines as comments, On, to parse lines as normal statements, Auto, to parse lines based on the program debugging mode.
Detect Potential Code Anomalies	Cobol	Enables generation of HyperView information on potential code anomalies.
Enable extended ASCII characters in SQL identifiers	Cobol, PL/I	Enables the parser to recognize extended ASCII characters in SQL identifiers. The extended ASCII characters set includes national language characters not included in the basic ASCII character set and special symbols used for drawing pictures.
Enable HyperView	Cobol, Natural, PL/I, RPG	Enables generation of HyperView information.

Option	Source File	Description
Enable Quoted SQL Identifiers	Assembler File, Cobol, DDL	Enables the parser to recognize quoted SQL identifiers. Strings delimited by double quotation marks are interpreted as object identifiers.
Enable Reference Reports	Cobol, Control Language, ECL, JCL, Natural, PL/I, RPG, W.L.	Enables generation of complete repository information for logical objects.
Enter classpath to JAR Files and/or path to external Java file root directories	Java	Specifies any combination of: • JAR files containing class libraries referenced in the Java application, or Zip files containing external Java files referenced in the application. Alternatively, specify the path of the folder for the JAR or Zip files, then select **Include Jar/Zip Files From Directories**. • Paths of the root folders for external Java files referenced in the application.  **Note:** This option is checked after the identical option in the project verification options. Setting the project verification option effectively overrides the setting here.
Extralingual Characters	System Definition File	Adds the specified lower-case national language characters to the supported character set. Do not separate characters with a space.
Extralingual Upper Characters	System Definition File	Adds the specified upper-case national language characters to the supported character set. Do not separate characters with a space.
Generate program entry points for functions with same name as file	C	Specifies that a program entry point be created for the function that has the same name as the file. Typically used to trace calls to C programs from Cobol, PL/I, Natural, RPG, or Assembler programs.
Ignore Duplicate Entry Points	All	Enables the parser to recognize duplicate entry points defined by the Cobol statement ENTRY ‘PROG-ID’ USING A, or its equivalent in other languages. The parser creates an entry point object for the first program in which the entry point was encountered and issues a warning for the second program. To use this option, you must select Enable Reference Reports. You cannot use this option to verify multiple programs with the same program ID.
Ignore Text After Column 72	DDL	Allows the parser to ignore trailing enumeration characters (columns 73 through 80) in source lines.
Libraries	PowerBuilder	Specifies the PowerBuilder libraries used by the application. Libraries must be listed in the order they appear in the PBL File folder in the Repository Browser. Add libraries as necessary.

Option	Source File	Description
Libraries support	Natural	Enables Natural library support. For more information, see "Natural Support" in the online help.
List of Include Directories	C, C++	Specifies the full path of the folders for include files (either original folders or Repository Browser folders if the include files were registered). Choose a recognized folder in the List of Include Directories pane. Add folders as necessary. You can also specify these folders in the Project Verification options, in which case the parser looks only for the folders for the project.
Number Sign	System Definition File	Specifies the national language character for the number symbol.
Perform Dead Code Analysis	Cobol, PL/I, RPG	Enables collection of dead code statistics.
Perform DSN Calling Chains Analysis	Control Language, ECL, JCL, WFL	Enables analysis of dataset calling chains.
Perform System Calls Analysis	JCL	Enables analysis of system program input data to determine the application program started in a job step.
Relaxed Parsing	AS400 Screen, BMS, Cobol, Copybook, CSD, DDL, Device Description, DPS, ECL, MFS, Natural, Netron Specification Frame, PL/I	Enables relaxed parsing.
Relaxed Parsing for Embedded Statements	Assembler File, Cobol, PL/I	Enables relaxed parsing for embedded SQL, CICS, or DLI statements.
Resolve Decisions Automatically	Control Language, WFL	Enables automatic decision resolution.
Show Macro Generation	C, C++	Specifies whether to display statements that derive from macro processing in HyperView.
Sort Program Aliases	JCL	Enables batch sort card analysis. Choose a recognized sort utility in the Sort Program Aliases pane. Add sort utilities as necessary.
SQL Statements Processor	Cobol	Specifies whether the SQL Preprocessor or Coprocessor was used to process embedded SQL statements.
System Procedures	JCL	Specifies the system procedures referenced by JCL files. Add system procedures as necessary.
Timeout in seconds to stop verification execution	All	The number of seconds to wait before stopping a stalled verification process
Treat every file with main procedure as a program	C, C++	Specifies whether to treat only files with main functions as programs.
Trim Date from Active Schedule Names	TWS Schedule	Specifies whether to append the effective date range to a TWS jobstream object.

Option	Source File	Description
Truncate Names of Absolute Elements	ECL	Allows the parser to truncate suffixes in the names of Cobol programs called by ECL. Specify a suffix in the adjoining text box.
Use Database Schema	Assembler File, Cobol, PL/I	Specifies whether to associate a program with a database schema. When this option is selected, the parser collects detailed information about SQL ports that cannot be determined from program text (SELECT *). If the schema does not contain the items the SQL statement refers to, an error is generated.
Workstation Report	TWS Schedule	Specifies the full path of the TWS workstation report (.wdr).

Enabling Staged Parsing

File verification generates repository information in four stages, as described in this section. You can control which stage the workbench parser performs by setting the staged parsing options on the Settings tab for Workspace Verification options. That may save you time verifying very large applications.

Rather than verify the application completely, you can verify it one or two stages at a time, generating only as much information as you need at each point. When you are ready to work with a full repository, you can perform the entire verification at once, repeating the stages you've already performed and adding the stages you haven't.

Basic Repository Information

To generate basic repository information only, deselect **Enable HyperView**, **Enable Reference Reports**, and **Perform Dead Code Analysis** on the Workspace Verification options Settings tab. The parser:

- Generates relationships between source files (Cobol program files and copybooks, for example).
- Generates basic logical objects (programs and jobs, for example, but not entry points or screens).
- Generates Defines relationships between source files and logical objects.
- Calculates program complexity.
- Identifies missing support files (Cobol copybooks, JCL procedures, PL/I include files, and so forth).

Note: If you generate only basic repository information when you verify an application, advanced program analysis information is not collected, regardless of your settings in the Project Options Verification tab.

Full Logical Objects Information

To generate complete repository information for logical objects, select **Enable Reference Reports** on the Workspace Verification options Settings tab. Set this option to generate all relationships between logical objects, and to enable non-HyperView analysis tools, including Reference Reports and Orphan Analysis.

Note: If you select this staged parsing option only, verify all legacy objects in the workspace synchronously to ensure complete repository information.

HyperView Information

To generate a HyperView construct model, select **Enable HyperView** on the Workspace Verification options Settings tab. A HyperView construct model defines the relationships between the constructs that comprise the file being verified: its sections, paragraphs, statements, conditions, variables, and so forth.

To generate HyperView information on potential code anomalies, select **Detect Potential Code Anomalies** ion the Workspace Verification options Settings tab.

Note: If you do not generate HyperView information when you verify an application, impact analysis, data flow, and execution flow information is not collected, regardless of your settings on the Project Verification options tab.

Dead Code Statistics

To generate dead code statistics, and to set the Dead attribute to True for dead constructs in HyperView, select **Perform Dead Code Analysis** on the Workspace Verification options Settings tab. The statistics comprise:

- Number of dead statements in the source file and referenced copybooks. A dead statement is a procedural statement that can never be reached during program execution.
- Number of dead data elements in the source file and referenced copybooks. Dead data elements are unused structures at any data level, all of whose parents and children are unused.
- Number of dead lines in the source file and referenced copybooks. Dead lines are source lines containing dead statements or dead data elements.

You can view the statistics in the Statistic tab of the Properties window for an object or in the Complexity Metrics tool.

Enabling Relaxed Parsing

The *relaxed parsing* option lets you verify a source file despite errors. Ordinarily, the parser stops at a statement when it encounters an error. Relaxed parsing tells the parser to continue to the next statement.

Use relaxed parsing when you are performing less rigorous analyses that do not need every statement to be modeled (estimating the complexity of an application written in an unsupported dialect, for example). Select **Relaxed Parsing** or **Relaxed Parsing for Embedded Statements** as appropriate on the Workspace Verification options Settings tab.

Note: Relaxed parsing may affect the behavior of other tools. You cannot generate component code, for example, from source files verified with the relaxed parsing option.

Enabling Advanced Data Flow Analysis for Control Language Files

Ordinarily, Modernization Workbench data flow analysis tools let you trace the flow of data into or out of a dataset only up to the program actually referenced in the control language file, whether or not that program writes to or reads from the dataset. If you need to trace the flow of data through the entire “calling chain,” that is, not only the referenced program, but also any programs that program calls, and any programs they call in turn:

- Select **Perform DSN Calling Chains Analysis** on the Workspace Verification options Settings tab for the control language file.
- Verify control language files *after* you verify the source files for the programs they use. If you reverify the source file for a program, you must also reverify the control language file that uses it.

If you verify an entire project, the workbench parses the files in appropriate order, taking account of the dependencies between control language and program files.

Enabling Sort Card Analysis

If you use sort utilities in JCL files, you can enable sort card analysis by specifying the names of the sort utilities to the parser in the Sort Program Aliases pane on the Workspace Verification options Settings tab. The parser creates an artificial program entity that defines the inputs and outputs for each sort utility invocation. The program has a name of the form *JCLFileName.JobName.StepName.SequenceNumber*, where *SequenceNumber* identifies the order of the step in the job.

Truncating Names of Absolute Elements

If you are verifying ECL files for an application in which absolute element names differ from program IDs, you can tell the parser to truncate suffixes in the names of Cobol programs called by ECL. Select **Truncate Names of Absolute Elements** on the Workspace Verification options Settings tab for the ECL file.

If a Cobol program named CAP13MS.cob, for example, defines the entry point CAP13M, and an ECL program named CAP13M.ecl executes an absolute element called CAP13MA, then setting this option causes the parser to create a reference to the entry point CAP13M rather than CAP13MA.

Setting Workspace Verification Options: Parallel Verification Tab

Use the Verification > Parallel Verification tab of the Workspace Options window to enable online or batch parallel verification and to specify the minimum number of HyperCode Converters the workbench should expect.

Important: When you run parallel verification on more than one machine, you need to make sure that workspace and project verification options are set identically on each machine. The easiest way to do this is to log in as the same Windows user on each machine. Alternatively, you can define a default option set that is automatically assigned to every user in the environment who has not explicitly defined a custom option set. See the related topics for more information on option sets.

1. Choose **Tools > Workspace Options**. The Workspace Options window opens. Click the Verification tab, then the Parallel Verification tab.
2. Select:
   - **Run Parallel Verification in the Online Tool** to enable parallel verification online. In the **Minimum HyperCode Converters** combo box, specify the minimum number of concurrent HyperCode Converters the workbench should expect.
   - **Run Parallel Verification in BRP** to enable parallel verification in the Batch Refresh Process (BRP) tool. In the **Minimum HyperCode Converters** combo box, specify the minimum number of concurrent HyperCode Converters the workbench should expect.

Note: Before running verification, start the necessary HyperCode Converters on the local and/or remote machines. If you start fewer than the minimum number of converters, the verification process starts the needed converters automatically on the local machine.

Setting Project Verification Options

Use the Verification tab of the Project Options window to specify verification behavior for the selected project. Among other tasks, you can:

- Specify how the parser treats environment-related code.
- Specify the conditional constants the parser uses to compile programs in the project.
- Specify schedule IDs for CA-7 jobs triggered by datasets.
- Optimize verification for advanced program analysis.

1. Choose **Tools > Project Options**. The Project Options window opens. Click the Verification tab.
2. In the Source Type pane, select the source file type whose verification options you want to specify.
3. Set verification options for the source file type. The table below shows the available options.

Note: Click the **Environments**, **CopyLibs**, **Advanced**, or **Cobol Dialect** button if an option listed below does not appear on the Verification tab.

Option	Source File	Description
AIM/DB Environment	Cobol	Specifies how the parser treats AIM/DB-environment-related code or its absence.
C/C++ Parser Parameters	C, C++	Specifies the parameters used to compile the application.
CAP Processor predefined variables	Netron Specification Frame	Specifies the values of predefined environment variables the parser uses to compile Netron Specification Frames. Select each predefined variable you want the parser to recognize, then enter its value. Add variables as necessary.
CICS Environment	Assembler, Cobol, PL/I	Specifies how the parser treats CICS-environment-related code or its absence.
Context-Sensitive Value Analysis	Cobol	Specifies that the parser perform context-sensitive automatic decision resolution for Unisys MCP COMS analysis. Choosing this option may degrade verification performance.
Dialect Specific Options	Cobol	Specifies dialect-specific options, including the conditional constants the parser uses to compile programs in the project. Select the Cobol dialect, then choose the constant in the Macro Settings pane. Add constants as necessary.
DMS Environment	Cobol	Specifies how the parser treats DMS-environment-related code or its absence.
DMSII Environment	Cobol	Specifies how the parser treats DMSII-environment-related code or its absence.

