

Micro Focus[®]

Modernization Workbench[™]

Creating Components



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

Copyright Micro Focus (IP) Limited. All Rights Reserved.

MICRO FOCUS, the Micro Focus logo and are trademarks or registered trademarks of Micro Focus (IP) Limited or its subsidiaries or affiliated companies in the United States, United Kingdom and other countries.

All other marks are the property of their respective owners.

Contents

Introducing Component Maker	6
Componentization Methods	6
Structure-Based Componentization	6
Computation-Based Componentization	6
Domain-Based Componentization	7
Event Injection	7
Dead Code Elimination (DCE)	7
Entry Point Isolation	7
Language Support	8
Componentization Outputs	8
Component Maker Basics	9
Getting Started in the Components Pane	10
Creating Components	12
Extracting Components	12
Converting Components	13
Deleting Components	13
Viewing the Text for Generated Files	13
Restricting the Display to Program-Related Components	13
Working with HyperView Lists	13
Viewing Audit Reports	14
Generating Coverage Reports	14
Exporting Logical Components	15
Generating CICS Components	16
Setting Component Maker Options	17
Setting General Options	17
Setting Interface Options	18
Setting Optimize Options	18
Setting Document Options	20
Setting Component Type-Specific Options	21
Setting Structure-Based Type-Specific Options	21
Setting Computation-Based Type-Specific Options	21
Setting Domain-Based Type-Specific Options	22
Setting Event Injection Type-Specific Options	23
Setting Component Conversion Options	23
Extracting Structure-Based Components	25

Understanding Ranges	25
Specifying Ranges for Cobol Programs	25
Specifying Ranges for PL/I Programs	26
Specifying Ranges for RPG Programs	26
Understanding Parameterized Slices	26
Cobol Naming Conventions	26
Parameterization Example	26
Extracting Structure-Based Cobol Components	27
Extracting Structure-Based PL/I Components	29
Extracting Structure-Based RPG Components	29
Extracting Computation-Based Components	31
Understanding Variable-Based Extraction	31
Understanding Blocking	31
Understanding Parameterized Slices	32
Cobol Naming Conventions	32
Parameterization Example	32
Extracting Computation-Based Cobol Components	33
Extracting Computation-Based Natural Components	34
Extracting Domain-Based Components	35
Understanding Program Specialization in Simplified Mode	35
Understanding Program Specialization in Advanced Mode	37
Understanding Program Specialization Lite	37
Extracting Domain-Based Cobol Components	38
Extracting Domain-Based PL/I Components	39
Injecting Events	41
Understanding Event Injection	41
Extracting Event-Injected Cobol Components	42
Eliminating Dead Code	44
Generating Dead Code Statistics	44
Understanding Dead Code Elimination	44
Extracting Optimized Components	45
Performing Entry Point Isolation	46
Extracting a Cobol Component with Entry Point Isolation	46
Technical Details	47
Verification Options	47
Use Special IMS Calling Conventions	47
Override CICS Program Terminations	47
Support CICS HANDLE Statements	47
Perform Unisys TIP and DPS Calls Analysis	48

Perform Unisys Common-Storage Analysis	48
Relaxed Parsing	48
PERFORM Behavior for Micro Focus Cobol	49
Keep Legacy Copybooks Extraction Option	49
How Parameterized Slices Are Generated for Cobol Programs	51
Setting a Specialization Variable to Multiple Values	52
Arithmetic Exception Handling	53

Introducing Component Maker

The Modernization Workbench Component Maker tool offers a variety of advanced algorithms for *slicing* logic from program source: all the code you need for a computation, for example, or the code you need to "specialize" a program based on the value of a variable. You can create a self-contained program, called a component, from the sliced code or simply generate a HyperView list of sliced constructs for further analysis. You can mark and colorize the constructs in the HyperView Source pane.

Both the component generation and list functions are supported in the full version of the Component Maker tool available to users of Application Architect. The list function only is supported in the restricted version of Component Maker, called Logic Analyzer, available to users of Application Analyzer.

Componentization Methods

The supported componentization methods slice logic not only from program executables but associated include files as well. Dead Code Elimination and Entry Point Isolation are optimization tools built into the main methods and offered separately in case you want to use them on a standalone basis.



Note: Component Maker does not follow CALL statements into other programs to determine whether passed data items are actually modified by those programs. It makes the conservative assumption that all passed data items are modified. That guarantees that no dependencies are lost.

Structure-Based Componentization

Structure-Based Componentization lets you build a component from a range of inline code, Cobol paragraphs, for example. Use traditional structure-based componentization to generate a new component and its *complement*. A complement is a second component consisting of the original program minus the code extracted in the slice. Component Maker automatically places a call to the new component in the complement, passing it data items as necessary.

For Cobol programs, you can generate *parameterized slices*, in which the input and output variables required by the component are organized in group-level structures. These standard object-oriented data interfaces make it easier to deploy the transformed component in modern service-oriented architectures.



Tip: You typically repeat structure-based componentization in incremental fashion until all of the modules you are interested in have been created. For Cobol programs, you can avoid doing this manually by specifying multiple ranges in the same extraction. Component Maker automatically processes each range in the appropriate order.

Computation-Based Componentization

Computation-Based Componentization lets you build a component that contains all the code necessary to calculate the value of a variable at a point in the program where it is used to populate a report attribute or screen. As with structure-based componentization, you can generate parameterized slices that make it easy to deploy the transformed component in distributed architectures.

For Cobol programs, you can use a technique called *blocking* to produce smaller, better-defined parameterized components. Component Maker will not include in the slice any part of the calculation that appears before the blocked statement. Fields from blocked input statements are treated as input parameters of the component.

Domain-Based Componentization

Domain-Based Componentization lets you "specialize" a program based on the values of one or more variables. The specialized program is typically intended for reuse "in place," in the original application, but under new external circumstances.

After a change in your business practices, for example, a program that invokes processing for a "payment type" variable could be specialized on the value `PAYMENT-TYPE = "CHECK"`. Component Maker isolates every process dependent on the `CHECK` value to create a functionally complete program that processes check payments only.

Two modes of domain-based componentization are offered:

- In *simplified mode*, you set the specialization variable to its value anywhere in the program *except* a data port. The value of the variable is "frozen in memory." Operations that could change the value are ignored.
- In *advanced mode*, you set the specialization variable to its value at a data port. Subsequent operations can change the value, following the data and control flow of the program.

Use the simplified mode when you are interested only in the final value of a variable. Use the advanced mode when you need to account for data coming into a variable.

Event Injection

Event Injection lets you adapt a legacy program to asynchronous, event-based programming models like MQ Series. You specify candidate locations for event calls (reads/writes, screen transactions, or subprogram calls, for example), the type of operation the event call performs (put or get), and the text of the message. For a put operation, for example, Component Maker builds a component that sends the message and any associated variable values to a queue, where the message can be retrieved by monitoring applications.

Dead Code Elimination (DCE)

Dead Code Elimination is an option in each of the main component extraction methods, but you can also perform it on a standalone basis. For each program analyzed for dead code, standalone DCE generates a component that consists of the original source code minus any unreferenced data items or unreachable procedural statements.

 **Note:** Use the batch DCE feature to find dead code across your project. If you are licensed to use the Batch Refresh Process (BRP), you can use it to perform dead code elimination across a workspace.

Entry Point Isolation

Entry Point Isolation lets you build a component based on one of multiple entry points in a legacy program (an inner entry point in a Cobol program, for example). Component Maker extracts only the functionality and data definitions required for invocation from the selected point.

Entry Point Isolation is built into the main methods as an optional optimization tool. It's offered separately in case you want to use it on a standalone basis.

Language Support

The following table describes the extraction methods available for Component Maker-supported languages.

Method	COBOL	PL/I	Natural	RPG
Structure-based	Yes	Yes	No	Yes
Computation-based	Yes	No	Yes	No
Domain-based	Yes	Yes	No	No
Event-Injection	Yes	No	No	No
Dead Code Elimination	Yes	Yes	Yes	Yes
Entry Point Isolation	Yes	No	No	No

Componentization Outputs

The first step in the componentization process, called *extraction*, generates the following outputs:

- The source file that comprises the component.
- An abstract repository object, or *logical component*, that gives you access to the source file in the workbench.
- A HyperView list of sliced constructs, which you can mark and colorize in the HyperView Source pane.



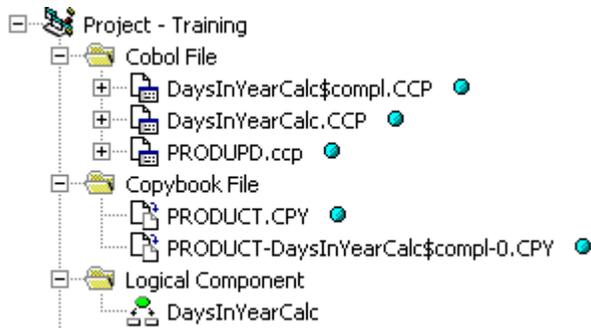
Note: For Logic Analyzer, sliced data declarations are not marked and colorized.

The second step, called *conversion*, registers the source files in your repository, creating repository objects for the generated components and their corresponding copybooks.

Component Maker lets you execute the extraction and conversion steps independently or in combination, depending on your needs:

- If you want to analyze the components further, transform them, or even generate components from them, you will want to register the component source files in your repository and verify them, just as you would register and verify a source file from the original legacy application.
- If you are interested only in deploying the components in your production environment, you can skip the conversion step and avoid cluttering your repository.

The figure below shows how the componentization outputs are represented in the Repository Browser after conversion and verification of a structure-based Cobol component called DaysInYearCalc. PRODUPD is the program the component was extracted from.



Component Maker Basics

Component Maker is a HyperView-based tool that you can invoke on a standalone basis or from within HyperView itself:

- Start the tool in HyperView by selecting the program you want to slice in the Modernization Workbench Repository Browser and choosing **Analyze > Interactive Analysis**. In the HyperView window, choose **View > Components**.



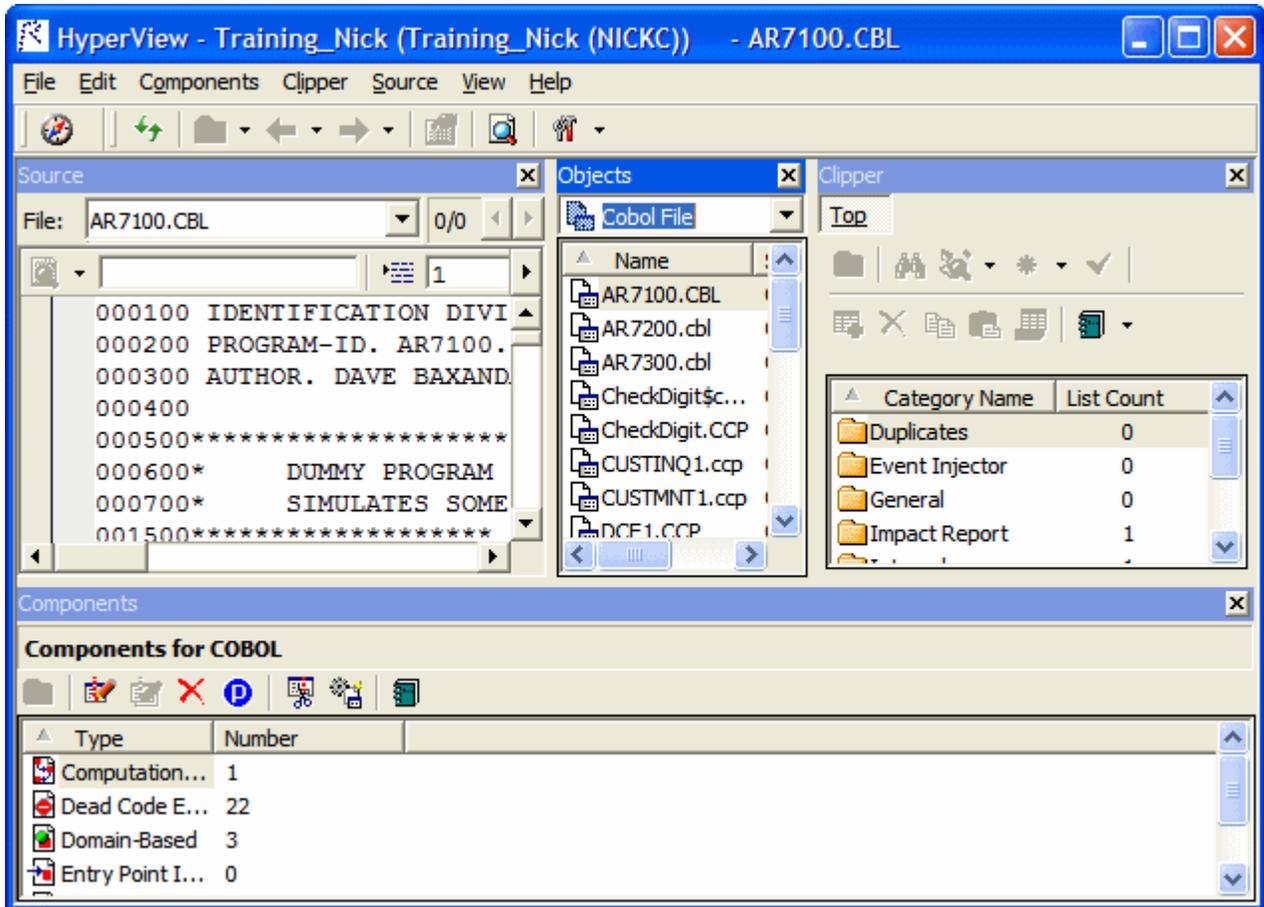
Note: Choose **View > Logic Analyzer** if you are using Logic Analyzer.