Option	Source File	Description
DPS routines may end with error	Cobol	Specifies that the parser perform call analysis of Unisys 2200 DPS routines that end in an error. Error-handling code for these routines is either analyzed or treated as dead code.
Enable Data Element Flow	Cobol, Natural, PL/I, RPG	Enables the Global Data Flow, Change Analyzer, and impact trace tools.
Enable Execution Flow	Cobol, PL/I	Enables the Execution Path tool.
Enable Extraction of Computation-Based Components	Cobol	Enables computation-based componentization with Application Architect.
Enable Impact Report	Cobol, Natural, PL/I, RPG	Enables the impact trace tools. You must also set Enable Data Element Flow to perform impact analysis.
Enable Parameterization of Components	Cobol	Enables parameterized structure- and computation-based componentization with Application Architect.
Enter classpath to JAR Files and/or path to external Java file root directories	Java	Specifies any combination of: • JAR files containing class libraries referenced in the Java application, or Zip files containing external Java files referenced in the application. Alternatively, specify the path of the folder for the JAR or Zip files, then select **Include Jar/Zip Files From Directories**. • Paths of the root folders for external Java files referenced in the application.  **Note:** This option is checked before the identical option in the workspace verification options. Setting it here effectively overrides the setting in the workspace verification options.
Helproutines	Natural Map	Specifies how you want the parser to treat helproutines, as programs or helpmaps.
IDMS Environment	Cobol	Specifies how the parser treats IDMS-environment-related code or its absence.
IMS Environment	Cobol	Specifies how the parser treats IMS-environment-related code or its absence.
Java Classpath	JSP	For JSP files that reference Java types or packages defined in JAR files, external Java files, or Java files registered with the **Preserve Folder Structure** option, specifies a list of patterns used to resolve the location of the files. For more information, see "Resolving the Location of Java Types and Packages Referenced in JSP Files" in the "JSP Support" section of the help.
Java Source Level	Java	Specifies the Java version. Set this option if your application uses constructs that are incompatible with the latest Java version. You would set this option to 1.4, for example, if your application used

Option	Source File	Description
		"enum" as an identifier, since "enum" is a reserved word in Java 5.
Job to Schedule Ids File	CA-7 Schedule	Specifies the full path of a file that supplies schedule IDs for CA-7 jobs triggered by the creation of a dataset.
Libraries	Natural	When Libraries support is selected in the Verification > Settings tab of the Workspace Options window for Natural files, specifies the order in which the parser searches for library names when it resolves program calls. Place a check mark next to a library to include it in the search. The parser always searches for the current library first. For more information, see "Natural Support" in the online help.
List of Include Directories	C, C++	Specifies the full path of folders for include files (either original folders or Repository Browser folders if the include files were registered). Choose a recognized folder in the List of Include Directories pane. Add folders as necessary.
List of Include Folders	Cobol	For applications using copybooks registered with the **Preserve Folder Structure** option, specifies the full path of folders for copybooks. Choose a recognized folder in the List of Include Folders pane. Add folders as necessary.  **Note:** Use this option to verify applications with identically named copybooks in different Partitioned Data Sets (PDS) on the mainframe. The order of folders in the list must correspond to the order of PDS files in the mainframe compilation job.
Maximum Number of Variable's Values	Cobol	Specifies the maximum number of values to be calculated for each variable during verification for advanced program analysis. Limit is 200.
Maximum Size of Variable to Be Calculated	Cobol	Specifies the maximum size in bytes for each variable value to be calculated during verification for advanced program analysis.
Override CICS Program Terminations	Cobol, PL/I	Specifies that the parser interpret CICS RETURN, XCTL, and ABEND commands as not terminating program execution. Error-handling code after these statements is either analyzed or treated as dead code.
Perform COMS Analysis	Cobol	Specifies that the parser define relationships for Unisys MCP COMS SEND statements.
Perform Generic API Analysis	Cobol, PL/I	Specifies that the parser define relationships with objects passed as parameters in calls to unsupported program interfaces, in addition to relationships with the called programs themselves.
Perform Program Analysis	Cobol	Enables program analysis and component extraction features.

Option	Source File	Description
Perform Unisys Common-Storage Analysis	Cobol	Specifies that the parser include in the analysis for Unisys Cobol files variables that are not explicitly declared in CALL statements, but that participate in interprogram communications. You must set this option to include Unisys Cobol common storage variables in impact traces and global data flow diagrams.
Perform Unisys TIP and DPS Calls Analysis	Cobol	Specifies that the parser perform Unisys 2200 TIP and DPS call analysis.
Report Writer Environment	Cobol	Specifies how the parser treats Report Writer-environment-related code or its absence.
Resolve Decisions Automatically	Cobol, Natural, PL/I, RPG	Specifies that the parser autoresolve decisions after successfully verifying files.
SQL Environment	Assembler, Cobol, RPG	Specifies how the parser treats SQL-environment-related code or its absence.
Support CICS HANDLE statements	Cobol	Specifies that the parser detect dependencies between CICS statements and related error-handling statements,
Use overwritten VALUEs	Cobol	Specifies that the parser use constants from VALUE clauses as known values even if they are overwritten in the program by unknown values.
Use Precompiled Header File	C, C++	Specifies that the parser verify the project with the precompiled header file you enter in the adjacent field. Do not specify the file extension. Using a precompiled header file may improve verification performance significantly. The content of the header file must appear in both a .c or .cpp file and a .h file. The precompiled header file need not have been used to compile the application.
Use VALUEs from Linkage Section	Cobol	Specifies that advanced analysis tools not ignore parameter values in the Linkage Section.

Specifying the Processing Environment

The Modernization Workbench parser autodetects the environment in which a file is intended to execute, based on the environment-related code it finds in the file. To ensure correct data flow, it sets up the internal parse tree for the file in a way that emulates the environment on the mainframe.

For Cobol CICS, for example, the parser treats an EXEC CICS statement or DFHCOMMAREA variable as CICS-related and, if necessary:

- Adds the standard CICS copybook DFHEIB to the workspace.
- Declares DFHCOMMAREA in the internal parse tree.
- Adds the phrase Procedure Division using DFHEIBLK, DFHCOMMAREA to the internal parse tree.

Autodetection is not always appropriate, of course. You may want the parser to treat a file as a transaction-processing program even in the absence of CICS- or IMS-related code, for example. For each autodetected environment on the Project Verification options tab, select:

- Auto, if you want the parser to autodetect the environment for the file.
- Yes, if you want to force the parser to treat the file as environment-related even in the absence of environment-related code.
- No, if you want to force the parser to treat the file as unrelated to the environment even in the presence of environment-related code. The parser classifies environment-related code as a syntax error.

Specifying Conditional Compiler Constants

Compiler constant directives let you compile programs conditionally. Specify the conditional constants the parser uses to compile programs in the project in the **Dialect Specific Options** for your dialect on the Project Verification options tab. For Micro Focus Cobol, two formats are supported:

- *constant_name=value* (where no space is allowed around the equals sign). In the following example, if you specify WHERE=PC on the Project Verification options tab, the source that follows the $if clause is compiled:

```
$if WHERE = "PC"
           evaluate test-field
               when 5 perform test-a
           end-evaluate
```

- *constant_name*. In the following example, if you specify NOMF on the Project Verification options tab, the source that follows the $if clause is compiled:

```
$if NOMF set
           $display Not MF dialect
               go to test-a test-b depending on test-field
           $end
```

Optimizing Verification for Advanced Program Analysis

When you enable advanced program analysis options for Cobol projects, the parser calculates constant values for variables at every node in the HyperView parse tree. That's one reason why very large Cobol applications may encounter performance or memory problems during verification.

You may be able to improve verification performance and avoid out-of-memory problems by manipulating the Maximum Number of Variable's Values and Maximum Size of Variable to Be Calculated options in the Project Verification options tab. The lower the maximums, the better performance and memory usage you can expect.

For each setting, you are warned during verification about variables for which the specified maximum is exceeded. It's usually best to increase the overflowed maximum and reverify the application.

Identifying System Programs

A *system program* is a generic program provided by the underlying operating system and used in unmodified form in the legacy application: a mainframe sort utility, for example. You need to identify system programs to the parser so that it can distinguish them from application programs and create relationships for them with their referencing files.

The most convenient way to identify the system programs your application uses is to run an unresolved report after verification. Once you learn from the report which system programs are referenced, you can identify them in the System Programs tab of the Workspace Options window and reverify any one of their referencing source files.

Note: The reference report tool lets you bring up the System Programs tab of the Workspace Options window while you are in the tool itself. Choose **View > System Programs** in the reference report window to display the tab.

1. Choose **Tools > Workspace Options**. The Workspace Options window opens. Click the System Programs tab.
2. In the System Program Patterns pane, select the patterns that match the names of the system programs your application uses. Add patterns as necessary.

Specifying Boundary Decisions

Specify a *boundary decision* object if your application uses a method call to interface with a database, message queue, or other resource. Suppose the function **`f1f()`** in the following example writes to a queue named **`abc`**:

```
int f1f(char*)
{
    return 0;
}
int f2f()
{
    return f1f(“abc”);
}
```

As far as the parser is concerned, **`f1f(“abc”)`** is a method call like any other method call. There is no indication from the code that the called function is writing to a queue.

When you specify the boundary decisions for a workspace, you tell the parser to create a decision object of a given resource type for each such call. Here is the decision object for the write to the queue:

```
int f2f().InsertsQueue.int f1f(char*)
```

You can resolve the decision objects to the appropriate resources in the Decision Resolution tool.

1. Choose **Tools > Workspace Options**. The Workspace Options window opens. Click the Boundary Decisions tab.
2. In the Decision Types pane, select the decision types associated with called procedures in your application. For the example, you would select the Queue decision type.
3. In the righthand pane, select each signature of a given method type you want to associate with the selected decision type. For the example, the method type signature would be **`int f1f(char*)`**. Add signatures as necessary. Do not insert a space between the parentheses in the signature. You can use wildcard patterns allowed in LIKE statements by Visual Basic for Applications (VBA).

   Note: Keep in mind that the signatures of C or C++ functions can contain an asterisk (*) character, as in the example. So if you specify a signature with a * character, you may receive results containing not only the intended signatures but all signatures matching the wildcard pattern. Delete the unwanted decision objects manually.
4. Select the project, folder, or source files for the application and choose:

   - **Prepare > Verify** if the source files have not been verified.
   - **Prepare > Apply Boundary Decisions** if the source files have been verified, but boundary decisions have not been specified for the application, or the specification has changed.

   A decision object is added to the tree for the source files in the Repository Browser.

Generating Copybooks

RPG programs and Cobol programs that execute in the AS/400 environment often use copy statements that reference Database Description or Device Description files rather than copybooks. MCP Cobol programs occasionally use copy statements that reference DMSII DASDL files. If your application uses copy statements to reference these types of files, you need to verify the files and generate copybooks for the application before you verify program files.

Copybook generation takes place in two steps:

- For each database and device file object generated at verification, the system creates a *target copybook* object.
- For each target copybook object, the system creates one or more *physical copybooks*.

Settings on the Generate Copybooks tab of the Project Options window determine conversion behavior. The default values should be appropriate for most installations.

1. Verify:
   - Database Description files. For each Database Description file, the system creates a *database file* object with the same name as the Database Description file.
   - Device Description files. For each Device Description file, the system creates a *device file* object with the same name as the Device Description file.
   - DMSII DASDL files. For each DMSII DASDL file, the system creates a *DMSII database file* object with the same name as the DMSII DASDL file.
2. In the Repository Browser, select the project, database file, or device file and choose:
   - **Prepare > Generate RPG Copybooks for Project** for RPG.
   - **Prepare > Generate Copybooks for Project** for AS/400 Cobol or MCP Cobol.

   Note: Skip this step for MCP Cobol if you selected **Generate After Successful Verification** on the Generate Copybooks tab of the Project options window.

   The system creates a target copybook for each database or device file object, with a name of the form database file.DBCOPYBOOK or device file.DVCOPYBOOK, in the Target Copybooks folder.
3. The system automatically converts the target copybooks to physical copybooks if you selected **Convert Target Copybooks to Legacy Objects** on the Generate Copybooks tab of the Project options window. If you chose not to convert target copybooks automatically, select the target copybooks and choose **Prepare > Convert to Legacy**. The system generates physical copybooks with names of the form DD_OF_database file.CPY or DV_OF_device file.CPY,

   Note: After generating copybooks, you can generate screens for AS/400 Cobol device file objects by selecting the objects in the Repository Browser and choosing **Prepare > Generate Screens**.

Setting Generate Copybooks Options

For RPG or AS/400 Cobol application that use copy statements to reference Database Description or Device Description files, or MCP Cobol applications that use copy statement to reference DMSII DASDL files, use the Generate Copybooks tab of the Project Options window to specify how the system converts the files to physical copybooks.

1. Choose **Tools > Project Options**. The Project Options window opens. Click the Generate Copybooks tab.
2. For Cobol MCP only, select **Generate After Successful Verification** if you want to generate target copybooks automatically on verification of DMSII DASDL files.
3. Select **Convert Target Copybooks to Legacy Objects** if you want to convert target copybooks automatically to physical copybooks when they are generated. then select:
   - **Assign Converted Files to the Current Project** if you want the system to create physical copybooks in the current project.
   - **Keep Old Legacy Object**s if you want the system not to overwrite existing physical copybooks, **Replace Old Legacy Objects** if you want the system to overwrite existing physical copybooks.
   - **Remove Target Copybooks After Successful Conversion** if you want the system to remove target copybooks from the current project after physical copybooks are generated.

Copybook Generation Order

It's usually best to generate copybooks for an entire project, because the system processes objects in the appropriate order, taking account of the dependencies between them. That is not the case when you generate copybooks for given objects. Follow these rules to ensure correct results:

- Copybooks for database file objects must be generated before copybooks for device file objects.
- Copybooks for referenced objects must be generated before copybooks for referencing objects.