- Start the standalone tool by selecting the project that contains the programs you want to slice in the Repository Browser and choosing **Architect > Logical Components**. In the HyperView window, select the program you want to slice in the Objects pane.

The Components pane consists of a hierarchy of views that let you specify the logical components you want to manipulate:

- The Types view lists the types of logical components you can create: structure-based, computation-based, domain-based, and so on.
- The List view displays logical components of the selected type.
- The Details view displays the details for the selected logical component in two tabs, Properties and Components. The Properties tab displays extraction properties for the logical component. The Components tab lists the files generated for the logical component.

The figure below shows a typical configuration of the Component Maker window. For HyperView usage, see *Analyzing Programs* in the workbench documentation set.



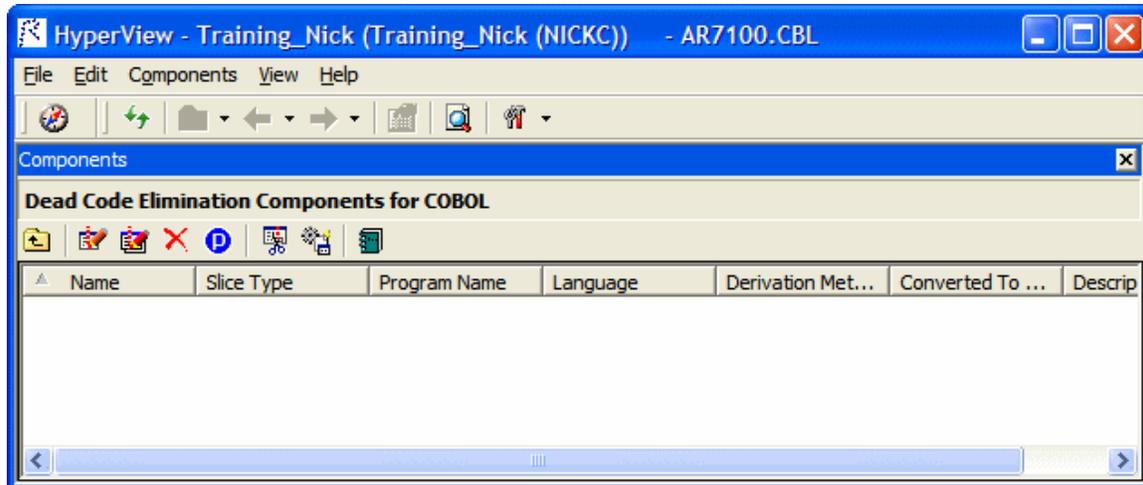
Getting Started in the Components Pane

You do most of your work in Component Maker in the Components pane. To illustrate how you extract a logical component in the Components pane, let's look at the simplest task you can perform in Component Maker, Dead Code Elimination (DCE).

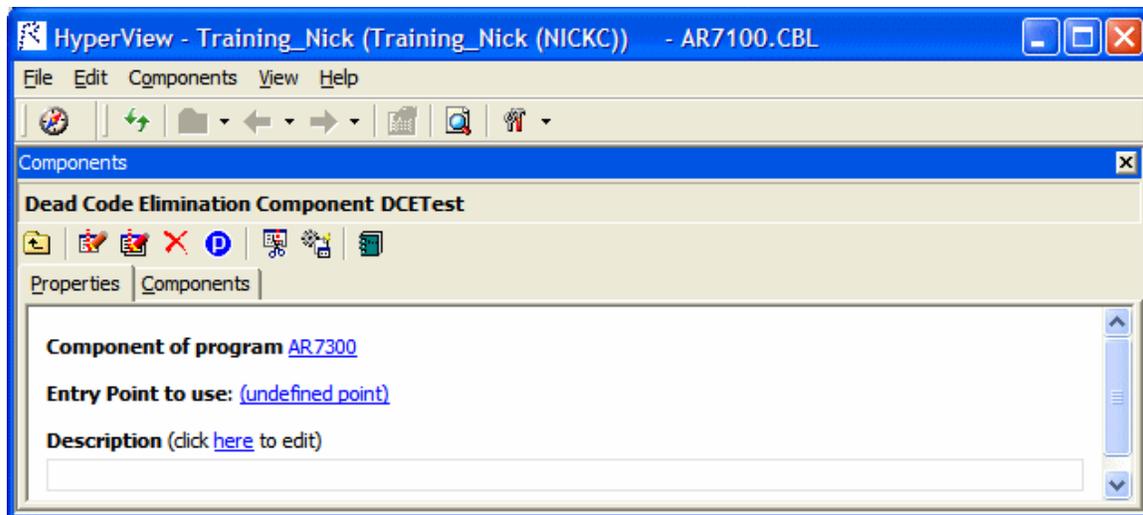
 **Note:** The following exercise deliberately avoids describing the properties and options you can set for DCE. See the relevant help topics for details.

1. In the Components pane, double-click **Dead Code Elimination**. The view shown in the figure below opens. This view shows the DCE-based logical components created for the programs in the current project.

 **Tip:** Click the  button on the tool bar to restrict the display to logical components created for the selected program.



2. Select the program you want to analyze for dead code in the HyperView Objects pane and click the  button. To analyze the entire project of which the program is a part, click the  button.
3. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new components to the list of components. If you selected batch mode, Component Maker creates a logical component for each program in the project, appending *_n* to the name of the component.
4. Double-click a component to edit its properties. The view shown in the figure below opens. The **Component of program** field contains the name of the selected program.

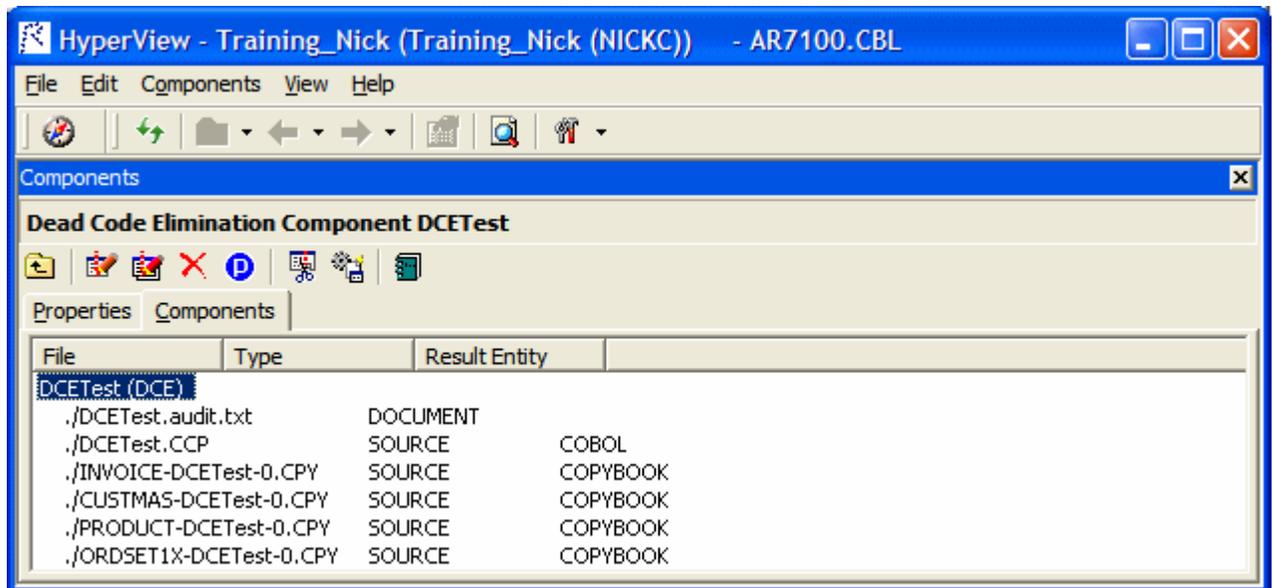


5. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.

 **Note:** This field is shown only for Cobol programs.

6. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.

7. Click the  button on the tool bar to navigate to the list of components, then repeat the procedure for each component you want to extract.
8. In the list of components, select each component you want to extract and click the  button on the tool bar. You are prompted to confirm that you want to extract the components. Click **OK**.
9. The Extraction Options dialog opens. Set extraction options as described in the relevant help topic. When you are satisfied with your choices, click **Finish**.
10. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** to view the errors or warnings in the Activity Log. Otherwise, click **No**.
11. Assuming the extraction executed without errors, the view shown in the figure below opens. Click the Components tab to display a list of the component source files that were generated for the logical component and an audit report if you requested one. Click an item in the list to view the read-only text for the item.



Creating Components

To create a component, select the program you want to slice in the HyperView Objects pane. In the Types view, select the type of logical component you want to create and click the  button on the tool bar. (You can also click the  button in the List or Details view.) A dialog opens where you can enter the name of the new component in the text field. Click **OK**.

Extracting Components

To extract a single logical component, select the component you want to extract in the List view and click the  button on the tool bar. To extract multiple logical components, select the type of the components

you want to extract in the Types view and click the  button. You are prompted to confirm that you want to continue. Click **OK**.

 **Tip:** Logical components are converted as well as extracted if the **Convert Resulting Components to Legacy Objects** is set in the Component Conversion Options pane.

Converting Components

To convert a single logical component, select the component you want to convert in the List view and click the  button on the tool bar. To convert multiple logical components, select the type of the components you want to convert in the Types view and click the  button. You are prompted to confirm that you want to continue. Click **OK**.

Deleting Components

To delete a logical component, select it in the List view and click the  button on the tool bar.

 **Note:** Deleting a logical component does not delete the component and copybook repository objects. You must delete these objects manually in the Repository Browser.

Viewing the Text for Generated Files

To view the read-only text for a generated file, click the file in the list of generated files for in the Components tab.

 **Tip:** You can also view the text for a generated file in the Modernization Workbench main window. In the Repository Browser Logical Component folder, click the component whose generated files you want to view.

Restricting the Display to Program-Related Components

To restrict the display to logical components of a given program, select the program and click the  button on the tool bar. The button is a toggle. Click it again to revert to the generic display.