Performing Post-Verification Program Analysis

Much of the performance cost of program verification for Cobol projects is incurred by the advanced program analysis options in the Project Verification Options window. These features enable impact analysis, data flow analysis, and similar tasks.

You can improve verification performance by postponing some or all of advanced program analysis until after verification. As long as you have verified source files with the **Enable Reference Reports** and **Enable HyperView** workspace verification options, you can use the post-verification program analysis feature to collect the remaining program analysis information without having to reverify your entire legacy program.

To perform post-verification program analysis, select the project verification options for each program analysis feature you want to enable. In the Repository Browser, select the programs you want to analyze (or the entire project) and choose **Prepare > Analyze Program**.

The system collects the required information for each analysis feature you select. And it does so incrementally: if you verify a Cobol source file with the **Enable Data Element Flow** option selected, and then perform post-verification analysis with both that option and the **Enable Impact Analysis** option selected, only impact analysis information will be collected.

The same is true for information collected in a previous post-verification analysis. In fact, if all advanced analysis information has been collected for a program, the post-verification analysis feature simply will not start. In that case, you can only generate the analysis information again by reverifying the program.

Restrictions on Cobol Post-Verification Program Analysis

With the exception of **Enable Impact Report** and **Enable Execution Flow**, you should select *all* of the **Perform Program Analysis** options you are going to need for Cobol program analysis the *first* time you

collect analysis information, whether during verification or subsequent post-verification analysis.This is because, with the exception of **Enable Impact Report** and **Enable Execution Flow**, selecting *any* of the options dependent on the **Perform Program Analysis** project verification option, whether during a previous verification or a previous program analysis, results in *none* of the information for those options being collected in a subsequent post-verification program analysis.

So if you verify a program with the **Resolve Decisions Automatically** option selected, then perform a subsequent program analysis with the **Perform Generic API Analysis** option selected, API analysis information is not collected. Whereas if you perform the subsequent program analysis with the **Enable Impact Report** option selected, impact analysis information is collected.

Similarly, if you perform program analysis with the **Enable Impact Report** option selected, then perform a subsequent program analysis with the **Enable Parameterization of Components** option selected, no parameterization information is collected. Whereas if you perform the subsequent program analysis with the **Enable Execution Flow** option selected, execution flow information is collected.

Restrictions on PL/I Post-Verification Program Analysis

For PL/I programs, selecting **Resolve Decisions Automatically** causes information for **Enable Data Element Flow** also to be collected, whether or not it already has been collected. Select these options together when you perform program analysis.

Using Post-Verification Reports

Use Modernization Workbench post-verification reports to check verification results, perform detailed executive assessments of the application, and view key data operations:

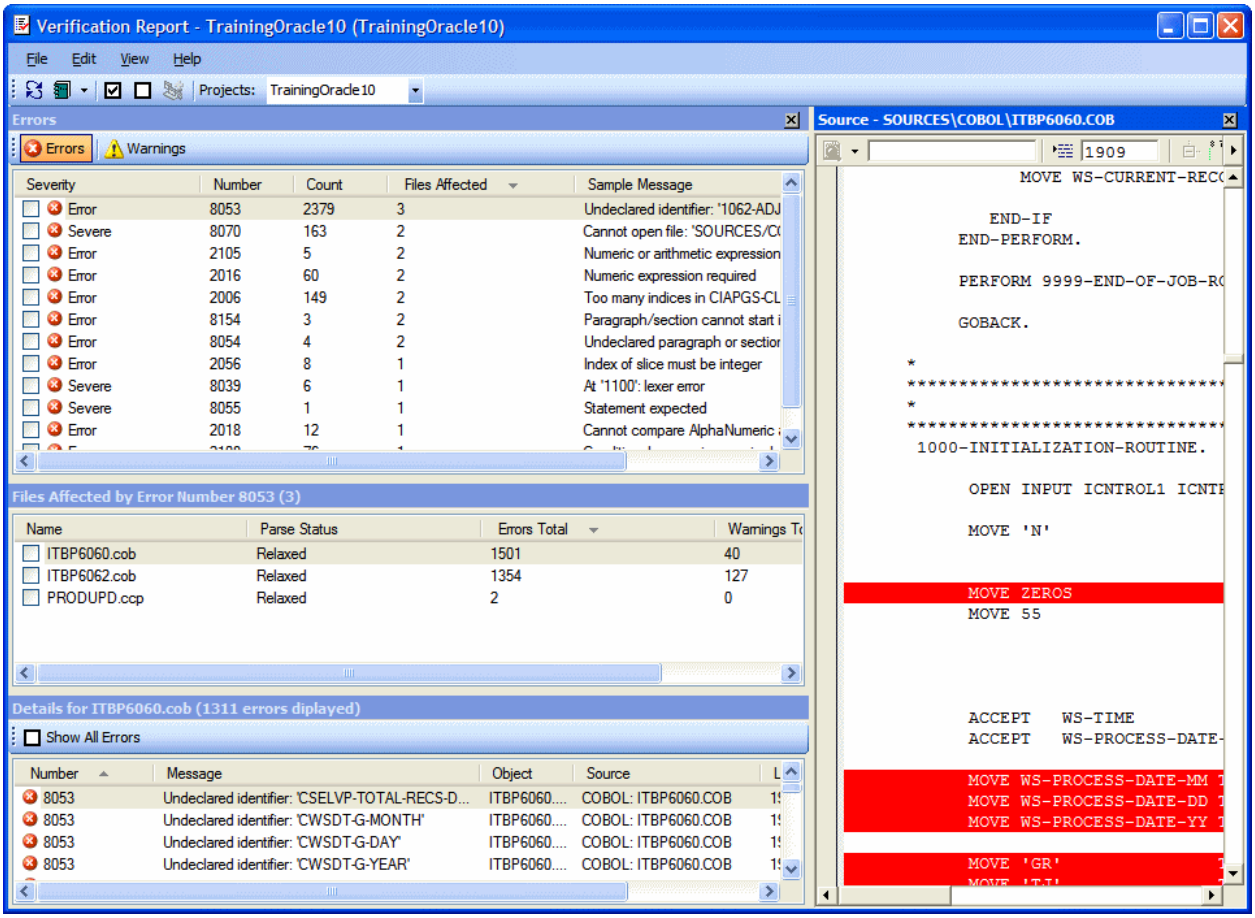
- Verification Reports offer a convenient way to analyze project verification results.
- Executive Reports offer HTML views of application inventories that a manager can use to assess the risks and costs of supporting the application.
- CRUD Reports show the data operations each program in the project performs, and the data objects on which the programs operate.

Viewing Verification Reports

Use Verification Reports to display the verification errors and warnings for each source file in the selected project and to navigate to the offending code in source. To open the Verification Report window, select a project in the Repository Browser and choose **Prepare > Verification Report**. The verification report is displayed in the Verification Report window.

To show the verification report for a different project, select the project in the **Projects** drop-down on the toolbar. To refresh the report after reverifying a file, choose **File > Generate Report**.

The figure below shows the Verification Report window. By default, the Report and Source windows are displayed. Select the appropriate choice in the **View** menu to hide a window. Select the choice again to show the window.



Errors Pane

The Errors pane of the Verification Report window displays the errors and warning for the project, sorted by the Files Affected column. Click the Errors button to display the errors for the project. Click the Warnings button to display the warnings for the project. The buttons are toggles. Click the buttons again to hide errors or warnings.

Click an error or warning to display the affected files in the Files Affected pane and to highlight each instance of offending code in the Source pane. Mark an error or warning to mark the affected files in the Files Affected pane. The table below describes the columns in the Errors pane.

Column	Description
Severity	The severity of the error or warning.
Number	The error or warning number.
Count	The number of occurrences of errors or warnings of this type in the project.
Files Affected	The number of files affected by the error or warning.

Column	Description
Sample Text	The text of the error or warning message. If there are multiple occurrences of the error or warning, and the text of the message differs among occurrences, then the text of a sample occurrence.

Files Affected Pane

The top portion of the Files Affected pane displays the files affected by the error or warning selected in the Errors pane, the verification status of the file, and the numbers of occurrences of the error and warning in the file. Click a file to display the occurrences in the Details portion of the Files Affected pane. Select **Show All Errors** to display every error and warning for the selected file.

Errors are indicated with a symbol. Warnings are indicated with a symbol. Click an occurrence of an error or warning to navigate to the offending code in the Source pane.

Source Pane

The Source pane displays view-only source for the file selected in the Files Affected pane. Offending code is highlighted in red.

Usage is similar to that for the Modernization Workbench HyperView Source pane. For more information, see *Analyzing Programs* in the workbench documentation set.

Marking Items

To mark an item, place a check mark next to it. To mark all the items in the selected tab, choose **Edit > Mark All**. To unmark all the items in the selected tab, choose **Edit > Unmark All**.

Including Files into Projects

In very large workspaces, you may find it useful to move or copy files into different projects based on the errors they contain. Follow the instructions below to move or copy files listed in a Verification Report into a project.

1. In the Files Affected pane, mark the items you want to move and choose **File > Include Into Project**. The Select Project window opens.
2. In the Select Project window, select the target project. Click **New** to create a new project.
3. Select:
   - **Include All Referenced Objects** if you want to include objects referenced by the selected object (the Cobol copybooks included in a Cobol program file, for example).
   - Select **Include All Referencing Objects** if you want to include objects that reference the selected object.

   Note: This feature is available only for verified files.
4. Select:

- **Copy** to copy the selection to the target project.
- **Move From Current Project** to move the selection to the target project.
- **Move From All Projects** to move the selection from all projects to the target project.

5. Click **OK** to move or copy the selection.

Generating HTML Reports

To generate HTML reports of Verification Report results, choose **File > Report > <*Report Type*>**. Before generating the Details for Checked Files report, mark each file on which you want to report.

Tip: The Missing Files report is a convenient alternative to an Unresolved Report when you are interested only in missing source files, and not in unresolved objects like system programs.

Viewing Executive Reports

Executive Reports offer HTML views of application inventories that a manager can use to assess the risks and costs of supporting the application:

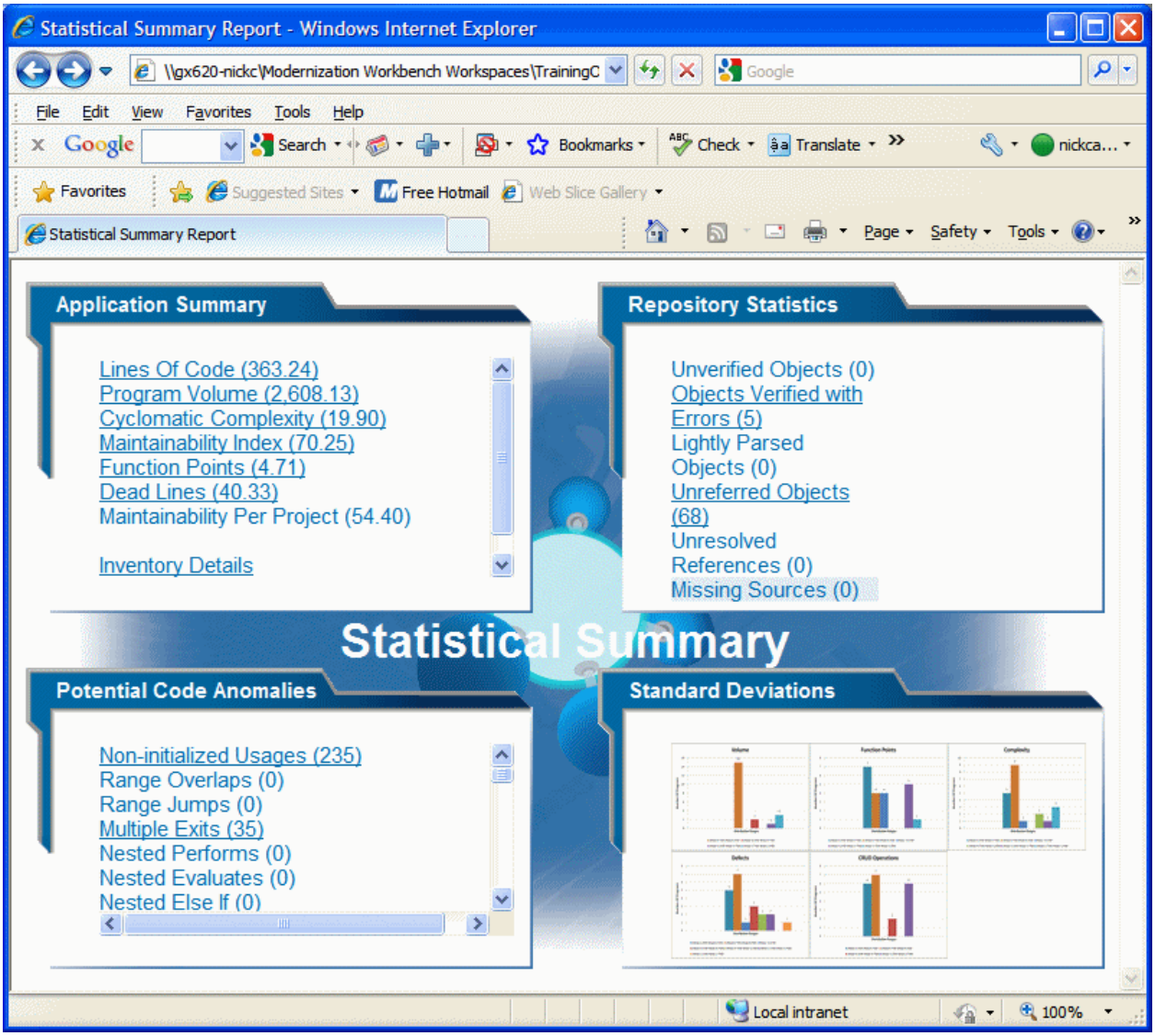
- The Application Summary view gives statistics for industry-standard metrics such as program volume, maintainability, cyclomatic complexity, and number of defects.
- The Potential Code Anomalies view gives statistics for potential code anomalies that may mark programs as candidates for re-engineering: GOTO non-exits, range overlaps, and the like. You can customize potential code anomalies using the Define Anomalies feature.