Working with HyperView Lists

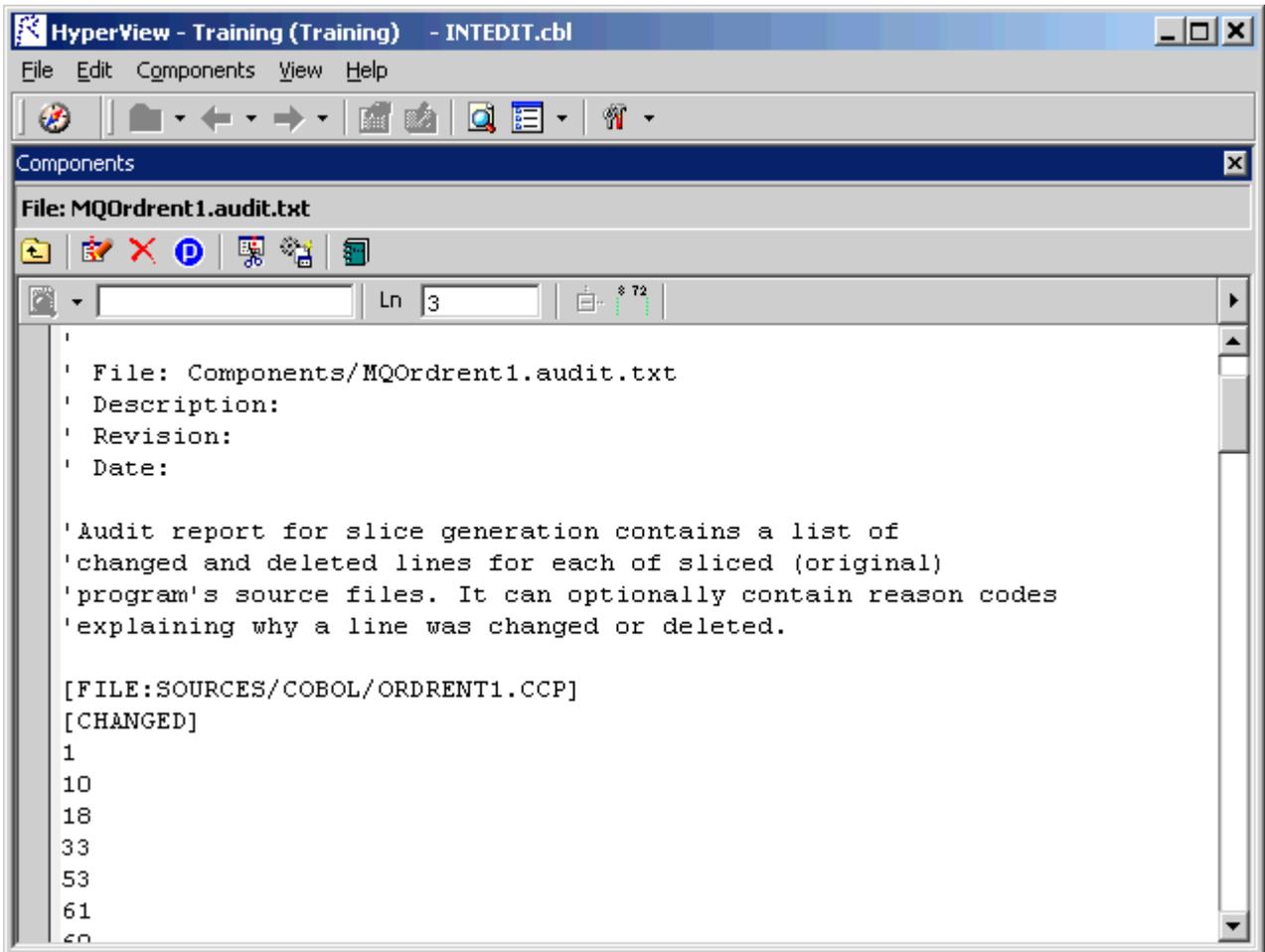
When you extract a logical component, Component Maker generates a HyperView list of sliced constructs. The list has the same name as the component. You can view the list in the Logic Analyzer folder in Clipper.

To mark and colorize sliced constructs in the list, select the list in Clipper and click the  button on the tool bar. To mark and colorize sliced constructs in a single file, select the file in the List view and click the  button. To mark and colorize a single construct, select it in the File view and click the  button. Click the  button again to turn off marking and colorizing.

Viewing Audit Reports

An *audit report* contains a list of changed and deleted lines in the source files (including copybooks) from which a logical component was extracted. The report has a name of the form *<component>.audit.txt*. Click the report in the Components tab to view its text.

An audit report optionally includes reason codes explaining why a line was changed or deleted. A reason code is a number keyed to the explanation for a change (for example, reason code 12 for computation-based componentization is RemoveUnusedVALUES).



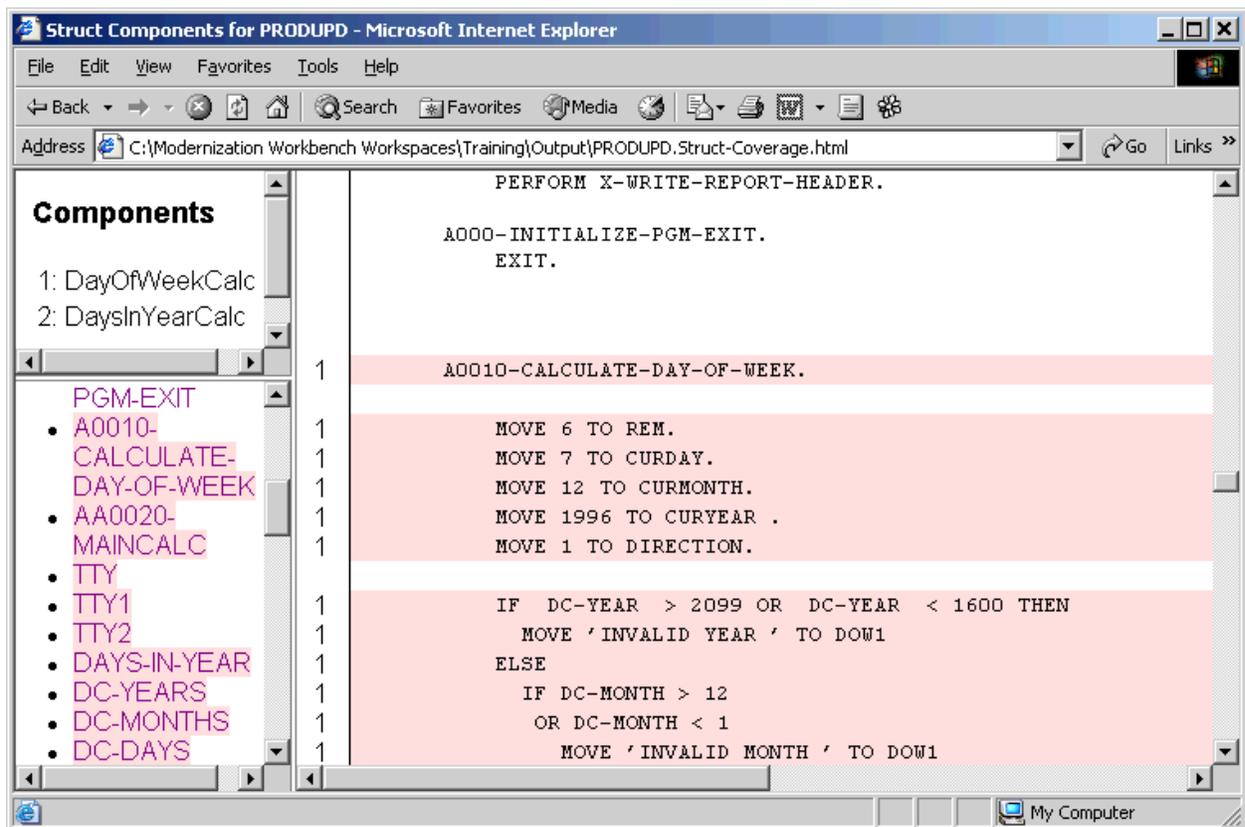
Generating Coverage Reports

A *coverage report* shows the extent to which a source program has been "componentized":

- The top-left pane lists each component of a given type (structure-based, computation-based, and so on) extracted from the program.
- The bottom-left pane lists the paragraphs in the program. Click on a paragraph to navigate to it in the righthand pane.
- The righthand pane displays the text of the program with extracted code shaded in pink. The numbers to the left of the extracted code identify the component to which it was extracted.

To generate coverage reports, click the  button on the Component Maker tool bar. The reports are listed in the Generated Document folder in the Repository Browser. Report names are of the form *<program>-<method>-Coverage*. Double-click a report to view it in a Web browser.

 **Note:** Reports are created for each program in the current project.



Exporting Logical Components

To move or copy the source files associated with a logical component (including any complement or copybooks) from the Modernization Workbench area to a specified location on your file system, select the component in the Repository Browser Logical Component folder and choose **Architect > Export**. A standard dialog appears where you can specify the destination for the source files. Select **Move files** if you want to move the files rather than copy them.

Generating CICS Components

Component Maker let you generate structure- and computation-based Cobol components as CICS programs, with COMMAREAS for parameter exchange. That means the component can be called through a CICS LINK or by some other middleware such as IBM's ECI.

A CICS component can be run directly on mainframes:

- The component's parameters, whether original (from USING) or created by Component Extraction, are packaged under the CICS variable DFHCOMMAREA. There is no PROCEDURE DIVISION USING phrase in the component.
- At all program points where the original program could exit, the component exits through a CICS RETURN statement. Any STOP RUN is replaced by CICS RETURN.

To generate CICS components, choose **Create CICS Program** in the interface component extraction options.

Setting Component Maker Options

It's a good idea to become familiar with the component extraction options before beginning your work in Component Maker. Each extraction method has a different set of options, and each set differs for the supported object types. Extraction options are project-based, so they apply to every program in the current Modernization Workbench project. Only relevant options are displayed for the restricted version of Component Maker, called Logic Analyzer, available to users of Application Analyzer.

 **Note:** For computation- and domain-based componentization of Cobol programs, and for structure-based componentization with parameterized slices, you must set **Perform Program Analysis** in the project verification options before verifying the program you want to slice.

You can set Component Maker extraction options in the standard Project Options window or in the extraction options dialog that opens when you create a component. To open the standard Project Options window, choose **Tools > Project Options**. In the Project Options window, click the Component Maker tab.

Setting General Options

The table below describes the Component Maker General extraction options.

Option	Language	Description
Add Program Name as Prefix	Cobol, Natural, PL/I, RPG	Prepend the name of the sliced program to the component name you specified when you created the component, in the form <i><program> \$<component></i> .
Generate Slice	Cobol, Natural, PL/I, RPG	Generate both a HyperView list of sliced constructs and a component.
Keep Legacy Copybooks	Cobol, RPG	Do not generate modified copybooks for the component. Modified copybooks have names of the form <i><copybook>-<component>-n</i> , where <i>n</i> is a number ensuring the uniqueness of the copybook name when multiple instances of a copybook are generated for the same component.  Note: Component Maker issues a warning if including the original copybooks in the component would result in an error.
Keep Legacy Includes	PL/I	Do not generate modified program include files for the component. The layout and commentary of the sliced program is preserved.
Keep Legacy Macros	PL/I	Do not expand macros for the component. The layout and commentary of the sliced program is preserved.
Preserve Legacy Includes	Natural	Do not generate modified program include files for the component.

Option	Language	Description
Rename Program Entries	Cobol	Prepend the name of the component to inner entry points, in the form <code><component>-<entrypoint></code> . This ensures that entry point names are unique and that the Modernization Workbench parser can verify the component successfully. Unset this option if you need to preserve the original names of the inner entry points.

Setting Interface Options

The table below describes the Component Maker Interface extraction options.

Option	Language	Description
Blocking	Cobol	If you are performing a parameterized computation-based extraction and want to use blocking, click the More button. A dialog opens, where you can select the blocking option and the types of statements you want to block.  Note: Choose Use Blocking from Component Definitions if you want to block statements in a HyperView list.
Create CICS Program	Cobol	Create COMMAREAS for parameter exchange in generated slices.
Generate Parameterized Components	Cobol	Extract parameterized slices.  Note: If you select this option for a structure-based extraction, you must set the Range Only option in the Component Type Specific pane.

Setting Optimize Options

The table below describes the Component Maker Optimize extraction options.

Option	Language	Description
No changes	Cobol, Natural, RPG	Do not remove unused data items from the component.
Preserve Original Paragraphs	Cobol	Generate paragraph labels even for paragraphs that are not actually used in the source code (for example, empty paragraphs for which there are no PERFORMs).  Note: This option also affects refactoring. When the option is set, paragraphs in the

Option	Language	Description
		<p>same "basic block" are defragmented separately. Otherwise, they are defragmented as a unit.</p>
Remove Redundant NEXT SENTENCE	Cobol	<p>Remove NEXT SENTENCE clauses by changing the bodies of corresponding IF statements, such that:</p> <pre data-bbox="922 464 1464 604"> IF A=1 NEXT SENTENCE ELSE ... END-IF.</pre> <p>is generated as:</p> <pre data-bbox="922 663 1464 747"> IF NOT (A=1) ... END-IF.</pre>
Remove/Replace Unused Fields with FILLERS	Cobol, Natural, RPG	<p>Remove unused any-level structures and replace unused fields in a used structure with FILLERS. Set this option if removing a field completely from a structure would adversely affect memory distribution.</p> <p> Note: If you select Keep Legacy copybooks in the General component extraction options, Component Maker removes or replaces with FILLERS only unused inline data items.</p>
Remove Unreachable Code	Cobol, RPG	Remove unreachable procedural statements.
Remove Unused Any-Level Structures	Cobol, Natural, RPG	<p>Remove unused structures at any data level, if all their parents and children are unused. For the example below, D, E, F, and G are removed:</p> <pre data-bbox="922 1251 1464 1476"> DEFINE DATA LOCAL 1 #A 2 #B 3 #C 2 #D 3 #E 3 #F 1 #G</pre>
Remove Unused Level-1 Structures	Cobol, Natural, RPG	<p>Remove only unused level-1 structures, and then only if all their children are unused. If, in the following example, only B is used, only G is removed:</p> <pre data-bbox="922 1633 1464 1801"> DEFINE DATA LOCAL 1 #A 2 #B 3 #C 2 #D 3 #E</pre>

Option	Language	Description
		<pre> 3 #F 1 #G </pre>
Replace Section PERFORMs by Paragraph PERFORMs	Cobol	Replace PERFORM section statements by equivalent PERFORM paragraph statements.
Roll-Up Nested IFs	Cobol	<p>Roll up embedded IF statements in the top-level IF statement, such that:</p> <pre> IF A=1 IF B=2 </pre> <p>is generated as:</p> <pre> IF (A=1) AND (B=2) </pre>

Setting Document Options

The table below describes the Component Maker Document extraction options.

Option	Language	Description
Comment-out Sliced-off Legacy Code	Cobol, RPG	Retain but comment out unused code in the component source. In the Comment Prefix field, enter descriptive text (up to six characters) for the commented-out lines.
Emphasize Component/Include in Coverage Report	Cobol, Natural, PL/I, RPG	Generate a HyperView list of sliced constructs and colorize the constructs in the Coverage Report.
Generate Audit Report	Cobol	Generate an audit report.
Generate Support Comments	Cobol, RPG	Include comments in the component source that identify the component properties you specified, such as the starting and ending paragraphs for a structure-based Cobol component.
Include Reason Codes	Cobol	<p>Include reason codes in the audit report explaining why a line was changed or deleted.</p> <p> Note: Generating reason codes is very memory-intensive and may cause crashes for extractions from large programs.</p>
List Options in Component Header and in Separate Document	Cobol, RPG	Include a list of extraction option settings in the component header and in a separate text file. The text file has a name of the form <code><component>.BRE.options.txt</code> .
Mark Modified Legacy Code	Cobol, RPG	Mark modified code in the component source. In the Comment Prefix field, enter descriptive text (up to six characters) for the modified lines.
Print Calculated Values as Comments	Cobol	For domain-based component extraction only, print the calculated values of variables as comments. Alternatively, you can substitute the

Option	Language	Description
		calculated values of variables for the variables themselves.
Use Left Column for Marks	Cobol, RPG	Place the descriptive text for commented-out or modified lines in the lefthand column of the line. Otherwise, the text appears in the righthand column.

Setting Component Type-Specific Options

Component type-specific extraction options determine how Component Maker performs tasks specific to each componentization method.

Setting Structure-Based Type-Specific Options

The table below describes the Component Maker structure-based type-specific extraction options.

Option	Language	Description
Dynamic Call	Cobol	Generate in the complement a dynamic call to the component. The complement will call a string variable that must later be set outside the complement to the name of the component.
Ensure Consistent Access to External Resources	Cobol	Monitor the integrity of data flow in the ranges you are extracting. If you select this option, for example, an extraction will fail if an SQL cursor used in the component is open in the complement.
Range Only	Cobol	Do not generate a complement. You must set this option to generate parameterized slices.
Restrict User Ranges to PERFORMed Ones	Cobol	Do not extract paragraphs that do not have a corresponding PERFORM statement. This option is useful if you want to limit components created with the Paragraph Pair or Section methods to PERFORMed paragraphs.
Suppress Errors	Cobol	Perform a "relaxed extraction," in which errors that would ordinarily cause the extraction to fail are ignored, and comments describing the errors are added to the component source. This option is useful when you want to review extraction errors in component source.

Setting Computation-Based Type-Specific Options

The table below describes the Component Maker computation-based type-specific extraction options.

Option	Language	Description
Generate HTML Trace	Cobol	Generate an HTML file with an extraction trace. The trace has a name of the form <i><component>.trace</i> . To view the trace, click the logical component for the extraction in the Repository Browser Logical Component folder. Double-click the trace file to view it in a Web browser.
Statement	Cobol	Perform statement-based component extraction.
Variable	Cobol	Perform variable-based component extraction.  Note: Even if you select variable-based extraction, Component Maker performs statement-based extraction if the variable you slice on is not an input variable for its parent statement: that is, if the statement writes to rather than reads from the variable.

Setting Domain-Based Type-Specific Options

The table below describes the Component Maker domain-based type-specific extraction options.

Option	Language	Description
Maximum Number of Variable's Values	Cobol	The maximum number of values to be calculated for each variable. Limit is 200. The lower the maximum, the better performance and memory usage you can expect.
Maximum Size of Variable to Be Calculated	Cobol	Maximum size in bytes for each variable value to be calculated. The lower the maximum, the better performance and memory usage you can expect.
Multiple Pass	Cobol, PL/I	Evaluate conditional logic again after detecting dead branches. Because the ELSE branch of the first IF below is dead, for example, the second IF statement can be resolved in a subsequent pass: <pre>MOVE 0 TO X. IF X EQUAL 0 THEN MOVE 1 TO Y ELSE /p> MOVE 2 TO Y. IF Y EQUAL 2 THEN . . . ELSE . . .</pre>  Note: Multi-pass processing is very resource-intensive, and not recommended for extractions from large programs.
Remove Unused Assignments	Cobol, PL/I	Exclude from the component assignments that cannot affect the computation (typically, an assignment after which the variable is not used until the next assignment or port).
Remove Unused Procedures	PL/I	Exclude unused procedures from the component.

Option	Language	Description
Replace Procedure Calls by Return Values	PL/I	Substitute the return values of variables for procedure calls in components.
Replace Variables by Their Calculated Values	Cobol	<p>Substitute the calculated values of variables for the variables themselves. Alternatively, you can print the values as comments.</p> <p> Note: Notice how the options in Remove Unused Assignments and Replace Variables by Their Calculated Values can interact. If both options are set, then the first assignment in the following fragment will be removed:</p> <pre>MOVE 1 TO X. DISPLAY X. MOVE 2 TO X.</pre>
Single Pass	Cobol, PL/I	Evaluate conditional logic in one pass.
VALUEs Initialize Data Items	Cobol	Set variables declared with VALUE clauses to their initial values. Otherwise, VALUE clauses are ignored.

Setting Event Injection Type-Specific Options

The table below describes the Component Maker event injection type-specific extraction options.

Option	Language	Description
Error Handling	Cobol	The type of statement to execute in case of an error connecting to middleware.
MQ	Cobol	Use an IBM MQ Series template for event injection.
MQPUT	Cobol	Use the MQPUT method.
MQPUT1	Cobol	Use the MQPUT1 method.
Queue Manager	Cobol	The name of the queue manager.
Target Queue Name	Cobol	The name of the target queue.
User Specified Event	Cobol	The name of the event to inject at the specified injection points.

Setting Component Conversion Options

The table below describes the Component Maker Component Conversion extraction options.

Option	Language	Description
Convert Resulting Components	Cobol, Natural, PL/I, RPG	Convert as well as extract the logical component.
Keep Old Legacy Objects	Cobol, Natural, PL/I, RPG	Preserve existing repository objects for the converted component (copybooks, for example). If you select this option, delete the repository object for the component itself before performing the extraction, or the new component object will not be created.
Remove Components after Successful Conversion	Cobol, Natural, PL/I, RPG	Remove logical components from the current project after new component objects are created.
Replace Old Legacy Objects	Cobol, Natural, PL/I, RPG	<p>Replace existing repository objects for the converted component.</p> <p> Note: This option controls conversion behavior even when you perform the conversion independently from the extraction. If you are converting a component independently and want to change this setting, select Convert Resulting Components to Legacy Objects, specify the behavior you want, and then deselect Convert Resulting Components to Legacy Objects.</p>

Extracting Structure-Based Components

Structure-Based Componentization lets you build a component from a range of inline code, Cobol paragraphs, for example. Use traditional structure-based componentization to generate a new component and its *complement*. A complement is a second component consisting of the original program minus the code extracted in the slice. Component Maker automatically places a call to the new component in the complement, passing it data items as necessary.

Alternatively, you can generate *parameterized slices*, in which the input and output variables required by the component are organized in group-level structures. These standard object-oriented data interfaces make it easy to deploy the transformed component in modern service-oriented architectures.

Understanding Ranges

When you extract a structure-based component from a program, you specify the *range* of code you want to include in the component. The range varies: for Cobol programs, a range of paragraphs; for PL/I programs, a procedure; for RPG programs, a subroutine or procedure.



Tip: You typically repeat Structure-Based Componentization in incremental fashion until all the modules you are interested in have been created. For Cobol programs, you can avoid doing this manually by specifying multiple ranges in the same extraction. Component Maker automatically processes each range in the appropriate order. No complements are generated.

Specifying Ranges for Cobol Programs

For Cobol programs, you specify the paragraphs in the range for structure-based component extraction in one of three ways:

- Select a paragraph PERFORM statement to set the range to the performed paragraph or paragraphs. Component Maker includes each paragraph in the execution path between the first and last paragraphs in the range, except when control is transferred by a PERFORM statement or by an implicit RETURN-from-PERFORM statement.
- Select a pair of paragraphs to set the range to the selected paragraphs. You are responsible for ensuring a continuous flow of control from the first to the last paragraph in the range.
- Select a section to set the range to the paragraphs in the section.



Note: For traditional structure-based COBOL components, Component Maker inserts in the complement the labels of the first and last paragraphs in the range. The first paragraph is replaced in the complement with a CALL statement followed by a GO TO statement. The last paragraph is always empty.

The GO TO statement transfers control to the last paragraph. If the GO TO statement and its target paragraph are not required to ensure correct call flow, they are omitted.

Specifying Ranges for PL/I Programs

For PL/I programs, the range you specify for structure-based component extraction is an internal procedure that Component Maker extracts as an external procedure. The slice contains the required parameters for global variables.

Specifying Ranges for RPG Programs

For RPG programs, the range you specify for structure-based component extraction is a subroutine or procedure to extract as a component.

Understanding Parameterized Slices

For Cobol programs, you can generate *parameterized slices*, in which the input and output variables required by the component are organized in group-level structures. The component contains all the code required for input/output operations.

To extract a parameterized slice, select the **Generate Parameterized Components** option in the extraction options dialog. Note that you cannot generate a complement for a parameterized Cobol slice.

 **Note:** For parameterized structure- and computation-based componentization of Cobol programs, you must select the **Perform Program Analysis** and **Enable Parameterization of Components** options in the project verification options.

Cobol Naming Conventions

- Component input structures have names of the form BRE-INP-<STRUCT-NAME>. Input fields have names of the form BRE-I-<FIELD-NAME>.
- Component Output structures have names of the form BRE-OUT-STRUCT-NAME. Output fields have names of the form BRE-O-<FIELD-NAME>.

Parameterization Example

The example below illustrates how Component Maker generates parameterized slices. Consider a Cobol program that contains the following structures:

```
WORKING-STORAGE SECTION.  
  01 A  
    03 A1  
    03 A2  
  01 B  
    03 B1  
    03 B2  
    03 B4
```

Suppose that only A1 has been determined by Component Maker to be an input parameter, and only B1 and B2 to be output parameters. Suppose further that the component is extracted with input and output data

structures that use the default names, BRE-INP-INPUT-STRUCTURE and BRE-OUT-OUTPUT-STRUCTURE, respectively, and with the default Optimization options set. The component contains the following code:

```
WORKING-STORAGE SECTION.  
  01 A  
    03 A1  
    03 A2  
  01 B  
    03 B1  
    03 B2  
    03 B4  
LINKAGE SECTION.  
  01 BRE-INP-INPUT-STRUCTURE  
    03 BRE-I-A  
      06 BRE-I-A1  
  01 BRE-OUT-OUTPUT-STRUCTURE  
    03 BRE-O-B  
      06 BRE-O-B1  
      06 BRE-O-B2  
PROCEDURE DIVISION  
  USING BRE-INP-INPUT-STRUCTURE BRE-OUT-OUTPUT-STRUCTURE.  
BRE-INIT-SECTION SECTION.  
  PERFORM BRE-COPY-INPUT-DATA.  
  .....  
  ....(Business Logic)....  
  .....  
  *Modernization Workbench added statement  
  GO TO BRE-EXIT-PROGRAM.  
BRE-EXIT-PROGRAM-SECTION SECTION.  
  BRE-EXIT-PROGRAM.  
    PERFORM BRE-COPY-OUTPUT-DATA.  
    GOBACK.  
BRE-COPY-INPUT-DATA.  
  MOVE BRE-I-A TO A.  
BRE-COPY-OUTPUT-DATA.  
  MOVE B TO BRE-O-B.
```

Extracting Structure-Based Cobol Components

Follow the instructions below to extract structure-based Cobol components.