  Note: You must set **Detect Potential Code Anomalies** in the Verification > Settings tab of the Workspace Options window to generate these statistics.
- The Repository Statistics view gives statistics for Modernization Workbench verification results and unresolved or unreferenced application elements.
- The Standard Deviations view displays graphs that plot the deviation of the programs in the application from the means for six key industry-standard metrics.

To generate an Executive Report, choose **Prepare > Executive Report**. Project options on the Report > Executive Report tab determine the entities included in the report, and the functions the report performs. The report is stored in \<*Workspace Home*>\Output\Executive Report\Project\index.htm.

The figure below shows an Executive Report. The top page in each view displays the available statistics and graphs. Click the links to view the detail for each type of statistic or graph. In the statistic or graph detail page, click the link for a program to view the detail for that program.

Note: After generating an Executive Report, use the Executive Report category in the HyperView Clipper tool to view potential code anomalies in program context.



Setting Executive Report Options

Use the Report > Executive Report tab of the Project Options window to specify the entities included in executive reports, and the functions the report performs. Generally, the fewer entities and functions you choose, the better report generation performance you can expect. You should especially consider not reporting on:

- Detail for code anomalies. Leave **Generate Details** unchecked.
- Data stores. Leave data store entities unchecked.
- Relationships, if you do not need cross-reference information. Leave **Relationships** unchecked.

1. Choose **Tools > Project Options**. The Project Options window opens. Click the Report tab, then the Executive Report tab.
2. Click **Functionality**, then select each report function you want to enable. For the Application Summary function, click **Select Metrics**. In the Application Summary (Averages) window, select **Application Summary (Averages)** to enable the function, then choose each metric you want to include in the report.
3. Select each type of entity you want to include in the report. To edit the attributes included in the report for the entity type, click **Attributes**. In the Attributes window, select each attribute of the selected entity type to include in the report.

Defining Potential Code Anomalies

You can view and modify existing definitions of potential code anomalies (other than range overlaps and range jumps) in the HyperView advanced search criteria in the Coding Standards folder.

To define a new code anomaly, you must define an advanced search criterion for the anomaly and a matching entry in the file <*Workbench Home*>\Data\CodeDefects.xml. The entry has the form:

```
<DEFECT Id="name"
Internal="True|False"
Enabled="True|False"
Caption="display name"
ListName="list name"
Criterion="path of criterion"/>
```

where:

- **`Id`** is a unique name identifying the code anomaly in the workbench.
- **`Internal`** specifies whether the anomaly is implemented internally in program code (True), or externally in an advanced search criterion (**`False`**).

  Note: You must specify False. Code anomalies with an Internal value of True cannot be modified.
- **`Enabled`** specifies whether the code anomaly is displayed in the Executive Report.
- **`Caption`** is the display name for the anomaly in the Executive Report.
- **`ListName`** is the name of the list of anomalous code constructs displayed in the Executive Report category of the HyperView Clipper tool.
- **`Criterion`** is the full path name of the criterion in the HyperView Advanced Search tool, including the tab name (General) and any folder names. For example, General:Coding Standards\MOVE Statements \Possible Data Padding.

You can display the anomaly caption in Japanese or Korean in the Executive Report by creating an entry for the anomaly in the file <*Workbench Home*>\Language\[Jpn|Kor]\CodeDefects.xrc. The entry has the form:

```
<String name="name"
listname="list name"
caption="translated display name"
description="description"/>
```

where:

- **`name`** is the unique name of the code anomaly in the workbench (**`Id`** attribute of CodeDefects.xml entry).
- **`listname`** is the name of the list of anomalous code constructs displayed in the Executive Report category of the HyperView Clipper tool (**`ListName`** attribute of CodeDefects.xml entry).
- **`caption`** is the translated display name for the anomaly in the Executive Report.
- **`description`** contains a description of the entry.

Cobol Range Overlaps and Range Jumps Detected in the Executive Report

This section lists Cobol range overlaps and range jumps detected in the Executive Report. S* defects appear in the report under “Range Overlaps” as the sum of all defects S1+S2+S3+S4+S5+S6. G* defects appear in the report under “Range Jumps” as the sum of all defects G1+G2+G3+G5+G6+G7.

S0. No defects

```
Perform A1 thru A2.
Perform B1 thru B2.
…
.--A1.
| StatementsA1.
| …
| A2.
’--- StatementsA2.
…
.--B1.
| StatementsB1.
| …
| B2.
’--- StatementsB2.
```

S1. Overlapped sections

```
Perform A1 thru A2.
Perform B1 thru B2.
…
.--A1.
| StatementsA1.
| …
| B1. --.
| StatementsB1. |
| … |
| A2. |
’--- StatementsA2. |
… |
B2. |
StatementsB2. –-’
```

S2. Overlapped sections

```
Perform A1 thru A2.
Perform B1 thru B2.
…
.--B1.
| StatementsB1.
| …
| A1. --.
| StatementsA1. |
| … |
| B2. |
’--- StatementsB2. |
```

```
… |
A2. |
StatementsA2. –-’
```

S3. Overlapped sections

```
Perform A1 thru A2.
Perform B1 thru B2.
…
.--A1.
| StatementsA1.
| …
| B1. --.
| StatementsB1. |
| … |
| B2. |
| StatementsB2. –-’
| …
| A2.
‘--- StatementsA2.
```

S4. Overlapped sections

```
Perform A1 thru A2.
Perform A1 thru B2.
…
.--A1. --.
| StatementsA1. |
| … |
| B2. |
| StatementsB2. –-’
| …
| A2.
‘--- StatementsA2.
```

S5. Overlapped sections

```
Perform A1 thru A2.
Perform B1 thru A2.
…
.--A1.
| StatementsA1.
| …
| B1. --.
| StatementsB1. |
| … |
| A2. |
‘--- StatementsA2. --’
```

S6. Overlapped sections

```
Perform A1 thru A2.
…
.--A1.
| StatementsA1.
| …
| Perform B1 thru B2.
| …
```

```
| B1. --.
| StatementsB1. |
| … |
| B2. |
| StatementsB2. –-’
| …
| A2.
’--- StatementsA2.
```

G0. No defects

```
Perform A1 thru A2.
…
.--A1.
| StatementsA1.
| …
| goto B1.
| …
| B1.
| StatementsB1.
| …
| A2.
’--- StatementsA2.
```

G0. No defects

```
Perform A1 thru A2.
goto B1.
…
.--A1.
| StatementsA1.
| …
| A2.
’--- StatementsA2.
…
B1.
StatementsB1.
```

G0. No defects

```
Perform A1 thru A2.
goto B1.
…
B1.
StatementsB1.
…
.--A1.
| StatementsA1.
| …
| A2.
’--- StatementsA2.
```

G0. No defects

```
Perform A1 thru A2.
.--A1.
| StatementsA1.
| …
```

```
| A2.
‘--- StatementsA2.
…
goto B1.
…
B1.
StatementsB1.
```

G1. Break-in goto

```
Perform A1 thru A2.
…
goto B1.
…
.--A1.
| StatementsA1.
| …
| B1.
| StatementsB1.
| …
| A2.
‘--- StatementsA2.
```

G2. Break-in goto

```
Perform A1 thru A2.
…
goto A1.
…
.--A1.
| StatementsA1.
| …
| A2.
‘--- StatementsA2.
```

G3. Break-in goto

```
Perform A1 thru A2.
…
goto A2.
…
.--A1.
| StatementsA1.
| …
| A2.
‘--- StatementsA2.
```

S3G4=G1. Overlapped sections, break-in goto

```
Perform A1 thru A2.
…
.--A1.
| StatementsA1.
| …
| Perform B1 thru B2.
| …
| goto C1.
| B1. --.
```

```
| StatementsB1. |
| … |
| C1. |
| … |
| B2. |
| StatementsB2. –-’
| …
| A2.
‘--- StatementsA2.
```

G5. Break-out goto

```
Perform A1 thru A2.
…
.--A1.
| StatementsA1.
| …
| goto B1.
| …
| A2.
‘--- StatementsA2.
…
B1.
StatementsB1.
```

G6. Break-out goto

```
Perform A1 thru A2.
…
.--A1.
| StatementsA1.
| …
| goto A1.
| …
| A2.
‘--- StatementsA2.
```

G7. Break-out goto

```
Perform A1 thru A2.
…
.--A1.
| StatementsA1.
| …
| goto A2.
| …
| A2.
‘--- StatementsA2.
```

G8. No Defects

```
Perform A1 thru A2.
…
.--A1.
| StatementsA1.
| …
| goto A2.
| …
```

```
| A2.
'--- EXIT.
```

S3G9=G5. Overlapped sections, break-out goto

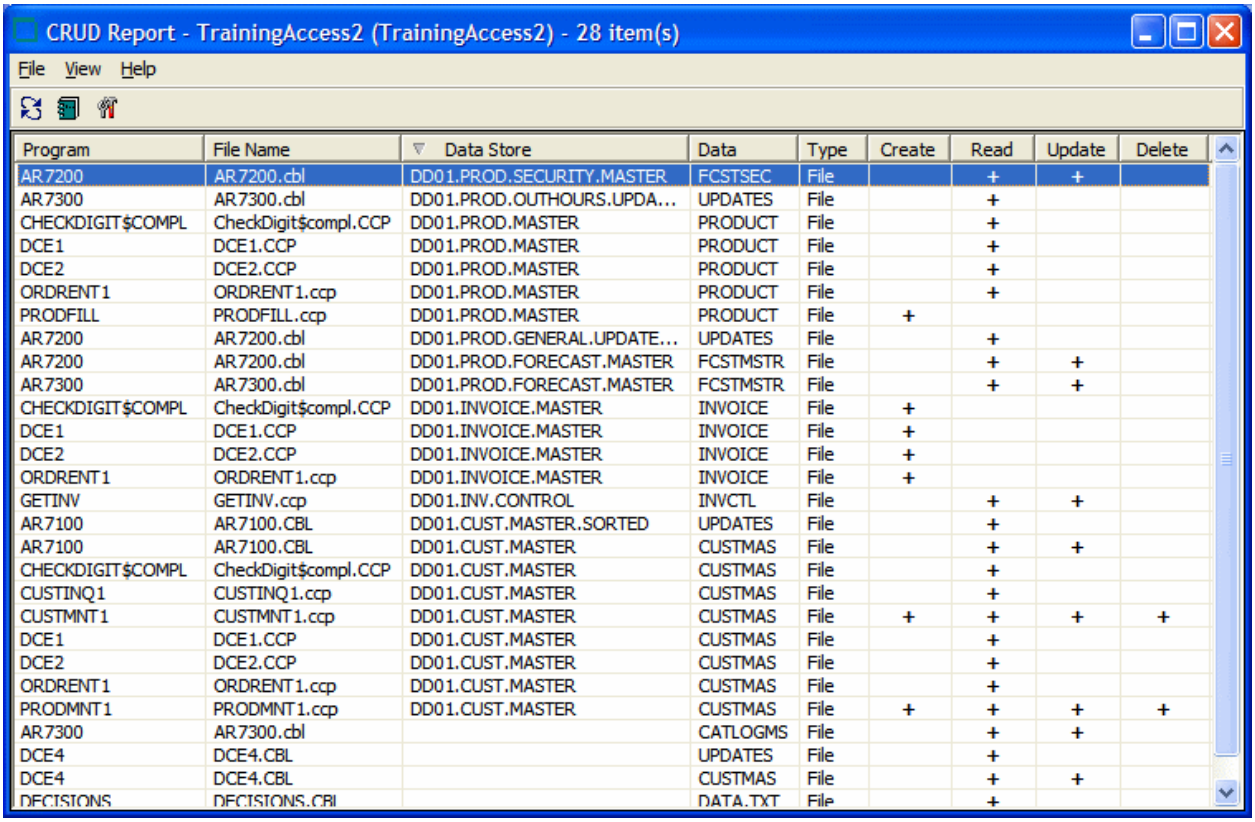
```
Perform A1 thru A2.
…
.--A1.
| StatementsA1.
| …
| Perform B1 thru B2.
| …
| B1. --.
| StatementsB1. |
| … |
| goto C1. |
| … |
| B2. |
| StatementsB2. –-'
| …
| C1.
| …
| A2.
'--- StatementsA2.
```

Viewing CRUD Reports

The CRUD Report for a project shows the data operations each program in the project performs, and the data objects on which the programs operate. To generate a CRUD Report, select a project in the Repository Browser and choose **Prepare > CRUD Report**. The figure below shows a CRUD Report.

Project options on the Report > CRUD Report tab determine the data operations and program-to-data object relationships displayed in CRUD reports. To refresh the report after modifying display options, choose **File > Refresh** in the CRUD Report window. To generate the report in HTML, choose **File > Report**.

Note: The IMS data column of the CRUD report behaves differently from the columns for other data types. What appears in the IMS data column cells depends on what can be determined. If the segment can be determined, the cell is populated with the PSB name and segment name. Otherwise, the segment name is left blank. The format is **xxxxxx.yyyyyyy**, where **xxxxxx** is the PSB name and **yyyyyyy** is the segment name or blank if the segment cannot be determined.



Program	File Name	Data Store	Data	Type	Create	Read	Update	Delete
AR7200	AR7200.cbl	DD01.PROD.SECURITY.MASTER	FCSTSEC	File		+	+	
AR7300	AR7300.cbl	DD01.PROD.OUTHOURS.UPDA...	UPDATES	File		+		
CHECKDIGIT$COMPL	CheckDigit$compl.CCP	DD01.PROD.MASTER	PRODUCT	File		+		
DCE1	DCE1.CCP	DD01.PROD.MASTER	PRODUCT	File		+		
DCE2	DCE2.CCP	DD01.PROD.MASTER	PRODUCT	File		+		
ORDRENT1	ORDRENT1.ccp	DD01.PROD.MASTER	PRODUCT	File		+		
PRODFILL	PRODFILL.ccp	DD01.PROD.MASTER	PRODUCT	File	+			
AR7200	AR7200.cbl	DD01.PROD.GENERAL.UPDATE...	UPDATES	File		+		
AR7200	AR7200.cbl	DD01.PROD.FORECAST.MASTER	FCSTMSTR	File		+	+	
AR7300	AR7300.cbl	DD01.PROD.FORECAST.MASTER	FCSTMSTR	File		+	+	
CHECKDIGIT$COMPL	CheckDigit$compl.CCP	DD01.INVOICE.MASTER	INVOICE	File	+			
DCE1	DCE1.CCP	DD01.INVOICE.MASTER	INVOICE	File	+			
DCE2	DCE2.CCP	DD01.INVOICE.MASTER	INVOICE	File	+			
ORDRENT1	ORDRENT1.ccp	DD01.INVOICE.MASTER	INVOICE	File	+			
GETINV	GETINV.ccp	DD01.INV.CONTROL	INVCTL	File		+	+	
AR7100	AR7100.CBL	DD01.CUST.MASTER.SORTED	UPDATES	File		+		
AR7100	AR7100.CBL	DD01.CUST.MASTER	CUSTMAS	File		+	+	
CHECKDIGIT$COMPL	CheckDigit$compl.CCP	DD01.CUST.MASTER	CUSTMAS	File		+		
CUSTINQ1	CUSTINQ1.ccp	DD01.CUST.MASTER	CUSTMAS	File		+		
CUSTMNT1	CUSTMNT1.ccp	DD01.CUST.MASTER	CUSTMAS	File	+	+	+	+
DCE1	DCE1.CCP	DD01.CUST.MASTER	CUSTMAS	File		+		
DCE2	DCE2.CCP	DD01.CUST.MASTER	CUSTMAS	File		+		
ORDRENT1	ORDRENT1.ccp	DD01.CUST.MASTER	CUSTMAS	File		+		
PRODMNT1	PRODMNT1.ccp	DD01.CUST.MASTER	CUSTMAS	File	+	+	+	+
AR7300	AR7300.cbl		CATLOGMS	File		+	+	
DCE4	DCE4.CBL		UPDATES	File		+		
DCE4	DCE4.CBL		CUSTMAS	File		+	+	
DECISIONS	DECISIONS.CBL		DATA.TXT	File		+		

Setting CRUD Report Options

Use the Report > CRUD Report tab of the Project Options window to specify the data operations and program-to-data object relationships displayed in CRUD Reports.

1. Choose **Tools > Project Options**. The Project Options window opens. Click the Report tab, then the CRUD Report tab.
2. Place a check mark next to each type of program-to-data object relationship you want to display.
3. Place a check mark next to each type of data operation you want to display.

Inventorying Applications

Users often ask why the Modernization Workbench parser encounters errors in working production systems. The reasons usually have to do with the source file delivery mechanism: incorrect versions or copybooks, corruption of special characters because of source code ambiguities, FTP errors, and so forth.

Use Modernization Workbench *inventory analysis* tools to ensure that all parts of your application are available to the parser:

- Reference Reports let you track referential dependencies in verified source.
- Orphan Analysis lets you analyze and resolve objects that do not exist in the reference tree for any top-level program object, so-called *orphans*. Orphans can be removed from a system without altering its behavior.
- Decision Resolution identifies and lets you resolve dynamic calls and other relationships that the parser cannot resolve from static sources in Cobol, PL/I, and Natural programs.

Using Reference Reports

When you verify a legacy application, the parser generates a model of the application that describes the objects in the application and how they interact. If a Cobol source file contains a COPY statement, for example, the system creates a relationship between the file and the Cobol copybook referenced by the statement. If the copybook doesn't exist in the repository, the system flags it as missing by listing it with a  symbol in the tree view of the Repository Browser.

Reference Reports let you track these kinds of referential dependencies in verified source:

- An Unresolved Report identifies missing application elements.
- An Unreferred Report identifies unreferenced application elements.
- A Cross-reference Report identifies all application references.
- An External Reference Report identifies references in object-oriented applications to external files that are not registered in the workspace, such as .java, Java Archive (JAR), or C++ include files (assuming you have identified the locations of these files in the Workspace Verification options window for the source files). These references are not reported as unresolved in the Unresolved Report.

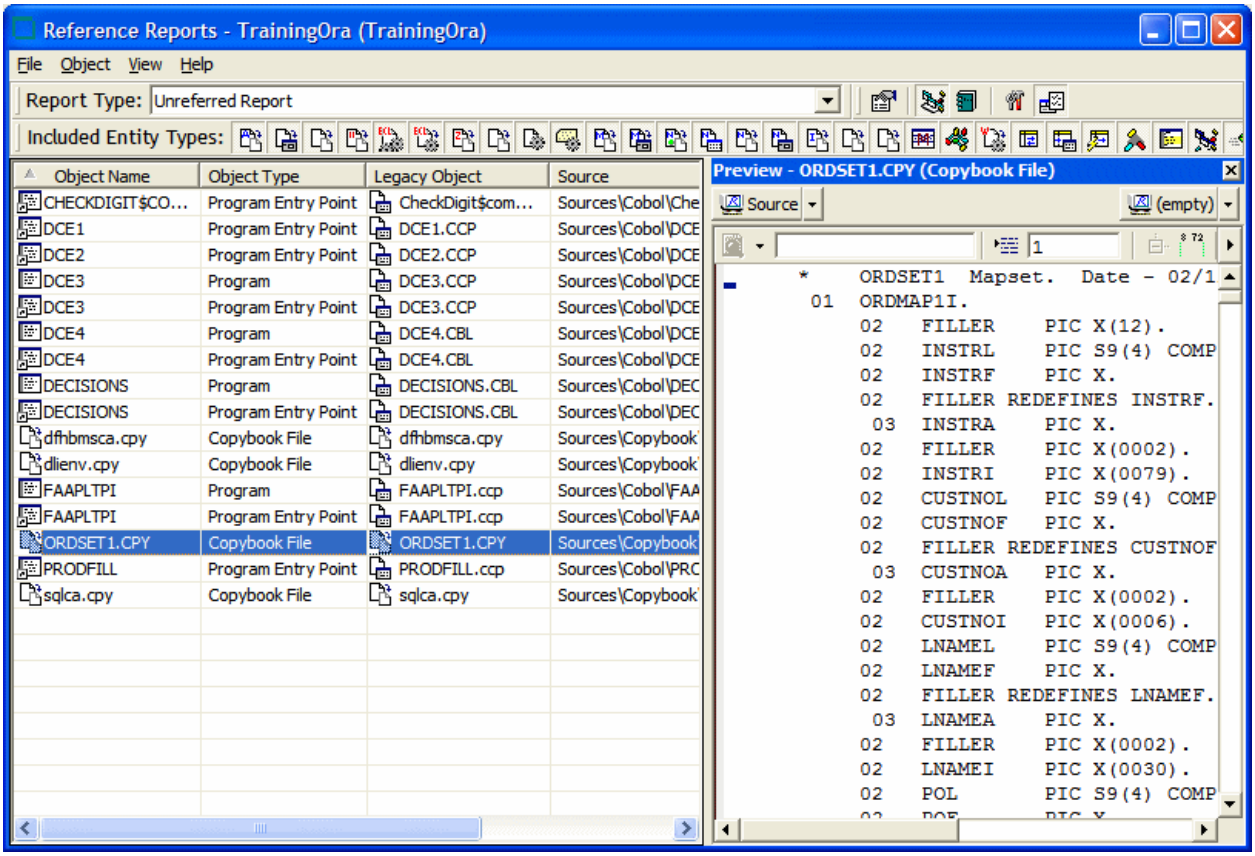
Tip: The Missing Files report in the Verification Report tool is a convenient alternative to an Unresolved Report when you are interested only in missing source files, and not in unresolved objects like system programs.

Understanding the Reference Reports Window

Use Reference Reports to track referential dependencies in verified project source. To open the Reference Reports window, select a project in the Repository Browser and choose **Prepare > Reference Reports**.

When the Reference Reports window opens, choose the Reference Report type in the **Report type** drop-down. To limit the report to references in the current project, choose **View > Restrict References to Project**. To generate the report in HTML, choose **File > Report**.

The figure below shows an Unreferred Report window. The windows for the other reports are similar. By default, all Reference Report panes are displayed. Select the appropriate choice in the **View** menu to hide a pane. Select the choice again to show the pane.



Main Pane

The Main pane displays the objects in the Reference Report and their relationships. The table below describes the columns in the Main pane.

Column	Report Type	Description
Object Name	All	The name of the unresolved, unreferenced, cross-referenced, or externally referenced object.
Object Type	All	The entity type of the unresolved, unreferenced, cross-referenced, or externally referenced object.
Legacy Object	Unreferred Report, Cross-reference Report	The source file that contains the unreferenced or cross-referenced object.
Source	Unreferred Report, Cross-reference Report	The location in the workspace folder of the source file that contains the unreferenced or cross-referenced object.

Column	Report Type	Description
Referred by	Unresolved Report, Cross-reference Report, External Reference Report	The name of the referring object.
Referring Object Type	Unresolved Report, Cross-reference Report, External Reference Report	The entity type of the referring object.
Relationship	Unresolved Report, Cross-reference Report, External Reference Report	The relationship between the unresolved, cross-referenced, or externally referenced object and the referring object.
Object Description	All	The description of the unresolved, unreferenced, cross-referenced, or externally referenced object entered by the user on the Description tab of the Object Properties window.

Preview Pane

The Preview pane lets you browse HyperView information for the object selected in the Report pane. The information available depends on the type of object selected. You see only source code for a copybook, for example, but full HyperView information for a program. Choose the information you want to view for the object from the **Source** drop-down.

Setting Reference Reports Options

Use the Report > Reference Reports tab of the Project Options window to specify the entity types for which reference report information is collected.

1. Choose **Tools > Project Options**. The Project Options window opens. Click the Report tab, then the Reference Reports tab.
2. Place a check mark next to each type of entity you want to be included in reference reports.

Using Orphan Analysis Reports

An object that does not exist in the reference tree for any top-level object is called an *orphan*. Orphans can be removed from a system without altering its behavior. Use the Orphan Analysis tool to find orphans.

What's the difference between an orphan and an unreferenced object?

- All unreferenced objects are orphans.
- Not every orphan is unreferenced.

Suppose an unreferred report shows that the copybook GSS3.CPY is not referenced by any object in the project. Meanwhile, a cross-reference report shows that GSS3.CPY references GSS3A.CPY and GSS3B.CPY.

These copybooks do not appear in the unreferred report because they are referenced by GSS3.CPY. Only orphan analysis will show that the two copybooks are not in the reference tree for the GSS program and, therefore, can be safely removed from the project.