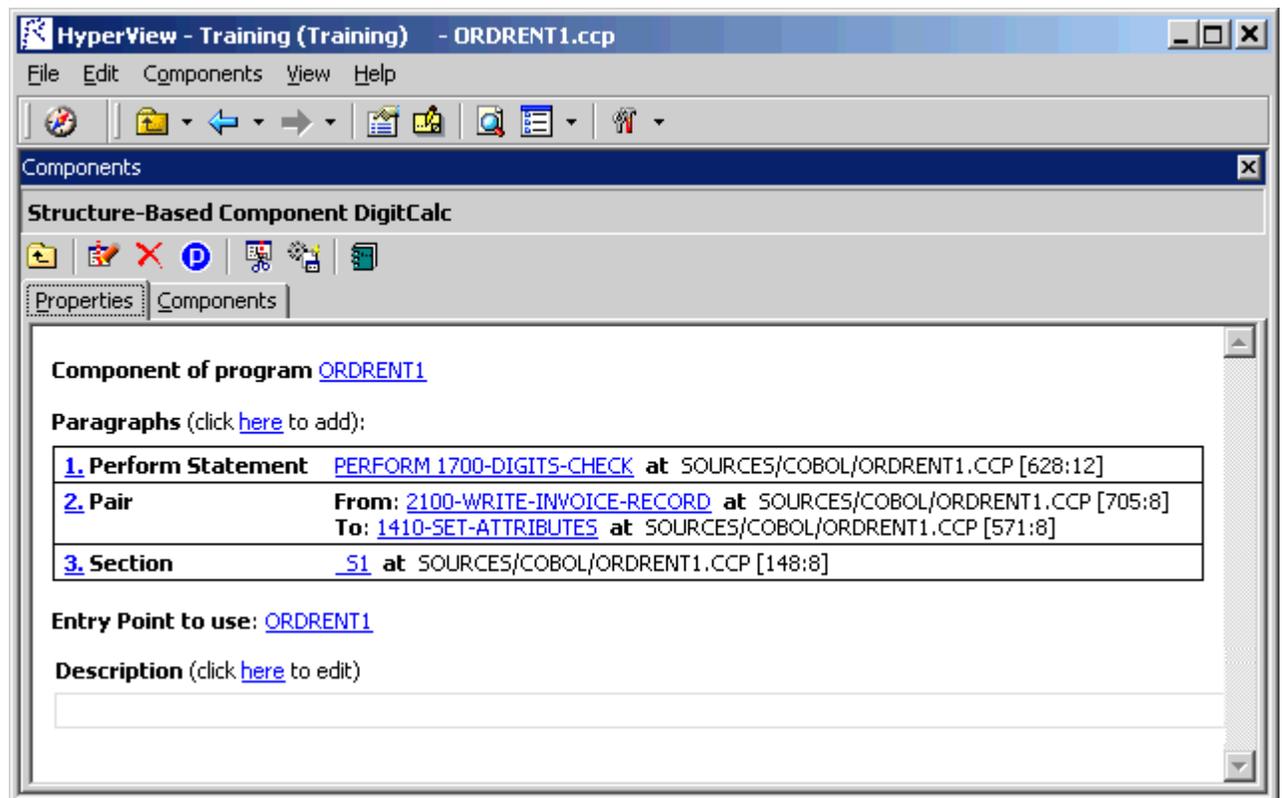
1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. In the **Paragraphs** field, click the **here** link. Choose one of the following methods in the pop-up menu:
 - **Paragraph Perform** to set the range to the paragraph or paragraphs performed by the selected PERFORM statement. Select the PERFORM statement in the Source pane, then click the link for the current selection and choose **Set** in the pop-up menu.
 - **Pair of Paragraphs** to set the range to the selected paragraphs. Select the first paragraph in the pair in the Source pane, then click the link for the current selection in the **From** field and choose **Set** in the drop-down menu. Select the second paragraph in the pair, then click the link for the current selection in the **To** field and choose **Set** in the pop-up menu.

 **Tip:** You can set the **From** and **To** fields to the same paragraph.

- **Section** to set the range to the paragraphs in the section. Select the section in the Source pane, then click the link for the current selection and choose **Set** in the pop-up menu.

 **Note:** To delete a range, select the link for the numeral that identifies the range and choose **Delete** in the pop-up menu. To unset a PERFORM, paragraph, or section, click it and choose **Unset** in the pop-up menu. To navigate quickly to a PERFORM, paragraph, or section in the source, click it and choose **Locate** in the pop-up menu.

3. Repeat this procedure for each range you want to extract. You can use any combination of methods. The figure below shows how the properties tab might look for a multi-range extraction.



4. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.
5. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
6. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
7. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
8. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Extracting Structure-Based PL/I Components

Follow the instructions below to extract structure-based PL/I components.

1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. Select the program entry point in the Source pane. In the **Point** field, click the link for the current selection and choose **Set** in the pop-up menu.
 **Note:** To unset an entry point, click it and choose **Unset** in the pop-up menu. To navigate quickly to an entry point in the source, click it and choose **Locate** in the pop-up menu.
3. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
4. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
5. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
6. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Extracting Structure-Based RPG Components

Follow the instructions below to extract structure-based RPG components.

1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. Select the subroutine or procedure you want to slice in the Source pane. In the **Point** field, click the link for the current selection and choose **Set** in the pop-up menu.
 **Note:** To unset an entry point, click it and choose **Unset** in the pop-up menu. To navigate quickly to an entry point in the source, click it and choose **Locate** in the pop-up menu.
3. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
4. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
5. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
6. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with

errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Extracting Computation-Based Components

Computation-Based Componentization lets you build a component that contains all the code necessary to calculate the value of a variable at a point in the program where it is used to populate a report attribute or screen. You can generate parameterized computation-based slices that make it easy to deploy the transformed component in distributed architectures.

Understanding Variable-Based Extraction

When you perform a computation-based extraction, you can slice by statement or by variable. What's the difference? Suppose you are interested in calculations involving the variable X in the example below:

```
MOVE 1 TO X  
MOVE 1 TO Y  
DISPLAY X Y.
```

If you perform statement-based extraction (if you slice on the statement DISPLAY X Y), all three statements will be included in the component. If you perform variable-based extraction (if you slice on the variable X), only the first and third statements will be included. In variable-based extraction, that is, Component Maker tracks the dependency between X and Y, and having determined that the variables are independent, excludes the MOVE 1 to Y statement.

 **Note:** If you slice on a variable for a Cobol component, you must select **Variable** in the Component Type Specific options for computation-based extraction.

Understanding Blocking

For Cobol programs, you can use a technique called *blocking* to produce smaller, better-defined parameterized components. Component Maker will not include in the slice any part of the calculation that appears before the blocked statement. Fields from blocked input statements are treated as input parameters of the component.

Consider the following fragment:

```
INP1.  
  DISPLAY "INPUT YEAR (1600-2099)".  
  ACCEPT YEAR.  
  CALL 'PROG' USING YEAR.  
  IF YEAR > 2099 OR YEAR < 1600 THEN  
    DISPLAY "WRONG YEAR".
```

If the CALL statement is selected as a block, then both the CALL and ACCEPT statements from the fragment are not included in the component, and YEAR is passed as a parameter to the component.

 **Tip:** Specify blocking in the blocking dialog accessed from the Interface options pane.

Understanding Parameterized Slices

For Cobol programs, you can generate *parameterized slices*, in which the input and output variables required by the component are organized in group-level structures. The component contains all the code required for input/output operations.

To extract a parameterized slice, select the **Generate Parameterized Components** option in the extraction options dialog. Note that you cannot generate a complement for a parameterized Cobol slice.

 **Note:** For parameterized structure- and computation-based componentization of Cobol programs, you must select the **Perform Program Analysis** and **Enable Parameterization of Components** options in the project verification options.

Cobol Naming Conventions

- Component input structures have names of the form BRE-INP-<STRUCT-NAME>. Input fields have names of the form BRE-I-<FIELD-NAME>.
- Component Output structures have names of the form BRE-OUT-STRUCT-NAME. Output fields have names of the form BRE-O-<FIELD-NAME>.

Parameterization Example

The example below illustrates how Component Maker generates parameterized slices. Consider a Cobol program that contains the following structures:

```
WORKING-STORAGE SECTION.  
  01 A  
    03 A1  
    03 A2  
  01 B  
    03 B1  
    03 B2  
    03 B4
```

Suppose that only A1 has been determined by Component Maker to be an input parameter, and only B1 and B2 to be output parameters. Suppose further that the component is extracted with input and output data structures that use the default names, BRE-INP-INPUT-STRUCTURE and BRE-OUT-OUTPUT-STRUCTURE, respectively, and with the default Optimization options set. The component contains the following code:

```
WORKING-STORAGE SECTION.  
  01 A  
    03 A1  
    03 A2  
  01 B  
    03 B1  
    03 B2  
    03 B4  
LINKAGE SECTION.  
  01 BRE-INP-INPUT-STRUCTURE  
    03 BRE-I-A
```

```

    06 BRE-I-A1
01 BRE-OUT-OUTPUT-STRUCTURE
    03 BRE-O-B
        06 BRE-O-B1
        06 BRE-O-B2
PROCEDURE DIVISION
    USING BRE-INP-INPUT-STRUCTURE BRE-OUT-OUTPUT-STRUCTURE .
BRE-INIT-SECTION SECTION.
    PERFORM BRE-COPY-INPUT-DATA .
    .....
    ....(Business Logic)....
    .....
*Modernization Workbench added statement
    GO TO BRE-EXIT-PROGRAM.
BRE-EXIT-PROGRAM-SECTION SECTION.
    BRE-EXIT-PROGRAM.
        PERFORM BRE-COPY-OUTPUT-DATA .
        GOBACK .
BRE-COPY-INPUT-DATA .
    MOVE BRE-I-A TO A .
BRE-COPY-OUTPUT-DATA .
    MOVE B TO BRE-O-B .

```

Extracting Computation-Based Cobol Components

Follow the instructions below to extract computation-based Cobol components.

1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. Select the variable or statement you want to slice on in the Source pane. In the **Point** field, click the link for the current selection and choose **Set** in the pop-up menu.

 **Note:** If you slice on a variable, you must select **Variable** in the Component Type Specific options for computation-based extraction.

To unset a variable or statement, click it and choose **Unset** in the pop-up menu. To navigate quickly to a variable or statement in the source, click it and choose **Locate** in the pop-up menu.

3. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.
4. If you plan to specify **Use Blocking from Component Definitions** in the Interface options, select the list of statements to block in Clipper, then click the link for the current selection in the **Block statements** field and choose **Set** in the drop-down menu.

 **Note:** Choose **Show** to display the current list in Clipper. Choose **(none)** to unset the list. For Clipper usage, see *Analyzing Programs* in the workbench documentation set.

5. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
6. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.

7. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
8. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Extracting Computation-Based Natural Components

Follow the instructions below to extract computation-based Natural components.

1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. Select the variable or statement you want to slice on in the Source pane. In the **Point** field, click the link for the current selection and choose **Set** in the pop-up menu.



Note: If you slice on a variable, you must select **Variable** in the Component Type Specific options for computation-based extraction.

To unset a variable or statement, click it and choose **Unset** in the pop-up menu. To navigate quickly to a variable or statement in the source, click it and choose **Locate** in the pop-up menu.

3. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
4. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
5. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
6. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Extracting Domain-Based Components

Domain-Based Componentization lets you "specialize" a program based on the values of one or more variables. The specialized program is typically intended for reuse "in place," in the original application but under new external circumstances.

After a change in your business practices, for example, a program that invokes processing for a "payment type" variable could be specialized on the value PAYMENT-TYPE = "CHECK". Component Maker isolates every process dependent on the CHECK value to create a functionally complete program that processes check payments only.

Two modes of domain-based componentization are offered:

- In simplified mode, you set the specialization variable to its value anywhere in the program except a data port. The value of the variable is "frozen in memory." Operations that could change the value are ignored.
- In advanced mode, you set the specialization variable to its value at a data port. Subsequent operations can change the value, following the data and control flow of the program.

Use the simplified mode when you are interested only in the final value of a variable, or when a variable never receives a value from outside the program. Use the advanced mode when you need to account for data coming into a variable (when the variable's value is repeatedly reset, for example). The next two sections describe these modes in detail.



Tip: Component Maker lets you set the specialization variable to a range of values (between 1 and 10 inclusive, for example) or to multiple values (not only CHECK but CREDIT-CARD, for example). You can also set the variable to all values not in the range or set of possible values (every value but CHECK and CREDIT-CARD, for example).

Understanding Program Specialization in Simplified Mode

In the simplified mode of program specialization, you set the specialization variable to its value anywhere in the program except a data port. The value of the variable is "frozen in memory." The table below shows the result of using the simplified mode to specialize on the values CURYEAR = 1999, MONTH = 1, CURMONTH = 12, DAY1 = 4, and CURDAY = 7.

Source Program	Specialized Program	Comment
<pre>INP3. DISPLAY "INPUT DAY". ACCEPT DAY1. MOVE YEAR TO tmp1. PERFORM ISV. IF DAY1 > tt of MONTHS (MONTH) OR DAY1 < 1 THEN DISPLAY "WRONG DAY".</pre>	<pre>INP3. DISPLAY "INPUT DAY". MOVE YEAR TO tmp1. PERFORM ISV. IF 0004 > TT OF MONTHS(MONTH) THEN DISPLAY "WRONG DAY" END-IF.</pre>	<p>ACCEPT removed.</p> <p>No changes in these statements (YEAR is a "free" variable).</p> <p>Value for DAY1 substituted. The 2nd condition for DAY1 is removed as always false. END-IF added.</p>

Source Program	Specialized Program	Comment
<pre> MAINCALC. IF YEAR > CURYEAR THEN MOVE YEAR TO INT0001 MOVE CURYEAR TO INT0002 MOVE 1 TO direction ELSE MOVE YEAR TO INT0002 MOVE 2 TO direction MOVE CURYEAR TO INT0001. </pre>	<pre> MAINCALC. IF YEAR > 1999 THEN MOVE YEAR TO INT0001 MOVE 1999 TO INT0002 MOVE 1 TO direction ELSE MOVE YEAR TO INT0002 MOVE 2 TO direction MOVE 1999 TO INT0001. </pre>	Value for CURYEAR substituted.
<pre> MOVE int0001 TO tmp3. MOVE int0002 TO tmp4. IF YEAR NOT EQUAL CURYEAR THEN PERFORM YEARS. </pre>	<pre> MOVE int0002 TO tmp4. IF YEAR NOT = 1999 THEN PERFORM YEARS. </pre>	Component Maker removes the first line for tmp3, because this variable is never used again. Value for CURYEAR substituted.
<pre> IF MONTH > CURMONTH THEN MOVE MONTH TO INT0001 MOVE CURMONTH TO INT0002 MOVE 1 TO direction </pre>		Value for MONTH substituted, making the condition (1>12) false, so Component Maker removes the IF branch and then the whole conditional statement as such.
<pre> ELSE MOVE MONTH TO INT0002 MOVE 2 TO direction MOVE CURMONTH TO INT0001. </pre>	<pre> MOVE 0001 TO INT0002 MOVE 2 TO direction MOVE 0012 TO INT0001. </pre>	The three unconditional statements remain from the former ELSE branch. Value for CURMONTH substituted.
<pre> IF MONTH NOT EQUAL CURMONTH THEN PERFORM MONTHS. </pre>	<pre> PERFORM MONTHS. </pre>	The condition is true, so the statement is made unconditional.
<pre> IF DAY1 > CURDAY THEN MOVE DAY1 TO INT0001 MOVE CURDAY TO INT0002 MOVE 1 TO direction </pre>		This condition (4>7) is false, so Component Maker removes the IF branch and then the whole conditional statement as such.
<pre> ELSE MOVE DAY1 TO INT0002 MOVE 2 TO direction MOVE CURDAY TO INT0001. </pre>	<pre> MOVE 4 TO INT0002 MOVE 2 TO direction MOVE 0007 TO INT0001. </pre>	The three unconditional statements remain from the former ELSE branch. Values for DAY1 and CURDAY substituted.

Source Program	Specialized Program	Comment
IF day1 NOT EQUAL CURDAY THEN PERFORM DAYS.	PERFORM DAYS.	The condition is true, so the statement is made unconditional.

Understanding Program Specialization in Advanced Mode

In the advanced mode of program specialization, you set the specialization variable to its value at a data port: any statement that allows the program to receive the variable's value from a keyboard, database, screen, or other input source. Subsequent operations can change the value, following the data and control flow of the program. The table below shows the result of using the advanced mode to specialize on the values MONTH = 1 and DAY1 = 4.

Source Program	Specialized Program	Comment
INP1. DISPLAY "INPUT YEAR (1600-2099)". ACCEPT YEAR. IF YEAR > 2099 OR YEAR < 1600 THEN DISPLAY "WRONG YEAR".	INP1. DISPLAY "INPUT YEAR (1600-2099)". ACCEPT YEAR. IF YEAR > 2099 OR YEAR < 1600 THEN DISPLAY "WRONG YEAR".	No changes in these statements (YEAR is a "free" variable).
INP2. DISPLAY "INPUT MONTH". ACCEPT MONTH. IF MONTH > 12 OR MONTH < 1 THEN DISPLAY "WRONG MONTH".	INP2. DISPLAY "INPUT MONTH". MOVE 0001 TO MONTH.	ACCEPT is replaced by MOVE with the set value for MONTH. With the set value, this IF statement can never be reached, so Component Maker removes it.
INP3. DISPLAY "INPUT DAY". ACCEPT DAY1. MOVE YEAR TO tmp1. PERFORM ISV. IF DAY1 > tt of MONTHS (MONTH) OR DAY1 < 1 THEN DISPLAY "WRONG DAY".	INP3. DISPLAY "INPUT DAY". MOVE 0004 TO DAY1. MOVE YEAR TO tmp1. PERFORM ISV. IF 0004 > TT OF MONTHS(MONTH) THEN DISPLAY "WRONG DAY" END-IF.	ACCEPT is replaced by MOVE with the set value for DAY1. No changes in these statements (YEAR is a "free" variable). The 2nd condition for DAY1 is removed as always false. END-IF added.

Understanding Program Specialization Lite

Ordinarily, you must turn on the **Perform Program Analysis** option in the project verification options before verifying the Cobol program you want to specialize. If your application is very large, however, and you know that the specialization variable is never reset, you can save time by skipping program analysis during verification and using the simplified mode to specialize the program, so-called "program specialization lite."

Component Maker gives you the same result for a lite extraction as it would for an ordinary domain extraction in simplified mode, with one important exception. Domain extraction lite cannot calculate the value of a variable that depends on the value of the specialization variable. Consider the following example:

```
01 X Pic 99.  
01 Y Pic 99.  
...  
MOVE X To Y.  
IF X = 1  
  THEN ...  
  ELSE ...  
END-IF.  
...  
IF Y = 1  
  THEN ...  
  ELSE ...  
END-IF.
```

If you set X to 1, both simplified mode and domain extraction lite resolve the IF X = 1 condition correctly. Only simplified mode, however, resolves the IF Y = 1 condition.

Extracting Domain-Based Cobol Components

Follow the instructions below to extract domain-based Cobol components.

1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. In the **Data Item Value** field, click the **here** link. Choose one of the following methods in the pop-up menu:
 - **HyperCode List** to set the specialization variable to the constant values in a list of constants.
 - **User Specified Value(s)** to set the specialization variable to a value or values you specify.
3. Select the specialization variable or its declaration in the Source pane. Click the link for the current selection in the **Data Item** field and choose **Set** in the drop-down menu. For advanced program specialization, you can enter a structure in **Data Item** and a field inside the structure in **Field**.
 **Note:** To delete an entry, select the link for the numeral that identifies it and choose **Delete** in the pop-up menu. To unset an entry, click it and choose **Unset** in the pop-up menu. To navigate quickly to a variable or declaration in the source, click it and choose **Locate** in the pop-up menu.
4. In the **Comparison** field, click the link for the current comparison operator and choose:
 - **equals** to set the specialization variable to the specified values.
 - **not equals** to set the specialization variable to every value but the specified values.
5. If you chose **HyperCode List**, select the list of constants in Clipper, then click the link for the current selection in the **List Name** field and choose **Set** in the drop-down menu.
 **Note:** Choose **Show** to display the current list in Clipper. Choose **(none)** to unset the list. For Clipper usage, see *Analyzing Programs* in the workbench documentation set.
6. If you chose **User Specified Value(s)**, click the **here** link in the **Values** field. Choose one of the following methods in the pop-up menu:
 - **Value** to set the specialization variable to one or more values. In the **Value** field, click the link for the current selection. A dialog opens where you can enter a value in the text field. Click **OK**.

 **Note:** Put double quotation marks around a string constant with blank spaces at the beginning or end.

- **Value Range** to set the specialization variable to a range of values. In the **Lower** field, click the link for the current selection. A dialog opens where you can enter a value for the lower range end in the text field. Click **OK**. Follow the same procedure for the **Upper** field.

 **Note:** For value ranges, the specialization variable must have a numeric data type. Only numeric values are supported.

7. Repeat this procedure for each value or range of values you want to set and for each variable you want to specialize on. For a given specialization variable, you can specify the methods in any combination. For a given extraction, you can specify simplified and advanced modes in any combination.

 **Note:** To delete a value or range, select the link for the numeral that identifies it and choose **Delete** in the pop-up menu.

8. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.
9. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
10. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
11. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
12. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Extracting Domain-Based PL/I Components

Follow the instructions below to extract domain-based PL/I components.

 **Note:** Not-equals comparisons and value ranges are not supported in PL/I.

1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. In the **Data Item Value** field (to set a single specialization variable) or the **Value for Data Item List** (to set a list of specialization variables), click the **here** link. Choose one of the following methods in the pop-up menu:
 - **HyperCode List** to set the specialization variable(s) to the constant values in a list of constants.
 - **User Specified Value(s)** to set the specialization variable(s) to a value or values you specify.
3. If you are setting:
 - A single specialization variable, select the specialization variable or its declaration in the Source pane. Click the link for the current selection in the **Data Item** field and choose **Set** in the drop-down

menu. For advanced program specialization, you can enter a structure in **Data Item** and a field inside the structure in **Field**.

- A list of specialization variables, click the link for the current selection and choose the list of variables or declarations to use in the pop-up menu.

 **Note:** To delete an entry, select the link for the numeral that identifies it and choose **Delete** in the pop-up menu. To unset an entry, click it and choose **Unset** in the pop-up menu. To navigate quickly to a variable or declaration in the source, click it and choose **Locate** in the pop-up menu.

4. If you chose **HyperCode List**, select the list of constants in Clipper, then click the link for the current selection in the **List Name** field and choose **Set** in the drop-down menu.

 **Note:** Choose **Show** to display the current list in Clipper. Choose **(none)** to unset the list. For Clipper usage, see *Analyzing Programs* in the workbench documentation set.

5. If you chose **User Specified Value(s)**, click the **here** link in the **Values** field. Choose one of the following methods in the pop-up menu:

- **Value** to set the specialization variable to one or more values. In the **Value** field, click the link for the current selection. A dialog opens where you can enter a value in the text field. Click **OK**.

 **Note:** Put double quotation marks around a string constant with blank spaces at the beginning or end.

- **Value Range** to set the specialization variable to a range of values. In the **Lower** field, click the link for the current selection. A dialog opens where you can enter a value for the lower range end in the text field. Click **OK**. Follow the same procedure for the **Upper** field.

 **Note:** For value ranges, the specialization variable must have a numeric data type. Only numeric values are supported.

6. Repeat this procedure for each value or range of values you want to set and for each variable you want to specialize on. For a given specialization variable, you can specify the methods in any combination. For a given extraction, you can specify simplified and advanced modes in any combination.