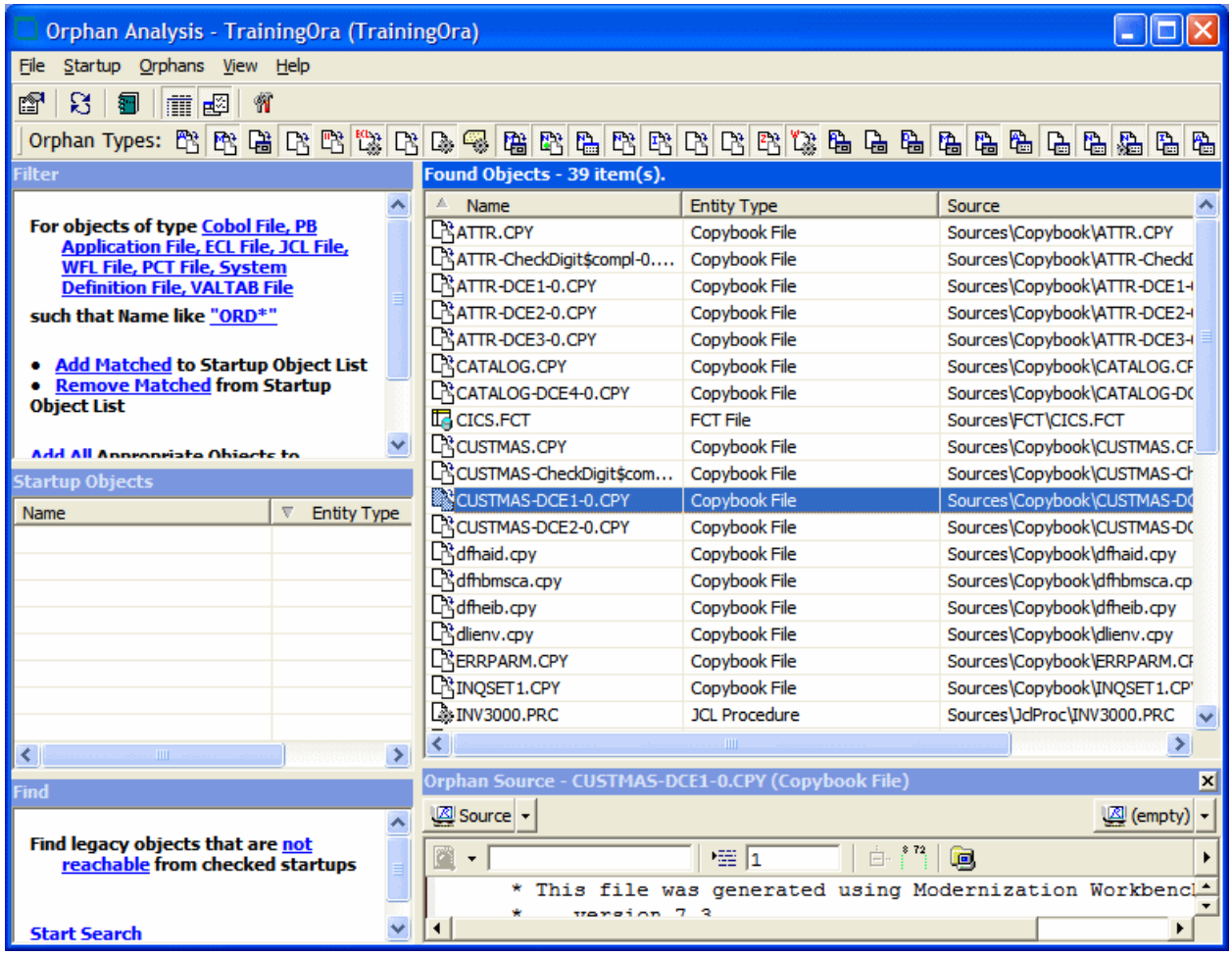
Understanding the Orphan Analysis Window

Use the Orphan Analysis tool to determine whether an object exists in the reference tree for a top-level program object. To open the Orphan Analysis tool window, select a project in the Repository Browser and choose **Prepare > Orphan Analysis**.

Project options on the Report > Orphan Analysis tab specify the search filter for the report. To refresh the report after modifying the options, choose **Orphans > Refresh** in the Orphan Analysis window. To generate the report in HTML, choose **File > Save Report As**.

Note: The Filter, Startup, and Find panes let you use hyperlinks to set up and apply the Orphan Analysis search filter. Use these panes instead of the options window if you prefer.

The figure below shows the Orphan Analysis window. By default, all Orphan Analysis panes are displayed. Select the appropriate choice in the **View** menu to hide a pane. Select the choice again to show the pane.



Found Objects Pane

The Found Objects pane shows the name, type, and source location of orphans. To show the list of orphans only, deselect **View > Report View**.

Orphan Source Pane

The Orphan Source pane lets you browse HyperView information for the object selected in the Found Objects pane. The information available depends on the type of object selected. You see only source code for a copybook, for example, but full HyperView information for a program. Choose the information you want to view for the object from the **Source** drop-down.

Setting Orphan Analysis Options

Use the Report > Orphan Analysis tab of the Project Options window to specify the search filter for an Orphan Analysis report.

1. Choose **Tools > Project Options**. The Project Options window opens. Click the Report tab, then the Orphan Analysis tab.
2. In the Startup pane, click **Select Startup Types**. The Startup dialog opens.
3. In the Startup dialog, set up a search filter for the startup objects in the orphan analysis. You can filter on entity type, entity name, or both:
   - To filter on entity type, place a check mark next to the entity type you want to search for in the Roots pane.
   - To filter on entity name, place a check mark next to a recognized matching pattern in the Like pane, the Unlike pane, or both. Add patterns as necessary. You can use wildcard patterns allowed in LIKE statements by Visual Basic for Applications (VBA).
4. When you are satisfied with your choices in the Startup dialog, click **OK**.
5. In the Find pane, define the terms of the search by selecting the appropriate choice in the **Relationships to Checked Startups** drop-down, the **Relationships to Unchecked Startups** drop-down, or both.
6. In the Entities pane, click **Displayed Types**. The Entities dialog opens. In the Entities dialog, place a check mark next to each type of entity to include in the report. When you are satisfied with your choices in the Entities dialog, click **OK**.

Deleting Orphans from a Project

To delete an orphan from a project (but not the workspace), select the orphan in the Found Objects pane and choose **Orphans > Exclude from Project**.

Deleting Orphans from a Workspace

To delete an orphan from the workspace, select the orphan in the Found Objects pane and choose **Orphans > Delete from Workspace**.

Resolving Decisions

You need to have a complete picture of the control and data flows in a legacy application before you can diagram and analyze the application. The parser models the control and data transfers it can resolve from static sources. Some transfers, however, are not resolved until run time. *Decision resolution* lets you identify and resolve dynamic calls and other relationships that the parser cannot resolve from static sources.

Understanding Decisions

A decision is a reference to another object, a program or screen, for example, that is not resolved until run time. Consider a Cobol program that contains the following statement:

```
CALL ‘NEXTPROG’.
```

The Modernization Workbench parser models the transfer of control to program NEXTPROG by creating a Calls relationship between the original program and NEXTPROG.

But what if the statement read this way instead:

```
CALL NEXT.
```

where NEXT is a field whose value is only determined at run time. In this case, the parser creates a Calls relationship between the program and an abstract decision object called PROG.CALL.NEXT, and lists the decision object with a  icon in the tree view of the Repository Browser.

The Decision Resolution tool creates a list of such decisions and helps you navigate to the program source code that indicates how the decision should be resolved. You may learn from a declaration or MOVE statement, for example, that the NEXT field takes either the value NEXTPROG or ENDPROG at run time. In that case, you would resolve the decision manually by telling the system to create resolves to relationships between the decision and the programs these literals reference.

Of course, where there are hundreds or even thousands of such decisions in an application, it may not be practical to resolve each decision manually. In these situations, you can use the *autoresolve* feature to resolve decisions automatically.

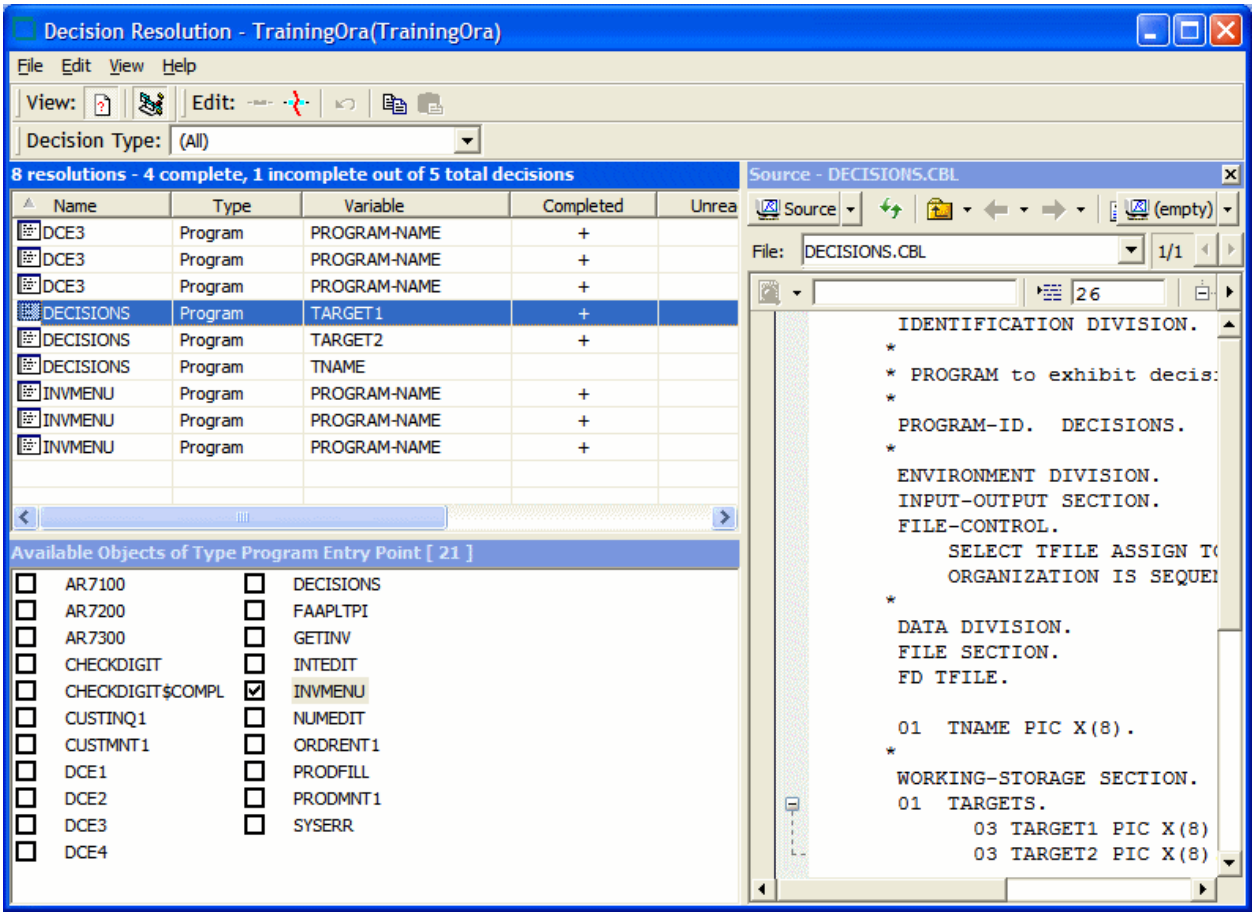
The Decision Resolution tool analyzes declarations and MOVE statements, and any other means of populating a decision point, to determine the target of the control or data transfer. The tool may not be able to autoresolve every decision, or even every decision completely, but it should get you to a point where you can complete decision resolution manually.

Understanding the Decision Resolution Tool Window

Use the Decision Resolution tool to view and manually resolve decisions. To open the Decision Resolution tool window, select a project in the Repository Browser and choose **Prepare > Resolve Decisions**.

To save decision resolutions to the repository, choose **File > Save**. To generate the Decision Resolution report in HTML, choose **File > Report**.

The figure below shows the Decision Resolution window. By default, all Decision Resolution panes are displayed. Select the appropriate choice in the **View** menu to hide a pane. Select the choice again to show the pane.



Decision List Pane

The Decision List pane displays the decisions in the project. To filter the list, choose the type of decision you want to display in the **Decision Type** drop-down. The table below describes the columns in the Decision List pane.

Column	Description
Name	The name of the object that contains the decision.
Type	The type of the object that contains the decision.
Variable	The program variable that requires the decision.
Completed	Whether the decision has been resolved.
Unreachable	Whether the decision is in dead code.
Manual	Whether the decision was resolved manually.

Column	Description
Resolved to	The target object the variable resolves to (an entry point, for example). An unresolved decision contains the grayed-out text Some *Object*.

Available Targets Pane

For the selected decision type, the Available Targets pane lists the objects in the workspace to which the decision can be resolved. To resolve a decision to an available target, select the decision in the Decision List pane and place a check mark next to the target.

To limit the targets to objects in the current project, choose **View > Restrict to Current Project.** To delete a decision resolution, remove the check mark next to the target. To undo changes, choose **Edit > Undo all changes**.

Source Pane

The Source pane lets you browse HyperView information for the object selected in the Decision List pane. The information available depends on the type of object selected. You see only source code for a copybook, for example, but full HyperView information for a program. Choose the information you want to view for the object from the drop-down in the upper lefthand corner of the pane.

Resolving Decisions Manually

Follow the instructions below to resolve decisions manually to targets in or out of the workspace.

1. To resolve decisions to available targets, select one or more entries in the Decision List pane and place a check mark next to one or more target objects in the Available Targets pane. If you link an entry to multiple targets, the Decision Resolution tool creates as many entries in the Decision List pane as there are targets.

   Note: If you are linking an entry to multiple targets, you can save time by selecting the targets and choosing **Edit > Link Selected Targets**. You can also choose **Edit > Copy** to copy selected targets to the clipboard, then **Edit > Paste** to link the targets to an entry.
2. To resolve decisions to targets not in the workspace, select one or more entries in the Decision List pane and choose **Edit > Link New Target**. The Link New Target window opens.
3. In the Link New Target window, enter the name of the new target in the field on the righthand side of the window, or populate the field by clicking a literal in the list of program literals on the Literals tab. Filter the list by using:
   - The **Minimum Literal Length** slider to specify the minimum number of characters the literal can contain.
   - The **Maximum Literal Length** slider to specify the maximum number of characters the literal can contain.
   - The **Names Like** field to enter a matching pattern for the literal. You can use wildcard patterns allowed in LIKE statements by Visual Basic for Applications (VBA).
4. Place a check mark next to **Completed** if you want the resolution to be marked as completed. When you are satisfied with your entry, click **OK**.

Restoring Manually Resolved Decisions

Reverifying a file invalidates all of its objects, including its manually resolved decisions. The *decision persistence* feature lets you restore manually resolved decisions when you return to the Decision Resolution tool.

After reverifying a file for which you have manually resolved decisions, reopen the Decision Resolution tool. A dialog box prompts you to restore manually resolved decisions. Click **Yes** if you want to restore the decisions. Click **No** otherwise.