 **Note:** To delete a value or range, select the link for the numeral that identifies it and choose **Delete** in the pop-up menu.

7. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.

8. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.

9. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.

10. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Injecting Events

Event Injection lets you adapt a legacy program to asynchronous, event-based programming models like MQ Series. You specify candidate locations for event calls (reads/writes, screen transactions, or subprogram calls, for example); the type of operation the event call performs (put or get); and the text of the message.

For a put operation, for example, Component Maker builds a component that sends the message and any associated variable values to a queue, where the message can be retrieved by monitoring applications.



Tip: The HyperView Clipper pane lets you create lists of candidate locations for event injection. Use the predefined searches for file ports, screen ports, and subprogram calls, or define your own searches. For Clipper pane usage, see Analyzing Programs in the workbench document set.

Understanding Event Injection

Suppose that you have a piece of code that checks whether the variables YEAR and MONTH belong to admissible ranges:

```
IF YEAR > 2099 OR YEAR < 1600 THEN
  MOVE "WRONG YEAR" TO DOW1
ELSE
  IF MONTH > 12 OR MONTH < 1 THEN
    MOVE "WRONG MONTH" TO DOW1
  ELSE
    MOVE YEAR TO tmp1
  PERFORM ISV
```

Suppose further that you want to send a message to your MQ Series middleware each time valid dates are entered in these fields, along with the value that was entered for YEAR. Here, in schematic form, is the series of steps you would perform in Component Maker to accomplish these tasks.

1. In HyperView, create a list that contains the MOVE YEAR TO tmp1 statement in Clipper.
2. In Component Maker, create a logical component with the following properties:
 - Component of program: select the program that contains the fragment.
 - List: select the HyperView list.
 - Insert: specify where you want event-handling code to be injected, before or after the injection point. In our case, after the MOVE statement.
 - Operation: select the type of operation you want the event-handling code to perform, put or get. Since we want to send a message to middleware, we choose put.
 - Include Values: specify whether you want the values of variables at the injection point to be included with the generated message. Since we want to send the value of YEAR with the message, we choose true.
 - Message: specify the text of the message you want to send. In our case, the text is "Valid dates entered".
3. In Component Maker, extract the logical component, making sure to set the **Use Middleware** drop-down in the Component Type Specific options for the extraction to MQ.

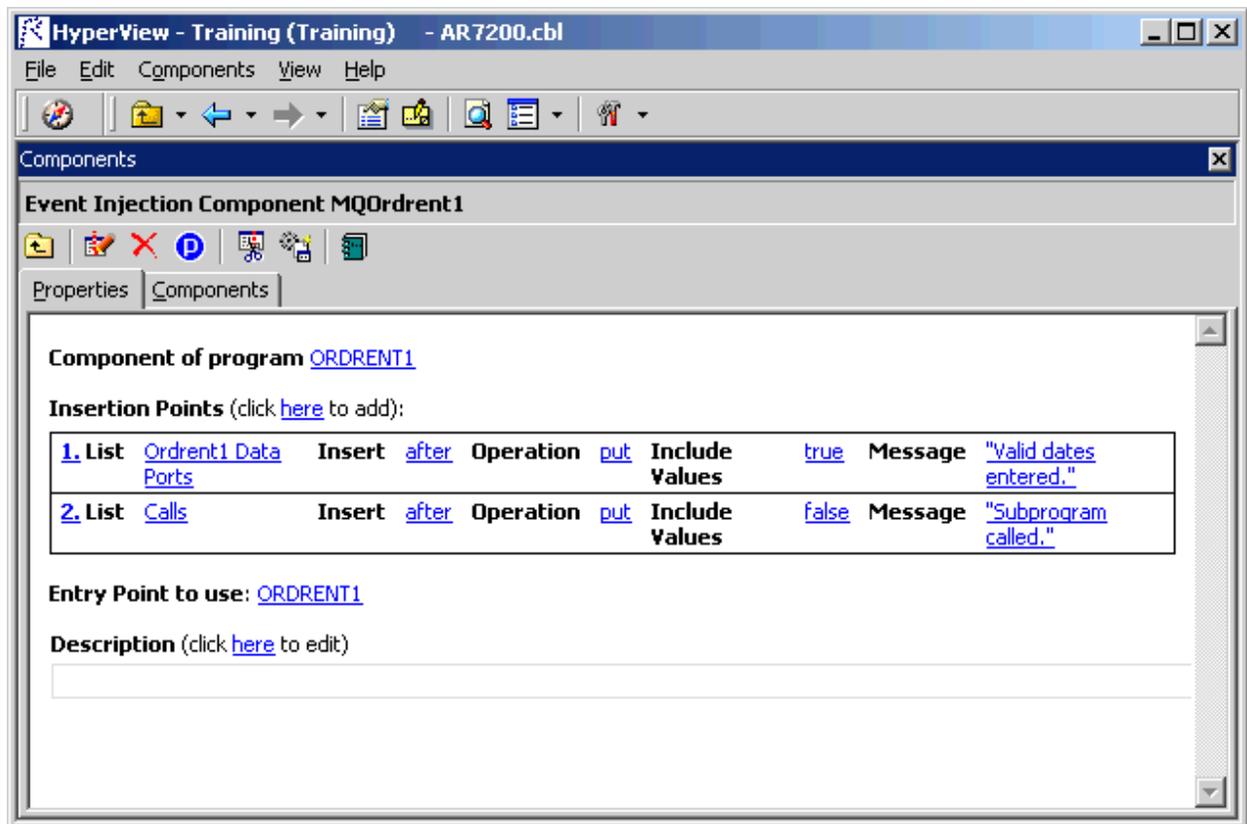
The result of the extraction appears below. Notice that Component Maker has arranged to insert the text of the message and the value of the YEAR variable into the buffer, and added the appropriate PERFORM PUTQ statements to the code.

```
IF YEAR > 2099 OR YEAR < 1600 THEN
  MOVE "WRONG YEAR" TO DOW1
ELSE
IF MONTH > 12 OR MONTH < 1 THEN
  MOVE "WRONG MONTH" TO DOW1
ELSE
  MOVE '<TEXT Value= "Valid dates
  entered"></TEXT>' TO BUFFER
  PERFORM PUTQ
  STRING '<VAR Name= "YEAR" Value=
  "' YEAR "'></VAR>'
  '<VAR Name= "TMP1" Value= "' TMP1 "'></VAR>'
  DELIMITED BY SIZE
  INTO BUFFER END-STRING
  PERFORM PUTQ
  MOVE YEAR TO tmp1
  PERFORM ISV
```

Extracting Event-Injected Cobol Components

Follow the instructions below to extract event-injected Cobol components.

1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. In the **Insertion Points** field, click the **here** link. In Clipper, select the list of injection points, then click the link for the current selection in the **List** field and choose **Set** in the drop-down menu.
 **Note:** Choose **Show** to display the current list in Clipper. Choose **(none)** to unset the list. For Clipper usage, see *Analyzing Programs* in the workbench documentation set.
3. In the **Insert** field, click the link for the current selection and choose:
 - **after** to inject event-handling code after the selected injection point.
 - **before** to inject event-handling code before the selected injection point.
4. In the **Operation** field, click the link for the current selection and choose:
 - **put** to send a message to middleware.
 - **get** to receive a message from middleware.
5. In the **Include Values** field, click the link for the current selection and choose **true** if you want the values of variables at the injection point to be included with the generated message, **false** otherwise.
6. In the **Message** field, click the link for the current message. A dialog opens where you can enter the text for the event message in the text field. Click **OK**.
7. Repeat this procedure for each list of candidate injection points. For a given extraction, you can specify the properties for the selected lists in any combination. The figure below shows how the properties tab might look for an extraction with multiple lists.



8. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.
9. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
10. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
11. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
12. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Eliminating Dead Code

Dead Code Elimination (DCE) is an option in each of the main component extraction methods, but you can also perform it on a standalone basis. For each program analyzed for dead code, DCE generates a component that consists of the original source code minus any unreferenced data items or unreachable procedural statements. Optionally, you can have DCE comment out dead code in Cobol and Natural applications, rather than remove it.

 **Note:** Use the batch DCE feature to find dead code across your project. If you are licensed to use the Batch Refresh Process (BRP), you can use it to perform dead code elimination across a workspace.

Generating Dead Code Statistics

Set the **Perform Dead Code Analysis** option in the project verification options if you want the parser to collect statistics on the number of unreachable statements and dead data items in a program, and to mark the constructs as dead in HyperView. You can view the statistics in the Legacy Estimation tool, as described in *Analyzing Projects* in the workbench documentation set.

 **Note:** You do not need to set this option to perform dead code elimination in Component Maker.

For Cobol programs, you can use a DCE coverage report to identify dead code in a source program. The report displays the text of the source program with its "live," or extracted, code shaded in pink.

Understanding Dead Code Elimination

Let's look at a simple before-and-after example to see what you can expect from Dead Code Elimination.

Before:

```
WORKING-STORAGE SECTION.  
  
    01 USED-VARS.  
       05 USED1 PIC 9.  
  
    01 DEAD-VARS.  
       05 DEAD1 PIC 9.  
       05 DEAD2 PIC X.  
  
PROCEDURE DIVISION.  
  
FIRST-USED-PARA.  
    MOVE 1 TO USED1.  
    GO TO SECOND-USED-PARA.  
    MOVE 2 TO USED1.  
  
DEAD-PARA1.  
    MOVE 0 TO DEAD2.
```

```
SECOND-USED PARA.  
MOVE 3 TO USED1.  
STOP RUN.
```

After:

```
WORKING-STORAGE SECTION.
```

```
01 USED-VARS.  
05 USED1 PIC 9.
```

```
PROCEDURE DIVISION.
```

```
FIRST-USED-PARA.  
MOVE 1 TO USED1.  
GO TO SECOND-USED-PARA.
```

```
SECOND-USED PARA.  
MOVE 3 TO USED1.  
STOP RUN.
```

Extracting Optimized Components

Follow the instructions below to extract optimized components for all supported languages.

1. Select the program you want to analyze for dead code in the HyperView Objects pane and click the  button. To analyze the entire project of which the program is a part, click the  button.
2. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new components to the list of components. If you selected batch mode, Component Maker creates a logical component for each program in the project, appending *_n* to the name of the component.
3. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.
4. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
5. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
6. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
7. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Performing Entry Point Isolation

Entry Point Isolation lets you build a component based on one of multiple entry points in a legacy program (an inner entry point in a Cobol program, for example) rather than the start of the Procedure Division. Component Maker extracts only the functionality and data definitions required for invocation from the selected point.

Entry Point Isolation is built into the main methods as an optional optimization tool. It's offered separately in case you want to use it on a stand-alone basis.

Extracting a Cobol Component with Entry Point Isolation

Follow the instructions below to extract a Cobol Component with entry point isolation.

1. Select the program you want to slice in the HyperView Objects pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new component to the list of components. Double-click the component to edit its properties.
2. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.
3. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
4. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
5. The Extraction Options dialog opens. Set options for the extraction and click **Finish**.
6. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No**.

Technical Details

This appendix gives technical details of Component Maker behavior for a handful of narrowly focused verification and extraction options; for Cobol parameterized slice generation; for domain-based extraction when the specialization variable is set to multiple values; and for Cobol arithmetic exception handling.

Verification Options

This section describes how a number of verification options may affect component extraction. For more information on the verification options, see *Preparing Projects* in the workbench document set.

Use Special IMS Calling Conventions

Select **Use Special IMS Calling Conventions** in the project verification options if you want to show dependencies and analyze CALL 'CBLTDLI' statements for the CHNG value of their first parameter, and if the value of the third parameter is known, then generate Calls relationship in the repository.

For example:

```
MOVE 'CHNG' TO WS-IMS-FUNC-CODE
MOVE 'MGRW280' TO WS-IMS-TRANSACTION
CALL 'CBLTDLI' USING WS-IMS-FUNC-CODE
    LS03-ALT-MOD-PCB
    WS-IMS-TRANSACTION
```

When both WS-IMS-FUNC-CODE = 'CHNG' and WS-IMS-TRANSACTION have known values, the repository is populated with the CALL relationship between the current program and the WS-IMS-TRANSACTION <value> program (in the example, 'MGRW280').

Override CICS Program Terminations

Select **Override CICS Program Terminations** in the project verification options if you want the parser to interpret CICS RETURN, XCTL, and ABEND commands in Cobol files as not terminating program execution.