Note: Place a check mark next to **Don't show me again** if you want the Decision Resolution tool to open without prompting you to restore manually resolved decisions. In the Decision Resolution Tool tab of the User Preferences window, place a check mark next to **Ask before restoring previous manual changes** if you want to be prompted again.

Resolving Decisions Automatically

You can autoresolve decisions during verification by setting the **Resolve decisions automatically** option on the Verification tab of the Project Options window for a source file type.

For programs only, you can autoresolve decisions after verification by selecting the project, folder, or files for which you want to autoresolve decisions and choosing **Prepare > AutoResolve Decisions**. Only a master user can autoresolve decisions in a multiuser environment.

Note: Decision Resolution cannot autoresolve every decision. The target name may be read from a data file, for example.

Setting Decision Resolution Tool User Preferences

Use the Decision Resolution Tool tab of the User Preferences window to specify whether you want to be prompted to restore invalidated manually resolved decisions when you reopen the Decision Resolution tool.

1. Choose **Tools > User Preferences**. The User Preferences window opens. Click the Decision Resolution Tool tab.
2. Select **Ask before restoring previous manual changes** if you want to be prompted to restore manually resolved decisions when you reopen the Decision Resolution tool.

Identifying Interfaces for Generic API Analysis

Use the Generic API Analysis feature if your legacy program calls an unsupported API to interface with a database manager, transaction manager, or similar external facility. In this call, for example:

```
CALL 'XREAD' using X
```

where X evaluates to a table name, the call to XREAD is of less interest than its parameter, the table the called program reads from. But because the parser does not recognize XREAD, only the call is modeled in the workbench repository.

You enable the Generic API Analysis feature by identifying unsupported APIs and their parameters in the file \<*Workbench Home*>\Data\Legacy.xml. Before you verify your application, set **Perform Generic API Analysis** on the Verification tab of the Project Options window. That option tells the parser to define relationships with the objects passed as parameters in the calls, in addition to relationships with the unsupported APIs themselves.

This section shows you how to identify the programs and parameters to the parser before verifying your application. You can specify both object and construct model information, and create different relationships or entities for the same parameter in a call.

The specification requires a thorough understanding of the Modernization Workbench repository models. For background on the repository models, see the *Software Development Kit* manual, available from support services.

Note: Only the predefined definitions described in this section are guaranteed to provide consistent data to workbench databases.

Identifying Unsupported API Calls to the Parser

Follow the instructions in the steps below and the detailed specifications in the following sections to identify unsupported API calls to the parser. For each call, you need to define an entry in \<*Workbench Home*>\Data\Legacy.xml that specifies at a minimum:

- The name of the called program and the method of invocation in the <match> tag.
- The program control flow in the <flow> tag, and the direction of the data flow through the parameters of interest in the <param> subtags.
- How to represent the call in the object model repository in the <rep> tag, and in the construct model repository in the <hc> tag.

Use the optional <vars> tag to extract values of a specified type, size, and offset from a parameter for use in a <rep> or <hc> definition.

Most repository entities can be represented in a <rep> or <hc> definition with the predefined patterns in \<*Workbench Home*>\Data\Legacy.xml.api. These patterns do virtually all of the work of the specification for you, supplying the relationship of the entity to the called program, its internal name, and so forth.

The syntax for specifying predefined patterns in a <rep> or <hc> definition is described in the section for the tag. Consult Legacy.xml.api for supported patterns and for required parameters and values.

1. Open the file \<*Workbench Home*>\Data\Legacy.xml in an editor.
2. Locate the <GenericAPI> section for the language and dialect you use.
3. Create entries for each unsupported API call, following the specifications in the sections below and the samples of Generic API usage in Legacy.xml.
4. Set **Perform Generic API Analysis** on the Verification tab of the Project Options window.
5. Verify the project.

Using the API Entry Tag

The *name* attribute of the <API Entry> tag is the name of the entry, used for error diagnostics only.

Using the match Tag

The *stmt* attribute of the <match> tag identifies the method of invocation: a CALL, LINK, or XCTL statement. The *value* attribute of the <name> subtag identifies the name of the program to be matched. It can also be used to specify an alternative name for the entry.

Note: The name of the program to be matched must be unique in the <GenericAPI> section. If names are not unique, the parser uses the last entry in which the name appears.

Example

```
<match stmt="CALL">
  <name value="XREAD"/>
</match>
```

Using the flow Tag

The <flow> tag characterizes the program control flow. The *halts* attribute of the <flow> tag specifies whether the calling program terminates after the call:

- yes, if control is not received back from the API.
- no (the default), if the calling program returns normally.

The <param> subtag identifies characteristics of the call parameters. Attributes are:

- *index* is the index of the parameter that references the item of interest, beginning with 1. Use an asterisk (*) to specify all parameters not specified directly.
- *usage* specifies the direction of the data flow through the parameter: r for input, w for output, rw for input/output. Unspecified parameters are assumed to be input/output parameters.

Note: *halts* is supported only for call statements. For PL/I, input parameters are treated as input/output parameters.

Example

```
<flow halts='no'>
```

```
  <param index='1' usage='r'/>
  <param index='2' usage='r'/>
  <param index='3' usage='rw'/>
  <param index='*' usage='rw'/>
</flow>
```

Using the vars Tag

Use the <vars> tag to extract values of a specified type, size, and offset from a call parameter. You can then refer to the extracted values in the <rep> and <hc> tags using the %var_name notation.

The <arg> subtag characterizes the argument of interest. Attributes are:

- *var* specifies the variable name.
- *param* specifies the index of the parameter.
- *type* specifies the variable type.
- *offset* specifies the offset of the field in bytes.
- *bitoffset* specifies the offset of the field in bits.
- *size* specifies the size of the field in bytes.
- *bitsize* specifies the size of the field in bits.

Additional attributes for PL/I are:

- *len* specifies the size of a character or bit string field.
- *mode* specifies the binary or decimal mode for a numeric field.
- *scale* specifies the scale of a fixed-point numeric field.
- *prec* specifies the precision of a fixed-point or floating-point numeric field.
- *varying* specifies whether a bit string variable is encoded as a varying-length string in the structure (yes or no, the default).

Supported data types are described in the language-specific sections below.

Example

Suppose a call to a database-entry API looks like this:

```
CALL 'DBNTRY' USING DB-USER-ID
                    DB-XYZ-REQUEST-AREA
                    XYZ01-RECORD
                    DB-XYZ-ELEMENT-LIST.
```

If the second parameter contains a 3-character table name in bytes 6-8, the following definition extracts the name for use as the right end of a relationship:

```
<vars>
  <arg var='TableName'
       param='2'
       type='auto'
       offset='5'
       size='3'/>
</vars>
<rep>
  <rel>
    <target type='TABLE'
```

```
            name='%TableName'/>
    .
    .
    .
  </rel>
</rep>
```

Cobol-Specific Usage

For Cobol, use the following data types in the <vars> tag:

- *data* extracts a subarea of the parameter as raw byte data. You must specify the size and offset.
- *auto* automatically determines the type of the variable, using the offset. If that is not possible, *auto* looks for a matching variable declaration and uses its type. You must specify the offset.
- *int* behaves as *auto*, additionally checking that the resulting value is a valid integer and converting it to the canonical form. Offset defaults to 0.

Note: *bitoffset* and *bitsize* are currently not supported. *auto* is not always reliable. Use *data* whenever possible.

PL/I-Specific Usage

For PL/I, use the following data types in the <vars> tag:

- *data* extracts a subarea of the parameter as raw byte data. You must specify the size and offset.
- *char* specifies a character variable, with attribute *varying* if the string is encoded as a varying-length string in the structure. Offset defaults to 0, and size is specified via the required *len* attribute, which specifies the string length.
- *bit* specifies a bit string variable, with attribute *varying* if the string is encoded as a varying-length string in the structure. Offset defaults to 0, and size is specified via the required *len* attribute, which specifies the string length in bits.
- *fixed* specifies a fixed-point numeric variable, with attributes *mode* (binary or decimal, the default), *scale* (default 0), and *prec* (precision, default 5). Offset defaults to 0, and size is overridden with a value calculated from the type.
- *float* specifies a floating-point numeric variable, with attributes *mode* (binary or decimal, the default) and *prec* (precision, default 5). Offset defaults to 0, and size is overridden with a value calculated from the type.

Note: Do not use *bitoffset* and *bitsize* for types other than bit string.

Using the rep and hc Tags

Use the <rep> tag to represent the API call in the object model repository. Use the <hc> tag and the <attr> subtag to represent the construct model (HyperCode) attributes of entities defined in the call.

You can use predefined or custom patterns to specify the relationship of interest. Expressions let you extract parameter values and context information for use in specifications of entity or relationship characteristics.

Using Predefined Patterns

Most repository entities can be represented with the predefined patterns in \<*Workbench Home*>\Data \Legacy.xml.api. These patterns do virtually all of the work of the specification for you, supplying the

relationship of the entity to the called program, its internal name, and so forth. They are guaranteed to provide consistent data to workbench databases.

To specify a predefined pattern, use the pattern name as a tag (for example, <tip-file>) anywhere you might use a <rel> tag. If the predefined pattern is specified at the top level of the entry, the parser creates a relationship with the calling program. If the predefined pattern is nested in an entity specification, the parser creates a relationship with the parent entity.

Each pattern has parameters that you can code as XML attributes or as subtags. So:

```
<transaction name='%2' params='' hc-kind='dpsSETRX'/>
```

Is equivalent to:

```
<transaction params=''>
  <name value='%2'/>
  <hc-kind value ='dpsSETRX'/
</transaction>
```

Use the subtag method when a parameter can have multiple values:

```
<file filename= '%2' data-record='%3'>
  <action switch-var='%op'>
    <case eq='1' value='Reads'/>
    <case eq='2' value='Reads'/>
    <case eq='4' value='Updates'/>
    <case eq='28' value='Inserts'/>
  </action>
  <hc-kind switch-var='%op'>
    <case eq='1' value='fcssRR'/>
    <case eq='2' value='fcssRL'/>
    <case eq='4' value='fcssWR'/>
    <case eq='28' value='fcssAW'/>
  </hc-kind>
</file>
```

Check Legacy.xml.api for further details of predefined pattern usage and for required parameters and values.

Using Custom Patterns

Use custom patterns only when a predefined pattern is not available. Custom patterns are not guaranteed to provide consistent data to workbench databases.

Using the entity Subtag

The <entity> subtag represents an entity in the object model repository. Attributes are:

- *type* specifies the entity type.
- *name* specifies the entity name.
- *produced* optionally indicates whether the entity is extracted, in which case it is deleted from the repository when the source file is invalidated (yes or no, the default).

Use the <attr> subtag to specify entity attributes. Attributes of the subtag are:

- *name* specifies the attribute name.
- *value* contains an expression that defines the attribute value.
- *join* specifies the delimiter to use if all possible variable values are to be joined in a single value.

Use the <cond> subtag to specify a condition.

Using the rel Subtag

The <rel> subtag represents a relationship in the object model repository. Attributes are:

- *name* specifies the relationship end name, which can be unrolled into a separate tag like the name or type of an entity.
- *decision* specifies a decision.

The <target> and <source> subtags represent, respectively, the right and left ends of the relationship. These subtags are equivalent in function and syntax to the <entity> tag. Use the <cond> subtag to specify a condition.

Note: As a practical matter, you will almost never have occasion to use the <entity> subtag.

If the <rel> subtag is specified at the top level of the entry, and no <source> tag is specified, the parser creates the relationship with the calling program; otherwise, it creates the relationship between the <source> and <target> entities. If the <rel> subtag is nested in an entity specification, the parser creates the relationship with the parent entity.

Example

Assume that we know that the second parameter in the API call described earlier for the <vars> tag contains a variable in bytes 1-3 that specifies the CRUD operation, in addition to the variable in bytes 6-8 specifying the table name. The following definition extracts the CRUD operation and table name:

```
<vars>
  <arg var='OpName'
       param='2'
       type='data'
       offset='0'
       size='3'/>
  <arg var='TableName'
       param='2'
       type='auto'
       offset='5'
       size='3'/>
</vars>
<rep>
  <rel>
    <target type='TABLE'
            name='%TableName'/>
    <name switch-var='OpName'>
      <case eq='RED' value='ReadsTable'/>
      <case eq='UPD' value='UpdatesTable'/>
      <case eq='ADD' value='InsertsTable'/>
      <case eq='DEL' value='DeletesTable'/>
    </name>
  </rel>
</rep>
```

Using Expressions

Expressions let you extract parameter values and context information for specifications of entity or relationship characteristics. You can use simple variable names in expressions, or apply a primitive function call to a variable.

Basic Usage

Use the %*var_name* or %*parameter_number* notation to define variables for parameter values. The number corresponds to the index of the parameter; parameters are indexed beginning with 1. Negative numbers index from the last parameter to the first.

Variables with names beginning with an underscore are reserved for special values. They generally have only one value. The table below describes the reserved variable names.

Name	Description
_line, _col	Line and column numbers of the call in source code.
_file	Index of the file in the file table.
_uid	UID of the node of the call in the syntax tree.
_fail	A permanently undefined variable. Use it to cause explicit failure.
_yes	A non-empty string for use as a true value.
_no	An empty string for use as a false value.
_pgmname	Name of the calling program.
_hcid	HyperCode ID of the call node.
_varname nn	If parameter number nn is passed using a simple variable reference (not a constant or an expression), this substitution variable contains its name. Otherwise, it is undefined.