If the source program contains CICS HANDLE CONDITION handlers, for example, some exceptions can arise only on execution of CICS RETURN. For this reason, if you want to see the code of the corresponding handler in the component, you need to check the override box. Otherwise, the call of the handler and hence the handler's code are unreachable.

Support CICS HANDLE Statements

Select **Support CICS HANDLE statements** in the project verification options if you want the parser to recognize CICS HANDLE statements in Cobol files. EXEC CICS HANDLE statements require processing to

detect all dependencies with error-handling statements. That may result in adding extra paragraphs to a component.

Perform Unisys TIP and DPS Calls Analysis

Select **Perform Unisys TIP and DPS Calls Analysis** in the project verification options if you are working on a project containing Unisys 2200 Cobol files and need to perform TIP and DPS calls analysis.

This analysis tries to determine the name (value of the data item of size 8 and offset 20 from the beginning of form-header) of the screen form used in input/output operation (at CALL 'D\$READ', 'D\$SEND', 'D\$SEND', 'D\$SEND1') and establish the repository relationships ProgramSendsMap and ProgramReadsMap between the program being analyzed and the detected screen.

For example:

```
01 SCREEN-946 .
  02 SCREEN-946-HEADER .
  05 FILLER PIC X(2)VALUE SPACES .
  05 FILLER PIC 9(5)COMP VALUE ZERO .
  05 FILLER PIC X(4)VALUE SPACES .
  05 S946-FILLER PIC X(8) VALUE '$DPS$SWS'
  05 S946-NUMBER PIC 9(4) VALUE 946 .
  05 S946-NAME PIC X(8) VALUE 'SCRN946' .
CALL 'D$READ USING DPS-STATUS, SCREEN-946 .
```

Relationship ProgramSendsMap is established between the program and screen 'SCRN946'.



Note: Select **DPS routines may end with error** if you want to perform call analysis of DPS routines that end in an error.

Perform Unisys Common-Storage Analysis

Select **Perform Unisys Common-Storage Analysis** in the project verification options if you want the system to include in the analysis for Unisys Cobol files variables that are not explicitly declared in CALL statements. This analysis adds implicit use of variables declared in the Common Storage Section to every CALL statement of the program being analyzed, as well as for its PROCEDURE DIVISION USING phrase. That could lead to superfluous data dependencies between the caller and called programs in case the called program does not use data from Common Storage.

Relaxed Parsing

The **Relaxed Parsing** option in the workspace verification options 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.

For code verified with relaxed parsing, Component Maker behaves as follows:

- Statements included in a component that contain errors are treated as CONTINUE statements and appear in component text as comments.
- Dummy declarations for undeclared identifiers appear in component text as comments.
- Declarations that are in error appear in component text as they were in the original program. Corrected declarations appear in component text as comments.
- Commented-out code is identified by an extra comment line: "Modernization Workbench assumption".

For Domain-Based Componentization, in particular:

- Data items with errors in declarations are treated as data items with unknown values.
- Statements with errors are treated as statements that do not change values.
- Whenever a calculation error occurs, the comment "Calculation has not been completed successfully by Modernization Workbench" is generated in the component before the erroneous operator, along with an error message.
- Component Maker ignores user values for duplicated identifiers (which may have an association with a wrong DECL); structures with fields marked with errors; and undeclared identifiers are ignored. A list of ignored values appears at the top of the component.
- Users cannot specify values for VARs with attribute errors (duplicated identifiers); VARs without DECLs (undeclared identifiers); and DECLs with attribute errors.

PERFORM Behavior for Micro Focus Cobol

For Micro Focus Cobol applications, use the PERFORM behavior option in the workspace verification options window to specify the type of PERFORM behavior the application was compiled for. You can select:

- **Stack** if the application was compiled with the PERFORM-type option set to allow recursive PERFORMS.
- **All exits active** if the application was compiled with the PERFORM-type option set to not allow recursive PERFORMS.

For non-recursive PERFORM behavior, a COBOL program can contain PERFORM mines. In informal terms, a PERFORM mine is a place in a program that can contain an exit point of some active but not current PERFORM during program execution.

The program below, for example, contains a mine at the end of paragraph C. When the end of paragraph C is reached during PERFORM C THRU D execution, the mine "snaps" into action: control is transferred to the STOP RUN statement of paragraph A.

```
A.
  PERFORM B THRU C.
  STOP RUN.
B.
  PERFORM C THRU D.
C.
  DISPLAY 'C'.
  * mine
D.
  DISPLAY 'D'.
```

Setting the compiler option to allow non-recursive PERFORM behavior where appropriate allows the Modernization Workbench parser to detect possible mines and determine their properties. That, in turn, lets Component Maker analyze control flow and eliminate dead code with greater precision. To return to our example, the mine placed at the end of paragraph C snaps each time it is reached: such a mine is called stable. Control never falls through a stable mine. Here it means that the code in paragraph D is unreachable.

Keep Legacy Copybooks Extraction Option

Select **Keep Legacy Copybooks** in the General extraction options for Cobol if you want Component Maker not to generate modified copybooks for the component. Component Maker issues a warning if including the original copybooks in the component would result in an error.

Example 1:

```
[COBOL]
01 A PIC X.
PROCEDURE DIVISION.
COPY CP.
[END-COBOL]
[ COPYBOOK CP.CPY]
STOP RUN.
DISPLAY A.
[END-COPYBOOK CP.CPY]
```

For this example, Component Maker issues a warning for an undeclared identifier after Dead Code Elimination.

Example 2:

```
[COBOL]
PROCEDURE DIVISION.
COPY CP.
STOP RUN.
P.
[END-COBOL]
[ COPYBOOK CP.CPY]
DISPLAY "QA is out there"
STOP RUN.
PERFORM P.
[END-COPYBOOK CP.CPY]
```

For this example, Component Maker issues a warning for an undeclared paragraph after Dead Code Elimination.

Example 3:

```
[COBOL]
working-storage section.
copy file.
PROCEDURE DIVISION.
p1.
  move 1 to a.
p2.
  display b.
  display a.
p3.
  stop run.
[END-COBOL]
[ COPYBOOK file.cpy]
01 a pic 9.
01 b pic 9.
[END-COPYBOOK file.cpy]
```

For this example, the range component on paragraph p2 looks like this:

```
[COBOL]
WORKING-STORAGE SECTION.
  COPY FILE1.
  LINKAGE SECTION.
  PROCEDURE DIVISION USING A.
[END-COBOL]
```

while, with the option turned off, it looks like this:

```
[COBOL]
WORKING-STORAGE SECTION.
  COPY FILE1-A$RULE-0.
  LINKAGE SECTION.
```

```
COPY FILE1-A$RULE-1.  
[END-COBOL.]
```

That is, turning the option on overrides the splitting of the copybook file into two files. Component Maker issues a warning if that could result in an error.

How Parameterized Slices Are Generated for Cobol Programs

The specifications for input and output parameters are:

- Input

A variable of an arbitrary level from the LINKAGE section or PROCEDURE DIVISION USING is classified as an input parameter if one or more of its bits are used for reading before writing.

A system variable (field of DFHEIB/DFHEIBLK structures) is classified as an input parameter if the **Create CICS Program** option is turned off and the variable is used for writing before reading.

- Output

A variable of an arbitrary level from the LINKAGE section or PROCEDURE DIVISION USING is classified as an output parameter if it is modified during component execution.

A system variable (a field of DFHEIB/DFHEIBLK structures) is classified as an output parameter if the **Create CICS Program** option is turned off and the variable is modified during component execution.

- For each input parameter, the algorithm finds its first usage (it does not have to be unique, the algorithm processes all of them), and if the variable (parameter from the LINKAGE section) is used for reading, code to copy its value from the corresponding field of BRE-INPUT-STRUCTURE is inserted as close to this usage as possible.
- The algorithm takes into account all partial or conditional assignments for this variable before its first usage and places PERFORM statements before these assignments.

If a PERFORM statement can be executed more than once (as in the case of a loop), then a flag variable (named BRE-INIT-COPY-FLAG-[<n>] of the type PIC 9 VALUE 0 is created in the WORKING-STORAGE section, and the parameter is copied into the corresponding variable only the first time this PERFORM statement is executed.

- For all component exit points, the algorithm inserts code to copy all output parameters from working-storage variables to the corresponding fields of BRE-OUTPUT-STRUCTURE.

Variables of any level (rather than only 01-level structures together with all their fields) can act as parameters. This allows exclusion of unnecessary parameters, making the resulting programs more compact and clear.

For each operator for which a parameter list is generated, the following transformations are applied to the entire list:

- All FD entries are replaced with their data descriptions.
- All array fields are replaced with the corresponding array declarations.
- All upper-level RENAMES clauses are replaced with the renamed declarations.
- All upper-level REDEFINES clauses with an object (including the object itself, if it is present in the parameter list) are replaced with a clause of a greater size.
- All REDEFINES and RENAMES entries of any level are removed from the list.
- All variable-length arrays are converted into fixed-length of the corresponding maximal size.
- All keys and indices are removed from array declarations.

- All VALUE clauses are removed from all declarations.
- All conditional names are replaced with the corresponding data items.

Setting a Specialization Variable to Multiple Values

For Domain-Based Componentization, Component Maker lets you set the specialization variable to a range of values (between 1 and 10 inclusive, for example) or to multiple values (not only CHECK but CREDIT-CARD, for example). You can also set the variable to all values not in the range or set of possible values (every value but CHECK and CREDIT-CARD, for example).

Component Maker uses multiple values to predict conditional branches intelligently. In the following code fragment, for example, the second IF statement cannot be resolved with a single value, because of the two conflicting values of Z coming down from the different code paths of the first IF. With multiple values, however, Component Maker correctly resolves the second IF, because all the possible values of the variable at the point of the IF are known:

```
IF X EQUAL Y
  MOVE 1 TO Z
ELSE
  MOVE 2 TO Z
DISPLAY Z.
IF Z EQUAL 3
  DISPLAY "Z=3"
ELSE
  DISPLAY "Z<>3"
```

Keep in mind that only the following COBOL statements are interpreted with multiple values:

- COMPUTE
- MOVE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

That is, if the input of such a statement is defined, then, after interpretation, its output can be defined as well.

Single-Value Example:

```
MOVE 1 TO Y.
MOVE 1 TO X.
ADD X TO Y.
DISPLAY Y.
IF Y EQUAL 2 THEN...
```

In this fragment of code, the value of Y in the IF statement (as well as in DISPLAY) is known, and so the THEN branch can be predicted.

Multiple-Value Example:

```
IF X EQUAL 0
  MOVE 1 TO Y
ELSE
  MOVE 2 TO Y.
ADD 1 TO Y.
IF Y = 10 THEN... ELSE...
```

In this case, Component Maker determines that Y in the second IF statement can equal only 2 or 3, so the statement can be resolved to the ELSE branch.

The statement interpretation capability is available only when you define the specialization variable "positively" (as equalling a range or set of values), not when you define the variable "negatively" (as not equalling a range or set of values).

Arithmetic Exception Handling

For Cobol, the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements can have ON SIZE ERROR and NOT ON SIZE ERROR phrases. The phrase ON SIZE ERROR contains an arithmetic exception handler.

Statements in the ON SIZE ERROR phrase are executed when one of the following arithmetic exception conditions take place:

- The value of an arithmetic operation result is larger than the resultant-identifier picture size.
- Division by zero.
- Violation of the rules for the evaluation of exponentiation.

For MULTIPLY arithmetic statements, if any of the individual operations produces a size error condition, the statements in the ON SIZE ERROR phrase is not executed until all of the individual operations are completed.

Control is transferred to the statements defined in the phrase NOT ON SIZE ERROR when a NOT ON SIZE ERROR phrase is specified and no exceptions occurred. In that case, the ON SIZE ERROR is ignored.

Component Maker specialization processes an arithmetic statement with exception handlers in the following way:

- If a (NOT) ON SIZE ERROR condition occurred in some interpreting pass, then the arithmetic statement is replaced by the statements in the corresponding phrase.
- Those statements will be interpreted at the next pass.

Index

A

Application Architect 6–10, 12–18, 20–23, 25–27, 29,
31–35, 37–39, 41, 42, 44–49, 51–53

C

Component Maker 6–10, 12–18, 20–23, 25–27, 29,
31–35, 37–39, 41, 42, 44–49, 51–53

L

Logic Analyzer 6–10, 12–18, 20–23, 25–27, 29,
31–35, 37–39, 41, 42, 44–49, 51–53