Simple Notation

The simplest way to use a variable is to include it in an attribute value, prefixed with the percent character (%). (%% denotes the character itself.) If the character directly after the % is a digit or a minus sign (-), the end of the variable name is considered to be the first non-digit character. Otherwise, the end of the name is considered to be the first non-alphanumeric, non-underscore character. In:

```
'%abc.%2def'
```

the first variable name is abc and the second is 2. It is also possible to specify the end of the variable name explicitly by enclosing the name in curly brackets:

```
'%{abc}.%{2}def'
```

When evaluated, a compound string like this produces a string value that concatenates variable values and simple text fragments, or fails if any of the variables is undefined.

Switch Usage

Use a *switch-var* attribute instead of the *value* attribute when a tag expects a value with a compound string expression. The *switch-var* attribute contains a single variable name (which may be prefixed by %, but

cannot be enclosed in curly brackets). Use <case>, <undef>, or <default> subtags to specify switch cases. These tags also expect the *value* attribute, so switches can be nested:

```
<name switch-var='var'>
  <case eq='value1' value='...'/>
  <case eq='value2' switch-var='%var2'>
    <undef value='...'/>
  </case>
  <undef value='...'/>
  <default value='...'/>
</name>
```

When a switch is evaluated, the value of the variable specified using the *switch-var* attribute is matched against the literal specified in the <case> tags. The literal must be the same size as the variable. (The literals **`value1`** and **`value2`** in the example assume that **`var`** is defined as having six bytes.) If an appropriate case is found, the corresponding case value is evaluated.

Set the value and type of the variable to `__fail__` (two underscores) if you do not want a relationship to be produced for the variable:

```
<progname switch-var="%fname">
  <case eq="LINKDATA" value="%pname" type="Calls" params="%3" />
  <case eq="LINK" value="%pname" type="Calls" params="%3" />
  <case eq="READQTS" value="__fail__" type="__fail__" />
  <default value="" />
  </progname>
```

If the variable is undefined, and the <undef> tag is specified, its value is used; if not, the switch fails. Otherwise, if the <default> case is specified, it is used; if not, the switch fails.

Fallback Chain Usage

Whenever multiple tags specifying a single attribute are presented in a <name>, <type>, or <case>/<undef>/<default> specification, those tags are joined into a *fallback chain*. If an entry in the chain fails, evaluation proceeds to the next entry. Only when the last entry of the chain fails is the failure propagated upward:

```
<name value='%a'/>
<name value='%b'/>
<name value='UNKNOWN'/>
```

If %a is defined, the name is its value. Otherwise, if %b is defined, the name is %b. Finally, if both are undefined, the name is UNKNOWN.

Fallback Semantics for Attributes

To determine the value of an attribute, the <attr> definitions for that attribute are processed one by one in order of appearance within the parent tag. For each definition, all combinations of variables used within it are enumerated, and all non-fail values produced are collected into a set:

- If the set contains exactly one value, it is taken as the value of the attribute.
- If the set contains multiple values, and the <attr> tag has a *join* attribute specified, the values are concatenated using the value of the *join* attribute as a delimiter, and the resulting string is used as the value for the repository attribute.
- Otherwise, the definition fails, and the next definition in the sequence is processed. If there are no definitions left, the attribute is left unspecified.

This behavior provides a way to determine if the variable has a specific value in its value set. The following example sets the attribute to False if the first parameter can be undefined, to True otherwise:

```
<attr name='Completed' switch-var='1'>
  <undef value='False'/>
</attr>
<attr name='Completed' value='True'/>
```

Using a Function Call

When a variable name contains commas, it is split into a sequence of fragments at their boundaries, and then interpreted as a sequence of function names and their parameters. In the following example:

```
%{substr,0,4,myvar}
```

the substr function extracts the first four characters from the value of %myvar. The table below describes the available functions.

Functions can be nested by specifying additional commas in the last argument of the preceding function. In the following example:

```
%{int,substr,0,4,map}
switch-var='trim,substr,4,,map'
```

the first line takes the first four characters of the variable and converts them to a canonical integer, the second line takes the remainder, removes leading and trailing spaces, and uses the result in a switch, and so forth.

Function	Description
substr,<start>,<size>,<variable>	Extracts a substring from the value of the variable. The substring begins at position <start> (counted from 0), and is <size> characters long. If <size> is an empty string, the substring extends up to the end of the value string.
int,<variable>	Interprets the value of the variable as an integer and formats it in canonical form, without preceding zeroes or explicit plus (+) sign. If the value is not an integer, the function fails.
trim,<variable>	Removes leading and trailing spaces from a string value of the variable.
const,<string> or =,<string>	Returns the string as the function result.
warning,<id-num>[,<variable>]	Produces the warning specified by <id-num>, a numeric code that identifies the warning in the backend.msg file, and returns the value of the variable. If the variable is not specified, the function fails. So %{warning,12345} is equivalent to %{warning,12345,_fail}.

Understanding Enumeration Order

If the definition of the name of a relationship or the name or type of an entity contains substitution variables that have several possible values, the parser enumerates the possible combinations. The loops are performed at the boundary of the innermost <entity> or <rel> tag that contains the reference. (Loops for the target or source are raised to the <rel> level.)

Once the value for a variable has been established at a loop insertion point, it is propagated unchanged to the tags within the loop tag. So an entity attribute specification that refers to a variable used in the name of the entity will always use the exact single value that was used in the name.

If the expression for a name or type fails, the specified entity or relationship is locked out from processing for the particular combination of values that caused it to fail. This behavior can be used to completely block whole branches of entity/relationship definition tags:

```
<entity ...>
  <type switch-var='a'>
    <case eq='1' value='TABLE'/>
  </type>
  <rel name='InsertsTable' ..../>
</entity>
<entity ...>
  <type switch-var='a'>
    <case eq='2' value='MAP'/>
  </type>
  <rel name='Sends'..../>
</entity>
```

If %a is 1, the first declaration tree is used, and the table relationship is generated; the second declaration is blocked. If %a is 2, the second declaration tree is used, and the map relationship is generated; the first declaration is blocked.

Note: These enumeration rules require that the value of a repository entity attribute not depend on variables used in the name of an enclosing <rel> tag, unless that variable is also used in the name of the entity itself. Otherwise, the behavior is undefined.

Understanding Decisions

A *decision* is a reference to another object (a program or screen, for example) that is not resolved until run time. If there are multiple possible combinations of values of variables used in the name of the target entity, or if some of the variables are undefined, the parser creates a decision entity, replacing the named relationship with a relationship to the decision and a set of relationships from the decision to each instance of the target entity.

When you use the <rel> tag at the top level of the repository definition, you can specify a *decision* attribute that tells the parser to create a decision regardless of the number of possible values:

- yes means that a decision is created regardless of the number of possible values.
- no means that a decision is never created (multiple values results in multiple direct relationships).
- auto means that a decision is created if more than one possible value exists, and is not created if there is exactly one possible value.

Both the relationship name and the type of the target entity must be specified as plain static strings, without any variable substitutions or switches:

```
<rep>
  <rel name='ReadsDataport' decision='yes'>
    <target type='DATAPORT' name='%_pgmname.%x'/>
  </rel>
</rep>
```

Understanding Conditions

The <cond> subtag specifies a condition that governs the evaluation of declarations in its parent <entity> or <relationship> tag. The evaluation semantics of the tag follow the semantics for the <attr> tag: a non-empty string as a result indicates that the condition is true, an empty string or a failure indicates that the condition is false. Multiple <cond> tags can be specified, creating a fallback chain with <attr>-style fallback semantics.

Notice in the example given in the section on decisions that the parser creates a decision entity even when the name of the target resolves to a single value. Use a <cond> subtag in the relationship definition to avoid that:

```
<rel name='ReadsDataportDecision'>
  <cond if-multi='%x' value='%_yes'/>
  <target type='DECISION'>
    <attr name='HCID' value='%_hcid'/>
    <attr name='DecisionType' value='DATAPORT'/>
    <attr name='AKA'
          value='%_pgmname.ReadsDataport.%_varname1'/>
    <attr name='AKA'
          value='%_pgmname.ReadsDataport.'/>
    <attr name='VariableName' value='%_varname1'/>
    <attr name='Completed' if-closed='%x'
          value='True'/>
    <rel name='ResolvesToDATAPORT'>
      <target type='DATAPORT'
              name='%_pgmname.%x'/>
    </rel>
  </target>
</rel>
<rel name='ReadsDataport'>
  <cond if-single='%x' value='%_yes'/>
  <target type='DATAPORT' name='%_pgmname.%x'/>
</rel>
```

This repository definition produces the same result as the example in the section on decisions, except that no decision is created when the name of the target resolves to a single value.

_yes and _no are predefined variables that evaluate, respectively, to a non-empty and empty string for true and false, respectively. The *if-single* attribute means that the <cond> tag should be interpreted only if the specified variable has a single defined value. The *if-multi* attribute means that the <cond> tag should be interpreted if the variable has multiple values, none, or can be undefined. The *if-closed* attribute blocks the <cond> tag if the variable has an undefined value.

Note: *if-single*, *if-multi*, and *if-closed* can also be used with the <attr> tag.

Conditions have *join* set to an empty string by default, resulting in a _yes outcome if any combination of values of the variables used in switches within causes it to evaluate to _yes. If a particular condition definition should fail when some of the values evaluate to _no and others to _yes, use a yes-only='yes' attribute specification. That causes *join* to be unset, and the condition to give a non-fail outcome only when all values evaluate to _yes.

In a relationship definition, <cond> determines whether the relationship is generated. For a decision relationship, it also determines whether the decision entity should be generated.

In an entity definition, <cond> governs all attribute and subrelationship definitions in the tag, and the creation of the entity in case of a standalone entity. For an entity specified in a <target> or <source> tag, instantiation of the relationship automatically spawns the corresponding entity, meaning that a false condition on the source or target of a relationship does not prevent creation of corresponding entities.

Usage Example

The following example illustrates use of the Generic API Analysis feature:

```
<APIEntry name='Call another program'>
  <match stmt="CALL">
    <name value="INVOKEPGM"/>
  </match>
  <flow halts='no'>
    <param index='1' usage='r'/>
    <param index='*' usage='w'/>
  </flow>
  <vars>
    <arg var='a' param='2' type='bit' len='5'/>
  </vars>
  <rep>
    <rel name='CallsDecision'>
      <target type='DECISION'>
        <attr name='AKA'
              value='%_pgmname.
              Calls.INVOKEPGM(%_varname1)'/>
        <attr name='AKA'
              value='%_pgmname.
              Calls.INVOKEPGM'/>
        <attr name='DecisionType'
              value='PROGRAMENTRY'/>
        <attr name='HCID' value='%_hcid'/>
        <attr name='VariableName'
              value='%_varname1'/>
        <attr name='Completed' switch-var='1'>
          <undef value='False'/>
        </attr>
        <attr name='Completed' value='True'/>
        <rel name='ResolvesToProgramEntry'>
          target type='PROGRAMENTRY'
                 name='%1'/>
        </rel>
      </target>
    </rel>
  </rep>
  <hc>
    <attr name='test' switch-var='a' join=','
      <case eq='00101' value='X'/>
      <undef value='?'/>
      <default value='%a'/>
    </attr>
  </hc>
</APIEntry>
```

Support for IMS Aliases

The <IMSC> subtag in the <Auxiliary> section of Legacy.xml contains definitions for the standard CBLTDLI or PLITDLI programs. You can also use it to define aliases for non-standard IMS batch interfaces.

If the order of parameters in the alias program is the same as the order of parameters in the standard program, simply enter the alias name in the <Detect> and <APIEntry> tags, as follows:

```
<IMSC>
  <Cobol>
    <Detect>
      <item> 'CBLTDLI' </item>
      <item> 'MYCBLTDLI' </item>
    </Detect>
    ...
    <Process>
      <APIEntry name='IMS call'>
        <match stmt="CALL">
          <name value="CBLTDLI"/>
          <name value="MYCBLTDLI"/>
        </match>
    ...
    </Process>
  </Cobol>
</IMSC>
```

If the order of parameters in the alias program differs from the order in the standard program, you also need to specify a full API entry, using the:

- <match> tag to define the alias name and method of invocation.
- <flow> tag to characterize the program control flow.
- <ims-call> tag to specify the call parameters.

Use the definitions for CBLTDLI or PLITDLI as examples.

Attributes of <ims-call> are:

- *count* specifies the index of the parameter that contains the argument count.
- *opcode* specifies the index of the parameter that contains the operation code.
- *pcb* specifies the index of the parameter that contains the Program Control Block (PCB) pointer.
- *arg-base* specifies the index of the first data parameter, usually io-area.

Note: Alternative parameter order is allowed only for the params-num, function-code, and pcb parameters. All other parameters (io-area and ssa) must appear in the same order as they do in the standard IMS call, at the end of the parameter list.

Skip Type Usage

Use the *skip-type* attribute of the <param> subtag in the <halts> section to ensure that the optional first parameter of a Cobol IMS CALL is parsed only if necessary. If the actual parameter passed by the program in the first position has the type specified by the regular expression in *skip-type*, the parameter is filled with a dummy value and the actual value is used in the next parameter.

Note: Skip definitions are also available for use in non-IMS generic API entries.

Example

If the first parameter in a call is a 4-character picture, the following definition inserts a dummy value in the first position and treats the actual value as that of the second parameter:

```
<param index='1'
       usage='r'
       skip-type='PIC:(X\(4\)'/>
```

Note: Skip definitions are currently limited to declarations having picture clauses. Use regular expression syntax to specify normalized picture strings.

Index